Which types have a unique inhabitant?

Gabriel Scherer

Doctorat d'Informatique
Université Paris-Diderot

Which types have a unique inhabitant?

Focusing on pure program equivalence

Gabriel Scherer
under the supervision of Didier Rémy

Defense: March 30th, 2016
Last manuscript update: April 30, 2016

Jury: Roberto Di Cosmo, Didier Rémy, Sam Lindley, Dale Miller, Gilles Dowek, Olivier Laurent

Abstract

Some programming language features (coercions, type-classes, implicits) rely on inferring a part of the code that is determined by its usage context. In order to better understand the theoretical underpinnings of this mechanism, we ask: when is it the case that there is a *unique* program that could have been guessed, or in other words that all possible guesses result in equivalent program fragments? Which types have a unique inhabitant?

To approach the question of unicity, we build on work in proof theory on more canonical representation of proofs. Using the proofs-as-programs correspondence, we can adapt the logical technique of focusing to obtain more canonical program representations.

In the setting of simply-typed lambda-calculus with sums and the empty type, equipped with the strong $\beta\eta$-equivalence, we show that uniqueness is decidable. We present a saturating focused logic that introduces irreducible cuts on positive types "as soon as possible". Goal-directed proof search in this logic gives an effective algorithm that returns either zero, one or two distinct inhabitants for any given type.

Contents

Introduction

Motivation

This thesis is concerned with the following question:

Which types have a *unique* inhabitant?

This is a question about computer programming.

A *program* is a recipe that a computer should follow to build a particular piece of data (file, picture, sound, etc.) or behavior (interaction with the user). We distinguish *executions* of the program (what happens when the computer follows the recipe) from the *description* of the program, in a symbolic form that humans programmers can understand, manipulate and change.

A *type* is an interface that computer programs, or fragments of computer programs, may or may not provide to its users. A program term is said to *inhabit* a type when it respects the described interface; a type for a function that adds number, for example, may specify that it takes two positive integers and returns a positive integer. A *type system* is a formal description of a collection of types and program terms, along with rules to explain which terms inhabit which types.

Which types have a unique inhabitant? I think that this question is both of theoretical and practical interest. In the following sections of this introduction, we discuss the motivations for studying this question, but we should first have a few words about the approach, and correspondingly the nature of the results that are presented in this document.

To make our question precise, we must precisely specify the type system we consider. There are many, from the very simple to the extremely sophisticated. We must also be more precise about what we mean by *unique*: a type is uniquely inhabited if all the programs at this type are equal, but what does it mean for two programs to be equal? Program equality, or equivalence, is a rich and subtle notion. Fortunately, for the simple enough type systems that we consider in this thesis, there is a natural choice of equivalence that we use to define unicity.

A result that is now folklore among programming language researchers is that there is a correspondence between computer programs and formal (mathematical) proofs: we can understand formal proofs as specific kinds of well-behaved programs, whose interface is described by the logical *statement* they prove. By giving two very different points of view on these objects, the correspondence has helped transfer intuitions, ideas and results in both directions: from proof theory to programming language theory, and vice-versa.

This correspondence, called the Curry-Howard correspondence, is of interest to us for at least two reasons. First, the question of *unicity* (is there exactly one ...) is closely related to its older sister, *existence* (is there at least one...). Existence of programs at a given arbitrary type is arguably peripheral for programming (it does have interesting application, but is not central to the craft) whereas it is a central question in logic, as it corresponds to the question of whether a given logical statement is *provable* – existence of at least one proof. There is a rich field of research concerned with the question of "which statements are provable?", looking for practical way to automatically answer it for restricted classes of statements. We may look at their ideas and techniques, and try to adapt them to answer the stronger question of unicity.

Second, proof theory is often concerned with the question of what is an *appropriate* representation of proofs. There is some intuition that some proofs are "obviously the same", and proof theoreticians tend to prefer representation systems where not too many proofs with distinct representations are "obviously the same". Eliminating such duplicates allow

them to better understand what proofs really are, and may also have practical benefits: in a tighter representation system with fewer possible proofs, it may be easier to tell if any given statement has a proof, because the proof search process has less proof candidates to consider. Proof theory comes with a large body of work on such representations (such as focused proofs, proof nets, connection-based methods, etc.), and a natural question is whether those can be re-used in our quest for unicity.

Unfortunately, it is not obvious to define which proofs are "obviously the same" (many different notions have been proposed), because it is not clear exactly which requirements such a notion should satisfy. In particular, while most mathematicians expect that the mathematical proofs they produce can be elaborated into fully formal proofs, the question of identity of proofs is not of central importance to them. Simplicity, generality of proofs matters; one also wonders about their dependencies (which existing theory they use in the course of the proof, for example). Two proofs may look very similar (if they share their key arguments), or completely unrelated, but there is no evident criterion for whether they are "the same".

On the contrary, there is a clear definition of whether two programs are equivalent: provided the same inputs, do they behave in the same way, in particular do they return the same outputs? There are other things we could wish to observe (is one program faster than the other?), but there is a natural idea of "same input, same outputs" or more generally "same environment, same observable behavior" that gives a good notion of equivalence.

And this is where we come in. This rich body of work on the question of existence and representations of *proofs* carries many interesting ideas, but before applying them to *programs* we need to pay close attention to their treatment of equality. Some techniques remove redundant proofs in a way that corresponds to throwing away some possible programs: they must be avoided. Some techniques remove only duplicates that are equivalent as programs, but not all of them, so they can be reused but need to be strengthened to precisely decide unicity. Furthermore, the fact that they preserve the identity of programs may be true but have never been proved (or even asked) before, as their authors were satisfied with the weaker property of preserving provability. It is time to revisit them with programming mind.

Some sensible thesis writing advice suggests to center the document around *a thesis*, a central claim that the whole document supports. Here is our take on this helpful exercise:

> The proof theoretic technique of *focusing* can be adapted and reused to reason about programs, provided one gives a careful look at its treatment of the identity of proofs.

Background: programming language design, and how we go about it

Humans programmers have invented many different symbolic representations for computer programs, which are called programming *languages*. One can think of them as languages used to communicate with the computer, but it is important to remember that programming is also a social activity, in the sense that many programs are created by a collaboration of several programmers, or that programs written by one programmer may be reused, inspected or modified by others. Programs communicate intent to a computer, but also to other human programmers.

Programmers routinely report frustration with the limitations of the programming language they use – it is very hard to design a *good* programming language. At least the three following qualities are expected:

- *concision*: Simple tasks should be described by simple, not large or complex programs. Complex tasks require complex programs, but their complexity should come

solely from the problem domain (the specificity of the required task), not accidental complexity imposed by the programming language.

For example, early Artificial Intelligence research highlighted the need for language-level support for *backtracking* (giving up on a series of decisions made toward a goal to start afresh through a different method), and some programming languages make this substantially easier than others.

- *clarity*: By reading a program description it should be easy to understand the *intent* of its author(s). We say that a program has a *bug* (a defect) when its meaning does not coincide with the intent of its programmers – they made a mistake when transcribing their thoughts into programs. Clarity is thus an essential component of safety (avoiding program defects), and should be supported by mechanized tools to the largest possible extent. To achieve clarity, some language constructions help programmers express their intent, and programming language designers work on tools to automatically verify that this expressed intent is consistent with the rest of the program description.

  For example, one of the worst security issues that was discovered in 2014 (failure of all Apple computers or mobile phones to verify the authenticity of connections to secure websites) was due to a single line of program text that had been duplicated (written twice instead of only once). The difference between the programmer intent (ensure security of communications) and the effective behavior of the program (allowing malicious network nodes to inspect your communications with your online bank) was dramatic, yet neither the human programmers nor the automated tools used by these programmers reported this error.

- *consistency*: A programming language should be regular and structured, making it easy for users to guess how to use the parts of the language they are not already familiar with. In particular, consistency supports clarity, as recovering intent from program description requires a good knowledge of the language: the more consistent, the more predictable, the lower the risks of misunderstanding. This is an instance of a more general design principle, the principle of least surprise.

Of course, the list above is to be understood as the informal opinion of a practitioner, rather than a scientific claim in itself. Programming is a rich field that spans many activities, and correspondingly programming language *research* can and should be attacked from many different angles: mathematics (formalization), engineering, design, human-machine interface, ergonomics, psychology, linguistics, sociology, and the working programmers all have something to say about how to make better programming languages.

This thesis was conducted within a research group – and a research sub-community – that uses mathematical formalization as its main tool to study, understand and improve programming languages. To work with a programming language, we give it one or several formal semantics (defining programs as mathematical objects, and their meaning as mathematical relations between programs and their behavior); we can thus prove theorems about programming languages themselves, or about formal program analyses or transformations.

The details of how mathematical formalization can be used to guide programming language design are rather fascinating – it is a very abstract approach of a very practical activity. The community shares a common baggage of properties that may or may not apply to any given proposed design, and are understood to capture certain usability properties of the resulting programming language. These properties are informed by practical experience using existing languages (designed using this methodology or not), and our understanding of them evolves over time.

Having a formal semantics for the language of study is a solid way to acquire an understanding of what the programs in this language *mean*, which is a necessary first step for

clarity – the meaning of a program cannot be clear if we do not first agree on what it is. Formalization is a difficult (technical) and time-consuming activity, but its simplification power cannot be understated: the formalization effort naturally suggests many changes that can dramatically improve consistency. By encouraging to build the language around a small core of independent concepts (the best way to reduce the difficulty of formalization), it can also improve concision, as combining small building blocks can be a powerful way to simply express advanced concepts. Finding the *right* building blocks, however, is still very much dependent of domain knowledge and radical ideas often occur through prototyping or use-case studies, independently of formalization. Our preferred design technique would therefore be formalization and implementation co-evolution, with formalization and programming activities occurring jointly to inform and direct the language design process.

The presented thesis work was not started as an attempt to design a complete programming language, but rather to study one specific aspect of programming: the situations where the computer can automatically "guess" some program fragments that the user left unfilled – we call this *code inference*. This capability exists in several existing programming languages, and is bound to be employed in many future languages as well. We wish to develop a theoretical understanding of the following question: when can we be *sure* that the guess made by the computer was the correct answer?

Motivation: Unicity as the ideal code inference criterion

Many existing code inference features have a history that can be traced to one or both of the following simple concepts, that occur rather naturally to programmers – or other users of formal, precise notations, such as engineers or mathematicians – during their activity.

- *coercions* are transformations of values from one representation to another that are natural, in the sense that there is one (and, hopefully, only one) way to do it that makes sense in all situations. For example, consider an information system that represents various categories of persons by a data record containing information about them. A "user" is described by a name and an email address, a "student" by a name, an email address, and a department of studies. There is a natural, obvious way to turn a data record representing a student into a data record representing an user, simply dropping the department information. A programmer that manipulates a student record may want to be able to pass it to another program fragment expecting a user record, without having to write tedious conversion code. It expects the programming language to implicitly "coerce" from one format to the other.

  In mathematics, some well-defined embeddings play this role of coercions; for example, any natural number can be "seen as" a real number (although the way they are defined often gives them different representation: the natural number may or may not have to be transformed to become its real number equivalent).

- *disambiguation* is the process of selecting, for a symbol that may have several distinct significations, one meaning that is appropriate in the context of a given use-case. For example, the addition symbol (+) may mean several different operations with fairly different properties, such as addition of natural numbers, of real numbers, of complex matrices, of ordinals, etc. But for any given occurrence of the symbol (+), it should be clear from the context which of these operation we mean.

Both these concepts have simple motivations but, once extended to the scale of full programming languages, they may become complex and raise difficult questions. For example, for some expressive enough programming language, the questions of whether a given type of data may be coerced into another may be an undecidable problem. Disambiguation may seem at first to be a purely local problem, but there are sometimes good reason to delay a disambiguation choice. For example, if I define the function $\texttt{double} \stackrel{\mathsf{def}}{=} x \mapsto x + x$,

I may not have a particular addition operation in mind, but instead expect to be able to use this function `double` on any sort of values for which an addition operation is defined. In other words, the meaning of the symbol (+) becomes an implicit parameter of the function `double`.

In the general case, the resolution of these situations has been formulated as a *constraint satisfaction* problem: given some basic facts (here are the basic coercion rules, the basic disambiguation rules) and a set of rules to deduce new facts, does the specific coercion or disambiguation problem posed by this specific point in the program have a solution that can be deduced from these facts and rules? If yes, we can inspect this solution to understand how to transform a data into another, or which unambiguous operation to use; if no, the program is ambiguous, incomplete and should be rejected.

In my opinion (this is not a scientific claim), it is time to recognize that we are actually inferring a program fragment, rather than an arbitrary constraint resolution problem. A coercion can be represented by the code of a transformation function. Disambiguation can be seen as the inference of a type-correct program among a set of programs determined by the symbol to disambiguate. The constraint problem can thus be formulated as the guessing of a program fragment; the state of the resolution, as a partially inferred program – plus some bookkeeping information. Of course, the set of possible programs representing valid solutions is constrained (not every function is a valid coercion). These restrained programs may be constrained by a stricter environment, stricter validity and formation rules, than arbitrary programs. The question of whether the situation is unambiguous now boils down to the question of whether, in this restricted setting, the possible programs are uniquely inhabited.

It may seem, at first, that this approach gives up on some flexibility and freedom in designing a constraint resolution strategy. However, thanks to the correspondence between proof and programs, we know that the design space remains huge: searching a program is just as expressive a satisfiability problem as search for a proof in these corresponding logics. The sophisticated techniques developed for proof search can be transferred to these term inference formulation.

Furthermore, forcing ourselves to formulate these problems as the search for a valid program in some type system has design benefits. When a first attempt at formulating an inference problem fails to be non-ambiguous (the types involved are not uniquely inhabited), the language designer needs to reformulate the problem and restrict it to recover unicity. A term-inference formulation gives us a natural toolkit of design choices to make: restricting the environment available to the elaborated programs, restricting the rules that determine program validity, etc. I believe that these techniques will "blend in" the general programming language design better than arbitrary rules coming from a constraint-solving formulation; at the very least, they can be explained to the programmer users using concepts they are already familiar with.

Method: focusing towards canonicity

Focusing is a technique of proof theory to restrict a given logic to eliminate sources of redundancy from its proof representation. One gets a logic that is equivalent to the original one in terms of expressivity – it proves the same statements – and has a more structured representation of proofs.

In the present thesis, we will instead make use of focusing on typed programs. The adaptation is direct, and gives a discipline that does not restrict a language's expressivity – all programs remain expressible – but imposes more structural constraints on program representations. This is good to decide unicity: we want to know if all possible programs are equivalent, and focusing gives us fewer candidates to test for equivalence.

Ideally, we would like a type system in which there are no duplicates to test: each expressible behavior has a unique representation. Focusing does not give us this property,

but it is a good building block to start from. In this thesis, we propose a variant of focusing, *saturated focusing*, that achieves this property for the simple type system we are studying.

An interesting side-result of this work is a focusing-based approach to program *equivalence*. Once we have a redundancy-free representation that is also complete, we can test whether two given programs are equivalent by converting them to this representation, and simply checking that they have the exact same representation. In particular, our technique provides a way to test equivalence between categories of program for which no equivalence procedure was previously known, namely simply-typed programs in presence of an empty type 0.

Plan

Background

The first part of this thesis, Part I (Background), gives an introduction to the scientific topics covered. Because a thesis is also a personal document, we get much more freedom to make choice about their structure than in an article (tightly constrained by size limits and presentation norms), and the choice made here is to attempt to be as self-contained as possible: I hope that, eventually (in the internet age, a thesis is also a living document that can be amended following readers' feedback), anyone with a scientific background should be able to follow those introductory chapters. Of course, we cannot be exhaustive in our covering of those introduction topics, but instead focus on the parts that are relevant to the later contributions. We will point to more complete reference material.

These chapters are thus mostly about things that are already well-known in the community, not new contributions of the thesis. I have made on a few occasions choices that may interest the reader, and will try to point them out in this plan. Expert readers, you should feel free to skip the rest. Really, do skip it! I have included back-references to these chapters in the other parts of the thesis, so there is no risk of missing important background content.

In Chapter 1 (Introduction to the formal study of logic: natural deduction), we give a self-contained introduction to a way to define proofs, and the formal study of logic, that is very close to the type theory also used to study typed programming languages. More precisely, we study natural deduction – in presence of disjunctions (sums). The chapter concludes on the usual proof of consistency (by measuring the types appearing in elimination-introduction pairs), in absence of disjunctions, as this usual proof fails in presence of disjunctions.

In Chapter 2 (Introduction to the formal study of programming: the λ-calculus), we give a self-contained introduction to the λ-calculus (untyped then typed), which is the fundamental calculus for the study of functional programming languages. Experts may be interested to note that we tried to actually motivate the apparition of types (instead of vaguely speaking of "reasoning about programs" or promising a normalization result that jumps out of nowhere), which is difficult in the pure λ-calculus (functions only) as dynamic failure never happens there. We made a detour through λ-calculi with several constructors to understand failure. I think this is important: the study of programming is not only about allowing programs to be expressed, but also about preventing errors in programs.

In Chapter 3 (Curry-Howard of reduction and equivalence), we expose the Curry-Howard correspondence, which is a tight relation between logics (and their proofs) and type systems (and their programs). We also study program equivalence, look at its counterpart on proofs, and take the tourist deviation through category theory (this part is not self-contained) to justify strong η-rules for disjunction and the empty type. As a side-result of the study of program equivalence in presence of disjunction elimination, we fix the consistency proof of Chapter 1 to prove consistency of the whole logic, sums included.

Expert readers will remark that because we have insisted that, in proofs, the hypothesis context are *sets* of formulas, our correspondence between proof and programs is not as tight as it is usually crafted to be (by cunningly defining logic judgments with multi-set of formulas, which makes little sense if one just wants to understand provability without any λ-afterthoughts); we only have a forward simulation result (β-reductions in programs are valid substitutions in proofs) instead of an isomorphism.

In Chapter 4 (A better proof system: sequent calculus), we introduce the sequent calculus. It is a better proof system than natural deduction, and in particular its cut-elimination procedure gives us a direct (no fix required) proof of consistency of intuitionistic logic with disjunctions. It is also a worse support for programming languages than natural deduction, and in particular the identity of proofs in sequent calculus looks more problematic – but we still define a term language for it for convenience, which may be of interest to readers that have never looked at a (naive and not that useful) term syntax for the sequent calculus. We will mention that sequent calculus is easily extended to define classical logic, and make a remark on the multi-succedent presentations of intuitionistic logic – there is an intuitionistic negative sum!

In Chapter 5 (The bothersome equivalence of cut-free sequent proofs), we discuss the identity of sequent proofs, and in particular the equivalence relations between sequents that correspond to $\beta\eta$-equivalence for natural deduction. While the sequent calculus is arguably more harmonious and regular than natural deduction, and thus better-suited for proof search, its notion of equivalence is rather more bureaucratic. Developing this bothersome equivalence is useful to serve as a basis specification, to evaluate refined notions equivalences in more structured calculi – focused or polarized calculi.

Chapter 6 (Proof and type systems, in general) gives a precise definition of concepts related to proof or type systems, and in particular how those systems relate to each other – for example, there are several notions of completeness of interest. As applications, we develop some concepts that hold for proof search in many of the systems we consider: the subformula property, and the positive (covariant) and negative (contravariant) positions for subformulas. Decidability of provability in intuitionistic and classical propositional logic is directly obtained from the subformula property.

Chapter 7 (Focusing in sequent calculus) is also mostly an introductory chapter, but it covers a more advanced topic that many in the programming-language community are not familiar with (focusing was discovered in proof theory in 1992 [Andreoli, 1992b]). Most introductions to focusing are done using linear logic; in the interest of space we present focusing for intuitionistic logic directly, but still in the usual setting of sequent-calculus. There are many different choices of presentation of focused systems that are made in the literature, and we try to cover the main ones and how to transition from one presentation to the other.

Finally, Chapter 8 (Semantics) presents some results on program equivalence. Contextual equivalence is defined, as well as a more "semantic" equivalence obtained by interpreting atoms as ground formulas. We show that $\beta\eta$-equivalence is sound with respect to those equivalences.

Focusing on program equivalence

The second part of this thesis, Part II (Focusing for program equivalence and unique inhabitation), presents contributions on questions related to program equivalence that have been obtained during this thesis. It is rather clear that a good understanding of equivalence is necessary to attack unicity, and these chapters, which started off as diversions from the thing I was supposed to work on, ended up being invaluable in understanding the relation between a practically-justified algorithm we were developing for unicity checking, on one hand, and focusing and polarization on the other.

In Chapter 9 (Counting terms and proofs), we give a preliminary result on the number of distinct programs that share the same shape as a logical derivation. In particular, we prove that it suffices to consider program environments with at most two variables of each type to tell if there are two distinct programs of the same logical shape. This result is used to obtain termination result for our unicity-checking algorithm.

In Chapter 10 (Focused λ-calculus), we give a term system for focused natural deduction that is designed to be as close as possible to the usual λ-calculus. This contrasts with the

usual presentations using the sequent calculus, more convenient to work with as a logic, but it has the advantage of easier transfer of our λ-calculist intuitions. The statement of our completeness result, which guarantees preservation of computational behavior, is slightly stronger than the usual completeness results for focused system which only claim preservation for provability – the usual proofs do not need to be changed to provide this strengthened statement.

Chapter 11 (Saturation logic for canonicity) is thus the first chapter to actually cover the question that started the thesis: "which types have a unique inhabitant?". It describes an algorithm to decide unicity for the simply-typed λ-calculus in presence of sums, building on the previous work on focusing – although we in fact understood the idea of the algorithm before we realized it was maximal multi-focusing in disguise. A central contribution to answer this question is a canonical *saturating* intuitionistic logic, a variant of multi-focusing that allows goal-directed proof search – whereas maximal multi-focused proofs have no goal-directed search procedure as the minimality criterion is highly non-local.

Finally, Chapter 12 (From the logic to the algorithm: deciding unicity) concludes by describing the unicity-checking algorithm arising from the canonical proof system of the previous chapter. We prove that it is correct and terminates on any input.

Part I.

Background

In this part we summarize the basic concepts needed to follow the exposition of the contributions of this thesis, which are presented in the following parts. It is also an occasion to present the particular notations and conventions we follow.

This is a curious exercise, because there is a natural urge to try to make it as self-contained as possible, so that non-specialists that would be curious about this work could have a good chance of following it if they are motivated. On the other hand, this is not the time and space to write a good *course* on these topics. We tried to only present results that will be useful for the other parts, and resisted to the temptation of including remarks on the many other perspectives opened by this introductory material.

1. Introduction to the formal study of logic: natural deduction

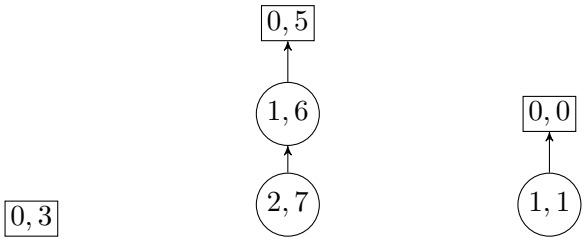
1.1. A first introduction to inference rules

1.1.1. Derivation trees and their notation

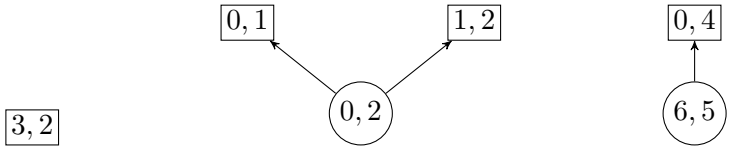
In this warm-up section, we will study a highly simplified notion of *formula* and *proof*, in order to introduce general concepts that will be used throughout this document. In the following chapters, we will use richer definitions, but for now we define a *formula* as a pair of natural numbers m and n, and a *proof* as a *tree* of formulas, satisfying the following rules:

- all the leaves of the tree are pairs of the form $(0, n)$
- all the nodes have exactly one child sub-tree (the tree is list-like), and if the sub-tree has the formula (m, n) at its root, then the tree has the formula $(m + 1, n + 1)$

For example, the following trees are valid proof trees (we draw leaves with a rectangular box):



and the following are not:



Question (to the reader) What are the integers (m, n) such that there exists a proof with root (m, n)?

Answer They are exactly the pairs (m, n) such that $m \leq n$.

As an introduction to the notions used in this thesis, we will prove it formally in this section.

Notation 1.1.1 (proof trees)**.**
We use the following notation to represent proof trees. We write

$$\frac{}{0 \preccurlyeq n}$$

for the leaf tree $(0, n)$, and

$$\frac{m \preccurlyeq n}{m+1 \preccurlyeq n+1}$$

for the node $(m+1, n+1)$, placing the sub-tree of root (m, n) above it.

For example, the valid trees we presented previously can be written as follows:

$$\frac{}{0 \preccurlyeq 3} \qquad \frac{\dfrac{\dfrac{}{0 \preccurlyeq 5}}{1 \preccurlyeq 6}}{2 \preccurlyeq 7} \qquad \frac{\dfrac{}{0 \preccurlyeq 0}}{1 \preccurlyeq 1}$$

We use the variable Π (and cousins: Π', Π_3, ...) to name proof trees. We can also write $\Pi :: m \preccurlyeq n$ to represent a proof, when Π has conclusion (m, n).

We have defined a very simple notion of *proofs*, specialized to proving facts of the form $m \leq n$, as purely syntactic objects (trees). In the rest of the section, we see how to formally prove things about those proof objects. This formal study will convince ourselves that they indeed capture evidence that $m \leq n$, and give us a few useful tools to work with such syntactic objects: structural induction, derivability, and admissibility. Logic, as done by computer scientists, uses these same techniques on more complex notions of proof objects, that represent richer mathematical propositions than just $m \leq n$.

We call the trees in our syntax using vertical bars *derivations*, and the tree-forming rules are called *inference rules*. The inference rules of our system (giving them names for future reference) are

$$\frac{}{0 \preccurlyeq n}\ \textsc{leq-zero} \qquad \frac{m \preccurlyeq n}{m+1 \preccurlyeq n+1}\ \textsc{leq-succ}$$

and they completely determine what the valid proofs are. Note that they are in fact "schemas", in the sense that each rule describes an infinite family of valid proof-formers, for all instances of m and n. In the other sections (for other logics), we will "define" the set of valid proofs by simply giving the inference rules that generate all valid proof trees.

In the general case an inference rule TOTO may be of the form

$$\frac{\mathcal{J}_1 \qquad \mathcal{J}_2 \qquad \dots}{\mathcal{J}}\ \textsc{toto}$$

with arbitrary many children. The $\mathcal{J}, \mathcal{J}_1, \dots$ are the things being proved, we call them *judgments*. The rule can be intuitively understood as "if all the things above the bar are true, then the thing below is true" – but the rule *is* the way to construct proofs, the syntactic evidence of truth. We call $\mathcal{J}$ the *conclusion* of this rule, and the $\mathcal{J}_1, \mathcal{J}_2, \dots$ the *premises* of this rule – they need to be filled with sub-derivations to get a complete derivation.

1.1.2. Proofs by structural induction

Theorem 1.1.1 (Soundness)**.**
For any natural numbers m and n, if there exists a valid derivation $\Pi :: m \preccurlyeq n$, then indeed we have $m \leq n$.

We claimed that proof trees of conclusion $m \preccurlyeq n$ represent proofs of the fact $m \leq n$. This theorem tells this *interpretation* of proof objects is *sound*, correct. To prove it, we use the proof technique of *structural induction*, which is a generalization of proofs by recurrence on natural numbers to arbitrary tree-like structures (in particular derivations).

Structural induction To prove that a property $P(\Pi)$ is true of all valid derivations Π, we prove that:

- P holds of all leaf derivations formed by rules without premises – in our case, LEQ-ZERO.
- For any inference rule

$$\frac{\mathcal{J}_1 \qquad \mathcal{J}_2 \qquad \dots}{\mathcal{J}}$$

  we can prove that it holds for a proof $\Pi :: \mathcal{J}$ whose root node uses this rule, assuming that P holds of all of its sub-proofs with conclusions $\mathcal{J}_1, \mathcal{J}_2, \dots$. In our case, we get to assume that $P(\Pi)$ holds for some $\Pi :: m \preccurlyeq n$, and we need to prove that P also holds of the derivation

$$\frac{\Pi :: m \preccurlyeq n}{m+1 \preccurlyeq n+1}\ \text{LEQ-SUCC}$$

This generalizes recurrence on natural numbers, as you can represent natural numbers as trees with one leaf rule (zero) and one one-child rule (successor), and then structural induction on those trees is exactly recurrence on natural numbers.

This can also be justified from recurrence on natural numbers, by presenting it as a recurrence on the *height* of proof derivations. Just as natural recurrence can be extended to "strong recurrence" (proving $P(m)$ by assuming $P(n)$ for all $n < m$), we will occasionally use "strong induction" (proving $P(\Pi)$ by using the induction hypothesis on all sub-derivations of Π, not only its direct children).

Finally, it is sometimes useful to reason by induction not only on the children of Π or on its sub-derivations, but on all valid derivations of strictly smaller size, or height (as trees), than Π. This let us inspect sub-derivations of Π and, sometimes, transform them in a way required by the proof, as long as we do not increase their size (or height). In this case, we will indicate that we are performing a strong induction on the size (or height) of the derivations, not the derivations themselves.

Proof (Theorem 1.1.1 (Soundness))**.** By structural induction on the proofs $\Pi :: m \preccurlyeq n$.

In the LEQ-ZERO case we have $\Pi :: 0 \preccurlyeq n$, and it is true that $0 \leq n$.

In the LEQ-SUCC case, Π is of the form

$$\frac{\Pi' :: m \preccurlyeq n}{m+1 \preccurlyeq n+1}$$

By induction hypothesis on Π' we may assume $m \leq n$, and thus we have $m+1 \leq n+1$. □

To the non-specialist reader: the small square □ at the right of the last paragraph is a conventional notation indicating that the proof that was ongoing is now finished. This visual cue is helpful if you want to browse the text quickly, skipping the proofs. Similarly, we give closing symbols to remarks (∗) and examples (◊).

1.1.3. Partial derivations and derivability

A derivation is *complete* if all the leaves of the proof correspond to rules with no premises (leaf rules). It is often convenient to manipulate *partial* derivations, that is valid compositions of rules with some missing subtree(s), for example:

$$\dfrac{\dfrac{\dfrac{m \preccurlyeq n}{m+1 \preccurlyeq n+1}\,\text{LEQ-SUCC}}{m+2 \preccurlyeq n+2}\,\text{LEQ-SUCC}}{m+3 \preccurlyeq n+3}\,\text{LEQ-SUCC}$$

This is a partial derivation: it is an *incomplete* proof of $m+3 \preccurlyeq n+3$, which needs a derivation of $m \preccurlyeq n$ to become a complete derivation. Note that the judgment at the leaf of the proof has no bar on top of it (it is *not* justified by a rule with no premises); we call it an *open leaf* of the partial proof – by opposition to *closed leaves*, the judgments justified by a rule with no premises, in our setting always of the form $0 \preccurlyeq n$.

Definition 1.1.1 Derivability**.**
We say that a judgment $\mathcal{J}$ is *derivable from* a set of judgments $\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_n$ if there exists a partial proof of $\mathcal{J}$, whose open leaves are among the $\mathcal{J}_1, \ldots, \mathcal{J}_n$. We just proved that $m+3 \preccurlyeq n+3$ is derivable from $m \preccurlyeq n$ for any m, n, and we can generalize this.

Lemma 1.1.2.
For any natural numbers m, n, k, the judgment $m+k \preccurlyeq n+k$ is derivable from $m \preccurlyeq n$.

Proof. Immediate, by recurrence/induction on k. For $k=0$ we take the empty partial derivation: if filled with a complete proof of $m \preccurlyeq n$, it becomes a complete derivation of $m+0 \preccurlyeq n+0$. In the successor case, assume we have a partial derivation Π_k of $m+k \preccurlyeq n+k$, with open leaf $m \preccurlyeq n$, then the derivation Π_{k+1} defined as

$$\dfrac{\dfrac{\Pi_k :: m+k \preccurlyeq n+k}{m+k+1 \preccurlyeq n+k+1}\,\text{LEQ-SUCC}}{}$$

is a partial derivation of $m+(k+1) \preccurlyeq n+(k+1)$ as expected, and it has an open leaf with the judgment $m \preccurlyeq n$, as part of its sub-derivation Π_k. □

Remark 1.1.1. "Lemma" is a technical word to describe something we claim is formally true because we have a proof, but which is of lesser importance than a "theorem". We usually demonstrate several auxiliary lemmas before claiming each theorem; theorems are supposed to formulate the grand results. ∗

1.1.4. Admissibility

Definition 1.1.2 Admissibility**.**
We say that a judgment $\mathcal{J}$ is *admissible* from the judgments $\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_n$ if, given complete proofs $\Pi_1 :: \mathcal{J}_1, \ldots, \Pi_n :: \mathcal{J}_n$, we can construct a complete proof of $\mathcal{J}$.

If by "construct" we meant just "plug the $\Pi_1, \ldots, \Pi_n$ as subtrees of a partial proof", this would be equivalent to the notion of derivability. Admissibility is more general, as we accept any procedure that produces a complete proof of $\mathcal{J}$; for example, it is possible to build a proof by case analysis on the structure of the proofs $\Pi_1, \ldots, \Pi_n$.

Notation 1.1.2 (admissible rule)**.**
To say that $\mathcal{J}$ is admissible from $\mathcal{J}_1, \ldots, \mathcal{J}_n$ for the reason FOO, we write

$$\dfrac{\mathcal{J}_1 \quad \cdots \quad \mathcal{J}_n}{\mathcal{J}}\,\text{FOO}$$

and we say that FOO is an *admissible rule*. This notation can be composed to create larger admissible rules, possibly mixed with valid inference rules.

Lemma 1.1.3 (Transitivity)**.**
The following transitivity rule is admissible:

$$\dfrac{m \preccurlyeq n \qquad n \preccurlyeq p}{m \preccurlyeq p}\ \textsc{leq-trans}$$

Proof. By structural induction on the proof $\Pi_{mn} :: m \preccurlyeq n$. (The induction property $P(\Pi :: m \preccurlyeq n)$ is the following: "for any p and proof Π_{np} of $n \preccurlyeq p$, the judgment $m \preccurlyeq p$ is provable".)

If Π_{mn} is of the form

$$\dfrac{}{0 \preccurlyeq n}$$

then we have $m = 0$, and we can prove $m \preccurlyeq p$ with simply

$$\dfrac{}{0 \preccurlyeq p}$$

If Π_{mn} is of the form

$$\dfrac{\Pi_{m'n'} :: m' \preccurlyeq n'}{m'+1 \preccurlyeq n'+1}$$

then $m = m'+1$ and $n = n'+1$. In this case, the proof $\Pi_{np} :: n \preccurlyeq p$ cannot be a LEQ-ZERO, as n is strictly larger than 0, so it is itself of the form

$$\dfrac{\Pi_{n'p'} :: n' \preccurlyeq p'}{n'+1 \preccurlyeq p'+1}$$

with $p = p'+1$. By induction hypothesis on $\Pi_{m'n'}$, a sub-derivation of Π_{mn}, we get a complete derivation $\Pi_{m'p'} :: m' \preccurlyeq p'$, and we can conclude with the complete derivation

$$\dfrac{\Pi_{m'p'} :: m' \preccurlyeq p'}{m'+1 \preccurlyeq p'+1}$$

of the judgment $m \preccurlyeq p$. □

Let us emphasize that this transitivity rule is admissible but not derivable: we have not simply plugged proofs of $m \preccurlyeq n$ and $n \preccurlyeq p$, unchanged, in a larger derivation. On the contrary, we have peeled them off, looking at sub-derivations of them. In fact, none of the inference rules of the final proof come of either input derivations, they were just built by looking at the shape of the inputs – looking at a strictly smaller sub-derivation at each induction step, which makes this proof technique valid.

1.1.5. Completeness

We have now seen the main tools used to work with proofs, presented as derivations of inference rules. We can conclude this section with the result of *completeness*, which tells us that whenever the mathematical fact $m \leq n$ (m is smaller than n) is true, then there is a corresponding derivation of the judgment $m \preccurlyeq n$ – soundness (Theorem 1.1.1) only told us that those of the $m \preccurlyeq n$ that could be proved really satisfied $m \leq n$, but there may be $m \leq n$ pairs for which no derivation exists. With soundness and completeness together, we know that $m \preccurlyeq n$ is provable exactly when $m \leq n$ holds.

Theorem 1.1.4.
If $m \leq n$, then $m \preccurlyeq n$ is provable.

Proof. To have a convincing proof, we need a precise definition of $m \leq n$ – we have

handled this statement rather informally so far. Let us define the relation $(\leq)$ between natural numbers as the reflexive transitive closure of the relation that has all $m \leq m+1$; that is, we say that $m \leq n$ if either $m = n$ (reflexivity), or $n = m+1$ (the relation $m \leq m+1$), or there is some k such that $m \leq k$ and $k \leq n$ (transitivity).

To show that $m \leq n$ implies that $m \preccurlyeq n$ is provable, is thus to prove that the three following rules are admissible:

$$\frac{}{n \preccurlyeq n}\ \textsc{leq-refl} \qquad \frac{}{n \preccurlyeq n+1}\ \textsc{leq-succ'} \qquad \frac{m \preccurlyeq k \qquad k \preccurlyeq n}{m \preccurlyeq n}\ \textsc{leq-trans}$$

The rule LEQ-TRANS has already been proved admissible by Lemma 1.1.3. The two other rules are direct consequences of Lemma 1.1.2 (derivability, and thus admissibility, of $m + k \preccurlyeq n + k$ from $m \preccurlyeq n$ for any k). The rule LEQ-REFL $(0 + n \preccurlyeq 0 + n)$ is shown admissible by plugging the proof of $0 \preccurlyeq 0$, and the rule LEQ-SUCC' $(0 + n \preccurlyeq 1 + n)$ by plugging the proof of $0 \preccurlyeq 1$, both constructed by LEQ-ZERO. □

1.2. Propositional intuitionistic logic

In the previous section, we used simple pairs of natural numbers (m, n) to represent the statement "m is less than n"; pairs are just inert mathematical objects, but we chose to *interpret* them as those statements. We extend this to a richer language of *formulas*, that we can interpret as describing many more statements.

What we call *logic* here is a set of judgments (the judgments that we interpret as being "true"). One may then study general properties of this set (for example, it may be the case that, for any judgment of a certain shape in the set, another judgment of a slightly different shape is also in the set), but this is not what we do directly.

Instead, we define a *proof system*, which is given by a family of inference rules as we have already seen: a *proof* of a judgment in this proof system is a valid derivation using these inference rules. Each proof system determines a logic (the set of judgments that have a valid complete proof), but different proof systems may correspond to the same logic (there are several examples in this thesis). Each proof system may make it easier or harder to study a particular aspect of the logic; choosing a good proof system is important, and we will also discuss some criteria that make a proof system comfortable and useful.

1.2.1. Formally defining the formulas

We describe in Figure 1.1 a *grammar* of the formulas we consider – they will be the judgments of our proof system.

Figure 1.1.: Formulas of the propositional intuitionistic logic

A, B, C, D	$::=$	formulas
	$\mid X, Y, Z$	atoms
	$\mid A \times B$	conjunction ("and")
	$\mid A + B$	disjunction ("or")
	$\mid A \to B$	implication ("if" .. "then" ..)
	$\mid 1$	true
	$\mid 0$	false

Figure 1.1 defines a grammar of formulas. Formulas are objects with the following structure: a formula is either an atom (or "atomic formula"), or a conjunction of formulas, or a disjunction of formulas, or an implication of formulas, or true or false. For example, $(X \to Y) \times 0$ is a valid formula: it is a conjunction whose left formula is an implication of atoms, and whose right formula is the false formula. We call the operator symbols $(\times, +, \to)$ *connectives*, as they connect formulas together to build larger formulas. Because

we are building a logic (rather than a type system), we call them *logical connectives*. We give a name to this logic we are defining, because we may also manipulate variants of it and other logics, using their names to distinguish them: we call this logic $\mathsf{PIL}(\to, \times, 1, +, 0)$ – I like descriptive names.

A quick remark on the notation – how to read Figure 1.1. The column on the right starting with "formulas" is not part of the definition, it is a series of informal annotations to help the reader by indicating the intuitive interpretation of the different cases. The weird equal sign ($::=$) is a way to say that the stuff of the left is defined by the description given on the right. On the left, the letters A, B, C, D mean that to denote a formula, we use one of these meta-variables or variations of them (A_3, B', $C_{\mathtt{obj}}$, etc.). On the right, you have a series of cases separated by vertical bars: ($| \cdots | \cdots | \cdots$). This means that an element of the syntactic class we are defining (here, formulas), is of one of these forms. It may be an atom X, Y, Z, or it may be a conjunction for example if it is of the form $A \times B$, where A and B denote any formula. When we write X, Y, Z to describe the syntactic class of atoms, we actually mean either one of these letters, or variations of them: X', Y_2, $Z_{\mathtt{foo}}$, etc.

Our previous example $(X \to Y) \times 0$ is a valid formula because it can be decomposed using the rules given in the figure: it is a product of the form $B \times C$, where B is the implication $X \to Y$, and 0 is the "false" formula – it is just the symbol 0, but we understand it informally as meaning "false".

The "atoms" are primitive formulas that reasoning cannot decompose further. If you wonder whether the sentence "If I am hungry and in a good mood, then I am hungry" is provable in the logic we are defining, you may model "I am hungry" and "I am in a good mood" as two atoms $X_{\mathtt{h}}$ and $Y_{\mathtt{gm}}$, and study the provability of the formula $(X_{\mathtt{h}} \times Y_{\mathtt{gm}}) \to X_{\mathtt{h}}$.

The symbols chosen to represent conjunction and disjunction are a bit unusual, because I reuse notations coming from the programming world. To my defense, on one hand the specialists will immediately make sense of those notations, and on the other the usual notations ($A \wedge B$ and $A \vee B$) are no easier to learn for non-specialists.

Remark 1.2.1. The usual rule about distributivity of multiplication over addition

$$A \times (B + C) \iff (A \times B) + (A \times C)$$

is true for our formulas: "A and (B or C)" is equivalent to "(A and B) or (A and C)" – both intuitively and in the formal logics we study.[1] ∗

On meta-variables In mathematics, if we express the function that doubles its input as $(x \mapsto 2 \times x)$, the symbol x as used in this expression is called a "variable" and is part of the formal mathematical object being defined, $(x \mapsto 2 \times x)$. If we then say "let's call this function f", and use this name f in a mathematical expression ($f(3)$), we mean that the name f should be (implicitly) replaced by its definition to understand the mathematical object that we are describing; the name f itself is not a formal variable of the object. This is another notion of "variable", present at the level of our discussion, the level of discourse, called the *meta*-level. Hence the name, "meta-variable". When we discuss the properties of a formula A, the symbol A is only a name for an actual formula (such as $Y \to Z$), it is a meta-variable.

In usual mathematics, the distinction between variables and meta-variables is rarely made, because we quickly add other layers of abstraction such that what were previously meta-variables become object variables. For example, it is natural to express higher-order functions that take functions as arguments, such as the object $f \mapsto f(3)$, where f now plays the role of a variable.

[1] You may be interested in the remark that, if we wrote B^A for the implication $A \to B$, the usual rules about exponentiation would hold as well. For more details on those non-coincidences, see Fiore, Di Cosmo, and Balat [2006] and Ilik [2014].

Remark 1.2.2. Early mathematicians did not think of functions as mathematical objects, only as descriptions of transformations between objects. Even "the square of x" was only an element of discourse at the meta-level. ∗

On the contrary, when defining logics and programming languages, distinguishing the object level and the meta-level can be important. For example, if the object level is a logic that is different from the logics we usually reason with (it does not accept the same reasoning principles), confusing the object and meta-level can lead to mistakes. In the theory of programming languages, some meta-level concepts can be turned into objects of discourse, but that requires a deep understanding of them that can take years of research. For example, polymorphism is the programming-language counterpart of turning formula metavariables into object variables, and it is a subtle and difficult notion. Explicit substitutions turned a meta-level notation of substitution into an object-level concept, and again it took years to find good formulations.

The prefix "meta" is a common modality meaning "one level up", by analogy with "meta-physics", often understood to describe what is "beyond physics", the truths independent from the physical world. This comes from the Greek prefix $\mu\epsilon\tau\alpha$, meaning "after" or "beside": the name "meta-physics" was given to the subject of the books of Aristotle that were placed, on the shelves of the library, next to his book on physics[2].

1.2.2. Formally defining the proofs

We have described a set of objects that we call *formulas*, whose structure is given by a grammar. Similarly, what we call *proof* is nothing more than a set of objects with a certain structure, which we can manipulate and study. Both are some sort of trees. It may be useful to consider why we could use a simple grammar to describe formulas, but had to use a more complex notation (inference rules) for proofs.

The reason for this different notation is that the structure of proofs is more complex than the structure of formulas, in the following sense. A formula is built up, recursively, of other formulas that appear inside it (we call them "subformulas"); an implication formula, for example, has left-hand and right-hand sides that are themselves formulas, and may contain subformulas. Any of those subformulas has exactly the same structure as any other: they are formulas.

On the contrary, the structure of proofs depends in important ways on what is being proved. The proof of a conjunction does not have the same formal structure as the proof of a disjunction. A proof contains sub-proofs that correspond to intermediate steps, but their structure depends on what this intermediate step is trying to establish. Simple recursive context-free grammars, as used to describe formulas, cannot capture this dependency.

Thus, to represent proofs, we use derivation trees as defined in Section 1.1 (A first introduction to inference rules). If we use logic formulas as the judgments of these derivations, we can have different rules to form derivations whose conclusion is the conjunction $A_1 \times A_2$ and derivations whose conclusion is the disjunction $A_1 + A_2$.

However, formulas are not quite enough to capture proofs. To prove the formula $A_1 \times A_2$, we could require complete proofs of the formulas A_1 and A_2:

$$\frac{A_1 \qquad A_2}{A_1 \times A_2}$$

But which premises should we require to prove the implication $A \to B$? We would like to say that "assuming A holds, we require a proof of B". This means that our notion of judgment should tell us not only what formula is to be proved, but what assumptions we have made in the process of proving it. We will use the syntax $A \vdash B$ to represent the

[2] http://en.wikipedia.org/wiki/Metaphysics#Etymology

judgment "prove B assume A". A rule for implication could be:

$$\frac{A \vdash B}{A \to B}$$

However, in the general case, we may have not only one assumption (here A) but a *set* of various assumptions. For example, to prove $A \to (B \to C)$ is to prove C under the assumptions $\{A, B\}$. We will use the meta-variable Γ to represent these sets of assumptions, that we call *contexts*, and the notation Γ, A to represent the addition of the assumption A to the set Γ – that is, the set $\Gamma \cup \{A\}$. The general rule for implication is thus, for any context Γ:

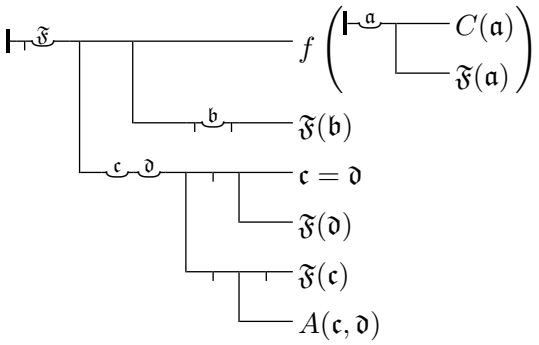
$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

It reads: "to prove that A implies B under the assumptions Γ, it suffices to prove B under the assumptions Γ plus the assumption A".

In other words, the judgments that we establish in our proof systems are not a single formula A, but a pair (Γ, A) of a context Γ that is assumed, and a formula A that is to be proved. The syntax $\Gamma \vdash A$ is just a notation for the pair (Γ, A), just as the judgment $m \preccurlyeq n$ of Section 1.1 was just a notation for the pair of natural numbers (m, n).

Remark 1.2.3. The symbol $\vdash$ is inspired by the weird notations proposed by Frege in his famous *Begriffsschrift* in 1879, probably the first attempt to represent mathematical statements and proofs as precise mathematical objects themselves, instead of just an informal mathematical text.

You will easily spot the $\vdash$ in the following example of Frege's notation – whose typesetting is attributed to Marcus Rossberg by Quirin Pamp[3]. It represents a particular rendition of the second-order Geach-Kaplan sentence[4], "Some critics (C) admire (A) only one another.", $\exists \mathfrak{F}, (\forall \mathfrak{a}, \mathfrak{F}(\mathfrak{a}) \to C(\mathfrak{a})) \times (\exists \mathfrak{b}, \mathfrak{F}(\mathfrak{b})) \times (\forall \mathfrak{c}\, \mathfrak{d}, (\mathfrak{F}(\mathfrak{c}) \times A(\mathfrak{c}, \mathfrak{d})) \to (\mathfrak{F}(\mathfrak{d}) \times \mathfrak{c} \neq \mathfrak{d}))$. This is for the reader amusement only, I will not attempt to explain or use the notation.



∗

In Figure 1.2, we describe the proofs of propositional intuitionistic logic in natural deduction style (there is another common presentation, called "sequent style" which we also describe in this document, in Chapter 4), specified as a system of inference rules.

For example, below is a complete proof of the formula $1 \times (0 + X)$ ("true and (either false or X)"), with the assumption that the atom X is true in context. That is, a proof of the judgment $X \vdash 1 \times (0 + X)$:

$$\frac{\dfrac{}{X \vdash 1} \qquad \dfrac{\dfrac{}{X \vdash X}}{X \vdash 0 + X}}{X \vdash 1 \times (0 + X)}$$

[3] http://mirrors.ctan.org/macros/latex/contrib/frege/frege.pdf
[4] https://en.wikipedia.org/wiki/Nonfirstorderizability

Figure 1.2.: Propositional intuitionistic logic $\mathsf{PIL}(\to, \times, 1, +, 0)$, in natural deduction style

$$\textsc{nd-axiom}\ \frac{}{\Gamma, A \vdash A}$$

$$\textsc{nd-and-intro}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \textsc{nd-and-elim}\ \frac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash A_i} i \in \{1, 2\}$$

$$\textsc{nd-or-intro}\ \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} i \in \{1, 2\} \qquad \textsc{nd-or-elim}\ \frac{\Gamma \vdash A + B \qquad \begin{array}{c}\Gamma, A \vdash C \\ \Gamma, B \vdash C\end{array}}{\Gamma \vdash C}$$

$$\textsc{nd-impl-intro}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \textsc{nd-impl-elim}\ \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\textsc{nd-true-intro}\ \frac{}{\Gamma \vdash 1} \qquad \text{(no elimination rule for 1)}$$

$$\text{(no introduction rule for 0)} \qquad \textsc{nd-false-elim}\ \frac{\Gamma \vdash 0}{\Gamma \vdash A}$$

The first rule of Figure 1.2, named ND-AXIOM (ND for "Natural Deduction"), tells us that if A is among the current assumptions (Γ, A) of the judgment, then A is provable. Note that Γ, A is a notation for the (non-disjoint) union of sets (or insertion of an element in a set): Γ may itself contain A, and then (Γ, A) and Γ are the same set.

$$\frac{}{\Gamma, A \vdash A}$$

Note the side-condition in the rule ND-OR-INTRO:

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} i \in \{1, 2\}$$

The formula A_i is either A_1 or A_2. The side-condition insists that i be either 1 or 2 – in the other systems of this thesis we often omit it, as it is often evident, from the way the index i is used. In the case of our example $X \vdash 1 \times (0 + X)$, we have used the rule with $i = 2$:

$$\frac{X \vdash X}{X \vdash 0 + X}$$

We proved the right-hand side of the disjunction, as the left-hand side would not have been provable.

Figure 1.2 has several kind of rules. The *axiom* rule ND-AXIOM is one-of-a-kind; it is called a *structural rule* because it does not pertain to a specific logical connective, but is a general principle for any formulas of the logic; some other logics have more such structural rules.

$$\textsc{nd-axiom}\ \frac{}{\Gamma, A \vdash A}$$

The *introduction rules* are the rules where a logical connective appears below the bar. Introduction rules allows us to prove a formula using this connective, from premises involving the sub-formulas: they *introduce* a proof of the connective. Here we have one such rule for each connective: ND-AND-INTRO for conjunction, ND-OR-INTRO for disjunction, and ND-IMPL-INTRO for implication. To prove ("introduce") an implication $A \to B$, it suffices to assume A by adding it to the context, and then prove B.

$$\text{ND-AND-INTRO}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \text{ND-OR-INTRO}\ \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}\ i \in \{1,2\} \qquad \text{ND-IMPL-INTRO}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

Finally, the *elimination rules* have a formula using a logical connective above the bar. Elimination rules describe how to use the proof of a formula starting with a given connective to prove things about its sub-formulas: they *use*, or *eliminate*, proofs of the connective. The elimination rule for the conjunction, ND-AND-ELIM, tells you that from a proof of $A \times B$ you can obtain a proof of A (when applying the rule with $i \stackrel{\text{def}}{=} 1$) or B (with $i \stackrel{\text{def}}{=} 2$). The rule ND-OR-ELIM uses a disjunction $A + B$ to prove a formula C; you have to be able to prove C assuming A, and also prove C assuming B. Finally the rule ND-IMPL-ELIM lets you use ("eliminate") the implication $A \to B$ to deduce B, provided that you can prove A.

$$\text{ND-AND-ELIM}\ \frac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash A_i}\ i \in \{1,2\} \qquad \text{ND-OR-ELIM}\ \frac{\Gamma \vdash A + B \qquad \begin{array}{c}\Gamma, A \vdash C \\ \Gamma, B \vdash C\end{array}}{\Gamma \vdash C} \qquad \text{ND-IMPL-ELIM}\ \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

The base formulas 0 and 1 are a bit special: the true formula has only one (trivial) introduction rule, but no elimination rule (it is not very useful to deduce other formulas). The false formula, on the contrary, has no introduction rule (you do not want to help the users of your logic prove false results!), only an elimination rule, which let us deduce anything from an absurdity.

You may have noticed a certain symmetry between some of those rules: the introduction rule for the disjunction ND-OR-INTRO resembles the elimination rule of the conjunction ND-AND-ELIM – it is a bit harder to see a relation between the two other rules, ND-AND-INTRO and ND-OR-ELIM.

$$\frac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash A_i} i \in \{1,2\} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} i \in \{1,2\}$$

This is no coincidence, but this particular logical system is not the best suited to exhibit and discuss this symmetry. We present a more symmetrical system in the form of a sequent calculus in Section 4.1.

Finally, let us come back to the opening remark of this section, that describing the structure of valid proofs required a richer formalism than the context-free grammar used to described valid formulas. System of inference rules permit exactly this: valid proofs are trees where not all parts of the tree have the same structure: the only rules that can be applied at a given point in the proof are those that match the current judgment $\Gamma \vdash A$. All context-free grammars could be presented as systems of inference rules, but the converse is not true. For your amusement, below is a system of inference rules for a different judgment A formula that defines the valid formulas (A is valid if and only if A formula is provable), just as the grammar of Figure 1.1 (Formulas of the propositional intuitionistic logic), only in a more verbose way.

$$\frac{A \text{ formula} \qquad B \text{ formula}}{A \times B \text{ formula}} \qquad \frac{A \text{ formula} \qquad B \text{ formula}}{A + B \text{ formula}} \qquad \frac{A \text{ formula} \qquad B \text{ formula}}{A \to B \text{ formula}}$$

$$\frac{A \in \{X, Y, Z \dots\}}{A \text{ formula}} \qquad \frac{}{1 \text{ formula}} \qquad \frac{}{0 \text{ formula}}$$

1.2.3. Rootward and leafward reading of inference rules

There are two natural ways to look at any inference rule of a logic, and people familiar with that notation often jump from one interpretation to the other, at the cost of confusing the less confident reader. A rule

$$\dfrac{\mathcal{J}_1 \qquad \mathcal{J}_2 \qquad \dots}{\mathcal{J}}\ \text{TATA}$$

can be read:

- Rootward (downward): if you have succeeded in proving the sequents $\mathcal{J}_1, \mathcal{J}_2, \dots$, then you can now prove $\mathcal{J}$. This is useful when you know what you have, and wonder where you can go with it.
- Leafward (upward): if your goal is to prove $\mathcal{J}$, then the rule tells you that one way you may try is to prove $\mathcal{J}_1, \mathcal{J}_2, \dots$. This is useful when you know what you want, but not how to go there.

Remark 1.2.4. The use of the words "rootward" and "leafward" is a personal choice; they have the advantage (compared to for example "downward" and "upward") of being independent of the particular spatial convention that we adopted to represent trees with their leaves up and their root down – this convention is absolutely omnipresent in the field, but some of our neighbors, for example in "tableaux proof systems", take the opposite one. I prefer to avoid using "up" and "down". People sometimes use "bottom-up" for leafward and "top-down" for rootward (I caught Stéphane Graham-Lengrand doing this in a talk); this is absolutely wrong, because for everyone else "bottom-up" means "from the small atomic parts to the composed result" (in our setting, that would be rootward), and "top-down" means "from the composed results to the atomic parts" (in our setting, leafward). ∗

There is no "right choice" between the rootward and leafward reading of inference rules. Different users of the logic have different biases. The designer of (goal-directed) proof search algorithms, for example, almost exclusively favor the leafward reading. Because we tend to think of logics in terms of "how can we prove its formulas?", it is a rather common view. Have a look at the order in which people write those rules on a blackboard: it may reveal their natural reading order. But there are also proof-search algorithms (such as the so-called "inverse" method) that work by saturation from the axioms, and thus rather use the rootward reading.

In the natural deduction system we have presented, introduction rules have a more natural leafward interpretation, and elimination rules are more naturally explained in a rootward way. When you see

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad\qquad \dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

a natural explanation of the first rule is "to prove an implication, it suffices to ..." (leafward), while the second is more "if you have proved an implication, then you can ..." (rootward).

A fairly confusing aspect of this ambiguity is that, even though I suspect most people naturally use the leafward reading most of the time, the naming of the rules consistently comes from their rootward interpretation. For example, we have defined in our logic contexts Γ as *sets* of hypotheses, in particular the sets $\{A, B, A\}$ and $\{B, A\}$ are the same, they contain the same elements A and B. Some other logics have contexts with a more restricted structure (multisets for example, or even ordered lists), and are careful about the number of time each hypothesis is used. They may have the following rule:

$$\dfrac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

I find natural to read this rule leafward: it is ok to split any hypothesis of the context in two, to be used in two different ways by the leafward derivation. But it is named the *contraction* rule, from its rootward interpretation: if you have a proof that uses two distinct copies of a hypothesis, you can *contract* it in a proof using only one copy. Similarly, a rule that is commonly discussed is

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

My personal intuition for this rule is a form of "hiding": we are trying to prove B in the context Γ, A, but we claim that it is ok to forget about the hypothesis A, we decide to be brave and look for a proof that does not use it at all. (That is also the natural interpretation in terms of programming language design, through the proof-as-programs correspondence.) Yet this rule is named *weakening* from its rootward interpretation: if we have a complete proof of B using the assumptions Γ, then we may *weaken* it by adding the unnecessary hypothesis A – which makes it applicable in less situations, hence the idea of weakness.

This confusion is to be taken as a strength: by tilting your head and inverting your reading order, you get to see new things about the same rule, maybe connections to other concepts and new ideas.

1.2.4. Structural presentations vs. Hilbert-style proofs

In my first course of mathematical logic, I was presented a very different formal structure for proofs. We would first pick a closed set of reasoning axioms (formulas that are assumed to be always true in classical logic), then define a proof of a statement S as a sequences of propositions $P(i)$ indexed by natural numbers $[0; n]$, such as each step is either one of the axioms, or the modus ponens of two previous steps (that is, deducing B from A and $A \Rightarrow B$), and such that $P(n) = S$.

For example, a proof with conclusion

$$\frac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$$

would be represented as the linear sequence of steps:

- a sequence of formulas representing Π_1, ending with $\Gamma \Rightarrow A_1$, at step index (k_1),
- followed by a sequence of formulas representing Π_2, ending with $\Gamma \Rightarrow A_2$, at step index (k_2),
- followed by the classical tautology $\forall X, Y, Z, (X \Rightarrow Y) \Rightarrow ((X \Rightarrow Z) \Rightarrow (X \Rightarrow (Y \wedge Z)))$, at index $(k_2 + 1)$,
- followed by the formula $(\Gamma \Rightarrow A_2) \Rightarrow (\Gamma \Rightarrow (A_1 \wedge A_2))$, as the modus ponens of steps $(k_2 + 1)$ (with choice of parameters $X \stackrel{\text{def}}{=} \Gamma$, $Y \stackrel{\text{def}}{=} A_1$, $Z \stackrel{\text{def}}{=} A_2$) and (k_1), at index $(k_2 + 2)$,
- ending with the formula $\Gamma \Rightarrow (A_1 \wedge A_2)$ as the modus ponens of steps $(k_2 + 2)$ and (k_2).

This representation, called Hilbert-style[5], is simple to define, but it also feels very "low-level". In particular, the representation we propose, as a tree of inference rules, exposes more information on the "structure" of the proof. Our connective $(\times)$ is given meaning by the inference rules to form and use proofs of $A \times B$. In a Hilbert-style system, it is the set

[5] https://en.wikipedia.org/wiki/Hilbert_system

of axioms (a list with no particular structure) that gives meaning to all the connectives of the logic.

Our more structural definition of proofs allows us to prove important results by inspection of the tree structure, by induction over it; in particular, we can prove this way that our logic is consistent: it admits no proof of *false* in the empty context (Theorem 3.3.9 (Consistency of PIL$(\to, \times, 1, +, 0)$)). With Hilbert-style proof, the structure of proofs contains almost no information, and any consistency result must be obtained by studying the set of all tautologies.

Structural presentations also give stronger intuitions of what the "computational" meaning of each logic may be. That said, Hilbert-style proofs can be extended with more structural proof-formers (combinatory logics), and then have corresponding notion of programs (combinator languages). Combinatory logics[6] have been a fertile ground for research with many applications to computer science, for example for type inference and term rewriting.

1.3. On the meaning of logical connectives: testing a logic

We have been a bit dishonest in the previous section: we introduced some formal symbols and some formal rules, and at the same time we gave them the names of existing concepts ("and", "or", "implies", "true", "false"), speaking informally of those formal objects as if they corresponded to those notions that we all already understand intuitively. But do the connectives that we defined correspond to those informal notions? Does the notion of proof that we define correspond to what is usually understood as a proof?

The answer is "no, and that is sort of the point". We are not trying to make a philosophical stance on what Reasoning and Truth are about, or even about what would be the Right way to reason and prove. We are defining one precise, simple, formal notion of reasoning and proofs, and studying how far we can go with it, which interesting things it allows us to do – a parallel (or nested) world where the rules are purposely different. The meaning of our connective $(\times)$ is not defined as an approximation of what we understand of the informal notion of "and", but precisely by "whatever you can do using the rules we have given", in the present case ND-AND-INTRO and ND-AND-ELIM.

We could make a parallel with the non-euclidean geometries that emerged during the nineteenth century: there are many possible geometries that can be defined formally, and they are not (only) judged on the resemblance to our physical space, but also on their interesting properties and applications – they turn out to be useful to think about other things than the physical world. We have scratched the beginning of a toolbox to define new logics, which can then be judged on their properties and applications.

(The specialist have recognized that this logic is intuitionistic rather than classical, and thus does not prove certain things that most people would consider intuitively valid. I explain this difference and bridge the gap in Section 4.3 (Classical logic).)

Of course, the ability to define arbitrary proof systems and play with them does not mean that *anything goes*: some definitions are better (more interesting, have more applications, etc.) than others. What defines a *good* proof system is extremely subtle, but there are some *tests* that can tell us if we did an obvious mistake. For computer programs, a test is a particular input to feed the running program, along with a description of the expected output. For mathematical objects the notion of *test* is more delicate; a common form of test is a particular mathematical property that most objects of this class verify – that *good* objects of this class ought to verify: lemmas as tests. Following this idea, we now discuss two kinds of properties of proof systems, some that are *local* in nature – they test each connective separately – and some that are *global* – they test the logic as a whole.

Remark 1.3.1. We could also try to prove *soundness* and *completeness* of our rules, as we did for the simpler judgments $m \preccurlyeq n$ in Section 1.1 (A first introduction to inference rules). There is no hope to prove soundness and completeness of our formal definition

[6] http://plato.stanford.edu/entries/logic-Tcombinatory/

of proofs against the *informal* idea that most people (including mathematicians) have of what a proof is. To rigorously formulate soundness and completeness conjectures and prove them, we need another formal description of logic to compare against.

This can be done (for example, comparing our rules for intuitionistic logic with Kripke semantics), but it requires to develop these alternate formal presentations of logics, which we will not do, by lack of space. In any case, our proofs of soundness and completeness would require most of the results that we describe as *tests* below.

When you create a new logic, or a new programming language, finding the right alternative formal model to compare to is an important first step towards validating your new idea, and it can be difficult. ∗

1.3.1. Global tests: weakening and substitution

A first example of test is the following: is the logic monotonic? That is, does adding new hypotheses to the context Γ always let us prove *more* things? Or is it the case that some formula A is provable in a given context Γ, but not in a larger context Γ, A'? This is a *global* test, because we test all the rules of the logic together: any of them could break this monotonicity property.

The logic we have defined is monotonic, in other words the following property holds.

Lemma 1.3.1 (Weakening for $\mathsf{PIL}(\to, \times, 1, +, 0)$)**.**
If $\Gamma \vdash A$ admits a partial proof Π, then $\Gamma, A' \vdash A$ admits a partial proof Π', for any additional hypothesis A'. The open leaves of the form $\Delta \vdash B$ in Π becomes leaves of the form $\Delta, A' \vdash B$ in Π', and there are no other open leaves in Π'.

The proof which follows is a constructive procedure that takes as input the partial derivation $\Pi :: \Gamma \vdash A$, and returns a new derivation $\Pi' :: \Gamma, A' \vdash A$. Because it is constructive, we can use this proof as if it was a reasoning step: this almost as if it was a rule of the logic (this is the concept of admissibility, explained in definition 1.1.2). We write:

$$\dfrac{\Pi :: \Gamma \vdash A}{\Gamma, A' \vdash A}\ \text{WK}$$

to denote the (uniquely defined) proof obtained by applying this procedure to Π. The dotted horizontal line is here to remind us that this is not a built-in inference rule. This particular admissible rule is called *weakening* because we obtain a "weaker" judgment with more hypotheses.

Proof. The first case of the proof is simple: if Γ already contains A', then the proof is immediate: Π is already a proof of $\Gamma, A' \vdash A$ as (Γ, A') and Γ are the same set of hypotheses.

Otherwise, the proof proceeds by induction on the structure of Π. For example, consider the weakening of a proof concluded by the conjunction introduction rule,

$$\dfrac{\dfrac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}\ \text{ND-AND-INTRO}}{\Gamma, A' \vdash A_1 \times A_2}\ \text{WK}$$

Doing an induction on the sub-proofs of the judgments $\Pi_i :: \Gamma \vdash A_i$ gives us, as induction hypotheses, two proofs $\Pi'_i :: \Gamma, A' \vdash A_i$ that we write with the same "admissible rule" notation:

$$\dfrac{\Pi_1 :: \Gamma \vdash A_1}{\Pi'_1 :: \Gamma, A' \vdash A_1}\ \text{WK} \qquad\qquad \dfrac{\Pi_2 :: \Gamma \vdash A_2}{\Pi'_2 :: \Gamma, A' \vdash A_2}\ \text{WK}$$

From these two induction hypotheses, we can form a valid proof of the desired goal $\Gamma, A' \vdash A_1 \times A_2$ as follows:

$$\dfrac{\Pi'_1 :: \Gamma, A' \vdash A_1 \qquad \Pi'_2 :: \Gamma, A' \vdash A_2}{\Gamma, A' \vdash A_1 \times A_2}\ \text{ND-AND-INTRO}$$

In other words (using a more compact notation), we can define the weakening of the conjunction introduction rule as follows:

$$\dfrac{\dfrac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}\,\textsc{nd-and-intro}}{\Gamma, A' \vdash A_1 \times A_2}\,\textsc{wk}$$

$$\stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash A_1}{\Gamma, A' \vdash A_1}\,\textsc{wk} \qquad \dfrac{\Gamma \vdash A_2}{\Gamma, A' \vdash A_2}\,\textsc{wk}}{\Gamma, A' \vdash A_1 \times A_2}\,\textsc{nd-and-intro}$$

The rest of this proof uses a similar notation for all other rules.

$$\dfrac{\dfrac{}{\Gamma, A \vdash A}\,\textsc{nd-axiom}}{\Gamma, A, A' \vdash A}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{}{\Gamma, A, A' \vdash A}\,\textsc{nd-axiom}$$

$$\dfrac{\dfrac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash A_i}\,\textsc{nd-and-elim}}{\Gamma, A' \vdash A_i}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash A_1 \times A_2}{\Gamma, A' \vdash A_1 \times A_2}\,\textsc{wk}}{\Gamma, A' \vdash A_i}\,\textsc{nd-and-elim}$$

$$\dfrac{\dfrac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}\,\textsc{nd-or-intro}}{\Gamma, A' \vdash A_1 + A_2}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash A_i}{\Gamma, A' \vdash A_i}\,\textsc{wk}}{\Gamma, A' \vdash A_1 + A_2}\,\textsc{nd-or-intro}$$

$$\dfrac{\dfrac{\Gamma \vdash A_1 + A_2 \qquad \Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma \vdash C}\,\textsc{nd-or-elim}}{\Gamma, A' \vdash C}\,\textsc{wk}$$

$$\stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash A_1 + A_2}{\Gamma, A' \vdash A_1 + A_2}\,\textsc{wk} \qquad \dfrac{\Gamma, A_1 \vdash C}{\Gamma, A_1, A' \vdash C}\,\textsc{wk} \qquad \dfrac{\Gamma, A_2 \vdash C}{\Gamma, A_2, A' \vdash C}\,\textsc{wk}}{\Gamma, A' \vdash C}\,\textsc{nd-or-elim}$$

$$\dfrac{\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,\textsc{nd-impl-intro}}{\Gamma, A' \vdash A \to B}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash B}{\Gamma, A', A \vdash B}\,\textsc{wk}}{\Gamma, A' \vdash A \to B}\,\textsc{nd-impl-intro}$$

$$\dfrac{\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\textsc{nd-impl-elim}}{\Gamma, A' \vdash B}\,\textsc{wk}$$

$$\stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash A \to B}{\Gamma, A' \vdash A \to B}\,\textsc{wk} \qquad \dfrac{\Gamma \vdash A}{\Gamma, A' \vdash A}\,\textsc{wk}}{\Gamma, A' \vdash B}\,\textsc{nd-impl-elim}$$

$$\dfrac{\dfrac{}{\Gamma \vdash 1}\,\textsc{nd-true-intro}}{\Gamma, A' \vdash 1}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{}{\Gamma, A' \vdash 1}\,\textsc{nd-true-intro}$$

$$\dfrac{\dfrac{\Gamma \vdash 0}{\Gamma \vdash A}\,\textsc{nd-false-elim}}{\Gamma, A' \vdash A}\,\textsc{wk} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma \vdash 0}{\Gamma, A' \vdash 0}\,\textsc{wk}}{\Gamma, A' \vdash A}\,\textsc{nd-false-elim}$$

Note that, in the rules that add hypotheses to the context, ND-IMPL-INTRO for example, it may be the case that A' belongs to the context of the premise – when introducing an

implication of the form $A' \to B$. In this case the definition of the admissible rule applies, but the inductive weakening of the premise is just the premise itself. □

Notice that all inference rules of our logic have been considered in this proof – this is a global test. For an example of rule that would break this monotonicity property, consider the following restricted axiom rule:

$$\text{ND-AXIOM-LINEAR}\ \frac{}{A \vdash A}$$

This rule let us prove $A \vdash A$, but not $A, B \vdash A$ for $B \neq A$: it breaks monotonicity. It is also a central rule in *linear logic*, a beautiful and useful logic – failing some tests can be a good thing.

Remark 1.3.2. A less obvious example is a rule, in a different logic with a different notion of implication, that tells you, for each implication, what was the environment at the place where the implication was proved (in terms of programming, that tells us about the variables captured by the function closure):

$$\text{ND-CLOSURE-INTRO}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash [\Gamma](A \to B)}$$

To make this logic monotonic, you need to ensure that whenever $[\Gamma](A \to B)$ is provable, then $[\Gamma, A'](A \to B)$ is also provable. I designed such a system once, and at first I forgot to describe this lifting of implication – it was kept implicit in the representation of proofs. An anonymous reviewer promptly reminded me that monotonicity is an important property that should be explicitly preserved, and fixing this mistake led to a more precise, better description of the system. ∗

The second global test is even more important than weakening. Suppose we have a proof Π of B assuming A, and independently we came up with a proof of A. Can we transform Π into a proof of B that does *not* assume A anymore (as it is proved)?

Theorem 1.3.2 (Substitution for $\mathsf{PIL}(\to, \times, 1, +, 0)$)**.**
The following rule is admissible

$$\frac{\Pi_B :: \Gamma, A \vdash B \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

In particular, some valid proofs of $\Gamma \vdash B$ *are all the proofs that have the same structure as the complete proof* Π_B*, except that some axiom rules on* A *have been replaced by (weakenings of)* Π_A*.*

We speak of the *proofs* resulting of substitution, instead of *the proof*, because there may be several possible admissible proofs. For example, consider the proof $\Pi :: A \vdash A \to A$ defined as

$$\frac{\dfrac{}{A \vdash A}\ \text{ND-AXIOM}}{A \vdash A \to A}$$

If we provide a proof $\Pi_A :: \emptyset \vdash A$, we may substitute it inside Π to obtain a proof of $\emptyset \vdash A \to A$, but there are two possible such proofs:

$$\frac{\dfrac{}{A \vdash A}\ \text{ND-AXIOM}}{\emptyset \vdash A \to A} \qquad \frac{\dfrac{\Pi_A :: \emptyset \vdash A}{A \vdash A}\ \text{WK}}{\emptyset \vdash A \to A}$$

In a sense, the two proofs result from different views of where the hypothesis used in the axiom rules of the original proof Π "comes from". If it comes from the hypothesis A added by implication introduction, it should not be replaced by Π_A (first proof). If it comes from the hypothesis A that was present in the context, it is the one assumption

that we remove and substitute with Π_A (second proof). We cannot distinguish these two cases in the original proof, as the two hypotheses A are merged, in the context, as the singleton set $\{A\}$.

Remark 1.3.3. It is possible to make this idea more precise by using not a *set* of hypotheses, but a *multi-set* of hypotheses. We would have several copies of A in the context of the axiom rule, one coming from the root context and one from the implication introduction. Some authors prefer this approach (which is closer to the way we name variables when programming; and thus makes it easier to have a correspondence between proof derivations and programs), and arguably it is closer to some historic presentations of natural deduction, where there are no contexts, and introducing an hypothesis A is done by "discarding" a *subset* of the open leaves of type A.

I still prefer the set-of-hypotheses approach, which I find a more faithful rendering of intuitionistic (or classical) logic: there is no reason we should care about having different hypotheses of the same formula, either it is provable or it is not – this is the approach taken by search procedures, for example. ∗

Our statement of the theorem says that the notation

$$\dfrac{\Pi :: A \vdash A \to A \qquad \Pi_A :: \emptyset \vdash A}{\emptyset \vdash A \to A}\ \text{SUBST}$$

may denote either of those proofs, and no other: those two are all the proofs obtained by replacing some of the axiom rules of Π by (weakenings of) the proof Π_A. In the general case there may exist other valid proofs of the desired judgment (consider the case $A \stackrel{\text{def}}{=} 1$ for example), but by this admissible notation we mean one of the proofs with the shape we described. Restricting the set of proofs meant by this notation is important in later sections where we prove and use properties of substitution.

Proof (Theorem 1.3.2 (Substitution for $\mathsf{PIL}(\to, \times, 1, +, 0)$))**.** As for the case of weakening, we define the admissible rule

$$\dfrac{\Pi_B :: \Gamma, A \vdash B \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

by induction on the structure of Π_B.

The axiom case is the most delicate. We have either

$$\dfrac{}{\Pi_B :: \Gamma, B, A \vdash B}\ \text{ND-AXIOM}$$

with $B \neq A$, and we can return the proof

$$\dfrac{}{\Pi_B :: \Gamma, B \vdash B}\ \text{ND-AXIOM}$$

Or we are in the case where $B = A$, namely

$$\dfrac{}{\Pi_B :: \Gamma, A \vdash A}\ \text{ND-AXIOM}$$

and we may simply return the proof $(\Pi_A :: \Gamma \vdash A)$, but if furthermore we have $A \in \Gamma$ (that is the set (Γ, A) is in fact equal to the set Γ), we also have the choice of returning the proof

$$\dfrac{}{\Pi_B :: \Gamma \vdash A}\ \text{ND-AXIOM}$$

It is the choice between those two latter proofs (Π_A or an axiom rule), in the case where $A \in \Gamma$, that makes this admissible rule non-deterministic: there are two possible proofs that we could return. They both respect the structure described in the lemma: they have the structure of the initial proof Π_B where *some* (but maybe not *all*) axiom rules on A have been replaced by (weakenings of) Π_A.

There is no such choice in the other cases, which simply mirror the structure of the proof Π_B; they are directly handled by a case analysis on the root inference rule.

$$\dfrac{\dfrac{\Gamma, A \vdash B_1 \qquad \Gamma, A \vdash B_2}{\Gamma, A \vdash B_1 \times B_2} \qquad \Gamma \vdash A}{\Gamma \vdash B_1 \times B_2}\ \text{SUBST}$$

$$\stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash B_1 \qquad \Gamma \vdash A}{\Gamma \vdash B_1}\ \text{SUBST} \qquad \dfrac{\Gamma, A \vdash B_2 \qquad \Gamma \vdash A}{\Gamma \vdash B_2}\ \text{SUBST}}{\Gamma \vdash B_1 \times B_2}$$

Note that using substitution as an admissible rule on the premises $\Gamma, A \vdash B_i$ is an induction hypothesis that may return one of several possible substitutions proof, and that the proof structure described for $\Gamma \vdash B_1 \times B_2$ thus denotes many possible proofs, all sharing the structure of the initial proof.

$$\dfrac{\dfrac{\Gamma, A \vdash B_1 \times B_2}{\Gamma, A \vdash B_i} \qquad \Gamma \vdash A}{\Gamma \vdash B_i}\ \text{SUBST} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash B_1 \times B_2 \qquad \Gamma \vdash A}{\Gamma \vdash B_1 \times B_2}\ \text{SUBST}}{\Gamma \vdash B_i}$$

$$\dfrac{\dfrac{\Gamma, A \vdash B_i}{\Gamma, A \vdash B_1 + B_2} \qquad \Gamma \vdash A}{\Gamma \vdash B_1 + B_2}\ \text{SUBST} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash B_i \qquad \Gamma \vdash A}{\Gamma \vdash B_i}\ \text{SUBST}}{\Gamma \vdash B_1 + B_2}$$

$$\dfrac{\dfrac{\Gamma, A \vdash C_1 + C_2 \qquad \Gamma, A, C_1 \vdash B \qquad \Gamma, A, C_2 \vdash B}{\Gamma, A \vdash B} \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST} \qquad \stackrel{\text{def}}{=}$$

$$\dfrac{\dfrac{\Gamma, A \vdash C_1 + C_2 \qquad \Gamma \vdash A}{\Gamma \vdash C_1 + C_2}\ \text{S} \qquad \dfrac{\dfrac{\Gamma, A, C_1 \vdash B \qquad \dfrac{\Gamma \vdash A}{\Gamma, C_1 \vdash A}\ \text{WK}}{\Gamma, C_1 \vdash B}\ \text{S} \qquad \dfrac{\dfrac{\Gamma, A, C_2 \vdash B \qquad \dfrac{\Gamma \vdash A}{\Gamma, C_2 \vdash A}\ \text{WK}}{\Gamma, C_2 \vdash B}\ \text{S}}{\Gamma \vdash B}}$$

$$\dfrac{\dfrac{\Gamma, A, B_1 \vdash B_2}{\Gamma, A \vdash B_1 \to B_2} \qquad \Gamma \vdash A}{\Gamma \vdash B_1 \to B_2}\ \text{SUBST} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A, B_1 \vdash B_2 \qquad \dfrac{\Gamma \vdash A}{\Gamma, B_1 \vdash A}\ \text{WK}}{\Gamma, B_1 \vdash B_2}\ \text{SUBST}}{\Gamma \vdash B_1 \to B_2}$$

$$\dfrac{\dfrac{\Gamma, A \vdash B_1 \to B_2 \qquad \Gamma, A \vdash B_1}{\Gamma, A \vdash B_2} \qquad \Gamma \vdash A}{\Gamma \vdash B_2}\ \text{SUBST}$$

$$\stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash B_1 \to B_2 \qquad \Gamma \vdash A}{\Gamma \vdash B_1 \to B_2}\ \text{SUBST} \qquad \dfrac{\Gamma, A \vdash B_1 \qquad \Gamma \vdash A}{\Gamma \vdash B_1}\ \text{SUBST}}{\Gamma, A \vdash B_2}$$

$$\dfrac{\dfrac{}{\Gamma, A \vdash 1} \qquad \Gamma \vdash A}{\Gamma \vdash 1}\ \text{SUBST} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{}{\Gamma \vdash 1}$$

$$\dfrac{\dfrac{\Gamma, A \vdash 0}{\Gamma, A \vdash B} \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST} \qquad \stackrel{\text{def}}{=} \qquad \dfrac{\dfrac{\Gamma, A \vdash 0 \qquad \Gamma \vdash A}{\Gamma \vdash 0}\ \text{SUBST}}{\Gamma, A \vdash B}$$

One may notice that, in these proofs, whenever the induction hypothesis (the ability to perform substitution on a sub-derivation) is called in the leaves of a rule that adds a new hypothesis in context, a weakening is applied to the substituted $\Gamma \vdash A$ proof. This will be familiar to people that have worked with De Bruijn indices. $\square$

1.3.2. Derivability and Admissibility

A priori, one could distinguish many different notions of "A is more general, stronger than B" in our system, among which:

- Implication: A implies B (under a context Γ) if the judgment $\Gamma \vdash A \to B$ is provable.
- Provability: B is provable from A (under a context Γ) if $\Gamma, A \vdash B$ is provable.
- Derivability: B is derivable from A (under a context Γ) if the judgment $\Gamma \vdash B$ is derivable from a judgment (in a weaker context) of the form $\Gamma, \Delta \vdash A$.
- Admissibility: B is admissible from A (under a context Γ) if the judgment $\Gamma \vdash B$ is admissible from a judgment (in a weaker context) of the form $\Gamma, \Delta \vdash A$.

Luckily, those notions are not unrelated – a jungle of distinct concepts would make our logic rather difficult to work with. Implication and provability are equivalent (inter-derivable). They are also equivalent to derivability of formulas. In the general case of judgments, rather than simple formulas, derivability implies admissibility. These relations are established using weakening and substitution: this is one of the reasons why those global properties are important.

The equivalence between "B is provable from A" and "A implies B" is by design: it is a direct consequence of the introduction and elimination rules for implication, but note that we need to use weakening in one direction:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad\qquad \frac{\dfrac{\Gamma \vdash A \to B}{\Gamma, A \vdash A \to B}\,\textsc{wk} \qquad \dfrac{}{\Gamma, A \vdash A}}{\Gamma, A \vdash B}$$

The fact that a connective of our logic, implication, completely captures provability is an important property of the logic. It avoids having to make distinctions between the provability results that can be described as formulas, and those that can only be discussed at the meta-level.

The fact that provability implies derivability is precisely the substitution principle: if $\Gamma, A \vdash B$ has a complete proof Π, we can compute the substitution with the open leaf $\Gamma \vdash A$,

$$\frac{\Pi :: \Gamma, A \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\textsc{subst}$$

and this gives a partial proof of $\Gamma \vdash B$ whose open leaves are all $\Gamma \vdash A$.

Conversely, if we have a proof of $\Gamma \vdash B$ whose open leaves are each of the form $\Gamma, \Delta \vdash A$ for some Δ, then weakening the root judgment into $\Gamma, A \vdash B$ gives a proof whose open leaves are of the form $\Gamma, A, \Delta \vdash A$, and can thus be closed by an axiom rule. In other words, if $\Gamma \vdash B$ is derivable from judgments of the form $\Gamma, \Delta \vdash A$ for some Δ, then B is provable from A under Γ.

The fact that derivability implies admissibility is trivial: a partial proof Π of $\mathcal{J}$ from the $\mathcal{J}_1, \ldots, \mathcal{J}_n$ gives a direct way to obtain a complete proof of $\mathcal{J}$ from complete proofs of the $\mathcal{J}_1, \ldots, \mathcal{J}_n$ – just plug these complete proofs at the leaves of Π.

An important difference between derivability and admissibility is that derivability is stable with respect to the addition of new inference rules to the logic, while admissibility is not in general: a partial proof of $\mathcal{J}$ using $\mathcal{J}_1$ (witnessing derivability) will remain a valid partial proof, but a case-analysis on all possible complete proofs of $\mathcal{J}_1$ may start failing if new rules are added, which the case-analysis cannot handle. This (meta-level) reasoning proves that admissibility does not imply derivability.

Contravariance and equiprovability If B is admissible from A then, by weakening, $\Gamma \vdash B$ is admissible from $\Gamma \vdash A$ for any context Γ, that is, B can always be replaced by the stronger A in succedent position. But then it is also the case that $\Gamma, A \vdash C$ is admissible from $\Gamma, B \vdash C$, that is, A can always be replaced by the weaker B in hypothesis position. Indeed, we have:

$$\dfrac{\dfrac{\Gamma, B \vdash C}{\Gamma, A, B \vdash C}\,\text{WK} \qquad \dfrac{\Gamma, A \vdash A}{\Gamma, A \vdash B}}{\Gamma, A \vdash C}\,\text{SUBST}$$

In particular, if two formulas are equiprovable, ($A \vdash B$ and $B \vdash A$, or equivalently $\vdash A \to B$ and $\vdash B \to A$), one can always be replaced by the other in any part of a judgment, and the new judgment is provable if and only if the old judgment was provable.

The fact that "succedent position" and "hypothesis position" play symmetric roles will be extended to a more general notion of (sub)-formula occurrences in Section 6.2.4 (Positive and negative positions in a formula).

1.3.3. Local tests: reduction and expansion

The local tests below apply to each logical connective separately. They guarantee that the handling of each connective in the logic is "harmonious" in some sense. There are two natural local tests:

- *reduction*: If we apply the elimination rule of some connective on a proof that is the direct result of an introduction rule, can we reduce this proof to something simpler that does not use the introduction rule?
- *expansion*: By applying the elimination rule of some connective to a proof, can we extract enough information to be able to re-apply the corresponding introduction rule(s)?

Reducing the conjunction

Suppose we have two proofs $\Pi_A :: \Gamma \vdash A$ and $\Pi_B :: \Gamma \vdash B$. We can introduce the product $A \times B$ using ND-AND-INTRO, and immediately destruct it with ND-AND-ELIM, for example in the case $i \stackrel{\text{def}}{=} 1$:

$$\dfrac{\dfrac{\Pi_A :: \Gamma \vdash A \qquad \Pi_B :: \Gamma \vdash B}{\Gamma \vdash A \times B}}{\Gamma \vdash A}$$

You see that using the elimination rule for the product gives us something we already knew: we already have a simpler proof of the conclusion $\Gamma \vdash A$, namely Π_A. The application of this elimination rule to the result of the introduction rule could be *reduced* to the simpler proof Π_A.

The point is not that this particular use of the elimination rule (with $i \stackrel{\text{def}}{=} 1$) has this property, but that *all* the possible uses of elimination rules for conjunctions share it. Indeed, the only other form of elimination is to use ND-AND-ELIM with $i \stackrel{\text{def}}{=} 2$, and then we would get a sophisticated proof of $\Gamma \vdash B$ that can be reduced to Π_B.

This test tells us that the elimination rule are in a sense "reasonable" with respect to the introduction rule. They do not allow us to deduce new stuff that was not already there at introduction time – it would be highly suspicious. For example, a way to break this property would be to add the following rule to our logic:

$$\dfrac{\dfrac{\Gamma \vdash A \times B}{\Gamma \vdash C}}{}\,\text{SUSPICIOUS-AND-ELIM-RULE}$$

This rule is obviously wrong, as it let us prove the false proposition 0. But one more systematic way to realize quickly that it is wrong is to check that it breaks the reduction principle for the conjunction connective: when this rule is applied rootward from a conjunction introduction, the derivation cannot be reduced to any simpler proof not using the introduction.

Expanding the conjunction

By looking at what happens when we eliminate the result of an introduction rule, we have shown that the elimination rule for the conjunction does not allow to deduce more than what the introduction rule requires: eliminations are not "stronger" than introductions. Expansion is the other way around: given the results of all possible eliminations of a proof of a formula, can we re-introduce the eliminated connective? This tests whether the elimination rule let us deduce "enough", or whether the introduction rule requires "too much".

Assume we have a proof $\Pi :: \Gamma \vdash A \times B$. We can eliminate its conclusion in two different ways:

$$\frac{\Pi :: \Gamma \vdash A \times B}{\Gamma \vdash A} \qquad \frac{\Pi :: \Gamma \vdash A \times B}{\Gamma \vdash B}$$

It is not a coincidence that the conclusions of these eliminations are exactly the requirement of the introduction rule. We say that Π can be *expanded* into the more complex proof of the same judgment

$$\frac{\dfrac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \dfrac{\Gamma \vdash A \times B}{\Gamma \vdash B}}{\Gamma \vdash A \times B}$$

A rule that would violate that test would be the following introduction rule:

$$\frac{\Gamma \vdash A \qquad \emptyset \vdash B}{\Gamma \vdash A \times B}\ \textsc{suspicious-and-intro-rule}$$

which requires its right premise to be in the *empty* context, disallowing the use of any assumption in the current context Γ.

This suspicious introduction rule requires "too much" – with respect to the available elimination rule. In particular, if this rule replaced the usual ND-AND-INTRO, then we would not be able to prove $\Gamma \vdash (A \times B) \to (B \times A)$ for non-empty contexts Γ, which violates our intuition of what our logic should allow.

1.3.4. A notation for reductions and expansions

Notation 1.3.1 $\Pi \triangleright_R \Pi'$**.**
We write $\Pi \triangleright_R \Pi'$ to say that Π' can be considered a reduction of a proof Π which eliminates an introduction – both proofs prove the same judgment. Similarly we write $\Pi \triangleright_E \Pi'$ to say that Π can be *expanded* into Π' by introducing the result(s) of elimination.

We can similarly check that other connectives are reducible (the elimination of an introduction can be simplified) and expansible (the result(s) of eliminations can be re-introduced). In some places we need to use the admissible operations of substitution and weakening introduced in Section 1.3.1.

1.3.5. Reducing, expanding implications

$$\dfrac{\dfrac{\Pi_B :: \Gamma, A \vdash B}{\Gamma \vdash A \to B} \quad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B} \quad \triangleright_R \quad \dfrac{\Pi_B :: \Gamma, A \vdash B \quad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

$$\Pi :: \Gamma \vdash A \to B \quad \triangleright_E \quad \dfrac{\dfrac{\dfrac{\Pi :: \Gamma \vdash A \to B}{\Gamma, A \vdash A \to B}\ \text{WK} \quad \dfrac{}{\Gamma, A \vdash A}}{\Gamma, A \vdash B}}{\Gamma \vdash A \to B}$$

The substitution rule on the right-hand side of the reduction relation may mean many different proofs. We consider that the left-hand side is in relation to any of those proofs.

1.3.6. Reducing, expanding disjunctions

$$\dfrac{\dfrac{\Pi :: \Gamma \vdash A_i}{\Pi :: \Gamma \vdash A_1 + A_2} \quad \Pi_1 :: \Gamma, A_1 \vdash C \quad \Pi_2 :: \Gamma, A_2 \vdash C}{\Gamma \vdash C}$$

$$\triangleright_R \quad \dfrac{\Pi :: \Gamma \vdash A_i \quad \Pi_i :: \Gamma, A_i \vdash C}{\Gamma \vdash C}\ \text{SUBST}$$

$$\Pi :: \Gamma \vdash A_1 + A_2 \quad \triangleright_E \quad \dfrac{\Pi :: \Gamma \vdash A_1 + A_2 \quad \dfrac{\Gamma, A_1 \vdash A_1}{\Gamma, A_1 \vdash A_1 + A_2} \quad \dfrac{\Gamma, A_2 \vdash A_2}{\Gamma, A_2 \vdash A_1 + A_2}}{\Gamma \vdash A_1 + A_2}$$

1.3.7. No reductions/expansions for true and false

The true and false formulas have no reduction or expansion to check, because they lack either an elimination or an introduction rule.

1.3.8. Reducing, expanding sub-proofs

The relation $\Pi \triangleright_R \Pi'$ holds if Π ends with an elimination of an introduction rule, and if Π' is its simplification removing these two reasoning steps.

We extend this simplification relation ($\triangleright_R$) to what is called a *congruent* relation ($\to_R$), which also considers simplification of elimination-introduction pairs that are not at the conclusion of the proof, but anywhere inside the proof. For example, we have

$$\dfrac{\dfrac{\dfrac{\Pi_A :: \Gamma \vdash A \quad \Pi_B :: \Gamma \vdash B}{\Gamma \vdash A \times B}}{\Gamma \vdash A}}{\Gamma \vdash 0 + A} \quad \to_R \quad \dfrac{\Pi_A :: \Gamma \vdash A}{\Gamma \vdash 0 + A}$$

while these two proofs are not in the relation ($\triangleright_R$) because the last rule is not an elimination rule applied to an introduction rule, it is the disjunction-introduction rule ND-OR-INTRO.

Notation 1.3.2.
More generally, for a relation whose notation is a variation on the notation ($\triangleright$), we use the same variation on the notation ($\to$) to indicate its congruent extension – for example, ($\to_E$) is the relation that let us expand a subproof by eliminating then reintroducing the same connective.

Credits I realized that the reduction and expansion principles were a thing in logic alone (I was familiar with their programming-language counterpart) and could be used to test the logic during excellent lectures by Frank Pfenning at the Oregon Programming Language Summer School. Frank Pfenning has lecture notes on Linear Logic available online, that

present this idea (named *harmony*, in reference to the philosophy of logic of Michael Dummett) and many more – warmly recommended. Noam Zeilberger also has lecture notes discussing harmony [Zeilberger, 2013], which are a pleasure to read; they are also meant to introduce *focusing*, an important notion in proof theory that we study in Chapter 7.

1.4. Proving consistency (without disjunctions) by normalization

1.4.1. Defining consistency

The global and local properties discussed in Sections §1.3.1 and §1.3.3 are relatively easy to establish: they can serve as an obvious sanity check for whatever new logic system you just came up with.

A more ambitious property to establish is *consistency*: can our logic prove false? Notice that due to the elimination rule ND-FALSE-ELIM for the false formula 0, if we can build a proof of 0 in our logic then we can prove *any* formula A in our logic:

$$\frac{\emptyset \vdash 0}{\emptyset \vdash A}\text{ND-FALSE-ELIM}$$

A subtlety is that not all logics have an explicit false formula such as our 0 formula. In this case, what is a consequence of inconsistency (being able to prove false) in our logic can be taken as its definition, and we ask: can the logic prove *all* formulas? A logic that can prove all formulas (or no formula) is not very interesting.

When we ask whether a logic can "prove false", we mean a proof of false in the empty context $\emptyset$. Indeed, it is not very hard to prove false if we can choose the context: we just need to add 0 as an assumption in the context, and use the axiom rule ND-AXIOM.

A more general way to phrase this question would be to ask for the set of *consistent* contexts, those which cannot prove false, to be non-empty: then the logic is consistent if there exists a context for which there exists a formula that cannot be proved. The concept of consistent context is important (for example in terms of language design [Scherer and Rémy, 2015]) and offers a richer dichotomy than just "empty" vs. "non-empty". In a logic that admits weakening (such as the one of Figure 1.2), asking for the empty context to be consistent is the strongest requirement: if $\emptyset \vdash 0$, then any context Γ is inconsistent ($\Gamma \vdash 0$) by weakening.

1.4.2. A plan to prove consistency

Consistency is a very important property for a logic to have, but it is sensibly harder to establish than the previous checks we discussed. In fact, our proof of consistency relies on one of those sanity checks, the idea of *reduction* of eliminations of introductions – which in turns relies on the global property that weakening and substitutions are admissible.

The idea of the consistency proof is to study the subset of proofs that have a very particular form: they never apply an elimination rule to an introduction rule for the same connective – in other words, they cannot be reduced to a simpler proof. Those proofs, which we call *normal* proofs, have a specific structure that we can reason about. In particular, we can prove that there exists no normal proof of the false judgment $\emptyset \vdash 0$. To then deduce that the whole logic is consistent, we also prove that we can repeatedly reduce any proof of a judgment $\mathcal{J}$ to simpler proofs, until we obtain a normal proof of $\mathcal{J}$. In other words, if we had a (non-normal) proof of $\emptyset \vdash 0$, then we could repeatedly reduce it into a normal proof of $\emptyset \vdash 0$, which is absurd as no such proof exists. Thus, our logic is consistent. We formalize this argument in the rest of this section.

This proof technique introduces two important ideas. The first idea is to look at a particular subset of proofs which have a stronger, more precise structure (here the *normal* proofs). Many interesting logics can be first discovered as particular restrictions of existing logics. It is often possible to present them more directly, by giving a system of inference

rules characterizing exactly those proofs – we see this idea at work in Chapter 7 (Focusing in sequent calculus). But studying the properties of the proofs in this subset, typically showing the substitution principle, often requires reasoning in the larger world of the initial space of less-structured proofs.

The second idea is the process of repeated reduction. It is a powerful idea that is useful in many other situations than proving consistency; to the point that (although proving consistency is essential), some would say that the essential property a proof system should have to be a "good logic" is a form of normalization – often called the *cut elimination* property for reasons that are explained in Chapter 4 (A better proof system: sequent calculus). It is also the principle at the core of the correspondence between proofs and programs, as detailed in Chapter 3 (Curry-Howard of reduction and equivalence).

Unfortunately, the presence of disjunctions makes it sensibly more difficult to study the normalization properties of our proof system. For now we must restrict our cut-elimination result to the disjunction-free fragment of our logic, $\mathsf{PIL}(\to, \times, 1, 0)$. This is a good illustration of the issues raised by disjunctions (sums). Consistency of the full logic will be shown in Theorem 3.3.9 (Consistency of $\mathsf{PIL}(\to, \times, 1, +, 0)$), Chapter 3, using ideas coming from program equivalence.

1.4.3. Defining normalization

The idea of normalization is very simple: a proof is *normal* if it does not contain an elimination of an introduction (for the same connective). To show that if there exists a proof Π of a given judgment $\mathcal{J}$, then there exists a normal proof of $\mathcal{J}$, we show that we can repeatedly apply one of the reduction rules:

$$\Pi \to_R \Pi_1 \to_R \Pi_2 \to_R \dots \to_R \Pi_n \not\to_R$$

After some number of reduction steps (which depends on the initial proof Π), this repeated simplification stops, because Π_n is normal: no more reduction rule can be applied. Because the reduction relation $(\triangleright_R)$ preserves the conclusion of proofs, we know that Π_n is a normal proof of the judgment $\mathcal{J}$.

Notation 1.4.1.
For a relation $(\mathcal{R})$ we write $(\not\mathcal{R})$ for its negation: $a \not\mathcal{R}\ b$ holds if and only if $a\ \mathcal{R}\ b$ does not hold.

Notation 1.4.2.
For a relation $(\mathcal{R})$ we write $(a \not\mathcal{R})$ to say that there exists no b such that $a\ \mathcal{R}\ b$.

Under this apparent simplicity lies a surprise. A proof may have elimination-introduction pairs in several places, so there may be several possible choices of where to simplify the proof. These choices could, in turn, lead to simplified proofs that themselves have several possible elimination-introduction pairs, leading to several choices, etc. (Furthermore, because some reductions use the substitution principle which may duplicate some sub-derivation, they may duplicate existing elimination-introduction pairs and thus increase the number of possible choices.)

This means that many different reduction sequences could be possible, starting from a given proof. Some could lead to a normal proof, and some could *not* lead to a normal proof by introducing new elimination-introduction pairs (by substitution) indefinitely. Or maybe *all* possible choices lead to a normal proof? There are in fact two closely related statements:

- *weak normalization*: for any proof $\Pi :: \mathcal{J}$, there exists a finite sequence of reductions (one sequence of simplification choices) that lead to a normal proof of $\mathcal{J}$
- *strong normalization*: for any proof $\Pi :: \mathcal{J}$, *any* sequence of reductions (for any choice of simplification) leads to a normal proof of $\mathcal{J}$ – after a finite number of reduction steps

Strong normalization being a stronger property (it implies weak normalization), it is harder to prove. In this thesis we only prove weak normalization, which suffices to prove that the logic is consistent.

Remark 1.4.1. I find it surprising that, in the case of the propositional logic we are working with, it is relatively *easy* to prove weak normalization (done in Section 1.4.4), and it is *hard* to prove strong normalization.[7]

One explanation for that difference is that difficulty of proofs is intuitively related to the *length* of those reduction sequences (the larger the number of repeated reductions, the harder to prove finite). To prove weak termination, it suffices to describe a particular reduction *strategy*, an algorithm to decide which reduction to perform next. If there is a strategy that is easy to define (there is) and gives short enough reduction sequences (they are), proving weak reduction is easy.[8] On the contrary, strong reduction forces us to consider the *worst case*, the longest sequence among all sequences, and even for the simply-typed lambda-calculus, it can be very large.[9] In some more powerful logics or type systems, there are terms for which *all* reduction sequences are extremely long, so both weak and strong normalization are hard. ∗

1.4.4. Weak normalization

Theorem 1.4.1 (Weak normalization of $\mathsf{PIL}(\to, \times, 1, 0)$ – no sums)**.**
For any proof $\Pi :: \mathcal{J}$ *without sums, there exists a finite sequence of reductions (for the relation* $(\to_R)$ *defined in Section 1.3.4)*

$$\Pi \to_R \Pi_1 \to_R \Pi_2 \to_R \dots \to_R \Pi' \not\to_R$$

such that $\Pi' :: \mathcal{J}$ *is a* normal *proof, in the sense that there does not exist any* Π'' *such that* $\Pi' \to_R \Pi''$.

(More generally, for any relation $(\to)$ we say that a proof Π is $(\to)$-normal if there does not exist any proof Π' such that $\Pi \to \Pi'$.)

Proof. To obtain a weak normalization proof, we use a *complexity measure*: a quantity that we compute for each proof, such that we can always choose to reduce a proof in another proof of strictly smaller measure. We can then argue that this measure cannot decrease indefinitely, and thus that the reduction sequence must be finite – it stops at some point on a proof that cannot be reduced anymore, a normal proof.

Consider a reduction pair, for example

$$\dfrac{\dfrac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}}{\Gamma \vdash A_i} \qquad \triangleright_R \qquad \Pi_i :: \Gamma \vdash A_i$$

The initial proof contains an intermediary judgment proving a conjunction, $\Gamma \vdash A_1 \times A_2$, which has disappeared from the reduced proof. The idea to prove weak normalization is to formalize the intuition that reduction reduces the *complexity* of the formulas appearing in a proof. We can formally define the complexity $\|A\|$ of a formula, a natural number, as follows:[10]

$$\|X\| = \|0\| = \|1\| \stackrel{\mathsf{def}}{=} 1 \qquad \|A \times B\| = \|A \to B\| \stackrel{\mathsf{def}}{=} 1 + \max(\|A\|, \|B\|)$$

To find a terminating sequence of reductions from an initial proof Π, we look, along all the elimination-introduction pairs in Π, at the one(s) whose succedent is of maximal complexity. We obtain a terminating sequence by reducing one of those pairs at each step.

[7] All proof techniques I know of are inspired of either Tait's strong computability or Gandy's strictly monotonic functionals.

[8] On the other hand, the strategy we pick may not be *optimal*, there may exist other strategies that perform less reduction steps. But those are much, much harder to describe.

[9] Not elementary recursive, see Statman [1977].

[10] This is the height of the formula seen as a tree.

One would be tempted to use, as a complexity measure for complete proofs, the complexity of the most complex elimination-introduction pair. As the complexity of a formula is a positive natural number, it cannot strictly decrease indefinitely – this is a good complexity measure. There are however two difficulties.

The first difficulty is that there may be several pairs of maximal complexity. Reducing one of them does not necessarily reduce the maximal complexity of the whole proof. To avoid this problem, we can measure both the maximal complexity of any pair, and the *number* of pairs that have this maximal complexity. If we remove one such pair, the number of pairs decreases strictly, and if we had the last pair of maximal complexity, then the maximal complexity of all pairs decreases strictly.

More precisely, we define the complexity measure $\|\Pi\|$ of the proof Π as a *couple* (m, c) of natural numbers: m is the maximal Measure of all elimination-introduction pairs, and c is the Count of pairs with this measure. A couple is strictly smaller than another, written $(m, c) < (m', c')$, if either m is strictly smaller than m' (the maximal formulas are strictly less complex) or m and m' are equal but c is strictly smaller than c' (the maximal formulas are equally complex, but there are less of them). Notice that there infinite descending chains of strictly smaller measures do not exist: for any couple (m, c), there is a finite number of smaller couples with the same m (the couples $(m, 0), \ldots, (m, c-1)$), and one can descend to a strictly smaller m only finitely many times (formula complexities cannot be smaller than 1).

The second difficulty is that a simplification may in fact introduce new elimination-introduction pairs. Consider the following example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_{B_1} :: \Gamma, A \vdash B_1 \qquad \Pi_{B_2} :: \Gamma, A \vdash B_2}{\Gamma, A \vdash B_1 \times B_2}}{\Gamma \vdash A \to (B_1 \times B_2)} \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B_1 \times B_2}}{\Gamma \vdash B_i} \to_R$$

$$\dfrac{\dfrac{\Pi_{B_1} :: \Gamma, A \vdash B_1 \qquad \Pi_{B_2} :: \Gamma, A \vdash B_2}{\Gamma, A \vdash B_1 \times B_2} \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B_1 \times B_2}\ \text{SUBST}}{\Gamma \vdash B_i} =$$

$$\dfrac{\dfrac{\Pi_{B_1} :: \Gamma, A \vdash B_1 \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B_1}\ \text{SUBST} \qquad \dfrac{\Pi_{B_2} :: \Gamma, A \vdash B_2 \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B_2}\ \text{SUBST}}{\Gamma \vdash B_1 \times B_2}}{\Gamma \vdash B_i}$$

The first step simplifies the elimination-introduction pair for implication, and the second step is just an unfolding of the definition of substitution (Section 1.3.1) in the case of the introduction rule for conjunction ND-AND-INTRO – the third proof is equal to the second proof, just written differently.

We can see that an elimination-introduction pair for conjunction appears that was not present in the original proof, as the elimination and introduction rules of this pair were separated by the implication pair. But this is not a problem for our complexity measure, as the new pair has complexity $\|B_1 \times B_2\|$, which is strictly smaller than the complexity of the eliminated pair, $\|A \to (B_1 \times B_2)\|$.

We will do a case analysis of all possible reductions, looking at the "new pairs" they can form. For each form of reduction we have to prove, as in this example, that all newly introduced pairs are on formulas of strictly smaller complexity.

Before this, we should remark that, as we have seen in Section 1.3.1, the substitution operation may make zero, one or several copies of the subproof Π_A; in particular, it may increase the number of elimination-introduction pairs of maximal complexity if some of them are present in Π_A – these pairs are not really "new", they are copies of existing pairs.

To avoid this increase in complexity, we must be careful when picking a maximal pair to reduce. We need to choose a maximal pair such that its subproofs themselves contain no maximal pair, but only elimination-introduction pairs of strictly smaller complexity. This is always possible: otherwise, if all pairs contained a subterm of the same complexity, the proof would be infinite, and we have defined valid proofs as *finite* derivation trees of inference rules. Making this choice of a specific pair to reduce (instead of considering any reducible pairs) defines a reduction strategy: we are proving *weak* reduction.

New elimination-introduction pairs after implication reduction Consider the reduction of implication:

$$\dfrac{\dfrac{\Pi_B :: \Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B} \quad \triangleright_R \quad \dfrac{\Pi_B :: \Gamma, A \vdash B \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

How can this reduction create new elimination-introduction pairs that were not present in the initial proof? As we have seen in our previous example, this can happen if Π_B contains an axiom rule for A that is the eliminated premise of an elimination rule, and the rootwardmost rule of Π_A is an introduction that gets substituted there. This can also happen if the rootwardmost rule of Π_B is an introduction, and the whole proof is a premise of an elimination rule on B. These are the two only possible cases.

In these two cases, the new pairs that appear cut on formulas that are smaller than the first simplified pair: we simplify a pair of the form $A \to B$, and create new pairs either on B or A, whose complexities are strictly smaller than $\|A \to B\|$, by definition of the latter as $1 + \max(\|A\|, \|B\|)$.

New elimination-introduction pairs after conjunction reduction

$$\dfrac{\dfrac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}}{\Gamma \vdash A_i} \quad \triangleright_R \quad \Pi_i :: \Gamma \vdash A_i$$

The only possible new elimination-introduction pair in this case is on A_i: it can appear if Π_i starts with an introduction rule, and the simplification is the eliminated premise of an elimination rule. We have, as desired, that $\|A_i\| < \|A_1 \times A_2\|$.

Negative result: new elimination-introduction pairs after disjunction reduction We have explicitly excluded disjunctions of our weak normalization result, because our proof technique fails in this case. Consider the reduction

$$\dfrac{\dfrac{\Pi :: \Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} \quad \Pi_1 :: \Gamma, A_1 \vdash C \quad \Pi_2 :: \Gamma, A_2 \vdash C}{\Gamma \vdash C} \quad \triangleright_R$$

$$\dfrac{\Pi_i :: \Gamma, A_i \vdash C \quad \Pi :: \Gamma \vdash A_i}{\Gamma \vdash C}\ \text{SUBST}$$

New elimination-introduction pairs may come from the substitution of $\Pi :: \Gamma \vdash A_i$, creating a new pair on A_i if Π_A has an introduction at its root. This new pair on A_i is strictly simpler than the reduced pair on $A_1 + A_2$.

Unfortunately, if the rootwardmost rule of Π_i is an introduction, and the simplification is the eliminated premise of an elimination rule, we may have a new pair on C. We have no control over the complexity measure $\|C\|$, which is unrelated to $\|A_1 + A_2\|$.

(A tempting idea (we tried) is to define the complexity of such introduction-elimination pairs as $\max(\|A_1 + A_2\|, 1+\|C\|)$. Then the complexity of this reduction seems to decrease strictly, but the problem is reported on substitutions: if one of the A_i is itself a sum, and gets substituted in place of an axiom rule just above a disjunction elimination on some gigantic formula C', then the maximal complexity of the proof can grow arbitrarily.)

Conclusion We have been able to show that, in absence of disjunctions, the new introduction-elimination pairs introduced by reducing a pair of maximal complexity that has no pair of maximal complexity in its subproof are strictly less complex. This means that our notion of complexity is a valid complexity measure: the complexity of the whole proof decreases strictly at each reduction step, so reduction eventually stops. This specific choice of elimination-introduction pair to reduce always results in a normal proof after a finite number of reductions. □

1.4.5. Consistency

Recall the plan to prove consistency (§1.4.2): we have proved that for any valid proof $\Pi :: \mathcal{J}$ there is a $(\to_R)$-normal proof of the same judgment $\mathcal{J}$. We now prove that there is no $(\to_R)$-normal proof of the false judgment $\emptyset \vdash 0$.

Because 0 has only an elimination rule and no introduction rule, we know that a normal proof of $\emptyset \vdash 0$ necessarily starts with an elimination proof; this elimination could be, for example, an elimination of the conjunction $A \times 0$ for an arbitrary A, or an implication $A \to 0$, or a disjunction $A + B$. To prove that these cannot happen, we need to prove a stronger result than consistency (we strengthen our induction hypothesis); we do not only prove that there is no proof of $\emptyset \vdash 0$ that starts with an elimination, but that there is no proof of $\emptyset \vdash A$, for any A, starting with an elimination.

Lemma 1.4.2 (Closed normal proofs are not eliminations)**.**
Any $\mathsf{PIL}(\to, \times, 1, +, 0)$ *judgment in the empty context* $\emptyset \vdash A$ *has no* $(\to_R)$*-normal proof starting with an elimination rule.*

Proof. We prove (by induction on non-necessarily-normal proofs) that, inside a complete proof, an elimination rule in the empty context either is part of an elimination-introduction pair, or has one premise that is also an elimination rule in the empty context.

This suffices to conclude our proof. Indeed, in $(\to_R)$-normal proofs the case of an elimination-introduction pair is impossible, so a proof starting with an elimination in the empty context would necessarily have one premise starting with an elimination, also in the empty context. Unfolding this reasoning, we see that such a proof would need to be infinite – said otherwise, we can prove by structural induction that no finite proof starts with an elimination: no leaf rule is an elimination, and if a proof started with an elimination it would have a subproof that is also an elimination, which is impossible by induction hypothesis.

Conjunction

$$\frac{\Pi :: \emptyset \vdash A_1 \times A_2}{\emptyset \vdash A_i}$$

The proof Π either introduces the conjunction, forming an elimination-introduction pair, or starts with an elimination rule itself – in the empty context. (Besides introduction and elimination rules, the only other rule of natural deduction is the axiom rule, which is not applicable in the empty context.)

Implication

$$\frac{\Pi :: \emptyset \vdash A \to B \qquad \emptyset \vdash A}{\emptyset \vdash B}$$

The proof Π either introduces the implication, forming an elimination-introduction pair, or starts with an elimination rule itself – in the empty context.

Disjunction

$$\frac{\Pi :: \emptyset \vdash A + B \qquad A \vdash C \qquad B \vdash C}{\emptyset \vdash C}$$

The proof Π either introduces the disjunction, forming an elimination-introduction pair, or starts with an elimination rule itself – in the empty context. □

Using our weak $(\triangleright_R)$-normalization result, we can prove consistency of the disjunction-free fragment $\mathsf{PIL}(\to, \times, 1, 0)$. For the full logic $\mathsf{PIL}(\to, \times, 1, +, 0)$, we can only show that the $(\triangleright_R)$-*normal* fragment is consistent.

Corollary 1.4.3 (Consistency of $(\triangleright_R)$-normal $\mathsf{PIL}(\to, \times, 1, +, 0)$)**.**
There is no valid $(\triangleright_R)$*-normal proof of* $\emptyset \vdash 0$ *in* $\mathsf{PIL}(\to, \times, 1, +, 0)$.

Proof. By Lemma 1.4.2 (Closed normal proofs are not eliminations), the first rule of a proof of $\emptyset \vdash 0$ cannot be an elimination. It cannot be an axiom rule either, as the context is empty. Finally, it cannot be an introduction, as 0 has no introduction rule. □

Theorem 1.4.4 (Consistency of $\mathsf{PIL}(\to, \times, 1, 0)$)**.**
There is no valid proof of $\emptyset \vdash 0$ *in the disjunction-free propositional intuitionistic logic* $\mathsf{PIL}(\to, \times, 1, 0)$.

Proof. If there was a proof $\Pi :: \emptyset \vdash 0$, then by Theorem 1.4.1 (Weak normalization of $\mathsf{PIL}(\to, \times, 1, 0)$ – no sums) there would also be a $(\to_R)$-normal proof $\Pi' :: \emptyset \vdash 0$. This would contradict the previous corollary. □

Consistency of the full logic will be shown in Theorem 3.3.9 (Consistency of $\mathsf{PIL}(\to, \times, 1, +, 0)$).

Credits Much of what I know about logic comes from the seeds planted by the "Groupe de travail de logique" at École Normale Supérieure, a working group that was completely organized and run by students (in particular Marc Bagnol). Around the same time I prepared my undergraduate "mémoire" in collaboration with Silvain Rideau. I presented a consistency proof for Peano Arithmetic, using an infinitary rule for induction to justify the ordinal numbering used in the proof – as explained by Wilfried Buchholz. Silvain presented a model-theoretic demonstration that first-order Peano Arithmetic cannot prove termination of Goodstein sequences. Contrasting my intuition (and lack of, respectively) for these two techniques was one more reason to jump ship from mathematical logic to proof theory as computer scientists do it.

2. Introduction to the formal study of programming: the λ-calculus

2.1. The (untyped) λ-calculus

In Chapter 1 (Introduction to the formal study of logic: natural deduction), we have precisely defined the notions of *logic* and *proofs* as mathematical objects, and demonstrated how to prove some properties of a given logic and its proofs. The purpose of this section is to introduce similarly formal definitions of the notions of *program* and *computation*.

The logic of Section 1.2 is a toy system: it is useful to understand what logic is about, but it is too simple to fully model the reasoning tools of working mathematicians. For example, we allow proof by *implication* (the implication elimination rule corresponds to the *modus ponens* principle), but not proof by *contradiction* – proving A by proving that $A \to 0$ leads to falsity. Neither does it express general statements such as "for all $n \in \mathbb{N}$ there exists a m such that ...". Still, it is a useful first model to develop a theory of proofs, which can be extended in many ways, and made powerful enough to capture mathematical practice.

Similarly, you should expect our notion of program to be highly simplified, missing many aspects that working programmers feel essential. For example, we do not say anything about interaction with the user – our programs just compute results. Still, it is a useful basis to study programming concepts and notions of computations, which can be extended in many ways into full-blown programming languages.

2.1.1. The essence of programming

When describing programming to students that have never programmed before (the coolest class, even if you have to use Java), I usually tell them to think of a computer as a very dumb person, who does what you ask very fast, and never gets bored. You can use this marvelous speed to efficiently automate many useful things, but the price to pay is that you have to describe what you want in an extremely simple, very precise way, without being able to make assumptions about what it already knows and understands.

An immediate temptation when describing tasks to a computer is to copy-paste instructions whenever we want to do the same thing several times, or slight variations of the same things. This is problematic if you later change your mind about what the computer should do, and have to modify dozens or hundreds of instructions. Consider the following program, computing a multiplication table for 7 as a list of triples $(a, 7, a \times 7)$:

```
[
  (1, 7, 1*7);
  (2, 7, 2*7);
  (3, 7, 3*7);
  (4, 7, 4*7);
  (5, 7, 5*7);
  (6, 7, 6*7);
  (7, 7, 7*7);
  (8, 7, 8*7);
  (9, 7, 9*7);
  (10, 7, 10*7);
]
```

If I ask my computer to evaluate this program, it instantly returns the following answer:

```
[(1, 7, 7); (2, 7, 14); (3, 7, 21); (4, 7, 28); (5, 7, 35); (6, 7, 42);
 (7, 7, 49); (8, 7, 56); (9, 7, 63); (10, 7, 70)]
```

Unfortunately, if we decide to change it to get the multiplication table for 6, we have 20 changes to make to the program. On the contrary, consider:

```
let k = 7 in
[
  (1, k, 1*k);
  (2, k, 2*k);
  (3, k, 3*k);
  (4, k, 4*k);
  (5, k, 5*k);
  (6, k, 6*k);
  (k, k, k*k);
  (8, k, 8*k);
  (9, k, 9*k);
  (10, k, 10*k);
]
```

We have given a *name* to the multiplicative coefficient 7. To compute the table for 6, we now have only one place to modify, the definition of `k`. This program computes the same result but it is objectively *better*, as it is easier to evolve and adapt to changing needs.

There is still a lot of redundancy in the way the rows of the table are computed. Yet we cannot just give a name to the row computation, because they are not exactly the same: the multiple of `k` is different in each row. The solution is to give a name to the moving part, and define rows parametrized over that name.

```
let k = 7 in
let row(a) = (a, k, a*k) in
[
  row(1);
  row(2);
  row(3);
  row(4);
  row(5);
  row(6);
  row(7);
  row(8);
  row(9);
  row(10);
]
```

Finally, we can get a more compact description of our program by relying on pre-existing functions provided to us under the form of *software libraries*: the function `List.init` takes an integer `n`, a function `f`, and return the list of values `[f(0); f(1); ...; f(n-1)]`.

```
let k = 7 in
let row(a) = (a, k, a*k) in
List.init 10 (fun i -> row(i+1))
```

We have used several different ways to eliminate redundancies and make the code easier to change. We named expressions of the program, without parameters (`k`) or with parameters for varying subexpressions (`row(a)`). Finally, we built a function (`fun i -> row(i+1)`) to pass to an existing library function.

2.1.2. The minimal λ-calculus

The minimal lambda-calculus (from the Greek letter λ, spoken "lambda"; we sometimes write λ-calculus) is a formal toy programming language whose programs are described by the following grammar (reusing the description language presented in Section 1.2.1). We name it $\Lambda\mathsf{C}(\to)$ – the first letter is an uppercase "lambda".

Figure 2.1.: Terms of the minimal λ-calculus $\Lambda\mathsf{C}(\to)$

$$
\begin{array}{rll}
t, u, r ::= & & \text{terms} \\
& \mid x, y, z & \text{variables} \\
& \mid \lambda x.\, t & \text{abstraction} \\
& \mid t\ u & \text{application}
\end{array}
$$

Instead of "program", we often speak of "terms", or "expressions"; these traditional names make it clearer that we look not only at complete programs, but also program fragments in isolation.

The term $\lambda x.\, t$ represents an expression t parametrized over a variable x. In terms of programming, it can be understood as "the function that, given input x, returns output t", but parametrization has other uses. This syntax was in fact introduced in the 1930s by Alonzo Church who was trying to get the essence, not of programming, but of the first-order quantifiers $\forall x.\, P$ and $\exists x.\, P$ of mathematical logic.

If t is a parametrized expression of the form $\lambda x.\, t'$, then $t\ u$ is the expression that fixes the value of the parameter x to be u. Rather conveniently, if you interpret $\lambda x.\, t'$ as a function, then this also corresponds to applying the function t to the parameter u.

For example, the expression $y\ z$ may be understood as the specialization of the general pattern $x\ z$, with the parameter x instantiated by y. It can thus also be written $(\lambda x.\, x\ z)\ y$.

More generally, $(\lambda x.\, t)\ u$ can also be seen as a way to "give the name x to the term u" inside the term t: this is equivalent to what we wrote $\texttt{let}\ x = u\ \texttt{in}\ t$ in the previous section. In other words, λ-calculus does not only allow to define functions, it also captures the central idea of giving a name to reduce redundancy. This is a good formal vehicle to explore the essence of programming.

Finally, a word on priorities in our syntax. We consider that application is left-associative, that is, write $t\ u\ r$ as an equivalent to $(t\ u)\ r$. We also consider that application has precedence over abstraction: $\lambda x.\, t\ u$ is equivalent to $\lambda x.\, (t\ u)$ – intuitively, λ-abstraction scopes as far to the right as possible.

2.1.3. Binding, bound, free variables, and shadowing

Consider the term $\lambda x.\, y\ x\ x$ – with parentheses, this is $\lambda x.\, ((y\ x)\ x)$. The variable x occurs three times in this term, but the occurrences do not all play the same role. In $\lambda x.$, the parameter x is introduced. In $(y\ x\ x)$, the variable x refers to this parameter that has been introduced. We say that uses of a parameter are *bound* to its definition – the concept at work is *variable binding*. We say that the occurrence of x in $\lambda x.$ is a *binding occurrence*, that the two occurrences in $(y\ x\ x)$ are *bound occurrences* (bound to the binding occurrence), and that λ, as a programming language construction, is a *binder*.

A variable may have several binding occurrences, and several bound occurrences that are not bound to the same binder. This is the case for example of x in the term $(\lambda x.\, x)\ (\lambda x.\, x)$.

Finally, sometimes variables appear that are bound to nothing. For example in the term $x\ (\lambda y.\, y)$, the occurrence of the variable x has no corresponding binder. We say that it is a a *free occurrence*, that the variable x is *free* in this term.

The same variable may have both free and bound occurrences in the same term, for example in $x\ (\lambda x.\, x)$. A variable may even be bound at a place where it was already bound to some binding: consider $\lambda x.\, x\ (\lambda x.\, x)$ for example. In this case, we say that the

innermost binding *shadows* the previous bindings: inside the scope of this binding, x refers to it, and the "previous" definitions of x cannot be referred to – they are shadowed.

This should all be familiar to mathematicians: in the function definition $x \mapsto \int_0^r t^x dt$, the variables x and t are bound (the binding occurrences are in "$x \mapsto$" and "dt"), while the variable r is free.

2.1.4. On α-equality

It is not quite right to say that λ-terms are exactly the terms described by the grammar in Figure 2.1. The binding structure is what matters, but the precise choice of variable names does not matter so much. For example, we consider that $\lambda x.\, x$ and $\lambda y.\, y$ are "the same" object (the function that returns its argument), while in terms of concrete syntax they use different names. On the contrary, x and y are *not* the same object, as they refer to distinct free variables.

We could formally define an equivalence relation (traditionally named α-equivalence) that captures this notion of being "the same" modulo renaming of bound variables, but it is tedious, technical, and not the point of interest of this thesis. We skip over this difficulty, and simply consider two α-equivalent terms as equal. Formally, we are working on the objects represented by the grammar quotiended over the α-equivalence relation.

Again, mathematicians will not be surprised by the fact that $\int_0^1 t^2 dt$ and $\int_0^1 s^2 ds$ have the same meaning.

2.1.5. Substitution of variables

We write $t[u/x]$ for the term t where all free occurrences of x have been replaced by the term u. This meta-operation is called *substitution*. For example, $(x\ (\lambda y.\, x))[u/x]$ is a notation for $u\ (\lambda y.\, u)$. There are many other notations for this operation, such as $t\{u/x\}$, $t[x \backslash u]$ or even $[u/x]t$; the important thing to notice is the direction of the slanted bar: the stuff "on top" of the bar replaces the stuff "below" the bar. Syntactically, we give substitution the highest precedence: $t\ t'[u/x]$ means $t\ (t'[u/x])$, and $\lambda y.\, t[u/x]$ means $\lambda y.\, (t[u/x])$.

We use substitution a lot so it makes sense to give a formal definition of it, usable in proofs. It is defined in Figure 2.2 (Substitution for the minimal λ-calculus $\Lambda\mathsf{C}(\to)$).

Figure 2.2.: Substitution for the minimal λ-calculus $\Lambda\mathsf{C}(\to)$

$$
\begin{array}{lcl}
x[u/x] & \stackrel{\text{def}}{=} & u \\
y[u/x] & \stackrel{\text{def}}{=} & y \\
(\lambda y.\, t)[u/x] & \stackrel{\text{def}}{=} & \lambda y.\, t[u/x] \\
(t\ t')[u/x] & \stackrel{\text{def}}{=} & t[u/x]\ t'[u/x]
\end{array}
$$

There are two notational assumptions that are left implicit in this definition. First, the first two cases handle all cases where the substitution is performed on a variable: $x[u/x]$ is the case where this variable is equal to the variable being substituted (we replace it by u), and $y[u/x]$ implicitly defines the meaning on all variables that are distinct from x (we leave them unchanged).

Second, when defining substitution on a λ-abstraction, we have explicitly used a binder that is different from the one on which the substitution is performed. If we did not take α-equivalence into account (Section 2.1.4), this would not cover all cases: $\lambda x.\, x$ is a perfectly valid term and we may want to compute $(\lambda x.\, x)[u/x]$. The idea is that $\lambda x.\, x$ is equal to $\lambda y.\, y$ by α-equivalence, and thus $(\lambda x.\, x)[u/x]$ is necessarily equal to $(\lambda y.\, y)[u/x]$; the later is clearly well-defined in the notation of Figure 2.2, equal to $\lambda y.\, y[y/x]$, that is $\lambda y.\, y$.

When we "open" a λ-abstraction during substitution $(\lambda \ldots)[u/x]$, α-equivalence let us

pick any name for the λ-bound variable. We have argued that we never choose the same name as the variable x being substituted. There is one last subtlety: we should also avoid variable names that appear free in the substituted term u. This is always possible because there are finitely many free variables in u, and infinitely many possible choices of variable names. Consider for example the substitution $(\lambda y.\, x)[(y\ y)/x]$. If we naively perform the substitution without α-renaming the binder $\lambda y.$ first, we would get the result $\lambda y.\, y\ y$. If we rename it into z, we get the result of $\lambda z.\, x[y\ y/x]$, which is just $\lambda z.\, y\ y$. The first choice is wrong: it places the term $y\ y$, where the variable y is free, inside a subterm where the variable y is bound: we say that the free occurrences of y would be *captured* by the binder $\lambda y.$ during the substitution. By (implicitly) requesting that the binder y in the definition of $(\lambda y.\, t)[u/x]$ not appear in the free variables of u, we avoid this case: our notion of substitution is *capture-avoiding*. Capture-avoiding substitution is almost always the right thing for substitutions of variables that can be bound locally; other notions of substitutions are sometimes used, but then it is always explicitly stated.

This subsection shares the general embarrassment of this field, that one of our central notions (variable binding) is actually quite subtle to define formally. The good news is that, to humans that have some habit of working with variables, what I just described is obvious (and boring); this let us leave most of the obvious details out, and express ourselves concisely (the verbiage on shadowing and capture will hopefully remain limited to very specific places in this document). The bad news is that this thorn resurfaces when doing computer-checked proof, a laudable tendency of computer science in general, where one must be fully-explicit about the intricacies of variable bindings – again. Whole PhD theses have been written about this. The present thesis is about something else, so we have to tolerate a certain degree of imprecision.

Remark 2.1.1. For the working mathematician this should again be an obvious (if somewhat nitpicky) remark: if I define $f : x \mapsto \int_0^1 t^x dt$, and then try to integrate f itself, for example $\int_1^2 f(t)dt$, this is *not* equal to the double integral $\int_1^2 (\int_0^1 t^t dt)dt$, but it is equal to $\int_1^2 (\int_0^1 s^t ds)dt$ for example. When replacing f by its definition, we have performed a capture-avoiding substitution. Note that mathematicians most often do not discuss these issues at all and take them as granted. There are two reasons why we are more precise here:

- In our field, unlike in mathematics (sadly), it is common to implement on an actual computer the programs or algorithms we describe formally, and the issue of variable representations is thus one that is encountered in practice, in a setting where the usual human way to keep those subtleties unspoken does not suffice.

- From a didactic point of view, one reason why mathematicians have a harder time understanding the λ-calculus than integrals is that meaning of λ-terms is to be found in their *syntax*, while integrals are often understood from their *semantics* (as real numbers) first. That is, mathematical students think of the sequence of symbols $\int_0^r t^x dt$ as a description of how to compute a particular number; so this expression is perceived to reduce to something they are already familiar with. On the contrary, the "meaning" of the expression $(\lambda x.\, (y\ x))$ is the expression itself (this is syntax: uninterpreted data), we cannot explain it away by saying that it computes to something we already know. The same phenomenon happens in mathematics: there are many different notions of integrals, that are all specific interpretations of a common syntax; or people can manipulate what they call "formal sums" $\Sigma_{i=0}^{\infty} f(i)$ as pure syntax when they do not know how to give them a meaning as convergent sequences – in general "formal foo" means "foo as syntax". But it happens in more advanced setting, while here syntax must be understood, or at least accustomed to, as the first step towards the study of programming language theory.

*

Remark 2.1.2. This underlying worry about variable binding is not only of consequence inside the ivory tower of cloud-gazing academics. Practical technologies, in particular programming languages, have displayed tremendous creativity in getting variable binding *wrong*. These failures would be amusing if they didn't impose a tax on programmer efforts, occasionally distracting them from real work in insidious ways. Lisp had dynamic scope and we made jokes about it, but Python has `nonlocal`, most languages wrongly assume that loop indices are mutated rather than rebound, and Coffeescript doesn't allow to express variable scope (and thus shadowing) without elaborate defensive strategies. Even the high-brow languages of the ML family (OCaml, SML, Haskell) long failed to recognize the utility of proper binders for *type* variables. ∗

2.1.6. Reducing λ-terms

If those λ-terms are formal objects representing programs, there should also be a formal notion of computation. There are several good ways of defining computation used by programming language researchers. The one we present here is called *small-step semantics*, and consists in seeing the execution of a program as a series of program transformations, from the initial program to simpler and simpler programs. When a program cannot be simplified anymore, computation stops, and this final program is "the result". This is a very simple way to define computation as it does not require defining an additional class of "program results" (those are just programs); but it would need to be extended to cover many computational phenomena happening in realistic programming languages, such as user interaction and mutable memory.

For historical reasons, we call β-reduction (this is the Greek letter "beta") this reduction relation on programs. It is defined in a very simple way. First, we define a "head β-reduction" relation ($\triangleright_\beta$) as follows:

Figure 2.3.: Head reduction for the minimal λ-calculus $\Lambda\mathsf{C}(\to)$

$$(\lambda x.\, t)\ u \quad \triangleright_\beta \quad t[u/x]$$

Then we define the full β-reduction ($\to_\beta$) as the congruence closure of ($\triangleright_\beta$), that is the relation that let us apply ($\triangleright_\beta$) anywhere inside a subterm. For example, if $t \triangleright_\beta t'$, then we have $\lambda z.\, z\ t \to_\beta \lambda z.\, z\ t'$. More precisely, we can define this congruence closure as a system of inference rules:

Figure 2.4.: Full β-reduction for the minimal λ-calculus $\Lambda\mathsf{C}(\to)$

$$\frac{t \triangleright_\beta t'}{t \to_\beta t'} \qquad \frac{t \to_\beta t'}{\lambda x.\, t \to_\beta \lambda x.\, t'} \qquad \frac{t \to_\beta t'}{t\ u \to_\beta t'\ u} \qquad \frac{u \to_\beta u'}{t\ u \to_\beta t\ u'}$$

Notation 2.1.1.
For any relation ($\mathcal{R}$) from a set to itself, we write ($\mathcal{R}^*$) for its reflexive transitive closure: we have $a\ \mathcal{R}^*\ b$ if there is a chain

$$a = a_0\ \mathcal{R}\ a_1\ \mathcal{R}\ a_2 \ldots \mathcal{R}\ a_n = b$$

This chain may be empty if $a = b$; in other words we always have $c\ \mathcal{R}^*\ c$.

For example, we write $t \to_\beta^* u$ if u can be reached from t by a (possibly empty) sequence of β-reductions.

Notation 2.1.2.
For any relation ($\mathcal{R}$) we use the symmetric notation ($\mathcal{R}$) for the symmetric relation. For example, $a \triangleleft b$ if and only if $b \triangleright a$, and $a \leftarrow b$ if and only if $b \to a$.

Notation 2.1.3 Equivalence closure**.**
We write $(\approx_\beta)$ for the smallest equivalence[1] containing $(\to_\beta)$. In other words, $t \approx_\beta u$ if there is a chain $t_0, t_1, \ldots, t_n$ such that $t_0 = t$, $t_n = u$, and for each $i \in [1; n[$ we have $t_i \to_\beta t_{i+1}$ or $t_i \leftarrow_\beta t_{i+1}$.

In general we write $(\approx_\mathcal{R})$ for the equivalence closure of an arbitrary congruent relation $(\to_\mathcal{R})$.

2.1.7. Computing with λ-terms

In Section 2.1.2 we mentioned that the simple mechanisms of λ-abstraction and application could express several different programming patterns: both the creation of parametrized functions and the introduction of auxiliary definitions to decrease redundancy.

We can, in fact, go much further than that: those two constructions are enough to express many interesting computational behaviors, such as booleans and conditionals, natural numbers and arithmetic operations on them, or even aggregation of data such as pairs, optional data, lists, etc. In fact, the functions from natural numbers to natural numbers definable as λ-terms are exactly those definable using a Turing Machine, usually considered as the gold standard for the notion of "computable" – λ-calculus is just as expressive as a foundational formalism to define computability. We say that the minimal λ-calculus is Turing-complete.

Booleans To represent booleans, the core idea is that the conditional test $(\texttt{if}\ t\ \texttt{then}\ u_1\ \texttt{else}\ u_2)$ can be represented as just a function application with two parameters, $(t\ u_1\ u_2)$. Because $(\texttt{if true then}\ u_1\ \texttt{else}\ u_2)$ should be equal to u_1, we ask that $(\texttt{true}\ u_1\ u_2)$ reduce to u_1; it suffices to define$(\texttt{true} \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, x)$. Conversely, we pose $\texttt{false} \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, y$. We have, as expected

$$
\begin{array}{ll}
 & \texttt{if true then}\ u_1\ \texttt{else}\ u_2 \\
= & (\lambda x.\, \lambda y.\, x)\ u_1\ u_2 \\
\to_\beta & (\lambda y.\, x)[u_1/x]\ u_2 \\
= & (\lambda y.\, u_1)\ u_2 \\
\to_\beta & u_1[u_2/y] \\
= & u_1
\end{array}
$$

Remark 2.1.3 (Encoding arbitrary datatypes)**.** Retrospectively, there is another way to read this definition that generalizes to other encodings into the λ-calculus. The idea is that we want to define booleans in a language that has no data, only parametrization. How can we define `true` and `false`? Well, let's just parametrize over them! We can represent all boolean values as terms of the form $\lambda x.\, \lambda y.\, t$, where t is the definition of the boolean we want, *assuming* the parameter x represents "true" and y represents "false". From this point of view, `true` is just x; with the parametrization made explicit, this is $\lambda x.\, \lambda y.\, x$, our previous definition. Then, our definition for $\texttt{if}\ t\ \texttt{then}\ u_1\ \texttt{else}\ u_2$ has an interesting interpretation: by defining it as $(t\ u_1\ u_2)$, we say that the result is the boolean t, where specific choices have been made for the meaning of "true" and "false": truth is locally defined as u_1, and falsity as u_2. This does correspond to a conditional branch. $*$

Natural numbers Natural numbers can be built from just two concepts: the natural number 0, and the successor operation $n \mapsto (n + 1)$. Any natural number can be written as 0, on which the successor operation is called several times: 3 is $((0 + 1) + 1) + 1$.

We can use the idea of parametrization above to encode natural numbers into untyped λ-terms: we will parametrize over the definition of zero, z, and the definition of successor, s. The number 3 is represented by $s\ (s\ (s\ z))$ or, with the parametrization made

[1] An equivalence is a relation $(\mathcal{R})$ that is reflexive $(a\ \mathcal{R}\ a)$, transitive (if $a\ \mathcal{R}\ b$ and $b\ \mathcal{R}\ c$ then $a\ \mathcal{R}\ c$) and symmetric (if $a\ \mathcal{R}\ b$ then $b\ \mathcal{R}\ a$).

explicit, $\lambda s.\, \lambda z.\, s\ (s\ (s\ z))$. There is another way to read this definition: we are defining the operation that takes a function s and a value z, and applies the function s three times to z; this corresponds to what mathematicians sometimes write $s^3(z)$ (the composition operator is the natural choice of product for endofunctions).

This reading makes it easy to define operations on the natural numbers represented as λ-terms. For example, the addition of two natural numbers m and n is just $\lambda s.\, \lambda z.\, m\ s\ (n\ s\ z)$: we first repeat n times the application of the function s to the value z, and then apply it again m times to the result. In total, the function s was applied $m + n$ times to z. The reader may be interested in the following table of simple definitions:

$$
\begin{array}{lcl}
0 & \stackrel{\text{def}}{=} & \lambda s.\, \lambda z.\, z \\
1 & \stackrel{\text{def}}{=} & \lambda s.\, \lambda z.\, s\ z \\
2 & \stackrel{\text{def}}{=} & \lambda s.\, \lambda z.\, s\ (s\ z) \\
3 & \stackrel{\text{def}}{=} & \lambda s.\, \lambda z.\, s\ (s\ (s\ z)) \\
(m+1) & \stackrel{\text{def}}{=} & \lambda s.\, \lambda z.\, s\ (m\ s\ z) \\
\texttt{succ} & \stackrel{\text{def}}{=} & \lambda m.\, (m+1) \\
(m+n) & \stackrel{\text{def}}{=} & n\ \texttt{succ}\ m \\
(m \times n) & \stackrel{\text{def}}{=} & n\ (\lambda k.\, (k+m))\ 0 \\
(m^n) & \stackrel{\text{def}}{=} & n\ (\lambda k.\, (k \times m))\ 1 \\
(n = 0) & \stackrel{\text{def}}{=} & n\ (\lambda x.\, \texttt{false})\ \texttt{true}
\end{array}
$$

It is easy (but very boring; good for computers rather than humans) to check, for example, that the encoding of 3^2 reduces to the encoding of 9 after a large number of β-reduction steps.

Unfortunately, this definition of natural numbers makes it difficult to write the predecessor function $n \mapsto (n-1)$. The integer n makes it easy to iterate a transformation n times, and as we have seen with the definition of $n = 0$ we can easily turn this into "zero or more times", but there is no simple solution to iterate exactly $n - 1$ times – the reader might want to consider this a puzzle, we will give a solution in the next paragraph.

Pairs To define a pair (t, u), the only operator to parametrize on is the "comma" used in the pair construction: we can simply encode this term as $(\lambda c.\, c\ t\ u)$. Then, to obtain the first (respectively, second) element of a pair, is suffices to instantiate the comma with the function that returns its first (respectively, second) element.

$$
\begin{array}{lcl}
(t, u) & \stackrel{\text{def}}{=} & \lambda c.\, c\ t\ u \\
\pi_1\ p & \stackrel{\text{def}}{=} & p\ \texttt{true} \\
\pi_2\ p & \stackrel{\text{def}}{=} & p\ \texttt{false}
\end{array}
$$

Pairs allow to define $n - 1$. The idea is that instead of a single number on which to perform an operation n times, we will operate on a *pair* of numbers, one representing the current iteration of the operation, and one recording the result we had one step before. Repeating n times the increment operation $m \mapsto (m+1)$, starting from 0, gives the same number n we started from, and the value of the previous round gives $n-1$. We first define an auxiliary function $\texttt{shift}$ such that $\texttt{shift}\ f\ (x, y) \stackrel{\text{def}}{=} (f(x), x)$ – and in particular $\texttt{shift}\ f\ (f^n(x), f^{n-1}(x)) = (f^{n+1}(x), f^n(x))$ – and use this to define $(n-1)$.

$$
(n-1) \stackrel{\text{def}}{=} \left(\begin{array}{l} \texttt{let shift} = (\lambda f.\, \lambda p.\, \texttt{let}\ x = \pi_1\ p\ \texttt{in}\ (f\ x, x))\ \texttt{in} \\ \quad \pi_2\ (n\ (\texttt{shift succ})\ (0, 0)) \end{array} \right)
$$

Remark 2.1.4. This is admittedly scary. There are other ways to represent the natural

numbers that make the predecessor function much easier to define – and give its computation a better complexity in term of number of necessary reduction steps. But this encoding is simple and neat, and the predecessor function provides an amusing exercise to combine already-seen notions – it serves as the first non-trivial program.

The present encoding is named the "Church Encoding", and was extended in Berarducci and Böhm [1985] into a general encoding scheme for inductive datatypes, informally described in Remark 2.1.3 (Encoding arbitrary datatypes). It is simple and adequate when one is interested in the expressivity of a system, rather than more precise, demanding notions of complexity or ease of programming. Other usual encodings include the Scott encoding (which makes predecessor trivial but does not provide recursion for free), and a combination of both Church and Scott techniques named Church-Scott or Parigot encoding by Geuvers [Geuvers, 2015]. ∗

Bugs It is possible to write many useful programs in the λ-calculus, but also relatively useless ones. Let us define $\texttt{auto} \stackrel{\text{def}}{=} \lambda x.\, x\ x$ the function that applies its parameter to itself. This already reeks of paradoxes: the element that does not belong to itself, the parameter that applies to itself, etc. We make this suspicion precise by considering the term $\texttt{auto}\ \texttt{auto}$, which has the following reduction sequence (and no other):

$$\begin{array}{ll} & \texttt{auto}\ \texttt{auto} \\ = & (\lambda x.\, x\ x)\ \texttt{auto} \\ \triangleright_\beta & (x\ x)[\texttt{auto}/x] \\ = & \texttt{auto}\ \texttt{auto} \\ \triangleright_\beta & \texttt{auto}\ \texttt{auto} \\ \triangleright_\beta & \texttt{auto}\ \texttt{auto} \\ \triangleright_\beta & \dots \end{array}$$

This term reduces to itself in(de)finitely. In particular, there is no reduction sequence that eventually reaches an irreducible term – our notion of "result" of a computation. $\texttt{auto}\ \texttt{auto}$ is a computation with no result.

Remark 2.1.5. There are some programs for which never stopping is actually useful: an alarm system, a mail server, etc. Those programs do something along the way: they loop, but they are "reactive" or "productive" in senses that can be precisely defined [Turner, 1995]. In contrast, our $\texttt{auto}\ \texttt{auto}$ program loops and does nothing (it reduces to itself, producing no data or interaction along the way). We may call this *silent* non-termination. ∗

Fixpoints and general recursion We can define in the λ-calculus a fixpoint operator Y such that $Y\ f \approx_\beta f\ (Y\ f)$: applying the function f to the argument $Y\ f$ leaves it unchanged. Several distinct definitions share this property, but the following (called the Y combinator) relies on the same bag of tricks used to define our looping program:

$$Y \stackrel{\text{def}}{=} \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

$$\begin{array}{ll} & Y\ f \\ \triangleright_\beta & (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)) \\ = & (f\ (x\ x))[(\lambda x.\, f\ (x\ x))/x] \\ = & f\ ((\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))) \\ \leftarrow_\beta & f\ (Y\ f) \end{array}$$

Such a fixpoint combinator let us define arbitrary recursive functions. For example, the reader may want to check that the following expression reduces to (the λ-term encoding of) 40320, the factorial of 8:

$$Y\ (\lambda f.\, \lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n \times f\ (n-1))\ 8$$

Remark 2.1.6. $Y\ (\lambda f.\, f)$ reduces to $\texttt{auto}\ \texttt{auto}$, our previous looping term: the function defined only as "the fixpoint of itself" is a simple example of silent non-termination. ∗

Expressibility of general recursive functions Turing machines are relatively complex to define and inelegant to work with, so we will not attempt to show the correspondence with the minimal λ-calculus in this introduction. There is a third computational model, however, that is known to be equivalent to both, namely the so-called μ-recursive partial functions, a generalization of the class of primitive recursive functions. We now prove that all such functions can be represented as λ-terms.

The class of μ-recursive functions is the smallest set of partial functions from tuples of natural numbers to a natural number which contains constant functions (we have those), the successor function `succ` (we have it), projection functions (representing tuples as pairs of pairs of pairs... defining projections is immediate), and is closed by

- composition, trivially defined in the λ-calculus
- primitive recursion, a glorified way to define functions by induction on natural numbers (we can define those easily with a fixpoint iteration, and a bit more subtly without)
- minimization, that is the existence for any partial function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ of a partial function $\mu(f) : \mathbb{N}^k \to \mathbb{N}$ such that $\mu(f)(p)$ returns the smallest natural number n such that $f((n, p)) = 0$, if it exists. This can be easily encoded in the λ-calculus as

$$\mu(f) \stackrel{\text{def}}{=} \lambda p.\, Y\ (\lambda K.\, \lambda n.\, \texttt{if}\ f\ (n, p) = 0\ \texttt{then}\ n\ \texttt{else}\ K\ (n + 1))\ 0$$

We have described how to build a λ-term from any of the "building blocks" of recursive functions: this let us "encode" any recursive function as a λ-term. The correspondence between the encoding result and the initial recursive function f is then as follows: f is defined on some input p and $f(p) = n$ if and only if the encoding of f applied to the encoding of p reduces to the encoding of n (which is a normal form).

2.2. Programming errors and the λ-calculus

2.2.1. To understand failure, we should first allow it

For all its expressive power, the minimal λ-calculus has one irritating defect: it is unable to represent a very common form of failure in actual programming languages, the *invalid state* error. Sometimes programs end up in a situation where they are asked to perform invalid operations, that have no meaning. They have no better choice than stopping their normal execution.

This is not the case of the λ-calculus we have seen so far, because all the term-forming operations are total: in particular, anything can be applied to anything. For example, the application $(1\ x)$ makes sense, because the number 1 is defined as a function. In fact, everything is a function, a very strange property that most other programming languages do not share.

The minimal λ-calculus is a reasonable choice to talk about what *can* be expressed as a program, what is computable. But if the only available notion of failure is non-termination, which is fairly hard to manipulate, it is insufficient as a formal vehicle to study *errors* in programming.

Remark 2.2.1. We are in a situation similar to the literal reading of foundational works on mathematics, claiming that everything can be defined as sets. We can indeed define all mathematical objects as sets, but this abstraction-free view has the downside that, for example, $1 \subseteq ((x : \mathbb{N}) \mapsto x + 1)$ is a valid propositional statement: is 1, seen as a set, included into the successor function on natural numbers, also a set? We would rather reject this question as nonsensical than attempt to answer it.

A common (categorical) way to handle the problem is to remark that the answer depends on the specific choice of sets used to encode these objects, and to only consider statements

that are *well-defined* in the sense that they do not depend on such choices. This requires, of course, to be explicit about the level of abstraction at which we are presently speaking; are sets, or natural numbers, the objects of consideration? ∗

To fix this issue in the simplest possible way, we will add a concept of "boxes" to our untyped λ-calculus, that adds absolutely no expressive power but is an easy and useful way to obtain terms to which it is *invalid* to apply arguments. This is done by introducing a new term-former $\mathtt{box}(t)$ that puts its subterm "in a box", and saying that boxes are not functions: the application $\mathtt{box}(t)\ u$ is an error. To manipulate boxed terms we provide the symmetric construction $\mathtt{unbox}(t)$ that removes a box around a term – and is invalid if the inner term is a λ-abstraction, not in a box. In other words, we introduce an ability to *fail* in order to be able to study failure.

2.2.2. The administrative λ-calculus

We call *administrative λ-calculus* the extension of the minimal λ-calculus described in Figure 2.5. We name it $\Lambda\mathsf{C}(\to, \mathtt{box})$.

Figure 2.5.: Syntax of the administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$

$$
\begin{array}{rll}
t, u, r ::= & & \text{terms} \\
& \mid x, y, z & \\
& \mid \lambda x.\, t & \\
& \mid t\ u & \\
& \mid \mathtt{box}(t) & \\
& \mid \mathtt{unbox}(t) &
\end{array}
$$

Notation 2.2.1.
To extend a previous grammar, we may specify the extended grammar by using ... to denote the rules of the old grammar, and only explicitly write the new rules. For example, the grammar of Figure 2.5 may be rewritten as:

$$
\begin{array}{rll}
t, u, r ::= & & \text{terms} \\
& \mid \ldots & \text{minimal } \lambda\text{-calculus } \Lambda\mathsf{C}(\to) \\
& \mid \mathtt{box}(t) & \\
& \mid \mathtt{unbox}(t) &
\end{array}
$$

The head-reduction relation extends the relation $(\triangleright_\beta)$ of the minimal λ-calculus (Figure 2.3), with another rule to say that unboxing a box removes the box.

To distinguish the two relations, we will write $(\triangleright_\beta{}^{\Lambda\mathsf{C}(\to)})$ for the reduction relation of $\Lambda\mathsf{C}(\to)$ and $(\triangleright_\beta{}^{\Lambda\mathsf{C}(\to,\mathtt{box})})$ for the reduction relation of $\Lambda\mathsf{C}(\to, \mathtt{box})$.

Figure 2.6.: Head reduction for the administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$

$$(\triangleright_\beta{}^{\Lambda\mathsf{C}(\to)}) \subseteq (\triangleright_\beta{}^{\Lambda\mathsf{C}(\to,\mathtt{box})}) \qquad \mathtt{unbox}(\mathtt{box}(t)) \triangleright_\beta{}^{\Lambda\mathsf{C}(\to,\mathtt{box})} t$$

Notation 2.2.2.
When we want to be explicit about the domain[2] Dom of a relation $(\mathcal{R})$, we write $(\mathcal{R}^{\mathsf{Dom}})$ instead. In general it should be clear from the context.

We can see in particular that applying an argument to a λ-term reduces, that applying an $\mathtt{unbox}(\_)$ to a $\mathtt{box}(\_)$ reduces, but that applying an argument to a $\mathtt{box}(\_)$ or unboxing a

[2]The domain of a relation is the set of objects that may be related together by this relation. The domain of a function is the set of objects to which the function can be applied.

λ-abstraction creates an irreducible term: a term, for example $(\mathtt{box}(t)\ u)$, from which no head-reduction is possible. This is different in nature from the status of $x\ u$ (applying an argument to a variable), which also cannot perform any head-reduction; it may be the case that later a λ-abstraction is substituted for x, allowing reduction. In the case of $(\mathtt{box}(t)\ u)$ we know that it cannot reduce now, but it will also never head-reduce in the future, after applying substitutions. This is, by essence, a failure of computation. We could be even more explicit in our presentation, by listing not only the cases that do reduce, but also those that will never reduce.

$$(\lambda x.\, t)\ u \rhd_\beta^{\Lambda\mathsf{C}(\to,\mathtt{box})} t[u/x] \qquad \mathtt{unbox}(\mathtt{box}(t)) \rhd_\beta^{\Lambda\mathsf{C}(\to,\mathtt{box})} t$$

$$\mathtt{box}(t)\ u \not\rhd_\beta^{\Lambda\mathsf{C}(\to,\mathtt{box})} \qquad \mathtt{unbox}(\lambda x.\, t) \not\rhd_\beta^{\Lambda\mathsf{C}(\to,\mathtt{box})}$$

Note the difference between $\mathtt{unbox}(\lambda x.\, t)$, which we know will never reduce and, informally, is an "invalid" term, and terms like $\mathtt{unbox}(x)$ or $\mathtt{unbox}(t\ u)$ that are also not head-reducible, but may become reducible after a substitution is applied, or after some sub-term is itself reduced: in the second example, $t\ u$ could reduce into some $\mathtt{box}(r)$.

2.2.3. Reduction contexts to define full reduction

We could extend the definition of the full β-reduction relation ($\to_\beta$) given in Figure 2.4, but this would be cumbersome. Notice that relation already requires a rule for reduction under $\lambda x\_$ and two rules for applications (depending on whether one reduces on the left or on the right). This requires adding two extra rules, one for $\mathtt{box}(\_)$ and one for $\mathtt{unbox}(\_)$, and in the general case this definition would grow to have uncomfortably many rules – which also makes *reasoning* on the relation rather tedious. To wit:

$$\frac{t \rhd_\beta t'}{t \to_\beta t'} \qquad \frac{t \to_\beta t'}{\lambda x.\, t \to_\beta \lambda x.\, t'} \qquad \frac{t \to_\beta t'}{t\ u \to_\beta t'\ u} \qquad \frac{u \to_\beta u'}{t\ u \to_\beta t\ u'}$$

$$\frac{t \to_\beta t'}{\mathtt{box}(t) \to_\beta \mathtt{box}(t')} \qquad \frac{t \to_\beta t'}{\mathtt{unbox}(t) \to_\beta \mathtt{unbox}(t')}$$

A good solution to reduce this redundancy, proposed in the seminal article Wright and Felleisen [1994], is to factorize these rules using a grammar of *contexts*. Intuitively, those rules say that a full reduction may "go under" the term-forming constructs of the language: you can reduce under a $\lambda x\_$, on the left of an application $(\_\ u)$, etc. Those things reduction may "go under" are not terms, there are term fragments with one part unfilled. We will reify this intuition into a syntax of partial terms, where the missing part, the *hole*, is written $\Box$. This grammar of *contexts* is defined in Figure 2.7.

Figure 2.7.: Reduction contexts of the administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$

$$\begin{array}{rcll} E, F, G & ::= & & \text{contexts} \\ & & \mid \Box & \\ & & \mid \lambda x.\, E & \\ & & \mid E\ u & \\ & & \mid t\ E & \\ & & \mid \mathtt{box}(E) & \\ & & \mid \mathtt{unbox}(E) & \end{array}$$

Notation 2.2.3.
If E is a context with one hole $\Box$, we write $E\,[t]$ for the *non*-capture-avoiding substitution of t for $\Box$ in E. This operation is often called *plugging the term t in the context E*

By extension, if E and F are contexts, we also write $E\,[F]$ for the *non*-capture-avoiding substitution of F for $\Box$ in E. This operation is often called *composing the context E and the context F*.

The full reduction can then be defined with a *single* rule over contexts, instead of the six rules we used previously:

Figure 2.8.: Full reduction for the administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$

$$\frac{t \rhd_\beta t'}{E\,[t] \to_\beta E\,[t']}$$

The full-reduction rule can be applied for *any* context E. In particular, when E is just a hole $\Box$, we recover the rule saying that $(\rhd_\beta)$ is included in $(\to_\beta)$.

We insist that plugging a term in a context is *not* a capture-avoiding substitution. For example to justify that $\lambda x.\,(\lambda y.\,y)\;x \to_\beta \lambda x.\,x$, we use the decomposition

$$\frac{(\lambda y.\,y)\;x \rhd_\beta x}{(\lambda x.\,\Box)\;[(\lambda y.\,y)\;x] \to_\beta (\lambda x.\,\Box)\;[x]}$$

where the variable x is (intentionally) captured by the context $(\lambda x.\,\Box)$.

Remark 2.2.2. A contrarian reader may remark that we only need one rule instead of our six previous rules because we introduced an extra object with six different cases – we have not really reduced the complexity of the system as a whole. We have two different answers:

- A context-free grammar is a simpler object than an arbitrary system of inference rules. Doing the same with simpler concepts is a win, even at equal sizes. For example, all six rules had both a premise and a conclusion, and the premises were all the same; we factored this redundancy out.
- Contexts will be reused to get further simplifications. In Section 2.2.4 (Formally defining failure), for example, we will reuse our definition of contexts to uniformly define the notion of *failure terms*. Without contexts, we would again add six extra inference rules to express that failures can occur deep inside a term.

*

Another benefit of contexts is that it gives us an easy way to change the reduction relation, if we wish to do so. For example, practical programming language often do not allow to reduce under a λ-abstraction (the body of a function is "frozen", not to be evaluated, even partially, before the function is applied). We could represent this by simply removing the case $\lambda x.\,E$ from the grammar of contexts – no rule change is needed. Reduction contexts are the right formal representation of many different evaluation strategies.

2.2.4. Formally defining failure

We need one more remark to be able to formally define a class of failures corresponding to the "invalid state" error of general programming languages. If you look at the definition of head reduction $(\rhd_\beta{}^{\Lambda\mathsf{C}(\to,\mathtt{box})})$,

$$(\lambda x.\,t)\;u \rhd_\beta t[u/x] \qquad\qquad \mathtt{unbox}(\mathtt{box}(t)) \rhd_\beta t$$

it seems rather clear that the term-former go by pairs: λ-abstraction and application are related by a reduction rule, and boxing and unboxing are similarly related. A reduction may happen exactly when two related term-formers meet. In both cases there is a term ($\lambda x.\,t$ and $\mathtt{box}(t)$ respectively) that *constructs* some structure (a function, a box), and a term-with-a-hole ($\Box\;u$ and $\mathtt{unbox}(\Box)$ respectively) that *destructs* it. We can formally define a grammar of *constructor* terms and a grammar of *destructor* contexts as follows:

Figure 2.9.: Constructors and destructors of the administrative λ-calculus $\mathsf{\Lambda C}(\to, \mathtt{box})$

$$\begin{array}{rlll} t_{\mathtt{c}}, u_{\mathtt{c}}, r_{\mathtt{c}} ::= & & \text{constructors} & E_{\mathtt{d}}, F_{\mathtt{d}}, G_{\mathtt{d}} ::= \\ & \mid \lambda x.\, t & & \mid \square\, t \\ & \mid \mathtt{box}(t) & & \mid \mathtt{unbox}(\square) \end{array} \qquad \text{destructors}$$

Notice that we have not specified the pairing between constructors and destructors. This is unnecessary, as this information is contained in the head reduction relation: $t_{\mathtt{c}}$ and $E_{\mathtt{d}}$ are paired if and only if their composition can perform a head reduction ($E_{\mathtt{d}}\,[t_{\mathtt{c}}] \rhd_\beta$). We call a *redex* this reducible meeting of a constructor and a destructor.

Conversely, a failing term is a term where a constructor $t_{\mathtt{c}}$ meets a destructor $E_{\mathtt{d}}$ to which it is *not* paired (a *failing redex*) – possibly under some reduction context F. We thus define the set $\mathcal{F}$ of failing terms:

Figure 2.10.: Failures in the administrative λ-calculus $\mathsf{\Lambda C}(\to, \mathtt{box})$

$$\mathcal{F}^{\mathsf{\Lambda C}(\to,\mathtt{box})} \overset{\text{def}}{=} \{F\,[E_{\mathtt{d}}\,[t_{\mathtt{c}}]] \mid E_{\mathtt{d}}\,[t_{\mathtt{c}}] \not\rhd_\beta^{\mathsf{\Lambda C}(\to,\mathtt{box})}\}$$

Notation 2.2.4 $\{a \mid P\}$**.**
The notation $\{a \mid P\}$ is standard in mathematics, it means "the subset of the a such that the statement P is true" – P may mention the variable(s) used in the expression a. It is called a *set comprehension*. For example, $\{(a,b,c) \in \mathbb{N}^3 \mid a^2 + b^2 = c^2\}$ describes the Pythagorean triples.

Remark 2.2.3. A failing term contains a failing redex, but it may still contain other reducible redexes somewhere in the term – this is a slightly different notion from the notion of *stuck* term that is often used to define program errors. When using *stuck* terms (terms that cannot perform any $(\to_\beta)$-reduction), one must distinguish the "good ones", such as $(x\ (\lambda y.\, y))$, from the "bad ones" that contain a failing redex, so it is no simpler than our definition.

For example, $\mathtt{box}(x)\ ((\lambda x.\, x)\ z)$ is a failing term (you cannot apply an argument to a box) which can also reduce to $\mathtt{box}(x)\ z$ – another failing term. In this example, the reducible redex is part of the reduction context around the failing redex. Another example would be $\mathtt{unbox}(\lambda x.\, (\lambda y.\, y)\ z)$, where the reducible redex is inside the failing redex. $*$

The set of failing terms is stable by substitution – if $t \in \mathcal{F}$ then $t[u/x] \in \mathcal{F}$ – but not by reduction: we can have $t \in \mathcal{F} \to_\beta u \notin \mathcal{F}$. For example, $(\lambda x.\, y)\ \mathtt{unbox}(\lambda z.\, z) \to_\beta y$. Failing term is to be understood as "may fail": depending on the choice of reduction to perform, the computation may or may not encounter an invalid operation.

2.2.5. An exercise in administration

You may wonder why the name *administrative* for the extension of the λ-calculus with boxes. The name comes from the existing concept in the programming language theory literature of "administrative variants" of term-formers, and "administrative reductions". In some situations it is convenient to have two different notions of λ-abstraction and of application: the usual one, and the "administrative"one. Consider a language defined as follows:

In addition to the usual abstraction $\lambda x.\, t$ and application $t\ u$, this language has an "administrative variant" of these constructions, $\lambda_{\mathtt{a}} x.\, t$ and $t \cdot_{\mathtt{a}} u$ respectively. It is an independent copy that behaves in the exact same way, but notice that the two copies do not mix: you cannot do a regular application on an administrative $\lambda_{\mathtt{a}}$-abstraction, nor do administratively apply an argument to a regular λ-abstraction.

It should be obvious that in terms of expressive power, this farcical copy of the core mechanism has no benefits: every computation we can express there could already be

Figure 2.11.: Minimal λ-calculus with administrative functions $\Lambda\mathsf{C}(\to, \lambda_\mathsf{a})$

$$
\begin{array}{rll}
t, u, r ::= & & \text{terms} \\
& \mid \ldots & \text{minimal } \lambda\text{-calculus } \Lambda\mathsf{C}(\to) \ (\S 2.1) \\
& \mid \lambda_\mathsf{a} x.\, t & \\
& \mid t \cdot_\mathsf{a} u &
\end{array}
$$

$$
\begin{array}{rll}
E, F, G ::= & & \text{reduction contexts} \\
& \mid \Box & \\
& \mid \lambda x.\, E \mid \lambda_\mathsf{a} x.\, E & \\
& \mid E\ u \mid E \cdot_\mathsf{a} u & \\
& \mid t\ E \mid t \cdot_\mathsf{a} E &
\end{array}
\qquad
\begin{array}{rll}
t_\mathsf{c}, u_\mathsf{c}, r_\mathsf{c} ::= & & \text{constructors} \\
& \mid \lambda x.\, t & \\
& \mid \lambda_\mathsf{a} x.\, t &
\end{array}
$$

$$
\begin{array}{rll}
E_\mathsf{d}, F_\mathsf{d}, G_\mathsf{d} ::= & & \text{destructors} \\
& \mid \Box\ t & \\
& \mid \Box \cdot_\mathsf{a} t &
\end{array}
$$

$$
(\lambda x.\, t)\ u \triangleright_\beta t[u/x] \qquad (\lambda_\mathsf{a} x.\, t) \cdot_\mathsf{a} u \triangleright_\beta t[u/x] \qquad \frac{t \triangleright_\beta t'}{E\,[t] \to_\beta E\,[t']}
$$

expressed without administrative functions. A way to formally demonstrate this is to define a translation, $[\![\_]\!]_\mathsf{a} : \Lambda\mathsf{C}(\to, \lambda_\mathsf{a}) \to \Lambda\mathsf{C}(\to)$, from the calculus with administrative functions to the minimal λ-calculus:

$$
[\![x]\!]_\mathsf{a} \overset{\text{def}}{=} x \qquad
\begin{array}{rcl}
[\![\lambda x.\, t]\!]_\mathsf{a} & \overset{\text{def}}{=} & \lambda x.\, [\![t]\!]_\mathsf{a} \\
[\![\lambda_\mathsf{a} x.\, t]\!]_\mathsf{a} & \overset{\text{def}}{=} & \lambda x.\, [\![t]\!]_\mathsf{a}
\end{array}
\qquad
\begin{array}{rcl}
[\![t\ u]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![t]\!]_\mathsf{a}\ [\![u]\!]_\mathsf{a} \\
[\![t \cdot_\mathsf{a} u]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![t]\!]_\mathsf{a}\ [\![u]\!]_\mathsf{a}
\end{array}
$$

For example, we have $[\![(\lambda_\mathsf{a} t.\, t) \cdot_\mathsf{a} u]\!]_\mathsf{a} =^{\Lambda\mathsf{C}(\to)} (\lambda t.\, t)\ u$. This very simple translation simply "forgets" about the administrative variants, by translating them to the usual abstraction and application.

Forward simulation Our intuition is that "everything the λ-calculus with administrative functions can compute, the translation into the minimal λ-calculus can compute as well". We will prove, more formally, that if two terms with administrative functions are in the full reduction relation, then their translation is in the full reduction relation of the minimal λ-calculus.

We assume (this is easily provable) that the definition of $(\to_\beta{}^{\Lambda\mathsf{C}(\to)})$ we have given for the minimal λ-calculus (before we introduced reduction contexts) is equivalent to one given with reduction contexts, with the "obvious" grammar of reduction contexts:

$$
E, F, G \quad ::=^{\Lambda\mathsf{C}(\to)} \quad \Box \mid \lambda x.\, E \mid E\ u \mid t\ E
$$

A reduction in either calculi is thus characterized by a context E and some head reduction $t \triangleright_\beta t'$. To show that the translation preserve reduction, we will show how to translate contexts, and how to translate head reductions. Only then will we be able to prove that full reductions can be translated.

The translation from the terms of the λ-calculus with administrative functions to the minimal λ-calculus can be extended to a translation of contexts:

$$
[\![\Box]\!]_\mathsf{a} \overset{\text{def}}{=} \Box \qquad
\begin{array}{rcl}
[\![\lambda x.\, E]\!]_\mathsf{a} & \overset{\text{def}}{=} & \lambda x.\, [\![E]\!]_\mathsf{a} \\
[\![\lambda_\mathsf{a} x.\, E]\!]_\mathsf{a} & \overset{\text{def}}{=} & \lambda x.\, [\![E]\!]_\mathsf{a}
\end{array}
\qquad
\begin{array}{rcl}
[\![E\ u]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![E]\!]_\mathsf{a}\ [\![u]\!]_\mathsf{a} \\
[\![E \cdot_\mathsf{a} u]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![E]\!]_\mathsf{a}\ [\![u]\!]_\mathsf{a} \\
[\![t\ F]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![t]\!]_\mathsf{a}\ [\![F]\!]_\mathsf{a} \\
[\![t \cdot_\mathsf{a} F]\!]_\mathsf{a} & \overset{\text{def}}{=} & [\![t]\!]_\mathsf{a}\ [\![F]\!]_\mathsf{a}
\end{array}
$$

The translation of terms and of contexts are compatible, in the sense that the translation of the term $E\,[t]$ is equal to plugging the translation of t in the translation of E.

Lemma 2.2.1 (Translation of context decomposition)**.**
For any term t and context E in the λ-calculus with administrative functions, we have

$$[\![E\,[t]\,]\!]_{\mathsf{a}} = [\![E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]$$

Proof. This is done by induction on the context E. In the base case $E \stackrel{\mathsf{def}}{=} \Box$, we have simply

$$[\![\Box\,[t]\,]\!]_{\mathsf{a}} = [\![t\,]\!]_{\mathsf{a}} = \Box\,[[\![t\,]\!]_{\mathsf{a}}] = [\![\Box\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]$$

We will only do one inductive case as they are all very similar. Suppose we want to prove this property of the context $\lambda_{\mathsf{a}} x.\, E$, assuming it is true for E. That is, our induction hypothesis is $[\![E\,[t]\,]\!]_{\mathsf{a}} = [\![E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]$. We want to prove that $[\![(\lambda_{\mathsf{a}} x.\, E)\,[t]\,]\!]_{\mathsf{a}} = [\![\lambda_{\mathsf{a}} x.\, E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]$. By definition of context plugging we have $(\lambda_{\mathsf{a}} x.\, E)\,[t] = \lambda_{\mathsf{a}} x.\, (E\,[t])$, and thus $[\![(\lambda_{\mathsf{a}} x.\, E)\,[t]\,]\!]_{\mathsf{a}} = [\![\lambda_{\mathsf{a}} x.\, E\,[t]\,]\!]_{\mathsf{a}} = \lambda x.\, [\![E\,[t]\,]\!]_{\mathsf{a}}$. Then, by induction hypothesis, we have $\lambda x.\, [\![E\,[t]\,]\!]_{\mathsf{a}} = \lambda x.\, ([\![E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}])$. We can finally conclude with $\lambda x.\, ([\![E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]) = (\lambda x.\, [\![E\,]\!]_{\mathsf{a}})\,[[\![t\,]\!]_{\mathsf{a}}] = [\![\lambda_{\mathsf{a}} x.\, E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]$.

Notation 2.2.5.
Another notation for the reasoning in the case above is the following presentation as a series of equalities (with justifications for the non-immediate steps):

$$\begin{array}{rcl}
[\![(\lambda_{\mathsf{a}} x.\, E)\,[t]\,]\!]_{\mathsf{a}} & = & \\
[\![\lambda_{\mathsf{a}} x.\, (E\,[t])\,]\!]_{\mathsf{a}} & = & \\
\lambda x.\, [\![E\,[t]\,]\!]_{\mathsf{a}} & = & \text{(by induction hypothesis)} \\
\lambda x.\, ([\![E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}]) & = & \\
(\lambda x.\, [\![E\,]\!]_{\mathsf{a}})\,[[\![t\,]\!]_{\mathsf{a}}] & = & \\
[\![\lambda_{\mathsf{a}} x.\, E\,]\!]_{\mathsf{a}}\,[[\![t\,]\!]_{\mathsf{a}}] & &
\end{array}$$

□

Lemma 2.2.2 (Translation of head reductions)**.**
The translation preserves head reductions: if $t \rhd_\beta^{\Lambda\mathsf{C}(\to,\lambda_{\mathsf{a}})} t'$, then $[\![t\,]\!]_{\mathsf{a}} \rhd_\beta^{\Lambda\mathsf{C}(\to)} [\![t'\,]\!]_{\mathsf{a}}$.

Proof. Translation of $(\lambda x.\, t)\ u \rhd_\beta^{\Lambda\mathsf{C}(\to,\lambda_{\mathsf{a}})} t[u/x]$:

$$\begin{array}{rcl}
[\![(\lambda x.\, t)\ u\,]\!]_{\mathsf{a}} & = & \\
(\lambda x.\, [\![t\,]\!]_{\mathsf{a}})\ [\![u\,]\!]_{\mathsf{a}} & \rhd_\beta^{\Lambda\mathsf{C}(\to)} & \\
[\![t\,]\!]_{\mathsf{a}}[[\![u\,]\!]_{\mathsf{a}}/x] & = & \text{(property of substitution)} \\
[\![t[u/x]\,]\!]_{\mathsf{a}} & &
\end{array}$$

The "property of substitution" needed for the last equality is the fact that the translation commutes with substitution: $[\![t\,]\!]_{\mathsf{a}}[[\![u\,]\!]_{\mathsf{a}}/x] = [\![t[u/x]\,]\!]_{\mathsf{a}}$ for all t, x, u. This is immediately proved by induction on t – much like we proved the translation of context decomposition.

The other case of head reduction, $(\lambda_{\mathsf{a}} x.\, t) \cdot_{\mathsf{a}} u \rhd_\beta^{\Lambda\mathsf{C}(\to,\lambda_{\mathsf{a}})} t[u/x]$, is almost identical:

$$\begin{array}{rcl}
[\![(\lambda_{\mathsf{a}} x.\, t) \cdot_{\mathsf{a}} u\,]\!]_{\mathsf{a}} & = & \\
(\lambda x.\, [\![t\,]\!]_{\mathsf{a}})\ [\![u\,]\!]_{\mathsf{a}} & \rhd_\beta^{\Lambda\mathsf{C}(\to)} & \\
[\![t\,]\!]_{\mathsf{a}}[[\![u\,]\!]_{\mathsf{a}}/x] & = & \text{(property of substitution)} \\
[\![t[u/x]\,]\!]_{\mathsf{a}} & &
\end{array}$$

□

We can finally prove our main result on this translation. This form of translation of a reduction relation is called a *forward simulation* result: we show that everything the initial programs do, the translation can do as well, it "simulates" (imitates) the initial system.

Theorem 2.2.3 (Forward simulation)**.**
If $t \to_\beta^{\Lambda\mathsf{C}(\to,\lambda_\mathsf{a})} t'$*, then* $[\![t]\!]_\mathsf{a} \to_\beta^{\Lambda\mathsf{C}(\to)} [\![t']\!]_\mathsf{a}$.

Proof. If $t \to_\beta^{\Lambda\mathsf{C}(\to,\lambda_\mathsf{a})} t'$, then t must be of the form $E\,[u]$, and t' of the form $E\,[u']$, with

$$\frac{u \rhd_\beta^{\Lambda\mathsf{C}(\to,\lambda_\mathsf{a})} u'}{E\,[u] \to_\beta E\,[u']}$$

By Lemma 2.2.2 (Translation of head reductions) we have that $[\![u]\!]_\mathsf{a} \rhd_\beta^{\Lambda\mathsf{C}(\to)} [\![u']\!]_\mathsf{a}$.

By Lemma 2.2.1 (Translation of context decomposition) we have that $[\![E\,[u]]\!]_\mathsf{a} = [\![E]\!]_\mathsf{a}\,[[\![u]\!]_\mathsf{a}]$, and similarly for u'. We can thus build the following proof of the desired goal $[\![E\,[u]]\!]_\mathsf{a} \to_\beta^{\Lambda\mathsf{C}(\to)} [\![E\,[u']]\!]_\mathsf{a}$:

$$\frac{[\![u]\!]_\mathsf{a} \rhd_\beta^{\Lambda\mathsf{C}(\to)} [\![u']\!]_\mathsf{a}}{[\![E]\!]_\mathsf{a}\,[[\![u]\!]_\mathsf{a}] \to_\beta^{\Lambda\mathsf{C}(\to)} [\![E]\!]_\mathsf{a}\,[[\![u']\!]_\mathsf{a}]}$$

□

Failure to simulate failure Our forward simulation result tells us that adding administrative functions does not add computational power, as any computation possible in this extended calculus could be computed in the minimal calculus. A property that we do *not* have, however, is that a failing term with administrative functions is translated into a failing term of the minimal λ-calculus. See the following counter-example:

$$[\![(\lambda x.\, x) \cdot_\mathsf{a} y]\!]_\mathsf{a} \quad = \quad (\lambda x.\, x)\; y$$

The term $(\lambda x.\, x) \cdot_\mathsf{a} y$ attempts an administrative application on a non-administrative λ-abstraction: a destructor meets an unrelated constructor, it is a failing term in $\mathcal{F}^{\Lambda\mathsf{C}(\to,\lambda_\mathsf{a})}$. Its translation, however, is *not* a failing term in $\mathcal{F}^{\Lambda\mathsf{C}(\to)}$, and it head-reduces to y.

This is problematic if we try to lift results about correctness of programs. Suppose I have proved that a certain class of terms of the minimal λ-calculus $\Lambda\mathsf{C}(\to)$ is "safe", that it never reduces to a failing term. Then I define a class of terms in $\Lambda\mathsf{C}(\to,\lambda_\mathsf{a})$, and show that they always get translated to safe terms. Intuitively, I would expect that those administrative terms are also safe, they never reduce to failing term. But this is wrong: a term that gets translated to a safe term could very well be a failing term itself, as in the example above. I cannot reuse results about correctness of $\Lambda\mathsf{C}(\to)$ programs to reason about programs in my extended calculus.

To solve this problem, we will change our translation. Instead of translating terms of the λ-calculus with administrative functions into terms of the minimal λ-calculus $\Lambda\mathsf{C}(\to)$, we will translate them into terms of the *administrative* λ-calculus $\Lambda\mathsf{C}(\to,\mathtt{box})$, as follows:

$$[\![x]\!]_\mathsf{a}^\mathtt{box} \stackrel{\text{def}}{=} x \qquad \begin{array}{lcl} [\![\lambda x.\, t]\!]_\mathsf{a}^\mathtt{box} & \stackrel{\text{def}}{=} & \lambda x.\, [\![t]\!]_\mathsf{a}^\mathtt{box} \\ [\![\lambda_\mathsf{a} x.\, t]\!]_\mathsf{a}^\mathtt{box} & \stackrel{\text{def}}{=} & \mathtt{box}(\lambda x.\, [\![t]\!]_\mathsf{a}^\mathtt{box}) \end{array}$$

$$\begin{array}{lcl} [\![t\; u]\!]_\mathsf{a}^\mathtt{box} & \stackrel{\text{def}}{=} & [\![t]\!]_\mathsf{a}^\mathtt{box}\; [\![u]\!]_\mathsf{a}^\mathtt{box} \\ [\![t \cdot_\mathsf{a} u]\!]_\mathsf{a}^\mathtt{box} & \stackrel{\text{def}}{=} & \mathtt{unbox}([\![t]\!]_\mathsf{a}^\mathtt{box})\; [\![u]\!]_\mathsf{a}^\mathtt{box} \end{array}$$

Our previous example $(\lambda x.\, x) \cdot_\mathsf{a} y$ now gets translated $\mathtt{unbox}(\lambda x.\, x)\; y$, a failing term. The additional level of boxing "protects" the translation of the administrative abstractions and applications from undesired interactions with the regular abstractions and applications.

Lemma 2.2.4 (Translation of context decomposition)**.**
$[\![E\,[t]\,]\!]_{\mathsf{a}}^{\mathtt{box}} = [\![E\,]\!]_{\mathsf{a}}^{\mathtt{box}}\,[[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}]$

Proof. As with the previous translation. For example,

$$\begin{array}{rcll}
[\![(\lambda_{\mathsf{a}} x.\, E)\,[t]\,]\!]_{\mathsf{a}}^{\mathtt{box}} & = & & \\
[\![\lambda_{\mathsf{a}} x.\, (E\,[t])\,]\!]_{\mathsf{a}}^{\mathtt{box}} & = & & \\
\mathtt{box}(\lambda x.\, [\![E\,[t]\,]\!]_{\mathsf{a}}^{\mathtt{box}}) & = & \text{(by induction hypothesis)} & \\
\mathtt{box}(\lambda x.\, ([\![E\,]\!]_{\mathsf{a}}^{\mathtt{box}}\,[[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}])) & = & & \\
(\mathtt{box}(\lambda x.\, [\![E\,]\!]_{\mathsf{a}}^{\mathtt{box}}))\,[[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}] & = & & \\
[\![\lambda_{\mathsf{a}} x.\, E\,]\!]_{\mathsf{a}}^{\mathtt{box}}\,[[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}] & & &
\end{array}$$

□

Lemma 2.2.5 (Translation of head reductions)**.**
If $t \triangleright_\beta t'$*, then* $[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}} \to_\beta^* [\![t'\,]\!]_{\mathsf{a}}^{\mathtt{box}}$ *in at most two reduction steps.*

Proof. As with the previous translation. The case which uses more than one step (in fact exactly two steps) is the redex for administrative functions

$$\begin{array}{rcl}
[\![(\lambda_{\mathsf{a}} x.\, t) \cdot_{\mathsf{a}} u\,]\!]_{\mathsf{a}}^{\mathtt{box}} & = & \\
\mathtt{unbox}(\mathtt{box}(\lambda x.\, [\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}))\;[\![u\,]\!]_{\mathsf{a}}^{\mathtt{box}} & \to_\beta & \\
\lambda x.\, [\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}\;[\![u\,]\!]_{\mathsf{a}}^{\mathtt{box}} & \triangleright_\beta & \\
[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}}[[\![u\,]\!]_{\mathsf{a}}^{\mathtt{box}}/x] & = & \\
[\![t[u/x]\,]\!]_{\mathsf{a}}^{\mathtt{box}} & &
\end{array}$$

□

Lemma 2.2.6 (Forward simulation)**.**
If $t \to_\beta^{\Lambda\mathsf{C}(\to,\lambda_{\mathsf{a}})} t'$*, then* $[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}} \to_\beta^* [\![t'\,]\!]_{\mathsf{a}}^{\mathtt{box}}$*, and this translated reduction takes at most two* $(\to_\beta^{\Lambda\mathsf{C}(\to,\mathtt{box})})$*-steps.*

Proof. The proof is exactly as for the previous translation. □

The interest of the translation is in the preservation of failures, which was not the case for our previous translation.

Theorem 2.2.7 (Failure simulation)**.**
If $t \in \mathcal{F}^{\Lambda\mathsf{C}(\to,\lambda_{\mathsf{a}})}$*, then* $[\![t\,]\!]_{\mathsf{a}}^{\mathtt{box}} \in \mathcal{F}^{\Lambda\mathsf{C}(\to,\mathtt{box})}$.

Proof. In both calculi the failing terms are formed by a failing constructor-destructor pair $E_{\mathsf{d}}\,[t_{\mathsf{c}}]$ plugged inside an arbitrary reduction context F:

$$\mathcal{F}^{\Lambda\mathsf{C}} \quad \stackrel{\mathsf{def}}{=} \quad \{F\,[E_{\mathsf{d}}\,[t_{\mathsf{c}}]] \mid E_{\mathsf{d}}\,[t_{\mathsf{c}}] \not\triangleright_\beta^{\Lambda\mathsf{C}}\}$$

We have $[\![F\,[E_{\mathsf{d}}\,[t_{\mathsf{c}}]]\,]\!]_{\mathsf{a}}^{\mathtt{box}} = [\![F\,]\!]_{\mathsf{a}}^{\mathtt{box}}\,[[\![E_{\mathsf{d}}\,[t_{\mathsf{c}}]\,]\!]_{\mathsf{a}}^{\mathtt{box}}]$, so we only need to check that failing destructor-constructor pairs $E_{\mathsf{d}}\,[t_{\mathsf{c}}]$ translate to failing terms. By case analysis:

1. $(\lambda x.\, t) \cdot_{\mathsf{a}} u$ translates to $\mathtt{unbox}(\lambda x.\, t)\; u$, which is a failing term
2. $(\lambda_{\mathsf{a}} x.\, t)\; u$ translates to $\mathtt{box}(\lambda x.\, t)\; u$ which is also a failing term.

□

This concludes our study of the λ-calculus with administrative arrows $\Lambda\mathsf{C}(\to, \lambda_{\mathsf{a}})$. This is one example of a λ-calculus with extra construction that is expressible in the minimal λ-calculus $\Lambda\mathsf{C}(\to)$, but needs at least the administrative calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$ to properly translate failures. There are many such calculi, which can all be translated in $\Lambda\mathsf{C}(\to, \mathtt{box})$. In a way, $\Lambda\mathsf{C}(\to, \mathtt{box})$ captures the essence of failure – a kind of failure that is easier to manipulate and more representative of real-world languages than silent non-termination. In Section 2.3 ((Simply) Typed λ-calculi), we will investigate formal ways to reason about correctness, that is the absence of failures.

Remark 2.2.4. We can, in fact, refine Lemma 2.2.6 (Forward simulation) by a more fine-grained analysis of the reduction steps involved. The idea is that reducing a λ-abstraction is a potentially costly process (we have to perform a substitution, etc.), this is where the real computation happens, while reducing a $\mathtt{box}$ is a trivial step that only removes marks on terms. We can define $(\triangleright_{\beta(\lambda)})$ as the subrelation of $(\triangleright_\beta)$ that only involves λ-reductions, and $(\triangleright_{\beta(\mathtt{box})})$ as the subrelation only involving boxes:

$$(\lambda x.\, t)\ u \triangleright_{\beta(\lambda)} t[u/x] \qquad \mathtt{unbox}(\mathtt{box}(t)) \triangleright_{\beta(\mathtt{box})} t \qquad (\triangleright_\beta) = (\triangleright_{\beta(\lambda)}) \cup (\triangleright_{\beta(\mathtt{box})})$$

We can then easily prove the following result:

Lemma 2.2.8.
If $t \triangleright_\beta^{\Lambda\mathsf{C}(\to,\lambda_\mathtt{a})} t'$, *then*

$$[\![t]\!]_\mathtt{a}^\mathtt{box} \to^*_{\beta(\mathtt{box})} \triangleright_{\beta(\lambda)} \to^*_{\beta(\mathtt{box})} [\![t']\!]_\mathtt{a}^\mathtt{box}$$

This tells us that each translated reduction contains exactly one λ-reduction step (actual computations), and an irrelevant number of administrative steps; zero or one steps, in fact, but this weaker result generalizes better to other calculi.

This refined relation $(\to^*_{\beta(\mathtt{box})} \triangleright_{\beta(\lambda)} \to^*_{\beta(\mathtt{box})})$ has a stronger property that $(\to^*_\beta)$ does not have, which is that whenever a translation $[\![t]\!]_\mathtt{a}^\mathtt{box}$ reduces to another translation $[\![u]\!]_\mathtt{a}^\mathtt{box}$ by this relation, then the source term t also reduces to the source term u in the source system. This would allow us to establish a bisimulation result – two simulations that are inverse from each other. ∗

Credits I learned this elegant characterization of failure in an untyped setting from Julien Crétin's PhD thesis [Crétin, 2014]. I had the pleasure of discussing with Julien as we were both students working with Didier Rémy during the same period. The (minor) idea of isolating a boxing construct (as a dynamic counterpart of abstraction sealing) to use in simulating translations is a side-effect of some more recent work by Didier and myself [Scherer and Rémy, 2015] which was motivated and inspired by Julien's thesis.

2.3. (Simply) Typed λ-calculi

2.3.1. Reasoning on programs: type systems for modular verification information

There is no free lunch, and in my field a free lunch would be a programming language that lets us easily express *any* desired program without ever letting us insert bugs.

Remark 2.3.1. A (software) bug is a mismatch between the behavior intended by the authors of a program, and the behavior actually specified by the program source code as they have written it. (People wrongly blame the machines for bugs. Machines do what they are told.) It is natural that programming language design, working on the interface between author intentions and their practical means of expressions, play a key role in the development of practices and tooling to reduce bugs. The mathematics-inspired formal study of programming languages, as we practice it in this thesis, is only one tool among many to help us improve programming languages: there are others engineering, psychological and sociological aspects to be considered. ∗

In the previous section we have seen examples of the immense expressive power of the minimal λ-calculus. With this power to express what we want comes the power to make mistakes, such as silent non-termination and failing terms. There are fundamental theorems, that can be transferred between calculability theory, logics, and programming languages, that seem to indicate that allowing these mistakes is unavoidable – forbidding them endangers the expressiveness of the calculus.

For example, we can prove that there is no algorithm that can tell for all programs of a Turing-complete programming language (for example the minimal or administrative λ-calculus) whether they eventually reduce to an irreducible value or not. There is no algorithm taking the description of a μ-recursive function and an input, able to always decide if the function is defined on this input. The intuition for these results is that there is no shortcut: informally, the only way to tell is to run the program, to start computing the function, and this may never stop.

In fact, no non-trivial property can be decided for all the programs of such a language, this is the Rice-Myhill-Shapiro theorem from the 1950s. In logic it is known since Gödel's theorems in the 1930s that no consistent logic (rich enough to express computation) can prove all true statements. There is a direct relation between consistency (for a logic) and allowing silently non-terminating programs (for a programming language).

This means that, to reliably reason on programs, to check that they respect some property such as termination, *we need extra help*, some additional information that let us verify the property of interest (for example, "the program does not crash and does not send privileged, security-sensitive information to an untrusted third-party"). A priori, the structure of this information could strongly depend on the property being checked.

In the limit case, this information could be a mathematical proof of the property of interest, accompanying the program. This is maximally expressive, but often impractical: few programmers are willing to write proofs, fewer to check them properly.

We are interested in two additional properties of such verification information. First, it should be expressed in a syntactic form that a computer can manipulate (not just humans); we generally expect that there is an algorithm that can take any program and its verification information, and checks that the information correctly justifies the property of interest. Some verification systems lift that restriction (for example most presentations of extensional type theory), but they can be seen as an erasure of a more explicit system that provides the extra information needed for decidability.

Second, we expect the verification information to be *modular*, in the sense that information for a program can be derived from information on its sub-parts (without inspecting the syntactic structure of those sub-parts). There are practical benefits to modularity. It means that the checking process can scale to very large programs; and indeed, large computer programs have reached a scale comparable or larger than the most complex human-built physical structures, with tens of millions of lines of code and as little repeated patterns as possible – the Eiffel tower, on the other hand, has many repeated patterns, and its design could be fully described by a comparatively smaller computer program. Yet the main benefit is conceptual: designing a modular verification structure forces us to understand the property of interest in depth.

There are non-modular program analyses, and they have their uses. For programs for which no verification information is provided, reconstructing it is undecidable in the general case, but we may design a best-effort algorithm that answers either "yes, here is the verification information", "no, the program does not satisfy the property", or "I don't know". For a given property of interest which has both modular and non-modular verification structures, it may be easier (or even faster) to automatically reconstruct the non-modular one – being less powerful, it is easier to describe. In term of *language design*, however, I think that modularity is key: if we are to co-design a programming language and a specific verification structure together, it should be modular, with non-modular analyses delegated to external tools.

Definition 2.3.1 Type system**.**
We call *type system* a verification structure for a given property which is syntactic and which is modular with respect to program structure.

Remark 2.3.2. In practice many things are called *type systems* by theoreticians and practitioners that do not quite fit this framework. For example, in many programming languages typing information is only available during the program execution, in a partial

way (these are called *dynamically typed* languages); they do play the role of a verification structure, in the sense that type errors abort the program before other, worse kind of failures happen, but it is unclear whether the typing information is computed "modularly" with respect to the program text itself.

On top of this dynamic verification system, some languages add a static system that reasons on the program test, but they make simplifying assumptions that make its verification incomplete (even for the class of errors it was designed to detect). This trade-off can give a simpler system that is easier for end-users to understand.

Finally, modularity can be hard to achieve and first attempts at static verification of a particular aspect often resort to less-modular approaches, even inside a language that is generally typed in a modular way. For example, verification of contractivity in the OCaml module system (which should allow to define fixpoints across module boundaries), or positivity of Coq inductive definitions (which should allow to split mutually recursive inductive definitions across module boundaries) are non-modular, in the sense that those cannot be expressed in the interface language describing the statically deducible information of program fragments/modules. ∗

2.3.2. A simple type system for the administrative λ-calculus

We will now present a simple type system for the administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$, whose purpose is to ensure that the reduction of verified programs never results in failing terms.

We can start building our system from very simple examples. The term $t\ u$ reduces to a failing term if t reduces to a box; as long as it remains a variable or a λ-abstraction, the application does not cause a failure – but other parts of the term may lead to failures. In essence, we want to verify that t "is a function". Correspondingly, $\mathtt{unbox}(t)$ does not cause failure if t "is a box". Let us call this information the *shape* of a term.

In fact, we need to know a bit more than that: if the shape of t is "box", then what is the shape of $\mathtt{unbox}(t)$? We do not know, it may be either a function or a box. We are not able to determine the shape of $\mathtt{unbox}(t)$ from the shape of its subpart t, so the shape system as suggested does not satisfy our modularity requirement. We should demand some information on the shape inside the box: our types for box will be of the form $\mathtt{box}(A)$ (rather than just $\mathtt{box}$), where A is some information about the shape of what is inside the box.

Similarly for functions, knowing that t is "a function" is not enough, because it does not let us deduce anything about $t\ u$, which may be either a function (if t is $\lambda x.\, \lambda y.\, y$ for example) or a box ($\lambda x.\, \mathtt{box}(x)$ for example). Besides the shape of the output of the function, we also need some information on the shape *expected* for the input of the function. Our types for functions will thus be of the form $A \to B$, where A is the type of the input of the function, and B the type of its output.

Finally, there are parts of the program about which we have little information, and on which we do not need extra information. For example if the whole program to verify is $\lambda x.\, x$ (this program is safe, it will never reduce to a failing term), we do not really need to know the shape of x, as it is neither used as a function or a box. We may assume that it has a "base type", for example X, on which we do not know or need anything. By assuming nothing on X, we have the guarantee that if we decide to assume a more informative shape for x later, then the program will remain correctly verified.

In Figure 2.12 we present our type system for $\Lambda\mathsf{C}(\to, \mathtt{box})$. It is described as (you guess) a system of inference rules, whose judgments are of the form $\Gamma \vdash t : A$. The term t is the program that is given a type, the type A is the given type, and Γ is a *typing environment* that records the type we assumed for each free variable of the program: it is a mapping from variables to types.

We claimed that type systems should be *compositional*. This can be seen by the fact that, to build a typing judgment $\Gamma \vdash t : A$ on a particular term t, we only require to know

Figure 2.12.: Typed administrative λ-calculus $\Lambda\mathsf{C}(\to, \mathtt{box})$

$$\textsc{talc-var}\ \frac{}{\Gamma, x : A \vdash x : A}$$

$$\textsc{talc-lam}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \textsc{talc-app}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\textsc{talc-box}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{box}(t) : \mathtt{box}(A)} \qquad \textsc{talc-unbox}\ \frac{\Gamma \vdash t : \mathtt{box}(A)}{\Gamma \vdash \mathtt{unbox}(t) : A}$$

typing judgments on its direct subterms, and do not otherwise inspect them. The *typing* pair (Γ, A) is the compositional verification invariant.

Remark 2.3.3 (Comma notation in typing environments)**.** When we introduced logical contexts that are sets of variable in Section 1.2.2 (Formally defining the proofs), we proposed to interpret the comma notation Γ, A as *non*-disjoint union (A may already be present in Γ). For typing environments that are mappings from variables to typing environments, however, the comma notation $\Gamma, x : A$ denotes *disjoint* union: x must not already occur in Γ.

Because we use the notation $\Gamma, x : A$ for variables x introduced by term binders – above, in the TALC-LAM rule, for the λ-bound variable – we can α-rename bound variables to ensure this condition is respected.

Of course, A may already be present in Γ under a different names. This notation is thus not inconsistent with its variable-less counterpart Γ, A: the set of types present in the typing environment formed by disjoint union of Γ and $\{x : A\}$ is the non-disjoint union of the set of types of Γ and of $\{A\}$. ∗

To validate the fact that our rules correctly verify what they were designed to verify, we need to establish the following theorem:

Theorem 2.3.1 (Soundness of the $\Lambda\mathsf{C}(\to, \mathtt{box})$ type system)**.**
If t is well-typed for $\Lambda\mathsf{C}(\to, \mathtt{box})$, then

$$\forall u, \quad t \to_\beta^* u \implies u \notin \mathcal{F}^{\Lambda\mathsf{C}(\to, \mathtt{box})}$$

There are many techniques to prove soundness, but the classic approach is to do a purely syntactic proof by combining *subject reduction* and *progress*. This was introduced by the same article [Wright and Felleisen, 1994] that proposed to define reduction by decomposing terms into reduction contexts and head redexes.

Lemma 2.3.2 (Substitution in $\Lambda\mathsf{C}(\to, \mathtt{box})$ preserves typing)**.**
If $\Gamma, x : A \vdash t : B$ and $\Gamma \vdash u : A$, then $\Gamma \vdash t[u/x] : B$. In other words, the following rule is admissible:

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : B}\ \textsc{subst}$$

Proof. By induction on the derivation of $\Gamma, x : A \vdash t : B$, replacing any use of the variable x by the complete derivation $\Gamma \vdash u : A$. For example, in the application case we have

$$\frac{\Gamma, x : A \vdash t : A \to B \qquad \Gamma, x : A \vdash t' : A}{\Gamma, x : A \vdash t\ t' : A}$$

and we build the partial derivation

$$\frac{\Gamma \vdash t[u/x] : A \to B \qquad \Gamma \vdash t'[u/x] : A}{\Gamma \vdash t[u/x]\ t'[u/x] : B}$$

by induction on the premises: $\Gamma, x : A \vdash t : A \to B$ can be turned into a full derivation of $\Gamma \vdash t[u/x] : A \to B$ by induction hypothesis, and similarly for $t' : A$. Finally, it suffices to remark that $t[u/x]\ t'[u/x]$ is equal to $(t\ t')[u/x]$ by definition to conclude.

In the variable case, we transform the full derivation

$$\frac{}{\Gamma, x : A \vdash x : A}$$

into the full derivation of the judgment $\Gamma \vdash u : A$ we assumed as hypothesis of this lemma. (If the variable is different from x, the derivation in unchanged.)

In the λ-abstraction case, we have

$$\frac{\Gamma, x : A, y : B \vdash t : C}{\Gamma, x : A \vdash \lambda y.\, t : B \to C}$$

which we transform by induction into

$$\frac{\Gamma, y : B \vdash t[u/t] : C}{\Gamma \vdash \lambda y.\, t[u/x] : B \to C}$$

Note that the bound variable's type B may or may not be equal to A, but that there is no ambiguity on which variable we should replace – this is not a non-deterministic definition as substitution in proof derivations (Theorem 1.3.2 (Substitution for $\mathsf{PIL}(\to, \times, 1, +, 0)$)).

Finally, the (un)boxing steps are directly solved by induction:

$$\frac{\Gamma \vdash t[u/x] : A}{\Gamma \vdash \mathtt{box}(t[u/x]) : \mathtt{box}(A)} \qquad \frac{\Gamma \vdash t[u/x] : \mathtt{box}(A)}{\Gamma \vdash \mathtt{unbox}(t[u/x]) : A}$$

□

Theorem 2.3.3 (Subject reduction for $\Lambda\mathsf{C}(\to, \mathtt{box})$)**.**
Reduction in $\Lambda\mathsf{C}(\to, \mathtt{box})$ *preserves typing: if* $\Gamma \vdash t : A$ *and* $t \to_\beta u$*, then* $\Gamma \vdash u : A$*.*

Proof. The reduction relation $t \to_\beta t'$ means that some subterm occurrence u of t is replaced in t' by a term u' in the head reduction relation, $u \triangleright_\beta u'$. Because typing is compositional – the typing of a term depends only on the typing judgments of its subterms, not their syntax – it suffices to prove that u' has the same typing judgment than u.

In other words, it suffices to prove that if $\Gamma \vdash t : A$ and $t \triangleright_\beta t'$, then $\Gamma \vdash t' : A$. We do this by case analysis on $t \triangleright_\beta t'$: from a typing derivation for t, we build one for t'.

$$\frac{\dfrac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{box}(t) : \mathtt{box}(A)}}{\Gamma \vdash \mathtt{unbox}(\mathtt{box}(t)) : A} \quad \triangleright_\beta \quad \Gamma \vdash t : A$$

$$\frac{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.\, t)\ u : B} \quad \triangleright_\beta \quad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : B}\ \text{SUBST}$$

□

Theorem 2.3.4 (Progress for $\Lambda\mathsf{C}(\to, \mathtt{box})$)**.**
Failing $\Lambda\mathsf{C}(\to, \mathtt{box})$ *terms are not well-typed.*

Proof. Immediate by inspection of failing constructor/destructor pairs, which are never well-typed. □

Combining these two theorems immediately gives soundness.

Proof (Theorem 2.3.1 (Soundness of the $\Lambda\mathsf{C}(\to, \mathtt{box})$ type system))**.** If we have $\Gamma \vdash t : A$ and $t \to_\beta^* u$, we can prove by induction on the reduction chain that $\Gamma \vdash u : A$ by Theorem 2.3.3 (Subject reduction for $\Lambda\mathsf{C}(\to, \mathtt{box})$). Then by Theorem 2.3.4 (Progress for $\Lambda\mathsf{C}(\to, \mathtt{box})$) we know that $u \notin \mathcal{F}$. □

2.3.3. Equivalence of $\Lambda\mathsf{C}(\to, \mathtt{box})$ terms

A first attempt at defining program equivalence would to say that two terms t, u are equal if one can be reached from the other by a series of β-reduction or β-expansion steps, a relation we defined as $(\approx_\beta)$ in Notation 2.1.3 (Equivalence closure).

This quickly shows its limit. For example, consider the two following programs of type $A \to (B \to C)$: $\mathtt{id}_1 \stackrel{\mathsf{def}}{=} \lambda f.\, f$ and $\mathtt{id}_2 \stackrel{\mathsf{def}}{=} \lambda f.\, \lambda x.\, f\ x$. One cannot be reached from the other by performing β-reductions, but we argue that they should be considered equal, as for any choice of arguments $t : A, u : B$, we have $\mathtt{id}_1\ t\ u \approx_\beta t\ u \approx_\beta \mathtt{id}_2\ t\ u$.

To equate $\mathtt{id}_1$ and $\mathtt{id}_2$, we thus add the following "equality principle", which is traditionally called an η-equality (η is a "long e" Greek letter pronounced "eta"):

$$(t : A \to B) \approx_\eta \lambda x.\, t\ x$$

Remark that this is exactly the term-level counterpart of the "expansion principle" for implication, presented in Section 1.3.3 (Local tests: reduction and expansion). This suggests that we should add another η-principle for the other connective in our type system, namely $\mathtt{box}(A)$:

$$\Gamma \vdash t : \mathtt{box}(A) \qquad \triangleright_\eta \qquad \frac{\Gamma \vdash t : \mathtt{box}(A)}{\Gamma \vdash \mathtt{unbox}(t) : A}{\Gamma \vdash \mathtt{box}(\mathtt{unbox}(t)) : \mathtt{box}(A)}$$

This expansion principle is useful, as without it we could not prove that the function that unboxes then re-boxes is the identity: we have $\lambda x.\, \mathtt{box}(\mathtt{unbox}(x)) \approx_\eta \lambda x.\, x$ at any type $\mathtt{box}(A) \to \mathtt{box}(A)$.

To summarize, our $\beta\eta$-equality is the smallest equivalence relation on well-typed terms $(\approx_{\beta\eta})$ that contains the congruence closure $(\to_\beta)$, $(\to_\eta)$ of the following β-reduction and η-expansion relations:

$$(\lambda x.\, t)\ u \triangleright_\beta u[t/x] \qquad (t : A \to B) \triangleright_\eta \lambda x.\, t\ x$$

$$\mathtt{unbox}(\mathtt{box}(t)) \triangleright_\beta t \qquad (t : \mathtt{box}(A)) \triangleright_\eta \mathtt{box}(\mathtt{unbox}(t))$$

3. Curry-Howard of reduction and equivalence

3.1. The Curry-Howard correspondence

We have seen the three ingredients, namely logics, programs, and type systems, that are necessary to present the Curry-Howard correspondence between proofs and programs. This correspondence holds between pairs of a logic (given as a system of inference rules) and a typed programming language whose programs are terminating. When it holds, the *proofs* of the logic correspond to the *terms* of the programming language; in particular, the proposition A is provable if the type A is inhabited by a program. Furthermore, reduction of programs and proofs correspond, in the sense that a proof is reducible if and only if the corresponding program is also reducible.

3.1.1. The full simply-typed λ-calculus $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

Let us present in Figure 3.1 the typed λ-calculus $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$, whose programs correspond to the proofs of the proof system $\mathsf{PIL}(\to, \times, 1, +, 0)$ we presented in Figure 1.2 – the term grammar is summarized in Figure 3.2.

Figure 3.1.: Full simply-typed lambda-calculus $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

$$\textsc{stlc-var}\ \frac{}{\Gamma, x : A \vdash x : A}$$

$$\textsc{stlc-lam}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \textsc{stlc-app}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\textsc{stlc-pair}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \times B} \qquad \textsc{stlc-proj}\ \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i\ t : A_i}$$

$$\textsc{stlc-inj}\ \frac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2} \qquad \textsc{stlc-case}\ \frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash u_1 : C \qquad \Gamma, x_2 : A_2 \vdash u_2 : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array} : C}$$

$$\textsc{stlc-trivial}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{stlc-absurd}\ \frac{\Gamma \vdash t : 0}{\Gamma \vdash \texttt{absurd}(t) : A}$$

We established our notations in Section 2.3.2 (A simple type system for the administrative λ-calculus). In a typing environment Γ is a mapping from term variables to types, and we assume, modulo α-equivalence, that variables bound in terms are not already present in the environment.

In the typing rules, we use colors to separate terms from types. This should allow the reader to more easily phase terms out of typing derivations, to see what each rule means in terms of typing only. This erasing reveals that the rules are in one-to-one correspondence

Figure 3.2.: Term grammar for $\mathsf{PIL}(\to, \times, 1, +, 0)$

$$
\begin{array}{lll}
t, u, r ::= & & \text{terms} \\
\quad \mid \ldots & & \text{minimal } \lambda\text{-calculus } \mathsf{\Lambda C}(\to)\ (\S 2.1) \\
\quad \mid (t, u) & & \text{pair} \\
\quad \mid \pi_i\ t & & \text{projection} \\
\quad \mid \sigma_i\ t & & \text{injection} \\
\quad \mid \texttt{match}\ t\ \texttt{with} & \begin{array}{l} \mid\ \sigma_1\ x_1 \to u_1 \\ \mid\ \sigma_2\ x_2 \to u_2 \end{array} & \text{case split} \\
\quad \mid () & & \text{trivial} \\
\quad \mid \texttt{absurd}(t) & & \text{absurd}
\end{array}
$$

with the rules of propositional intuitionistic logic $\mathsf{PIL}(\to, \times, 1, +, 0)$ in natural deduction style (Figure 1.2).

In particular, conjunction in logic corresponds to the Cartesian product type $A \times B$, whose inhabitants are pairs (t, u) of a term at type A and a term at type B (STLC-PAIR); projections $\pi_i\ t$ (for $i \in \{1, 2\}$) allow to recover either the first or second member of the pair (STLC-PROJ).

Disjunction in logic corresponds to the disjoint union type $A + B$, which is a simplified form of the "algebraic" or "sum" types used in functional programming: a value of type $A + B$ holds either a value of A or a value of B, and there is a tag $\sigma_i\ \_$ that explicitly indicates which side it comes from. In particular, $A + A$ is not in general isomorphic to A, but to two disjoint copies of A. The case splitting construction $\texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2$ inspects a value $t : A + B$ and branches on its tag. Either t reduces to a value of the form $\sigma_1\ t'$, and the program continues with u_1 after having bound the variable x_1 to the "payload" t' (of type A), or it is a $\sigma_2\ t'$, and the program continues with u_2 after having bound x_2 to t' (of type B).

There is exactly one inhabitant of the unit type 1 (in an empty environment), it is the trivial value $()$, and it has no use at all. While its *value* is useless, the unit type 1 itself may be useful. For example, consider the parametrized type $F(\alpha) \stackrel{\mathsf{def}}{=} A \times \alpha$ representing a type A with some extra information. Whenever one wishes to provide no extra information at all, one can just use $F(1)$. (In impure languages with side-effects, turning a type A into the thunk type $1 \to A$ may also help controlling evaluation order.)

There is no inhabitant of the empty type 0 (in an empty environment). Thus, subprograms of type t can only appear in portions of the code that are un-reachable – they are never executed when reducing a term in an empty environment. The construction $\texttt{absurd}(t)$ let us get any type out of these un-reachable fragments; one can think of it as signaling a dynamic failure. This is useful in a situation that is dual to the $\alpha \mapsto A \times \alpha$ example above.

Indeed, consider the parametrized type $G(\alpha) \stackrel{\mathsf{def}}{=} A + \alpha$, denoting values of type A or, depending on the instantiation of α, something else. For example, with $G(1)$ the value may simply be completely absent (this is the "option" or "maybe" type of functional languages), and with $G(E)$ it may be replaced by some explanation for the absence, of type E (one may also think of the E as exceptions raised during the computation of A). Finally, the type $G(0)$ can be used in the case where one is actually sure that a value of type A is present: the other case is impossible.

Then, we need a typing rule as strong as the one for $\texttt{absurd}(\_)$ to be able to effectively manipulate this type $G(0) = A + 0$. Consider for example the function $\texttt{extract} : G(0) \to A$:

$$
\texttt{extract} \stackrel{\mathsf{def}}{=} \lambda t.\, \texttt{match}\ t\ \texttt{with}\ \begin{array}{l} \mid\ \sigma_1\ x \to x \\ \mid\ \sigma_2\ y \to \texttt{absurd}(y) \end{array}
$$

The type of `extract` makes perfect sense: if we have a $A + 0$, as the second case is

impossible we know we can always extract the A. However, in the second branch of the case split, we are left in uncomfortable position of having to produce a A out of thin air, or rather out of a value of type 0; the typing rule for $\texttt{absurd}(\_)$, allowing us to turn a 0 into any type we want, is crucial to write this program. One can relate this to the `assert false` construction in the OCaml programming language, that immediately aborts the program (and should only be used in parts of the program that cannot be reached by program execution) and has any type. The current presentation is more principled, as one is required to provide a $t : 0$, acting as a proof that something indeed went wrong.

In Figure 3.3 we describe the reduction rules for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$, the computational meaning of the language. We use reduction contexts as in Section 2.2.3, which cover the full set of possible one-hole contexts for this grammar, and thus capture the *full* reduction relation, rather than any specific reduction strategy.

Figure 3.3.: Reduction for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

$$
\begin{array}{lll}
E, F, G ::= & & \text{one-hole contexts} \\
\quad \mid \Box & & \\
\quad \mid \lambda t.\, E & & \\
\quad \mid t\ E \mid E\ u & & \\
\quad \mid (t, E) \mid (E, u) & & \\
\quad \mid \pi_i\ E & & \\
\quad \mid \sigma_i\ E & & \\
\quad \mid \left(\begin{array}{l}\texttt{match}\ E\ \texttt{with} \\ \mid\ \sigma_1\ x_1 \to u_1 \\ \mid\ \sigma_2\ x_2 \to u_2\end{array}\right) \mid \left(\begin{array}{l}\texttt{match}\ t\ \texttt{with} \\ \mid\ \sigma_1\ x_1 \to E \\ \mid\ \sigma_2\ x_2 \to u_2\end{array}\right) \mid \left(\begin{array}{l}\texttt{match}\ t\ \texttt{with} \\ \mid\ \sigma_1\ x_1 \to u_1 \\ \mid\ \sigma_2\ x_2 \to E\end{array}\right) & & \\
\quad \mid \texttt{absurd}(E) & &
\end{array}
$$

$$
(\lambda x.\, t)\ u \triangleright_\beta t[u/y] \qquad \pi_i\ (t_1, t_2) \triangleright_\beta t_i \qquad \texttt{match}\ \sigma_i\ t\ \texttt{with} \begin{array}{l}\mid\ \sigma_1\ x_1 \to u_1 \\ \mid\ \sigma_2\ x_2 \to u_2\end{array} \triangleright_\beta u_i[t/x_i]
$$

$$
\frac{t \triangleright_\beta u}{E\,[t] \to_\beta E\,[u]}
$$

Lemma 3.1.1 (Substitution in $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ preserves typing)**.**
If $\Gamma, x : A \vdash t : B$ *and* $\Gamma \vdash u : A$*, then* $\Gamma \vdash t[u/x] : B$*. In other words, the following rule is admissible:*

$$
\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : B}\ \textsc{subst}
$$

Proof. The proof is done by an easy induction, exactly in the same style as the proof of Lemma 2.3.2 (Substitution in $\mathsf{\Lambda C}(\to, \texttt{box})$ preserves typing). □

Lemma 3.1.2 (Subject reduction for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$)**.**
if $\Gamma \vdash t : A$ *and* $t \to_\beta t'$*, then* $\Gamma \vdash t' : A$*.*

Proof. The proof is done by case analysis; the function case is identical to the one in the proof of Theorem 2.3.3 (Subject reduction for $\mathsf{\Lambda C}(\to, \texttt{box})$). The new cases, for pairs and sums, are given below.

$$\frac{\dfrac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2}}{\Gamma \vdash \pi_i\ (t_1, t_2) : A_i} \quad \triangleright_\beta \quad \Gamma \vdash t_i : A_i$$

$$\frac{\dfrac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2} \qquad \Gamma, x_1 : A_1 \vdash u_1 : C \qquad \Gamma, x_2 : A_2 \vdash u_2 : C}{\Gamma \vdash \texttt{match}\ \sigma_i\ t\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array} : C} \quad \triangleright_\beta$$

$$\frac{\Gamma, x_i : A_i \vdash u_i : C \qquad \Gamma \vdash t : A_i}{\Gamma \vdash u_i[t/x_i] :}$$

□

We could go on and establish a progress theorem, and thus soundness of the calculus, as we have done in Section 2.3.2 (A simple type system for the administrative λ-calculus). The focus of this chapter is rather on the correspondence with proof derivations, so we will skip this step.

3.1.2. The Curry-Howard isomorphism, technically

The correspondence between programming and logic is already apparent from the typing rules alone: forget the term parts, and voila, you see the logic underlying your programming language. But to get a full correspondence, we also want to study the relation between the reduction of programs and the reduction of proofs. Note that reduction, for programs, is defined on untyped terms; a first step is to show that it can be lifted into a reduction at the level of type derivations, that is, that our reduction semantics is sound with respect to our type system – this is exactly what we proved in Lemma 3.1.2 (Subject reduction for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$). Then, one expects to show that reduction on typing derivations corresponds to reduction of proofs.

Definition 3.1.1 $[\![\Gamma]\!]_{\mathtt{CH}}, [\![t]\!]_{\mathtt{CH}}$ (Curry-Howard erasure)**.**
If Γ is a typing environment (mapping from variables to types/formulas), we write $[\![\Gamma]\!]_{\mathtt{CH}}$ for the set of types in its range – a proof context.

If t is well-typed in $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$, we write $[\![t]\!]_{\mathtt{CH}}$ for the proof derivation of the judgment $[\![\Gamma]\!]_{\mathtt{CH}} \vdash A$ in $\mathsf{PIL}(\to, \times, 1, +, 0)$ obtained by erasure of the typing derivation of t – replacing each rule of $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ by the corresponding rule in $\mathsf{PIL}(\to, \times, 1, +, 0)$, for example

$$\left[\!\!\left[\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \right]\!\!\right]_{\mathtt{CH}} \quad \stackrel{\mathsf{def}}{=} \quad \frac{[\![t]\!]_{\mathtt{CH}} :: [\![\Gamma]\!]_{\mathtt{CH}}, A \vdash B}{[\![\lambda x.\, t]\!]_{\mathtt{CH}} :: [\![\Gamma]\!]_{\mathtt{CH}} \vdash A \to B}$$

We recall that the notation $\Pi :: \Gamma \vdash A$, introduced in Section 1.1 (A first introduction to inference rules), means that Π is a derivation tree for the judgment $\Gamma \vdash A$.

Note that the relation between proof contexts (sets of types) and typing environments (mappings from variables to types) is one-to-many: many different typing environments erase to the same proof context. Correspondingly, the relation between proofs and typed programs is one-to-many.

Because proofs have lost some structure that was present in programs, the operations on proofs may not respect this extra structure. We previously remarked – Theorem 1.3.2 (Substitution for $\mathsf{PIL}(\to, \times, 1, +, 0)$) – that many different proofs may be considered as the result of substituting one proof into another. Only one of those substitutions respect the variable relation present in the program derivation. So we cannot hope to have a bijection between proofs and programs that would preserve reduction; the best we can formulate is the fact that reduction of proofs can *simulate* reduction of programs.

Lemma 3.1.3 (Term substitutions erase to proof substitutions)**.**
If $\Gamma, x : A \vdash t : B$ *and* $\Gamma \vdash u : A$ *then there is a substitution*

$$\frac{[\![t]\!]_{\mathtt{CH}} :: [\![\Gamma]\!]_{\mathtt{CH}}, A \vdash B \qquad [\![u]\!]_{\mathtt{CH}} :: [\![\Gamma]\!]_{\mathtt{CH}} \vdash A}{\Pi :: [\![\Gamma]\!]_{\mathtt{CH}} \vdash B}\ \text{SUBST}$$

such that $\Pi = [\![u[t/x]]\!]_{\mathtt{CH}}$.

Proof sketch. Parallel inspection of the definition of substitution on proofs and terms shows that they preserve the erasing relation. Remember that we have a choice in the definition of substitution, when encountering an axiom rule on A when A is in the context. Erasure of the substituted variable, $[\![x]\!]_{\mathtt{CH}}$, corresponds to the case where an axiom rule is replaced by $[\![u]\!]_{\mathtt{CH}}$. Erasure of another variable of type A, $[\![y]\!]_{\mathtt{CH}}$, corresponds to the case where the axiom rule is unchanged. ⊔

Theorem 3.1.4.
Curry-Howard correspondence between $\mathsf{PIL}(\to, \times, 1, +, 0)$ *and* $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

Reduction of $\mathsf{PIL}(\to, \times, 1, +, 0)$ *proofs forward-simulates reduction of well-typed* $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ *terms: if* $t \triangleright_\beta u$*, then* $[\![t]\!]_{\mathtt{CH}} \triangleright_R [\![u]\!]_{\mathtt{CH}}$.

Proof. Let us write the proofs with reducible introduction-elimination pairs on one side, and well-typed terms with a head redex on the other. Preservation of erasure is evident, using Lemma 3.1.3 on substitutions.

$$\frac{\dfrac{\Pi_B :: \Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B} \quad \triangleright_R \quad \frac{\Pi_B :: \Gamma, A \vdash B \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

$$\frac{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.\, t)\ u : B} \quad \triangleright_\beta \quad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : B}\ \text{SUBST}$$

$$\frac{\dfrac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}}{\Gamma \vdash A_i} \quad \triangleright_R \quad \Pi_i :: \Gamma \vdash A_i$$

$$\frac{\dfrac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2}}{\Gamma \vdash \pi_i\ (t_1, t_2) : A_i} \quad \triangleright_\beta \quad \Gamma \vdash t_i : A_i$$

$$\frac{\dfrac{\Pi :: \Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} \qquad \Pi_1 :: \Gamma, A_1 \vdash C \qquad \Pi_2 :: \Gamma, A_2 \vdash C}{\Gamma \vdash C}$$

$$\triangleright_R \quad \frac{\Pi_i :: \Gamma, A_i \vdash C \qquad \Pi :: \Gamma \vdash A_i}{\Gamma \vdash C}\ \text{SUBST}$$

$$\frac{\dfrac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2} \qquad \Gamma, x_1 : A_1 \vdash u_1 : C \qquad \Gamma, x_2 : A_2 \vdash u_2 : C}{\Gamma \vdash \mathtt{match}\ \sigma_i\ t\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array} : C}$$

$$\triangleright_\beta \quad \frac{\Gamma, x_i : A_i \vdash u_i : C \qquad \Gamma \vdash t : A_i}{\Gamma \vdash u_i[t/x_i] : C}\ \text{SUBST}$$

□

3.1.3. Curry-Howard: discussion

The name "Curry-Howard correspondence" denotes a family of such correspondence theorems for many different logics (but not necessarily all of their equi-provably-expressive presentations as systems of inference rules). It evokes the now well-understood idea that logics, in general, correspond to strongly-typed programming languages, or equivalently that normalization of proof derivations is a rich model of computation – among others.

Another point of view would be to say that the programming languages equipped with a type system that guarantees (strong) normalization are called "logics", and that the part of proof theory concerned with normalization of cut-elimination can thus be understood as a well-defined subset of programming language theory – in general programming languages may accept non-terminating programs. The point is not, of course, to practice reductionism by saying that some field is "just" a sub-field of another, but instead to understand the relation between them in a way that allows transfer of inspiration, concepts and results in both directions. This approach has worked beautifully since the second half of the 20th century; the whole point of the present thesis is to solve programming-languages questions using proof-theoretic tools.

Once one has grown familiar with it, the correspondence result is so simple that one would even wonder what its value is: isn't it obvious that one can give a term syntax to derivations of inference rules? The value, of course, is to transfer intuition and results; but by now most of the community has grown accustomed to switching between the two points of view with such ease that this also seems rather automatic. It may be interesting to discuss how the two points of view *differ*. For example:

- As already pointed out, it is common for programming languages to allow and study non-terminating programs, while that is generally outside the scope of logics – it usually coincides with unsoundness, the ability to prove anything. Yet, we are slowly coming with more logical ways to reason about some forms of usefully non-terminating programs (in particular coinduction).

- Some logicians have a deep interest in decomposing logics into more atomic / restricted / simpler notions, which goes much farther what has been realistically achieved in minimalism for language design. Linear logic, for example, is a decomposition of classical and intuitionistic logics that let us control resources in a very fine-grained way. In some ways, that can be made precise, this goes in the same direction as studies in compilation, translating programs from a high to a low-level language, with a more algebraic perspective. It has also had fruitful applications to programming language design, for the control of program resources such as mutable memory.

- On a pragmatic level, term syntaxes are more concise and thus more comfortable to manipulate than partial derivations. Many transformation of proofs gain to be expressed at the level of terms instead.

- Programming language designers in fact distinguish several possible term syntaxes, some that are "fully explicit" and are trivially isomorphic to a typing derivation, and other where some of the typing information has been left implicit, and may or may not be inferrable statically without additional information – those syntaxes are interesting in their own right, and not isomorphic to typing derivations. For example, a λ-abstraction may be written $\lambda x.\, t$ as in the untyped λ-calculus, or for example $\lambda(x : A).\, t$, with type information about the variable t included in the syntax. In the general case, for expressive enough type systems, recovering the full typing derivation from a less-typed term syntax can be undecidable.

This idea of a distinction between various levels of explicitness of term syntaxes is crucial. For example, dependent type systems have a judgment $\Gamma \vdash t = u : A$ justifying

that two terms are equal, and we distinguish "extensional" and "intensional" systems. The "extensional" theories have a richer notion of equality than what can be recovered from the usual syntax of programs, which does not explicitly mention the use of equalities during type-checking. One need extra information, the equality derivations, to check that a term is well-typed. On the other hand, "intensional" theories have a weaker equality whose witnesses can be decidably recovered from the same term syntax. This distinction only makes sense because there is consensus on some "term syntax", shared by both families of systems, that has *less* information than the typing derivation – and this syntax is not just the untyped λ-calculus, as this would make type-recovery undecidable in all cases.

Another example of interesting notion based on this distinction is "type erasure": when a given system has a term syntax that is strictly less explicit than the typing derivations, a question is whether the dynamic semantics of the language can be defined on the less-explicit term syntax. There are two ways this could not be the case:

- It could be the case that reduction depends on information that is present in the typing derivation, but not in the less-explicit program syntax. For example, Haskell type-classes or Scala implicits affect the observable behavior of programs, and they are not elaborated in the surface term syntax – one need a more explicit form, closer to full typing derivations, for the dynamic behavior to be unambiguous.

  In a slightly more general way, one may wonder whether all possible ways to give a typing derivation for a given term or judgment give equivalent derivations. This property is called *coherence*; it guarantees that the term syntax is a proper quotient over the corresponding derivations, respecting their identity. This property of coherence is related to the question of which types have unique inhabitants: those are exactly the types whose terms can be elided without losing coherence.

- It could also be the case that reduction is definable on the program syntax, but cannot be lifted into a reduction relation on typing derivations, because some reduction steps break typing. This happens when the type system and the reduction relation do not match, in the sense that some well-typed programs may reduce into an error state the type system was designed to prevent. But more interestingly it can also happen that well-typed programs "never go wrong", in that their reducts never run into this class of errors, but still intermediate steps of the reduction cannot be typed (see for example Barbanera, Dezani-Ciancaglini, and de'Liguoro [1995], Schubert and Fujita [2014]).

  In this case, there should exist a richer term syntax that restores this property by allowing to reason on those intermediate states, but such richer syntax may not be known. We believe that this means that the correctness invariants of the analysis are not well-understood enough [Cretin and Rémy, 2014]; a semantic soundness proof may go through, but is not the end of the story.

3.2. Equivalence with sums and Curry-Howard-Lambek

3.2.1. $\beta\eta$-equivalence for $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$

In Section 2.3.3 (Equivalence of $\Lambda\mathsf{C}(\to, \mathtt{box})$ terms) we defined the $\beta\eta$-equality for $\Lambda\mathsf{C}(\to, \mathtt{box})$ as the congruence determined by the reduction and expansion principles on well-typed terms. Doing the same for $\mathsf{PIL}(\to, \times, 1, +, 0)$ would give the following rules:

$$(\lambda x.\, t)\ u \triangleright_\beta u[t/x] \qquad (t : A \to B) \triangleright_\eta \lambda x.\, t\ x$$

$$\pi_i\ (t_1, t_2) \triangleright_\beta t_i \qquad (t : A_1 \times A_2) \triangleright_\eta (\pi_1\ t, \pi_2\ t)$$

$$\mathtt{match}\ \sigma_i\ t\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} \triangleright_\beta u_i[t/y_i]$$

$$(t : A_1 + A_2) \triangleright_\eta \mathtt{match}\ t\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ y_1 \to \sigma_1\ y_1 \\ \sigma_2\ y_2 \to \sigma_2\ y_2 \end{array}$$

Weak and strong η-rules for sums Upon inspection the η-rule for sums above is found to be lacking. For example, we cannot prove the two following equivalences for $t : A_1 + A_2$

$$(t, u) \approx_? \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to (\sigma_1\ y_1, u)\\ \sigma_2\ y_2 \to (\sigma_2\ y_2, u)\end{array} \qquad x \approx_? \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to x\\ \sigma_2\ y_2 \to x\end{array}$$

This rule is called the "weak" η-rule ($\triangleright_{\texttt{weak}\,\eta}$). We shall use instead the "strong" η-rule, which quantifies over all the well-typed contexts $C\,[x : A_1 + A_2]$.

Notation 3.2.1 Non-linear contexts $C\,[x]$**.**
We write $C\,[x]$ for a context that may use its variable zero, one or several times; for example, (x, x) is such a non-linear context. The plugging operation $C\,[t]$ is similar to a substitution $C[t/x]$, except that it is not capture-avoiding.

Definition 3.2.1 Strong η-rule**.**

$$\forall C\,[x], \quad C\,[t : A_1 + A_2] \triangleright_\eta \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to C\,[\sigma_1\ y_1]\\ \sigma_2\ y_2 \to C\,[\sigma_2\ y_2]\end{array}$$

We can in particular recover the two equations above by taking

$$C_1\,[x] \stackrel{\text{def}}{=} (x, u) \qquad\qquad C_2\,[y] \stackrel{\text{def}}{=} x$$

η-rules for units Our previous notion of expansion did not suggest what the η-rules for 0 and 1 could be: they were defined on types that had both an introduction and elimination form. The generalized form of the "strong" η-rule for sum, however, can be derived into η-rules for those types. In the case of 1, there is no destructor, so we just expand to the constructor inside the context (as $C\,[\sigma_i\ \ldots]$ in the sum case); in the case of 0, there is no constructor, so we just expand to the destruction form — just as $(\texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ y_1 \to \_ \mid \sigma_2\ y_2 \to \_)$ in the sum case.

$$\forall C\,[x], C\,[t : 1] \triangleright_\eta C\,[()] \qquad\qquad \forall C\,[x], C\,[t : 0] \triangleright_\eta \texttt{absurd}(t)$$

The first rule say that any term of type 1 can be rewritten into $()$ under any context. As a consequence, the following typed equality holds:

$$\frac{}{\Gamma \vdash t \approx_\eta u : 1}$$

Notation 3.2.2 $\Gamma \vdash t\ \mathcal{R}\ u : A$**.**
We write $\Gamma \vdash t\ \mathcal{R}\ u : A$ when all of $\Gamma \vdash t : A$, $\Gamma \vdash u : A$ and $t\ \mathcal{R}\ u$ hold. In particular, $\Gamma \vdash t \approx u : A$ mean that the programs t, u are equivalent and of type $\Gamma \vdash A$.

The second rule implies that, in presence of a term t of type 0, *any term* t is equal to $\texttt{absurd}(t)$, as shown by considering the non-linear context that ignores its argument $C\,[x] \stackrel{\text{def}}{=} t$. If absurdity is provable, then the world explodes – at any type! This corresponds to the following typed equality rule:

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

We summarized the full, strong equivalence rules for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ in Figure 3.4 (Typed program equivalence for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$).

We should also define precisely the weak η-equivalence ($\approx_{\texttt{weak}\,\eta}$). The reduction and expansion principles we have given all use both the introduction and the elimination form of the corresponding connective, so in particular we have no expansion principle for units – which lack either forms. However, the generalization of strong η-equivalence to units suggests a weak η-equivalence principle for them, by instantiating the context parameter with the identity context: $C\,[x] \stackrel{\text{def}}{=} x$. The rules for weak η-equivalence are thus given in Figure 3.5.

Figure 3.4.: Typed program equivalence for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

$$\text{FUN-}\beta\quad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.\, t)\ u \triangleright_\beta t[u/x] : B} \qquad \text{FUN-}\eta\quad \frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t \triangleright_\eta \lambda x.\, t\ x : A \to B}$$

$$\text{PROD-}\beta\quad \frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash \pi_i\ (t_1, t_2) \triangleright_\beta t_i : A_i} \qquad \text{PROD-}\eta\quad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash t \triangleright_\eta (\pi_1\ t, \pi_2\ t) : A_1 \times A_2}$$

$$\text{SUM-}\beta\quad \frac{\Gamma \vdash t : A_i \qquad \begin{array}{l}\Gamma, y_1 : A_1 \vdash u_1 : C \\ \Gamma, y_2 : A_2 \vdash u_2 : C\end{array}}{\Gamma \vdash \texttt{match}\ \sigma_i\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2\end{array} \triangleright_\beta u_i[t/y_i] : C}$$

$$\text{SUM-}\eta\quad \frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma \vdash C\,[\square : A_1 + A_2] : B}{\Gamma \vdash C\,[t] \triangleright_\eta \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to C\,[\sigma_1\ y_1] \\ \sigma_2\ y_2 \to C\,[\sigma_2\ y_2]\end{array} : B}$$

$$\text{UNIT-}\eta\quad \frac{\Gamma \vdash t, u : 1}{\Gamma \vdash t \approx_\eta u : 1} \qquad \text{EMPTY-}\eta\quad \frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

Figure 3.5.: Weak η-equivalence for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$

$$\text{FUN-}\texttt{weak}\,\eta\quad \frac{\Gamma \vdash t : A \to B}{t \triangleright_{\texttt{weak}\,\eta} \lambda x.\, t\ x} \qquad \text{PROD-}\texttt{weak}\,\eta\quad \frac{\Gamma \vdash t : A_1 \times A_2}{t \triangleright_{\texttt{weak}\,\eta} (\pi_1\ t, \pi_2\ t)}$$

$$\text{SUM-}\texttt{weak}\,\eta\quad \frac{\Gamma \vdash t : A_1 + A_2}{t \triangleright_{\texttt{weak}\,\eta} \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y_1 \to \sigma_1\ y_1 \\ \sigma_2\ y_2 \to \sigma_2\ y_2\end{array}}$$

$$\text{UNIT-}\texttt{weak}\,\eta\quad \frac{\Gamma \vdash t : 1}{t \triangleright_{\texttt{weak}\,\eta} ()} \qquad \text{EMPTY-}\eta\quad \frac{\Gamma \vdash t : 0}{t \triangleright_{\texttt{weak}\,\eta} \texttt{absurd}(t)}$$

Fact 3.2.1.
If $t \approx_{\texttt{weak}\,\eta} u$ *then* $t \approx_\eta u$.

3.2.2. Curry-Howard-Lambek

There is a third angle to the Curry-Howard isomorphism, which is a correspondence between proofs (or programs) and morphisms in well-known categories; in the case of $\mathsf{PIL}(\to, \times, 1)$, that would be Cartesian closed categories.

We will not use category theory much in this PhD thesis – we have enough content already; but one should note that categories are opinionated about what the equality of their morphisms should be, and in particular they confirm our intuition of equality for $\mathsf{PIL}(\to, \times, 1, +, 0)$. This is the topic of this subsection, which will assume some category-theory background – feel free to skip it if you do not know what objects and arrows/morphisms are.

Categorical models of $\mathsf{PIL}(\to, \times, 1, +, 0)$ The usual way to use category theory when studying logics or typed programming language is to give a denotational semantics for a

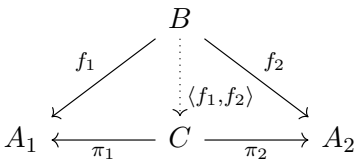
logic or type system. A denotational semantics is an interpretation, translation, modeling, of the logic or language into some mathematical structure, such that two equivalent proofs or programs are given the same interpretation. It translates simple syntactic objects and their complex equivalence relation into complex mathematical objects and their simple mathematical equality.[1]

Typically, the contexts and types of the language become objects of the category, and the terms (and/or substitutions or general contexts) of the language become morphisms (arrows) of the category. $\mathsf{PIL}(\to, \times, 1, +, 0)$ can be interpreted into any category which has a bicartesian closed structure[2]: it has finite products (product types and 1), exponentials (function types), and finite co-products (sum types and 0), and they work well together.

As any denotational semantics (model construction), this can be used to prove consistency of a logic, from non-inhabitation of the empty type 0: if there is no arrow from the "empty context" object to the "empty type" object, then we know there is no proof of false, or closed term of type 0. For reasons of space, we will not build the full interpretation here, but just remark on the equality of coproducts and units.

(Getting consistency proof from denotational semantics is not unique to category theory, any model construction will do. One interest of categorical models is that they are formulated in terms of the minimal structure required to model the syntax, and (in the good cases) those categorical conditions then capture the full generality of the syntax instead of being specialized, lossy interpretations. In other words, category theory serves as a toolkit to describe the "initial" models isomorphic to the syntax with the desired equivalence relations.)

$\beta\eta$ from the product object The definition of the categorical *product* object is as follows: an object C is the product of two objects A_1, A_2 if there exists two morphisms $\pi_1 : C \to A_1$, and $\pi_2 : C \to A_2$ such that, for any object B and pair of morphisms $f_1 : B \to A_1$, $f_2 : B \to A_2$, there exists a *unique* morphism $\langle f_1, f_2 \rangle : B \to C$ such that the following diagram commutes:



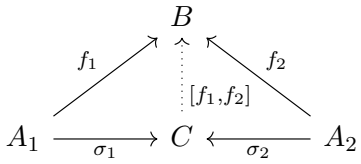
One can show that such a C, if it exists, is unique modulo (unique!) isomorphism; it can be written $A_1 \times A_2$. In a model of the lambda-calculus, where objects represent contexts and type, an arrow $f : \Gamma \to A$ represents a term $\Gamma \vdash t : A$. In particular, when B in the diagram above is a typing environment Γ, the $f_i : \Gamma \to A_i$ are terms $\Gamma \vdash t_i : A_i$, and the product-former $\langle f_1, f_2 \rangle$ is exactly the pair (t_1, t_2). The β-rule for pairs can then be read from the commutative diagram: the diagram says that following $\langle f_1, f_2 \rangle$ then π_i is equal to following f_i, that is $\pi_i \circ \langle f_1, f_2 \rangle = f_i$, in other words $\Gamma \vdash \pi_i\ (t_1, t_2) = t_i : A$.

Interestingly, the η-rule can also be read from this definition. For any B and morphism $p : B \to C$, we can define $f_i : B \to A_i$ $(i \in \{1, 2\})$ simply by taking $f_i \stackrel{\mathsf{def}}{=} \pi_i \circ p$, such that the product diagram commutes. But there is another arrow in $B \to C$ that makes this diagram commute, namely the arrow $\langle f_1, f_2 \rangle$. By unicity of the product morphism, these two morphisms are equal: $p = \langle \pi_1 \circ p, \pi_2 \circ p \rangle$. In particular, if we take B to be some typing environment Γ, a morphism $p : \Gamma \to C$ is just a term $\Gamma \vdash t : A_1 \times A_2$, and unicity gives us $\Gamma \vdash t = (\pi_1\ t, \pi_2\ t) : A_1 \times A_2$.

[1] As Alexandre Miquel once told me in front of a whiteboard explanation of some variant of classical realizability for extensional choice: "C'est ça les mathématiques, on définit des objets compliqués pour rendre les questions plus simples."

[2] It is hard to find a detailed exposition on bicartesian closed categories only, for example indicating how the isomorphism $0 \times A \simeq 0$ is derived from their definition. We refer the reader to the excellent book Lambek and Scott [1986], Section 8 (Cartesian closed categories with coproducts), page 65, easily found on Libgen.

$\beta\eta$ from the co-product object To obtain the β- and η-rules for sums (coproducts), we dualize the product diagram. C is the coproduct of A_1 and A_2 if there are $\sigma_i : A_i \to C$ ($i \in \{1,2\}$) such that, for any B with morphisms $f_i : A_i \to B$ ($i \in \{1,2\}$), there is a unique $[f_1, f_2] : C \to B$ such that the following diagram commutes:



Again, C is unique modulo unique isomorphism and represents the sum $A_1 + A_2$. In a model of the lambda-calculus, a "consumer" morphisms from type A to a type B represents a context $D\,[x]$ returning a B with a hole x of type A – this hole may appear several times in D. In particular, when the $f_i : A_i \to B$ in the diagram above are contexts $D_i\,[x]$, the morphism $[f_1, f_2]$ is exactly the case-splitter context

$$D\,[\square] \stackrel{\text{def}}{=} \texttt{match}\ \square\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to D_1\,[y_1] \\ \sigma_2\ y_2 \to D_2\,[y_2] \end{array} \right.$$

The β-rule arises from the fact that following σ_i then $[f_1, f_2]$ is equal to following f_i, that is $[f_1, f_2] \circ \sigma_i = f_i$; in particular, for any morphism $g : \Gamma \to A_i$ representing a term $\Gamma \vdash t : A_i$, we have $[f_1, f_2] \circ \sigma_i \circ g = f_i \circ g$, that is

$$\texttt{match}\ \sigma_i\ t\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to C_1\,[y_1] \\ \sigma_2\ y_2 \to C_2\,[y_2] \end{array} \right. = C_i\,[t]$$

Now, for any B and morphism $s : C \to B$, we can define $f_i : A_i \to B$ ($i \in \{1,2\}$) simply by taking $f_i \stackrel{\text{def}}{=} s \circ \sigma_i$, such that the co-product diagram commutes. But there is another arrow in $C \to B$ that makes this diagram commute, namely the arrow $[f_1, f_2]$. By unicity of the co-product morphism, these two morphisms are equal: $s = [s \circ \sigma_1, s \circ \sigma_2]$. In particular, if s is some context $D\,[x]$, then this equality gives us the strong η-rule, $D\,[x] = \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to D\,[\sigma_1\ y_1] \\ \sigma_2\ y_2 \to D\,[\sigma_2\ y_2] \end{array} \right.$.

Units 1 and 0 In a bicartesian closed category, the unit type 1 is interpreted as a *terminal* object, an object 1 such that for any other object A there exists a unique morphism $1_A : A \to 1$. The empty type 0 is interpreted as an *initial* object, an object 0 such that for any other object A there exists a unique morphism $0_A : 0 \to A$.

Our equality rule for 1 is easily derived from the unicity of morphisms into terminal objects: if we have two terms $\Gamma \vdash t, u : 1$, they correspond to morphisms $f_t, g_u : \Gamma \to 1$. Because there is a unique arrow 1_Γ from Γ to 1, we have $f_t = 1_\Gamma = g_u$, and thus we should have $\Gamma \vdash t = u : 1$.

We proposed two equalities for 0: the first says that any $C\,[t : 0]$ is equal to $\texttt{absurd}(t)$, and the second says that in a context where 0 is provable, all terms are equal. The first is easy to derive from the categorical structure: by definition of 0 as an initial object, any arrow $C\,[x] : 0 \to A$ is equal to $\texttt{absurd}(\square) : 0 \to A$.

Deriving the second equality rule – $\Gamma \vdash 0$ implies $\Gamma \vdash t \approx_\eta u : A$ – from the definition of a bicartesian closed category is a more difficult. Intuitively, this comes from the "high school algebra" equation $a^0 = 1$: as soon as a context Γ can prove 0, it becomes isomorphic to 0, and all terms in Γ become equal.

Lemma 3.2.2.
If there is an arrow $f : \Gamma \to 0$, then $0_\Gamma \circ f : \Gamma \to \Gamma$ is equal to the identity morphism $\texttt{id}_\Gamma$.

Proof (from Lambek and Scott [1986]). Let us first show that $0 \times \Gamma \simeq 0$. By definition of the exponential functor $(\Gamma \to \_)$ as the right adjoint of the product functor $(\_ \times \Gamma)$

– functional programmers call this "currying" – we have the bijection between sets of morphisms $\mathtt{Hom}(0 \times \Gamma, A) \simeq \mathtt{Hom}(0, \Gamma \to A)$. The latter set has a unique morphism, therefore there is always a unique morphism from $0 \times \Gamma$ to any A: the object $0 \times \Gamma$ is initial, thus isomorphic to 0.

In particular, there is a unique arrow from $0 \times \Gamma$ to itself. $0_{0\times\Gamma} \circ \pi_1$ is such an arrow $(0 \times \Gamma \xrightarrow{\pi_1} 0 \xrightarrow{0_{0\times\Gamma}} 0 \times \Gamma)$, and so is $\mathtt{id}_{0\times\Gamma}$, so they are equal:

$$0_{0\times\Gamma} \circ \pi_1 \quad = \quad \mathtt{id}_{0\times\Gamma}$$

Finally, remark that $f = \pi_1 \circ \langle f, \mathtt{id}_\Gamma \rangle$ (by projection) and $\pi_2 \circ 0_{0\times\Gamma} = 0_\Gamma$ (all functions in $0 \to \Gamma$ are equal). We thus have

$$\begin{array}{rcl} 0_\Gamma \circ f & = & (\pi_2 \circ 0_{0\times\Gamma}) \circ (\pi_1 \circ \langle f, \mathtt{id}_\Gamma \rangle) \\ & = & \pi_2 \circ (0_{0\times\Gamma} \circ \pi_1) \circ \langle f, \mathtt{id}_\Gamma \rangle \\ & = & \pi_2 \circ \mathtt{id}_{0\times\Gamma} \circ \langle f, \mathtt{id}_\Gamma \rangle \\ & = & \pi_2 \circ \langle f, \mathtt{id}_\Gamma \rangle \\ & = & \mathtt{id}_\Gamma \end{array}$$

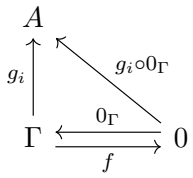
□

Corollary 3.2.3.
If there is a morphism $f : \Gamma \to 0$, then any two morphisms $g_1, g_2 : \Gamma \to A$ are equal.

Proof. We have $\mathtt{id}_\Gamma = 0_\Gamma \circ f$ from Lemma 3.2.2, so in particular for each g_i we have

$$g_i = g_i \circ \mathtt{id}_\Gamma = g_i \circ 0_\Gamma \circ f$$

that is, the following commuting diagram for each g_i:



By the initial object property, all morphisms $g_i \circ 0_\Gamma$ are equal, so we have

$$\begin{array}{rcl} g_1 & = & (g_1 \circ 0_\Gamma) \circ f \\ & = & (g_2 \circ 0_\Gamma) \circ f \\ & = & g_2 \end{array}$$

□

In particular, for any Γ such that $\Gamma \vdash t : 0$, we have $\Gamma \vdash u_1 = u_2 : A$ for any A.

Credits I discovered category theory thanks to a book recommendation by Brice Arnould ("Logique, ensemble, catégories (le point de vue constructif)", by Pierre Ageron), when I was just beginning to understand what mathematics and programming were about. Not using them daily in my own syntax-grounded work, it is not easy to preserve enough working memory to follow the work of the more category-infused authors of the field. For a positive example of excellent exposition of category-inspired ideas to programming layperson, see the work of Ralf Hinze, for example Henglein and Hinze [2013].

I learned the fact that the strong η-rule for sums can be justified from categorical coproducts from Andrej Filinski's master thesis – a very interesting read. I have been surprised to see people surprised by this fact; it is not as well-known as it should be.

3.3. Extrusion and commuting conversions

3.3.1. Splitting strong η: weak η plus extrusion

When we moved from the weak η-expansion to the strong η-expansion for sums – subsequently generalizing it to units (1 and 0) – we went outside the realm of the rules that had

been suggested by our study of the natural deduction in Section 1.3 (On the meaning of logical connectives: testing a logic). In this subsection, we repair that mismatch by showing that those equivalences suggested by programming intuition are reasonable operations on proof derivations; they correspond to the *permutability* of disjunction elimination – and unit rules.

Our strong η-rule is the following relation between well-typed programs:

$$\forall C\,[x : A_1 + A_2], \quad C\,[t] \triangleright_\eta \texttt{match}\ t\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to C\,[\sigma_1\ y_1] \\ \sigma_2\ y_2 \to C\,[\sigma_2\ y_2] \end{array} \right.$$

Because of the quantification on all contexts $C\,[x]$, this rule is non-local. Such contexts are easy to manipulate on term representations, but not so easily defined on proof derivations. Transformations on proof derivations are traditionally rather defined as local transformations on a small number of adjacent inference rules (what could be called a "small-step" style). By reformulating the η-transformation in this way, we recover permutation principles that are known in the logic community as (instances of) *commuting conversions*.

We decompose this strong η-expansion ($\triangleright_\eta$) into a combination of the weak η-expansion ($\triangleright_{\texttt{weak}\,\eta}$) under the context $C\,[x]$, and the *extrusion* ($\approx_{\texttt{extr}}$) of the disjunction elimination from the context $C\,[x]$, to be defined in this section:

$$\begin{array}{lll} & C\,[t : A_1 + A_2] & \\ \triangleright_{\texttt{weak}\,\eta} & C\left[\texttt{match}\ t\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to \sigma_1\ y_1 \\ \sigma_2\ y_2 \to \sigma_2\ y_2 \end{array} \right.\right] & \\ \approx_{\texttt{extr}} & \texttt{match}\ t\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to C\,[\sigma_1\ y_1] \\ \sigma_2\ y_2 \to C\,[\sigma_2\ y_2] \end{array} \right. & \end{array}$$

Let us define ($\approx_{\texttt{extr}}$) piece by piece, guided by the requirement that it captures exactly the transformation performed by the strong η-expansion. One can check that each definition below is sound, in the sense that it is derivable from ($\approx_{\beta\eta}$).

Extrusion out of non-binding contexts The rules to extrude a non-binding context are given in Figure 3.6. As an example, let us show how the first rule

$$t \left(\texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to r_1 \\ \sigma_2\ y_2 \to r_2 \end{array} \right.\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to t\ r_1 \\ \sigma_2\ y_2 \to t\ r_2 \end{array} \right.$$

is justified by ($\approx_{\beta\eta}$). For this purpose, let us define $C\,[x] \stackrel{\text{def}}{=} t\ (\texttt{match}\ x\ \texttt{with}\ \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2)$. Then we have

$$\begin{array}{lll} & t \left(\texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to r_1 \\ \sigma_2\ y_2 \to r_2 \end{array} \right.\right) & \\ = & C\,[u] & \\ \triangleright_\eta & \texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to C\,[\sigma_1\ z_1] \\ \sigma_2\ z_2 \to C\,[\sigma_2\ z_2] \end{array} \right. & \\ = & \texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to t \left(\texttt{match}\ \sigma_1\ z_1\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to r_1 \\ \sigma_2\ y_2 \to r_2 \end{array} \right.\right) \\ \sigma_2\ z_2 \to t \left(\texttt{match}\ \sigma_2\ z_2\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to r_1 \\ \sigma_2\ y_2 \to r_2 \end{array} \right.\right) \end{array} \right. & \\ \triangleright_\beta\ \triangleright_\beta & \texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to t\ (r_1[z_1/y_1]) \\ \sigma_2\ z_2 \to t\ (r_2[z_2/y_2]) \end{array} \right. & \\ =_\alpha & \texttt{match}\ u\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to t\ r_1 \\ \sigma_2\ y_2 \to t\ r_2 \end{array} \right. & \end{array}$$

Figure 3.6.: Extrusion of sum elimination out of non-binding contexts

$$t\ \left(\texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to t\ r_1\\ \sigma_2\ y_2 \to t\ r_2\end{array}$$

$$\left(\texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right)\ u \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\ u\\ \sigma_2\ y_2 \to r_2\ u\end{array}$$

$$\left(t, \texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to (t, r_1)\\ \sigma_2\ y_2 \to (t, r_2)\end{array}$$

$$\left(\texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\ , u\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to (r_1, u)\\ \sigma_2\ y_2 \to (r_2, u)\end{array}$$

$$\pi_j\ \left(\texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \pi_j\ r_1\\ \sigma_2\ y_2 \to \pi_j\ r_2\end{array}$$

$$\sigma_j\ \left(\texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \sigma_j\ r_1\\ \sigma_2\ y_2 \to \sigma_j\ r_2\end{array}$$

$$\texttt{match}\ \left(\texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right)\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ x_1 \to u_1\\ \sigma_2\ x_2 \to u_2\end{array}$$

$$\triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \texttt{match}\ r_1\ \texttt{with}\\ \qquad\qquad\qquad\qquad\ \ \sigma_1\ x_1 \to u_1\\ \qquad\qquad\qquad\qquad\ \ \sigma_2\ x_2 \to u_2\\ \sigma_2\ y_2 \to \texttt{match}\ r_2\ \texttt{with}\\ \qquad\qquad\qquad\qquad\ \ \sigma_1\ x_1 \to u_1\\ \qquad\qquad\qquad\qquad\ \ \sigma_2\ x_2 \to u_2\end{array}$$

Remark 3.3.1. In all these rules and the following ones, an implicit assumption is that extrusion preserves scoping of variables and well-typing. Consider for example the following extrusion:

$$t\ \left(\texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array}\right) \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ u\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to t\ r_1\\ \sigma_2\ y_2 \to t\ r_2\end{array}$$

This extrusion is only well-scoped if the y_i are not free in t – otherwise extrusion would capture these free occurrences. We leave these side-conditions implicit in the rules: they can be tediously derived from the constraint that typing and scoping be preserved. ∗

Extrusion out of binding contexts The rules for extruding a sum elimination out of a binding context are given in Figure 3.7.

As before, those rules carry an implicit side-condition on the extrusion relation to make sure that scoping (and thus typing) is preserved. For example, the first case could be written

$$\lambda x.\, \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to r_1\\ \sigma_2\ y_2 \to r_2\end{array} \quad \overset{x \notin t}{\triangleright_{\texttt{extr}}} \quad \texttt{match}\ t\ \texttt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \lambda x.\, r_1\\ \sigma_2\ y_2 \to \lambda x.\, r_2\end{array}$$

with an explicit condition $x \notin t$; if the variable x bound by the λ-abstraction is used in t, then moving t out of the binder scope breaks scoping and, in general, typing. This restriction is stronger than the capture-avoiding side-condition of Remark 3.3.1: if the variable condition is not satisfied, we cannot α-rename variables to satisfy it.

Also note that the two latter rules are reversible: the right-hand side may be an instance of the left-hand side. In particular, ($\triangleright_{\texttt{extr}}$) is not terminating as a rewrite rule, we need a

Figure 3.7.: Extrusion of sum elimination out of a binding context

$$\lambda x.\, \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2 \quad \triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \lambda x.\, r_1 \mid \sigma_2\ y_2 \to \lambda x.\, r_2$$

$$\texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2 \mid \sigma_2\ z_2 \to u'$$

$$\triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to r_1 \mid \sigma_2\ z_2 \to u' \mid \sigma_2\ y_2 \to \texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to r_2 \mid \sigma_2\ z_2 \to u'$$

$$\texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to u' \mid \sigma_2\ z_2 \to \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2$$

$$\triangleright_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to u' \mid \sigma_2\ z_2 \to r_1 \mid \sigma_2\ y_2 \to \texttt{match}\ u\ \texttt{with} \mid \sigma_1\ z_1 \to u' \mid \sigma_2\ z_2 \to r_2$$

particular strategy to stop expanding. This also implies that there is no strong notion of "canonical form" for this relation: if t and u are independent, there is no canonical reason for one to get split before the other.

"Extrusion" out of constant contexts The strange rules of Figure 3.8 arise from extrusion out of constant contexts. We will see a use for them in Chapter 10 (Focused λ-calculus).

Figure 3.8.: Extrusion of sum elimination out of a constant context

$$\texttt{absurd}(u) \quad \approx_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \texttt{absurd}(u) \mid \sigma_2\ y_2 \to \texttt{absurd}(u)$$

$$() \quad \approx_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to () \mid \sigma_2\ y_2 \to ()$$

$$x \quad \approx_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to x \mid \sigma_2\ y_2 \to x$$

Note that we consider them part of the equivalence relation ($\approx_{\texttt{extr}}$), but not in the (non-terminating) directed rewrite rule ($\triangleright_{\texttt{extr}}$).

Merging rules Consider the following generalized context: $C\,[x] \stackrel{\text{def}}{=} (x, x)$. The strong η-rule expands $C\,[t : A_1 + A_2]$ into $\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to (\sigma_1\ y_1, \sigma_1\ y_1) \mid \sigma_2\ y_2 \to (\sigma_2\ y_2, \sigma_2\ y_2)$. In particular, t is only case-split *once* in the final result.

It may reduce as follows using the rules we have seen so far:

$$
\begin{array}{lll}
C\,[t] & \triangleright_{\mathtt{weak}\,\eta} & C\left[\mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \sigma_1\ y_1\\ \sigma_2\ y_2 \to \sigma_2\ y_2\end{array}\right] \\
& = & \left(\mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \sigma_1\ y_1\\ \sigma_2\ y_2 \to \sigma_2\ y_2\end{array}\ ,\mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \sigma_1\ y_1\\ \sigma_2\ y_2 \to \sigma_2\ y_2\end{array}\right) \\
& \triangleright_{\mathtt{extr}} & \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \left(\sigma_1\ y_1, \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to \sigma_1\ z_1\\ \sigma_2\ z_2 \to \sigma_2\ z_2\end{array}\right)\\ \sigma_2\ y_2 \to \left(\sigma_2\ y_2, \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to \sigma_1\ z_1\\ \sigma_2\ z_2 \to \sigma_2\ z_2\end{array}\right)\end{array} \\
& \triangleright^*_{\mathtt{extr}} & \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_1\ y_1, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_1\ y_1, \sigma_2\ z_2)\end{array}\\ \sigma_2\ y_2 \to \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_2\ y_2, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_2\ y_2, \sigma_2\ z_2)\end{array}\end{array}
\end{array}
$$

The case-splits on t have been duplicated by this transformation: weak η-expansion generates as many case-splits as there are occurrences of the hole x in $C\,[x]$, and extrusion may create even more by duplicating code in the tail of an extruded case-split. In our example there is a first case-split on t and, in each case, a second case-split on the same term t.

Note that this is $\beta\eta$-equal to the result of the *strong* η-expansion above. To check this, it suffices to define a context equal to the whole term, with each occurrence of t replaced by a hole x, and then perform a strong η-expansion (along this context) and a series of β-reduction on this whole term. More precisely, we define the context

$$
D\,[x] \stackrel{\mathsf{def}}{=} \mathtt{match}\ x\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \mathtt{match}\ x\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_1\ y_1, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_1\ y_1, \sigma_2\ z_2)\end{array}\\ \sigma_2\ y_2 \to \mathtt{match}\ x\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_2\ y_2, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_2\ y_2, \sigma_2\ z_2)\end{array}\end{array}
$$

Note that for any $\sigma_i\ u$ we have

$$
\begin{array}{lll}
& D\,[\sigma_i\ u] & \\
= & \mathtt{match}\ \sigma_i\ u\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \mathtt{match}\ \sigma_i\ u\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_1\ y_1, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_1\ y_1, \sigma_2\ z_2)\end{array}\\ \sigma_2\ y_2 \to \mathtt{match}\ \sigma_i\ u\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_2\ y_2, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_2\ y_2, \sigma_2\ z_2)\end{array}\end{array} \\
\triangleright_\beta & \mathtt{match}\ \sigma_i\ u\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_i\ u, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_i\ u, \sigma_2\ z_2)\end{array} & \\
\triangleright_\beta & (\sigma_i\ u, \sigma_i\ u) &
\end{array}
$$

thus we have the η-equivalence:

$$
\begin{array}{lll}
& \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ y_1 \to \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_1\ y_1, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_1\ y_1, \sigma_2\ z_2)\end{array}\\ \sigma_2\ y_2 \to \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ z_1 \to (\sigma_2\ y_2, \sigma_1\ z_1)\\ \sigma_2\ z_2 \to (\sigma_2\ y_2, \sigma_2\ z_2)\end{array}\end{array} \\
= & D\,[t] & \\
\triangleright_\eta & \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ x_1 \to D\,[\sigma_1\ x_1]\\ \sigma_2\ x_2 \to D\,[\sigma_2\ x_2]\end{array} & \\
\triangleright_\beta\,\triangleright_\beta & \mathtt{match}\ t\ \mathtt{with}\ \middle|\ \begin{array}{l}\sigma_1\ x_1 \to (\sigma_1\ x_1, \sigma_2\ x_2)\\ \sigma_2\ x_2 \to (\sigma_1\ x_1, \sigma_2\ x_2)\end{array} &
\end{array}
$$

To resolve this difference between the equational theory of $(\approx_{\beta\eta})$ and $(\approx_{\mathtt{weak}\,\eta\cup\mathtt{extr}})$, we need to add the *merging* rules of Figure 3.9.

Extrusion of absurdity We have to add in Figure 3.10 a last case to our notion of extrusion, corresponding to the extrusion of an absurdity $\mathtt{absurd}(t)$ out any context.

Figure 3.9.: Extrusion of sum elimination: merging rules

$$\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ z_1 \to r_1 \mid \sigma_2\ z_2 \to r_2 \quad \triangleright_{\texttt{extr}}$$
$$\texttt{match}\ t\ \texttt{with} \mid \sigma_2\ y_2 \to u$$

$$\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to r_1[y_1/z_1] \mid \sigma_2\ y_2 \to u$$

$$\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to u \mid \sigma_2\ y_2 \to \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ z_1 \to r_1 \mid \sigma_2\ z_2 \to r_2 \quad \triangleright_{\texttt{extr}}$$

$$\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to u \mid \sigma_2\ y_2 \to r_2[y_2/z_2]$$

Figure 3.10.: Extrusion out of absurdity

$$\lambda x.\, \texttt{absurd}(t) \triangleright_{\texttt{extr}} \texttt{absurd}(t) \qquad t\ \texttt{absurd}(u) \triangleright_{\texttt{extr}} \texttt{absurd}(u)$$

$$\texttt{absurd}(t)\ u \triangleright_{\texttt{extr}} \texttt{absurd}(t) \qquad (t_1, \texttt{absurd}(t_2)) \triangleright_{\texttt{extr}} \texttt{absurd}(t_2) \quad \text{(and symmetric)}$$

$$\pi_i\ \texttt{absurd}(t) \triangleright_{\texttt{extr}} \texttt{absurd}(t) \qquad \sigma_i\ \texttt{absurd}(t) \triangleright_{\texttt{extr}} \texttt{absurd}(t)$$

$$\texttt{match}\ \texttt{absurd}(t)\ \texttt{with} \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2 \quad \triangleright_{\texttt{extr}} \texttt{absurd}(t)$$

$$\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ x_1 \to \texttt{absurd}(u_1) \mid \sigma_2\ x_2 \to u_2 \quad \triangleright_{\texttt{extr}} \texttt{absurd}(u_1) \quad \text{(and symmetric)}$$

$$\texttt{absurd}(t) \approx_{\texttt{extr}} \texttt{absurd}(u) \qquad () \approx_{\texttt{extr}} \texttt{absurd}(t) \qquad x \approx_{\texttt{extr}} \texttt{absurd}(t)$$

Formal results

Lemma 3.3.1 (Soundness of $(\approx_{\texttt{extr}})$)**.**
If $t \approx_{\texttt{extr}} u$ *then* $t \approx_{\beta\eta} u$.

Proof sketch. All cases work as in the example we gave earlier: apply a first strong η-reduction step (with the context being the whole term with occurrences of the extruded term replaced by holes), then β-reduce the result. ⊔

Theorem 3.3.2 (Completeness of $(\approx_{\texttt{extr}})$)**.**
For any $C\,[x]$ *and* t *of sum type, we have*

$$C\left[\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to \sigma_1\ y_1 \mid \sigma_2\ y_2 \to \sigma_2\ y_2\right] \quad \to^*_{\texttt{extr}} \quad \texttt{match}\ t\ \texttt{with} \mid \sigma_1\ y_1 \to C\,[\sigma_1\ y_1] \mid \sigma_2\ y_2 \to C\,[\sigma_2\ y_2]$$

For any $C\,[x]$ *and* t *of empty type, we have*

$$C\,[\texttt{absurd}(t)] \quad \to^*_{\texttt{extr}} \quad \texttt{absurd}(t)$$

Proof sketch. By induction on the context $C\,[x]$. If C has several subcontexts, we use as many merging rules. If C is a constant or a variable distinct from x, we use extrusion out of a constant context. ⊔

Corollary 3.3.3 (Completeness of extrusion)**.**
Strong η*-equivalence* $(\approx_\eta)$ *is equal to the congruent union of weak* η*-equivalence* $(\approx_{\texttt{weak}\,\eta})$ *and extrusion equivalence* $(\approx_{\texttt{extr}})$.

Definition 3.3.1 Standard extruded form (for λ-terms)**.**
A well-typed term t is in *standard extruded form* if no case-splits on sum or empty types

$$(\texttt{match}\ u\ \texttt{with}\ \mid \sigma_1\ x_1 \to \ldots \mid \sigma_2\ x_2 \to \ldots) \qquad \texttt{absurd}(u)$$

ever appears as the eliminated subterm of an elimination form. In other words, they may only appear

1. at the root of t, or
2. as a subterm of an introduction form, or
3. as a case of another case-split, or
4. as the argument of a function application

Theorem 3.3.4 (Standardization by extrusion)**.**
Any well-typed term t is $(\approx_{\texttt{extr}})$-equivalent to a standard extruded form.

Proof. This is immediate by inspection of the extrusion rules: as long as none of those conditions are met, we can extrude the case-split higher in the term. The only non-extrusible cases are the binding constructions (case-split or λ-abstractions) or the root. □

Remark 3.3.2. Our notion of "standard extruded form" does not attempt to be as strict as possible; we may further rule out case-splits that appear as subterms of any construction that does not bind variables (as they may always be extruded upward in this case), and even reason on the dependencies between these variables and the case splits.

The design requirement for standard extrusion form is rather that there are no hidden β-redexes: further extrusions may be possible, but they will not uncover additional β-redexes. This weaker notion suffices for this, as we show in the rest of this section.

We describe stronger notions of normal forms in Chapter 10 (Focused λ-calculus). ∗

Definition 3.3.2 Extruding reduction**.**
We define the *extruding reduction* relation as the relation $(\triangleright^*_{\texttt{extr}}\ \triangleright_\beta)$.

Definition 3.3.3 Extruded normal form**.**
A normal form for extruding reduction is called an *extruded normal form.*

Lemma 3.3.5.
A term t can perform a step of extruding reduction if and only if its standard extruded form can perform a β-reduction step.

Proof. The "if" direction is immediate, as the standard extruded form is reached by extrusion – Theorem 3.3.4 (Standardization by extrusion). In the "only if" relation, the difficulty comes from the fact that the extrusion rewriting is not normalizing: there may be several different choice of extrusions of t, some allowing more β-steps than others.

Let us prove that, if a standard extruded form is β-normal, then no directed extrusion step[3], backward or forward, may create a β-redex. A β-redex could appear if a case-split on a sum was blocking a beta-redex:

$$\pi_i \left(\texttt{match}\ t\ \texttt{with} \;\middle|\; \begin{array}{l}\sigma_1\ x_1 \to (u_1, r_1)\\ \sigma_2\ x_2 \to (u_2, r_2)\end{array}\right) \qquad \left(\texttt{match}\ t\ \texttt{with} \;\middle|\; \begin{array}{l}\sigma_1\ x_1 \to \lambda y_1.\, u_1\\ \sigma_2\ x_2 \to \lambda y_2.\, u_2\end{array}\right) r$$

However, none of these sub-terms may happen in a simplified form. □

Corollary 3.3.6 (Standard extruded β-normal forms are extruded normal forms)**.**
Terms that are both β-normal and in standard extruded form are extruded normal forms.

[3]The extrusions out of constant contexts, included in $(\approx_{\texttt{extr}})$ but excluded from $(\triangleright_{\texttt{extr}})$, can create β-redexes as it let us introduce arbitrary subterms of sum or empty type.

3.3.2. Normalization and consistency for $\mathsf{PIL}(\to, \times, 1, +, 0)$

As an example of the interest of studying those commuting conversions, the notion of standard extruded normal forms brings us the missing piece to prove $(\triangleright_R)$-normalization of proofs in presence of disjunction, which gives consistency for the full logic $\mathsf{PIL}(\to, \times, 1, +, 0)$.

In our consistency proof by normalization in Section 1.4 (Proving consistency (without disjunctions) by normalization), we used a specific normalization strategy with the following general structure:

$$\Pi \to_R \Pi_1 \to_R \Pi_2 \to_R \dots \to_R \Pi' \not\to_R$$

In this section, we want to ensure that the proof we reduce are in (the proof-derivation equivalent of) standard extruded form. We will perform extrusion steps before each reduction step, giving the following structure:

$$\Pi \to^*_{\mathtt{extr}} \Pi_0 \to_R \to^*_{\mathtt{extr}} \Pi_1 \to_R \to^*_{\mathtt{extr}} \Pi_2 \to_R \to^*_{\mathtt{extr}} \dots \to_R \to^*_{\mathtt{extr}} \Pi' \not\to_R$$

In particular, we choose the resulting normal proof Π' to also be in standard extruded form: disjunction or empty eliminations are at the root of the proof, or in a case of another disjunction elimination, or immediately after an implication or conjunction introduction.

The $(\to_{\mathtt{extr}})$ relation is not normalizing, so we need a particular reduction strategy. We will use the same as in Theorem 3.3.4 (Standardization by extrusion); in particular, we do not use the extrusions out of constant contexts, so no new proof fragments appear in the proof – but existing subtrees may be duplicated by extrusion.

The complexity measure on proofs used in Section 1.4.4 (Weak normalization) is straightforwardly extended to the full logic $\mathsf{PIL}(\to, \times, 1, +, 0)$, defining $\|A_1 + A_2\| \stackrel{\mathsf{def}}{=} 1+\max(\|A_1\|, \|A_2\|)$.

The fact that extrusion duplicates subterms means that extruding disjunction eliminations can increase the complexity measure of proofs; but if we consider the $(\to_R)$-reduction and the $(\to_{\mathtt{extr}})$-reductions together, we can show that the complexity measure decreases overall, as the formulas duplicated by extrusion are strictly simpler than the one removed by reduction.

Lemma 3.3.7.
For any relation $\Pi \to_R \Pi' \to^*_{\mathtt{extr}} \Pi''$ *such that*

- Π *is in standard extruded form*
- *the* $(\triangleright_R)$*-reduction is on an elimination-introduction pair of maximal complexity that contains no maximal pair in its sub-derivations*
- *the* $(\triangleright_{\mathtt{extr}})$*-steps only extrude through non-constant contexts*

we have $\|\Pi\| > \|\Pi''\|$.

Proof. The proof follows the same structure as the weak normalization proof in absence of disjunction: we reason on the new elimination-introduction pairs introduced by the transformation.

Implication reduction

$$\dfrac{\dfrac{\Pi_B :: \Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B} \quad \triangleright_R \quad \dfrac{\Pi_B :: \Gamma, A \vdash B \qquad \Pi_A :: \Gamma \vdash A}{\Gamma \vdash B}\ \text{SUBST}$$

New pairs formed by this reduction may be on the types A or B, which are strictly smaller than the formula $A \to B$ eliminated by the reduction.

Substitution and the subsequent extrusions may duplicate elimination-introduction pairs present in sub-derivations of the reduced derivation, but those are assumed to be of strictly smaller complexity.

Finally, note that extrusion may create new elimination-introduction pairs. Consider the following example (the Γ, A, B below are unrelated to those used in the reduction rule above):

$$\dfrac{\dfrac{\Gamma \vdash C_1 + C_2 \qquad \dfrac{\dfrac{\Gamma, C_1, A \vdash B}{\Gamma, C_1 \vdash A \to B} \qquad \Gamma, C_2 \vdash A \to B}{\Gamma \vdash A \to B} \qquad \Gamma \vdash A}{\Gamma \vdash \left(\begin{array}{l|l} \texttt{match}\ r\ \texttt{with} & \sigma_1\ x_1 \to \lambda x.\, t \\ & \sigma_2\ x_2 \to t' \end{array}\right) u : B}$$

$$\triangleright_{\texttt{extr}} \qquad \dfrac{\dfrac{\Gamma \vdash C_1 + C_2 \qquad \dfrac{\dfrac{\Gamma, C_1, A \vdash B}{\Gamma, C_1 \vdash A \to B} \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \dfrac{\Gamma, C_2 \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}}{\Gamma \vdash \texttt{match}\ r\ \texttt{with} \left|\begin{array}{l} \sigma_1\ x_1 \to (\lambda x.\, t)\ u \\ \sigma_2\ x_2 \to t'\ u \end{array}\right. : B}$$

In this example, an implication elimination and introduction that were separated by a disjunction elimination form a new pair after the disjunction elimination is extruded. We informally call "hidden pair" two matching introduction-elimination rules separated by disjunction eliminations that can be extruded.

In the general case, the formula of the new pair $A \to B$ may be unrelated to the previous introduction-elimination pairs of the term, and thus strictly increase the proof's complexity measure. However, notice that all reduction steps in this proof are performed on terms that are already in standard extruded form; in particular, there was no such "hidden pair" in the term before reduction – Corollary 3.3.6 (Standard extruded β-normal forms are extruded normal forms).

If a new "hidden pair" appears after the reduction step, it means that either the introduction or the elimination rule is new in the term. It must come from one of the sub-proofs substituted by the reduction. In the case of a reduction of an implication $A \to B$, this means that the new pairs produced are either on A or on B: the new proof (after extrusion) is still of strictly smaller measure than the pre-reduction proof.

The same reasoning on "hidden pairs" will apply in the two other reduction cases.

Conjunction reduction

$$\dfrac{\dfrac{\Pi_1 :: \Gamma \vdash A_1 \qquad \Pi_2 :: \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}}{\Gamma \vdash A_i} \qquad \triangleright_R \qquad \Pi_i :: \Gamma \vdash A_i$$

It is immediate that the new proof is of strictly smaller complexity as we only removed sub-proofs and one occurrence of the formula $A_1 \times A_2$.

Disjunction reduction

$$\dfrac{\dfrac{\Pi :: \Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} \qquad \Pi_1 :: \Gamma, A_1 \vdash C \qquad \Pi_2 :: \Gamma, A_2 \vdash C}{\Gamma \vdash C} \qquad \triangleright_R$$

$$\dfrac{\Pi_i :: \Gamma, A_i \vdash C \qquad \Pi :: \Gamma \vdash A_i}{\Gamma \vdash C}\ \text{SUBST}$$

New elimination-introduction pairs may come from the substitution of $\Pi :: \Gamma \vdash A_i$, creating a new pair on A_i if the last rule of Π_A is an introduction. This new pair is strictly simpler.

If the last rule of Π_i is an introduction, and the simplification is above an elimination rule, we may also have a new pair on C. In the previous proof, this justified removing

sums altogether: there is no link between $\|A_1 + A_2\|$ and $\|C\|$ justifying a strict decrease in complexity.

However, we now assume that the original proof is in simplified form. In particular, it is not possible for the reduced sub-proof to bring C in eliminable position: it is either at the root of the formula, a premise of an introduction rule, or a non-eliminated premise of an elimination rule.

We can thus conclude that all new elimination-introduction pairs are on A_i – those introduced by substitution, and those "hidden" after substitutions that are uncovered by extrusion. □

Corollary 3.3.8 (Weak reduction for $\mathsf{PIL}(\to, \times, 1, +, 0)$ natural deduction)**.**
Any $\mathsf{PIL}(\to, \times, 1, +, 0)$ *proof can be rewritten in* $(\triangleright_R)$*-normal form in a finite number of steps.*

Proof. Because we can choose each transformation step (reduction then extrusions) to strictly decrease the measure, and the measure is well-founded (no infinite descending chains), we reach an irreducible proof in a finite number of steps. □

Theorem 3.3.9 (Consistency of $\mathsf{PIL}(\to, \times, 1, +, 0)$)**.**
There is no valid $\mathsf{PIL}(\to, \times, 1, +, 0)$ *proof of* $\emptyset \vdash 0$*.*

4. A better proof system: sequent calculus

Historical context

Natural deduction was first defined by Gerhard Gentzen during his PhD research, in 1932. He was trying to formalize mathematical proofs as mathematical objects themselves, in a way that would be closer to the actual practice of mathematicians than Hilbert-style systems relying heavily on axioms and tautologies (we briefly mentioned Hilbert-style systems in Section 1.2.4).

Remark 4.0.3. The syntax of natural deduction proofs at this time did not use an assumption context Γ, it instead relied on a more arcane mechanism of "discharged assumptions" (I would give an example but cannot find a LaTeX package for this), where assumptions (mere propositions without context) at the leaves of the proof are "crossed out" during the downward construction of the derivation. Some people still use that style, but it should be avoided as it is much less convenient to manipulate than explicit context passing. It is slightly less verbose as contexts do not have to be repeated in each judgment, but if you want less verbose you should rather use λ-terms directly. ∗

Gentzen remarked that this presentation of natural deduction naturally resulting in a formal structure of proof for *intuitionistic* logic, while he was originally looking for a structure of proofs of *classical* logics that common mathematical practice rather uses. Note that the interest of intuitionistic logic was well-understood at the time and it was an active topic in logic; notably, Kurt Gödel and Gerhard Gentzen independently developed a translation from classical to intuitionistic arithmetic.

Sequent calculus arose as a solution to this problem of finding structural rules for classical logic. It represents a different point of view than natural deduction, more symmetric (and in particular more adapted to proof search), but has a slightly more complex notion of reduction. The problem it raises, namely commuting conversions, are interesting even in intuitionistic logic and in particular are central to the difficulty raised by sums.

4.1. Intuitionistic sequent calculus

In this section we will only discuss the intuitionistic sequent calculus. Classical logic will be discussed in Section 4.3.

Remark 4.1.1. I personally prefer to reserve the name *calculus* for term syntaxes equipped with a computational behavior (this is what allows to *run*, *compute* them, or *calculate* with them), and thus find the name *sequent calculus* slightly unfortunate, but it has always been named this way. ∗

Credits I was unfortunately never taught the history of ideas of our field in university courses; I discovered it through the "Groupe de travail de Logique", during lectures of Marc Bagnol and Maël Pégny. The online Stanford Encyclopedia of Philosophy is a trove of information on the history of logic in mathematics and computer science, and the article The Development of Proof Theory, by Jan von Plato, last revised in 2014, has many more details on the development of Gerhard Gentzen's work.

4.1.1. Left introduction rules

We presented the rules for logical connectives in natural deduction proofs, structured as introduction and elimination rules (§1.2). For the implication, for example, natural

deduction has:

$$\textsc{nd-and-elim}\ \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad\qquad \textsc{nd-and-intro}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

Elimination rules tell you how to *use* a complete proof of a connective to build new proofs (rootward). The sequent calculus uses *left-introduction* rules instead, that tell you how to *consume* a hypothesis present in context to prove your goal (leafward). The existing introduction rule is unchanged in sequent calculus (it is called "right-introduction"), and the left-introduction rule for implication is as follows:

$$\textsc{seq-and-left}\ \frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \qquad\qquad \textsc{seq-and-right}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

Suppose you have assumed an implication $A \to B$; how could you "consume" it to progress in your proof of some goal C? Well, if you can prove that A holds (left premise), then you can use B to prove your goal (right premise).

Remark 4.1.2. Note that, as for natural deduction, the comma notation for logical contexts Γ, A represents the *non*-disjoint union of sets: A may already be in Γ. In particular, in the rule SEQ-AND-LEFT presented above, having $\Gamma, A \to B$ in conclusion does not mean that $A \to B$ has been removed in the premises that use Γ only as context; $A \to B$ may or may not be present in the premises context, depending on how we decide to apply the inference rule. ∗

Elimination and left-introduction rules are read in opposite direction (rootward and leafward, respectively). As a consequence, natural deduction and sequent calculus proofs of the same judgment often look like one is the "upside down" version of the other. Compare for example those two proofs of the double-implication judgment $A \to B \to C, A, B \vdash C$ below. For readability, we have greyed out in each judgment the formulas (in context or goal) that are not used in the inference rule of the judgment.

$$\dfrac{\dfrac{\dfrac{}{A \to B \to C, A, B \vdash A \to B \to C} \qquad \dfrac{}{A \to B \to C, A, B \vdash A}}{A \to B \to C, A, B \vdash B \to C} \qquad \dfrac{}{A \to B \to C, A, B \vdash B}}{A \to B \to C, A, B \vdash C}$$

$$\dfrac{\dfrac{}{A \to B \to C, A, B \vdash A} \qquad \dfrac{\dfrac{}{A \to B \to C, A, B, B \to C \vdash B} \qquad \dfrac{}{A \to B \to C, A, B, B \to C, C \vdash C}}{A \to B \to C, A, B, B \to C \vdash C}}{A \to B \to C, A, B \vdash C}$$

In the natural deduction proof, the function $A \to B \to C$ is eliminated at the very leaf of the proof, and the goal C is used at the very root of the proof. In the sequent calculus proof, the function $A \to B \to C$ is left-introduced at the very root of the proof, and the goal C is only used in an axiom rule at one leaf of the proof. Those two derivations are, in a sense, "upside down" of each other.

The full rules of the sequent calculus, including the left-introduction rules for the other connectives, are given in §4.1.3.

4.1.2. Cut rule

In sequent calculus, how can one reuse an existing proof of some judgment $\Gamma \vdash A$ to build a larger proof, using the knowledge of A? In natural deduction, this is done by using elimination rules: they tell us precisely how to reuse a proof in a larger derivation. In the sequent calculus, right introduction rules let us construct results, left introduction rules let us deconstruct hypotheses, but there is no rule to turn a result into a hypothesis, which is what we need to reuse existing proofs – get them as hypothesis in our context, so that

we can manipulate them through left-introduction rules. Remark that the axiom rules, present in both systems, does the opposite: it turns a hypothesis into a proved result.

While reuse is allowed by elimination rules in natural deduction, the sequent calculus needs an additional rule, the *cut rule*, for this purpose:

$$\dfrac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ \text{SEQ-CUT}$$

This rule let us turn a proof of A, the left premise, into an hypothesis usable in the proof of B in the right premise. To see this rule in action, suppose we are given an arbitrary proof Π of an implication, $\Pi :: \Gamma \vdash A \to B$, which we want to reuse, along with a proof $\Pi_A :: \Gamma \vdash A$. We can reuse Π and Π_A to prove B as follows:

$$\dfrac{\Pi :: \Gamma \vdash A \to B \qquad \dfrac{\Pi_A :: \Gamma \vdash A \qquad \overline{\Gamma, A, B \vdash B}}{\Gamma, A \to B \vdash B}\ \text{SEQ-CUT}}{\Gamma \vdash B}\ \text{SEQ-CUT}$$

The use of the cut rule is crucial to turn the result of Π into a hypothesis on which left-introduction rule can be used. Note that we actually proved that the implication-elimination rule of natural deduction is admissible in the sequent calculus, using the cut rule. In §4.2.2 we will show that any proof of either logic can be translated into the other: in terms of provability, natural deduction and sequent calculus really model "the same logic" $\mathsf{PIL}(\to, \times, 1, +, 0)$.

4.1.3. $\mathsf{PIL}(\to, \times, 1, +, 0)$ in sequent style

The complete rules of the sequent calculus presentation of $\mathsf{PIL}(\to, \times, 1, +, 0)$ are given in Figure 4.1.

Remark 4.1.3. The reader may wonder why the elimination rule for pairs is

$$\dfrac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2 \vdash C}\ \text{SEQ-CONJ-LEFT}$$

instead of the arguably more natural

$$\dfrac{\Gamma, A, B \vdash C}{\Gamma, A \times B \vdash C}\ \text{SEQ-CONJ-LEFT-POS}$$

This choice corresponds to two different styles of "inspection" of pairs: SEQ-CONJ-LEFT uses projections, which return only one of the components of the pair, SEQ-CONJ-LEFT-POS uses pattern-matching (in term syntax, $\texttt{let}\ (x_1, x_2) = t\ \texttt{in}\ u$), adding both components at once to the environment. We chose the "projection" style here for consistency with our natural deduction system, which also eliminates pairs by projections – note that pattern-matching would be possible in natural deduction as well, with a different rule. The two visions of this connective are, of course, equivalent for intuitionistic logic and the λ-calculus, but they correspond to the choice of different "polarities" that influence proof search (products with projection are "negative", while products with pattern-matching are "positive") – we will discuss this further when presenting focusing in Chapter 7.

Those two rules are of course completely equivalent: projections are obtained from pattern-matching by shadowing, and pattern-matching from projections by contraction:

$$\dfrac{\dfrac{\Gamma, A_i \vdash C}{\Gamma, A_1, A_2 \vdash C}\ \text{WK}}{\Gamma, A_1 \times A_2 \vdash C}\ \text{SEQ-CONJ-LEFT-POS} \qquad \dfrac{\dfrac{\Gamma, A_1, A_2 \vdash C}{\Gamma, A_1 \times A_2, A_2 \vdash C}\ \text{SEQ-CONJ-LEFT}}{\Gamma, A_1 \times A_2, A_1 \times A_2 \vdash C}\ \text{SEQ-CONJ-LEFT}$$

Figure 4.1.: Sequent calculus presentation of $\mathsf{PIL}(\to, \times, 1, +, 0)$

$$\textsc{seq-axiom}\ \frac{}{\Gamma, A \vdash A}$$

$$\textsc{seq-cut}\ \frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}$$

$$\textsc{seq-impl-left}\ \frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \qquad \textsc{seq-impl-right}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\textsc{seq-conj-left}\ \frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2 \vdash C} \qquad \textsc{seq-conj-right}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B}$$

$$\textsc{seq-disj-left}\ \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A + B \vdash C} \qquad \textsc{seq-disj-right}\ \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}$$

$$\textsc{seq-true-right}\ \frac{}{\Gamma \vdash 1}$$

$$\textsc{seq-false-left}\ \frac{}{\Gamma, 0 \vdash A}$$

The weakening rule used in the derivation on the left is the exact correspondence of the weakening rule of natural deduction – see Lemma 1.3.1 (Weakening for $\mathsf{PIL}(\to, \times, 1, +, 0)$) – and is defined in the same way. ∗

Remark 4.1.4. In some sense the sequent calculus is more regular because it is a "least common denominator" system. In natural deduction (§1.2), the elimination rule for sums/disjunctions ND-DISJ-ELIM stands out of the other for introducing this type C that is external to the disjunction $A + B$ being eliminated – it feels heavier than other elimination rules. In the sequent calculus, all left-introduction rules have this extra type C in the goal; you could say that it is a regression, but this removes a discrepancy between the various connectives, and thus gives a more regular system.

Again, this irregularity can be explained through focusing (Chapter 7). Implications and products (with projections) have nice elimination rules in natural deduction because they are both "negative" connectives. Adding sums which are "positive" seems to introduce an irregularity, but it in fact reveals a fundamental phenomenon of logic and computation, occurring when both polarities are mixed.

One should be suspicious of claim that natural deduction (or term syntaxes in this style) is (much) simpler than sequent calculus (or term syntaxes in this style). This often comes from an incomplete study of the purely negative fragment of the system (forgetting about sums and idly hoping that they will be easy to add back afterwards). Making sure we add sums to the system we study keeps us honest, guarantees that our ideas will be more robust, and let us discover beautiful generalizations. ∗

4.1.4. A term syntax for the intuitionistic sequent calculus

Figure 4.2.: Terms of the sequent-form λ-calculus $\mathsf{S\Lambda C}(\to, \times, 1, +, 0)$

$$
\begin{array}{lll}
t, u, r ::= & & \text{terms} \\
\quad | \; x, y, z & & \text{variables} \\
\quad | \; \lambda x.\, t & & \text{abstraction} \\
\quad | \; \texttt{let}\ y = x\ t\ \texttt{in}\ u & & \text{left application} \\
\quad | \; (t, u) & & \text{pairs} \\
\quad | \; \texttt{let}\ y = \pi_i\ x\ \texttt{in}\ u & & \text{left projection} \\
\quad | \; \sigma_i\ t & & \text{injections} \\
\quad | \; \texttt{match}\ x\ \texttt{with} & \begin{array}{l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} & \text{left case split} \\
\quad | \; () & & \text{unit} \\
\quad | \; \texttt{absurd}(x) & & \text{absurd} \\
\quad | \; \texttt{let}\ x = t\ \texttt{in}\ u & & \text{cut}
\end{array}
$$

We provide in Figure 4.2 a small term syntax for the intuitionistic sequent calculus, which will make it easier to define its normalization process (cut-elimination) and study its relation with natural deduction in Section 4.2 (Reduction of sequent-calculus proofs). Because it is very close to λ-calculus $\mathsf{\Lambda C}$, call it the "sequent-form λ-calculus", $\mathsf{S\Lambda C}$.

The left-elimination rules are transcribed as `let`-binding construct with a slightly peculiar shape. They act on variables (note that the argument of a left-introduced function may be an arbitrary expression; in particular this is not an A-normal form).

Not all left rules use a `let`: sums and the empty type use the usual elimination construct, with their scrutinee restricted to be a variable.

Finally, we remark that there is no overlap between the various `let`-using rule: left rules such as $(\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ u)$ are not special cases of cuts $(\texttt{let}\ y = t\ \texttt{in}\ u)$, as $(\pi_i\ x)$ is not a valid expression t.

Remark 4.1.5. The beauty of the sequent calculus comes from its deep symmetries. This syntax is, on the contrary, not symmetric at all, and rather ugly. We chose it because it should be familiar to anyone knowing λ-calculus, which let us lower the accessibility barrier for defining manipulations of sequent terms. $*$

The correspondence between the term syntax and the typing rules is given in Figure 4.3. It is overall very natural, except for the use of $\Gamma \ni x : A$ instead of $\Gamma, x : A$ which should be explained in detail. $\Gamma \ni x : A$ means that the typing environment Γ, a mapping from variables to types, contains the mapping $x : A$.

For logical contexts (sets of formulas), the notation Γ, A denotes *non*-disjoint union : A may belong to the set Γ – see Remark 4.1.2. In typing environments (mappings from term variables to formulas), the comma notation $\Gamma, x : A$ denotes *disjoint* union on the contrary – see Remark 2.3.3. When writing $\Gamma \ni x : A$ instead, we insist that x *is* a binding of Γ, and thus that the subgoals using Γ also contain the binding for x. In other words, the two following rules are equivalent:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, y : A \vdash u : C}{\Gamma \ni x : A \to B \vdash \texttt{let}\ y = x\ t\ \texttt{in}\ u : C}$$

$$\frac{\Gamma, x : A \to B \vdash t : A \qquad \Gamma, x : A \to B, y : A \vdash u : C}{\Gamma, x : A \to B \vdash \texttt{let}\ y = x\ t\ \texttt{in}\ u : C}$$

We preferred the notation $\ni$ because it is more concise and closer to the corresponding rule SEQ-IMPL-LEFT of the sequent calculus (as an intuitionistic logic, that is, mostly unconcerned with matters of multiple variable usage).

Figure 4.3.: Typing rules of the sequent-form λ-calculus $\mathsf{S}\Lambda\mathsf{C}(\to, \times, 1, +, 0)$

$$\text{SLC-VAR}\quad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\text{SLC-CUT}\quad \frac{\Gamma \vdash t : A \qquad \Gamma, x : t \vdash u : B}{\Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ u : B}$$

$$\text{SLC-FUN-LEFT}\quad \frac{\Gamma \vdash t : A \qquad \Gamma, y : B \vdash u : C}{\Gamma \ni x : A \to B \vdash \texttt{let}\ y = x\ t\ \texttt{in}\ u : C}$$

$$\text{SLC-FUN-RIGHT}\quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B}$$

$$\text{SLC-PROD-LEFT}\quad \frac{\Gamma, y : A_i \vdash u : C}{\Gamma \ni x : A_1 \times A_2 \vdash \texttt{let}\ y = \pi_i\ x\ \texttt{in}\ u : C}$$

$$\text{SLC-PROD-RIGHT}\quad \frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2}$$

$$\text{SLC-SUM-LEFT}\quad \frac{\Gamma, y_1 : A_1 \vdash u_1 : C \qquad \Gamma, y_2 : A_2 \vdash u_2 : C}{\Gamma \ni x : A_1 + A_2 \vdash \texttt{match}\ x\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2\end{array}\right. : C}$$

$$\text{SLC-SUM-RIGHT}\quad \frac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2}$$

(no elimination rule for 1)

$$\text{SLC-UNIT-RIGHT}\quad \frac{}{\Gamma \vdash () : 1}$$

$$\text{SLC-EMPTY-LEFT}\quad \frac{}{\Gamma \ni x : 0 \vdash \texttt{absurd}(x) : C}$$

(no introduction rule for 0)

The alternative would be to write a different rule, namely

$$\text{SEQ-FUN-LEFT-NO-CONTRACTION}\quad \frac{\Gamma \vdash t : A \qquad \Gamma, y : B \vdash u : C}{\Gamma, x : A \to B \vdash \texttt{let}\ y = x\ t\ \texttt{in}\ u : C}$$

In this rule, the variable x is *not* available to the premises anymore. This is displeasing from a syntactic point of view, because it breaks the excepted rules for variable scoping: a variable defined by a `let` may suddenly become unavailable after it has been "consumed" by some left-introduction rule, without much marking in the syntax that this is happening. (Variables going out of scope is an interesting and useful phenomenon, but we should not reuse an existing syntax for it, `let`, that has a different meaning.)

More fundamentally: in the case of the cut-free sequent calculus, replacing the left-introduction rule for functions SEQ-FUN-LEFT with the rule SEQ-FUN-LEFT-NO-CONTRACTION gives a strictly weaker system that can prove less properties (some types that are inhabited in $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$ are uninhabited in this system). For example, consider the standard abbreviation $\neg A \stackrel{\text{def}}{=} A \to 0$, and the type $\neg(\neg(A + \neg A))$. It is inhabited by the following (admittedly hard to follow) program:

$$\lambda(f : \neg(A + \neg A)).\, \texttt{let}\ (x : 0) = f\ (\sigma_2\ \lambda(a : A).\, \texttt{let}\ y = f\ (\sigma_1\ a)\ \texttt{in}\ y)\ \texttt{in}\ x$$

Notice that the bound variable f is left-introduced twice in this program – once with a parameter of the form $\sigma_2\ \_$ (we claim to provide a $\neg A$ to f), once with a parameter of the form $\sigma_1\ a$ (we finally decide to provide a A). It would not be well-typed if we used SEQ-FUN-LEFT-NO-CONTRACTION instead of the rule SEQ-FUN-LEFT allowing variable reuse. In fact, we can check by exhaustive search, if we impose the use of SEQ-LEFT-NO-CONTRACTION, there is no well-typed *cut-free* program at this type.

Remark 4.1.6. This choice of formula is of course not arbitrary. $A + \neg A$ is the "excluded middle" formula for A (either A is true, or its negation is true), which characterizes classical logic and is unproveable in intuitionistic logic. Now, for any (quantifier-free) formula C provable in classical logic, it is a (non-trivial) theorem that its double-negation $\neg\neg C$ is provable in intuitionistic logic. It is a lesser-known fact that this proof will involve a contraction of a singly-negated hypothesis (for example $\neg C$) in an essential way if C cannot already be proved intuitionistically. We will discuss this in more details in Section 4.3. ∗

On the other hand, the cut rule allows duplicating a variable (by simply cutting on it). The following contraction rule (§1.2.3) is admissible in presence of SLC-CUT:

$$\frac{\Gamma, x : A, y : A \vdash B}{\Gamma, x : A \vdash B}\ \text{SLC-CONTR} \qquad \stackrel{\text{def}}{=} \qquad \frac{\dfrac{}{\Gamma, x : A \vdash x : A} \qquad \Gamma, x : A, y : A \vdash B}{\Gamma, x : A \vdash \texttt{let}\ y = x\ \texttt{in}\ t : B}\ \text{SLC-CUT}$$

The fact that a proof can be written with cuts and cannot be written without cuts means that, if we had used SEQ-FUN-LEFT-NO-CONTRACTION, we would be unable to perform cut-elimination! We would have to use an explicit rule for contraction, and then we could eliminate all cuts (and reduce variable-variable cuts to contractions).

On the contrary, the rule using $\Gamma \ni x : A \to B$ enjoys cut-elimination (as shown in the next section §4.2.1). By default, all left-introduction rules introduce implicit contractions. Focusing (Chapter 7) will again restrict the places where implicit contraction occurs. Note that if, in a particular proof, you wish to be more "in the spirit of sequent-calculus", with limited use of contraction, it is always possible to *not* use an implicit contraction using the following trick:

$$\frac{\Gamma, x : A_i \vdash u : B}{\Gamma, x : A_1 \times A_2 \vdash \texttt{let}\ x = \pi_i\ x\ \texttt{in}\ u : B}$$

By shadowing the old x variable with the result of the projection, we make sure that the proof term u cannot access the old x again: for this particular choice of new variable, there is no contraction. Interestingly, in presence of shadowing the function rule becomes:

$$\frac{\Gamma, x : A \to B \vdash t : A \qquad \Gamma, x : B \vdash u : C}{\Gamma, x : A \to B \vdash \texttt{let}\ x = x\ t\ \texttt{in}\ u : C}$$

With this rule, the function is still available in the left premise, but it is shadowed in the right one. This restriction let us write the program of type $\neg\neg(A + \neg A)$ below, and in fact one can show that it is complete for provability (all formulas that were provable with the full rule remain provable); this observation is the basis of so-called "contraction-free logics" introduced by Vorob'ev [Vorob'ev, 1958] and studied in particular by Dyckhoff [Dyckhoff, 1992, 2013].

Shadowing seems a rather superficial phenomenon, and I find surprising that it becomes interesting in this use-case (controlling contraction in a system that seems to impose contractions everywhere).

4.2. Reduction of sequent-calculus proofs

4.2.1. Normal sequent proofs: cut-elimination

Now that we have a term syntax, it is easy to define cut-elimination, as terms guide the intuition of what the computational behavior should be. This process is a bit more complex than in natural deduction, because cut rules can happen in any part of a sequent calculus proof, while natural deduction only reduces when an elimination rule encounters

a matching introduction rule, which is a more structural condition. We separate the reduction rules in three families: principal cases, initial cases, and commutative cases. We write ($\triangleright_{\mathtt{RP}}$), ($\triangleright_{\mathtt{RI}}$) and ($\triangleright_{\mathtt{RC}}$) for these relations, and ($\triangleright_{\mathtt{R}}$) for their union.

Remark 4.2.1. As in previous definition of equivalence or rewriting relations over well-typed terms – see Remark 3.3.1 – we implicitly restrict our relations to preserve well-typing and well-scoping. For example we will have the following rule

$$\mathtt{let}\ x = t\ \mathtt{in}\ \lambda y.\, u \qquad \triangleright_{\mathtt{RC}} \qquad \lambda y.\, \mathtt{let}\ x = t\ \mathtt{in}\ u$$

which implicitly assumes that y is not a free variable in t, as otherwise it would be captured by rewritten binder $\lambda y.\, \_$. $*$

The *principal cases* ($\triangleright_{\mathtt{RP}}$) correspond to elimination/introduction pairs of natural deduction; they occur when a right-introduction rule is cut over a left-introduction rule for the same variable – this is where the real computation happens. Note that because contraction is implicit, the cut variable x may always be used in latter parts of the term, so it does not disappear after the cut. The term is still simpler after reduction than before: the right-introduction rule on a connective $A_1 \times A_2$, $A_1 + A_2$ or $A \to B$ is replaced by cuts on strictly simpler formulas, A_1 or A_2, or A and B.

$$\mathtt{let}\ (x : A_1 \times A_2) = (t_1, t_2)\ \mathtt{in}\ (\mathtt{let}\ y = \pi_i\ x\ \mathtt{in}\ r)$$

$$\triangleright_{\mathtt{RP}} \qquad \mathtt{let}\ (x : A_1 \times A_2) = (t_1, t_2)\ \mathtt{in}\ (\mathtt{let}\ (y : A_i) = t_i\ \mathtt{in}\ r)$$

$$\mathtt{let}\ (x : A_1 + A_2) = \sigma_i\ t\ \mathtt{in}\ \mathtt{match}\ x\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ y_1 \to r_1 \\ \sigma_2\ y_2 \to r_2 \end{array}$$

$$\triangleright_{\mathtt{RP}} \qquad \mathtt{let}\ (x : A_1 + A_2) = \sigma_i\ t\ \mathtt{in}\ (\mathtt{let}\ (y_i : A_i) = t\ \mathtt{in}\ r_i)$$

$$\mathtt{let}\ (x : A \to B) = \lambda y.\, t\ \mathtt{in}\ (\mathtt{let}\ z = x\ u\ \mathtt{in}\ r)$$

$$\triangleright_{\mathtt{RP}} \qquad \mathtt{let}\ (x : A \to B) = \lambda y.\, t\ \mathtt{in}\ (\mathtt{let}\ (z : B) = (\mathtt{let}\ (y : A) = u\ \mathtt{in}\ t)\ \mathtt{in}\ r)$$

In the principal case for functions/implications, we have not used a substitution as in the natural deduction, but a cut instead. This means that the propagation of the environment is more local than in natural deduction: cuts are playing the role of explicit substitutions [Kesner, 2007]. Note that there is no case for 1 or 0 as they lack either a right- or left-introduction rule.

We also handle cuts on mere variables – the *initial cases* ($\triangleright_{\mathtt{RI}}$). There are in fact two cases: either the formula to the right of the cut is a variable (the body of the `let`), or the formula on the left of the cut is a variable (the definition of the `let`); the latter case corresponds to a form of contraction, as we have already noted, and it reduces to a variable-variable substitution – note that replacing a variable by a *term* would be invalid in general, if the variable is used in a left-introduction. In all cases, the cut disappears completely.

$$\mathtt{let}\ x = t\ \mathtt{in}\ x \qquad \triangleright_{\mathtt{RI}} \qquad t$$

$$\mathtt{let}\ x = t\ \mathtt{in}\ y \qquad \triangleright_{\mathtt{RI}} \qquad y$$

$$\mathtt{let}\ x = y\ \mathtt{in}\ t \qquad \triangleright_{\mathtt{RI}} \qquad t[y/x]$$

Finally, we come to *commutative cases* ($\triangleright_{\mathtt{RC}}$): neither sides of the cut is a variable, but neither are they matching left- and right-introduction rules. This relation is defined as the union of three relations ($\triangleright_{\mathtt{RCll}}$), ($\triangleright_{\mathtt{RCrr}}$) and ($\triangleright_{\mathtt{RCrl}}$): commutative cases can happen if *let*-definition starts with a left rule (instead of a right rule ready to match a principal case)

($\triangleright_{\texttt{RCll}}$), if the `let`-body is a right rule (instead of a left rule ready to match a principal case) ($\triangleright_{\texttt{RCrr}}$), and if the `let`-body is a left rule on a different variable than the cut variable ($\triangleright_{\texttt{RCrl}}$).[1] In all those cases, we can propagate the cut to strict subterms of the definition or body.

$$\texttt{let } x = t \texttt{ in } \lambda y.\, u \quad \triangleright_{\texttt{RCrr}} \quad \lambda y.\, \texttt{let } x = t \texttt{ in } u$$

$$\texttt{let } x = t \texttt{ in } (u_1, u_2) \quad \triangleright_{\texttt{RCrr}} \quad (\texttt{let } x = t \texttt{ in } u_1, \texttt{let } x = t \texttt{ in } u_2)$$

$$\texttt{let } x = t \texttt{ in } \sigma_i\, u \quad \triangleright_{\texttt{RCrr}} \quad \sigma_i\, (\texttt{let } x = t \texttt{ in } u)$$

$$\texttt{let } x = t \texttt{ in } () \quad \triangleright_{\texttt{RCrr}} \quad ()$$

$$\texttt{let } x = (\texttt{let } y = z\, t \texttt{ in } u) \texttt{ in } r \quad \triangleright_{\texttt{RCll}} \quad \texttt{let } y = z\, t \texttt{ in } (\texttt{let } x = u \texttt{ in } r)$$

$$\texttt{let } x = (\texttt{let } y = \pi_i\, z \texttt{ in } u) \texttt{ in } r \quad \triangleright_{\texttt{RCll}} \quad \texttt{let } y = \pi_i\, z \texttt{ in } (\texttt{let } x = u \texttt{ in } r)$$

$$\texttt{let } x = \texttt{match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\, z_1 \to u_1 \\ \sigma_2\, z_2 \to u_2 \end{array} \right. \texttt{ in } r$$

$$\triangleright_{\texttt{RCll}} \quad \texttt{match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\, z_1 \to \texttt{let } x = u_1 \texttt{ in } r \\ \sigma_2\, z_2 \to \texttt{let } x = u_2 \texttt{ in } r \end{array} \right.$$

$$\texttt{let } x = \texttt{absurd}(y) \texttt{ in } r \quad \triangleright_{\texttt{RCll}} \quad \texttt{absurd}(y)$$

$$\texttt{let } x = t \texttt{ in } (\texttt{let } y = z\, u \texttt{ in } r)$$

$$\triangleright_{\texttt{RCrl}} \quad \texttt{let } y = z\, (\texttt{let } x = t \texttt{ in } u) \texttt{ in } (\texttt{let } x = t \texttt{ in } r)$$

$$\texttt{let } x = t \texttt{ in } (\texttt{let } y = \pi_i\, z \texttt{ in } r) \quad \triangleright_{\texttt{RCrl}} \quad \texttt{let } y = \pi_i\, z \texttt{ in } (\texttt{let } x = t \texttt{ in } r)$$

$$\texttt{let } x = t \texttt{ in match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\, z_1 \to r_1 \\ \sigma_2\, z_2 \to r_2 \end{array} \right.$$

$$\triangleright_{\texttt{RCrl}} \quad \texttt{match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\, z_1 \to \texttt{let } x = t \texttt{ in } r_1 \\ \sigma_2\, z_2 \to \texttt{let } x = t \texttt{ in } r_2 \end{array} \right.$$

$$\texttt{let } x = t \texttt{ in } \texttt{absurd}(y) \quad \triangleright_{\texttt{RCrl}} \quad \texttt{absurd}(y)$$

Remark 4.2.2. The left-initial case, that is the reduction for $\texttt{let } x = y \texttt{ in } t$, seems a bit odd and superfluous: from a λ-calculus perspective there is no good intuition of why this should be a computation rule. However, it is really necessary to get cut-elimination; otherwise there are some variable-variable cuts that cannot be eliminated. Consider for example this term $(\lambda(x : 0).\, \texttt{let } y = x \texttt{ in } \texttt{absurd}(y))$, of type $0 \to A$. No other rule than the left-initial rule applies to remove this cut. $*$

Remark 4.2.3. The commutative rules introduce non-confluence: it may be the case that the `let`-definition and the `let`-body match a different reduction pattern. For example, $\texttt{let } x = \texttt{absurd}(y) \texttt{ in } ()$ may reduce to either $\texttt{absurd}(y)$ or $()$ depending on which side we choose to reduce first.

This example is not problematic from a program-equality point of view: if y has type 0, then the context are inconsistent and all terms in this context are semantically equal. $*$

We have defined the relation ($\triangleright_{\texttt{R}}$) and its sub-relations on *well-typed* terms of $\mathsf{S\Lambda C}(\to, \times, 1, +, 0)$; this is kept implicit in the presentation of the reduction, but the reader can check that

[1] Classical sequent calculus has a dual to this third case, where the `let`-definition is a right rule against a different co-variable.

any valid derivation on the left-hand side of a reduction determines a valid derivation on the right-hand side, for the same root judgment. For example, taking the most complex reduction rule:

$$\dfrac{\dfrac{\Gamma, y : A \vdash t : B}{\Gamma \vdash \lambda y.\, t : A \to B} \qquad \dfrac{\Gamma, x : A \to B \vdash u : A \qquad \Gamma, x : A \to B, z : B \vdash r : C}{\Gamma, x : A \to B \vdash \mathtt{let}\ z = x\ u\ \mathtt{in}\ r : C}}{\Gamma \vdash \mathtt{let}\ x = \lambda y.\, t\ \mathtt{in}\ (\mathtt{let}\ z = x\ u\ \mathtt{in}\ r) : C}\ \triangleright_{\mathtt{RP}}$$

$$\dfrac{\dfrac{\Gamma, x : A \to B \vdash u : A \qquad \dfrac{\Gamma, y : A \vdash t : B}{\Gamma, x : A \to B, y : A \vdash t : B}\ \mathrm{WK}}{\Gamma, x : A \to B \vdash \mathtt{let}\ y = u\ \mathtt{in}\ t : A}}{\vdots}$$

$$\dfrac{\dfrac{\Gamma, y : A \vdash t : B}{\Gamma \vdash \lambda y.\, t : A \to B} \qquad \dfrac{\Gamma, x : A \to B \vdash \mathtt{let}\ y = u\ \mathtt{in}\ t : A \qquad \Gamma, x : A \to B, z : B \vdash r : C}{\Gamma, x : A \to B \vdash \mathtt{let}\ z = (\mathtt{let}\ y = u\ \mathtt{in}\ t)\ \mathtt{in}\ r : C}}{\Gamma \vdash \mathtt{let}\ x = \lambda y.\, t\ \mathtt{in}\ (\mathtt{let}\ z = (\mathtt{let}\ y = u\ \mathtt{in}\ t)\ \mathtt{in}\ r) : C}$$

Lemma 4.2.1.
If u does not start with a cut, then $(\mathtt{let}\ x = t\ \mathtt{in}\ u)$ is a head redex for the $(\triangleright_{\mathtt{R}})$ reduction.
If a term contains a cut, then it is reducible for the congruent reduction $(\to_{\mathtt{R}})$.

Proof. The second part of the lemma ensures that our reduction indeed covers all cases of terms with cuts. It is a direct consequence of the first part: if u is starts with a sequence of cuts nested to the right, then the last one is a head redex, and u is reducible.

We prove the first part by case-distinction on u. If it starts with a right-introduction rule, one of the principal reductions $(\triangleright_{\mathtt{RP}})$ applies. If it starts with a variable, one of the initial reductions $(\triangleright_{\mathtt{RI}})$ applies. Finally, if it starts with a left-introduction rule, one of the $(\triangleright_{\mathtt{RCrr}})$ or $(\triangleright_{\mathtt{RCrl}})$ applies. □

Remark 4.2.4. One should not be suspicious of the fact that the rules $(\triangleright_{\mathtt{RCll}})$ are not used in this proof. What we are doing here is to prove that a *particular* reduction strategy, the one that always reduces the innermost cuts, can indeed reduce all cuts (and thus that the reduction relation in general can). The $(\triangleright_{\mathtt{RCll}})$ rules may be crucial to a different reduction strategy that would also correspond to interesting computational behavior. ∗

Lemma 4.2.2.
If t, u are cut-free SΛC *terms, then $\mathtt{let}\ x = t\ \mathtt{in}\ u$ has a $(\to_{\mathtt{R}})$-normal form.*

Proof sketch. The proof proceeds by induction on, by lexicographic ordering from the less to the more important: the structure of u, the number of bound occurrences of the cut variable x, and the type of t.

The initial $(\triangleright_{\mathtt{RI}})$ rules always remove the cut.

The commutative $(\triangleright_{\mathtt{RC}})$ rules transform the head cut into cuts on strictly smaller subterms (note that this may duplicate the *binding occurrence* of x, but does not change its number of *bound occurrences*).

The principal $(\triangleright_{\mathtt{RP}})$ rules first create new cuts, and re-applies the head cut to the resulting term. We can first normalize the new cuts by induction hypothesis, as they are on strictly smaller types than the head cut. We then reduce the remaining head cut, on a term with one less occurrence of the cut variable x.

(Note that, if u was not cut-free, reducing cuts in u could duplicate subterms containing occurrences of x.) ⊔

Theorem 4.2.3.
$(\to_{\mathtt{R}})$ is weakly normalizing into cut-free normal forms.

Proof. The innermost cut of a term t has cut-free subterms, and can thus be put in $(\to_{\mathtt{R}})$-normal form by Lemma 4.2.2. By Lemma 4.2.1, this normal-form is cut-free, so the result has strictly less cuts than t. Repeating this process gives a finite reduction sequence to a cut-free term that is also a $(\to_{\mathtt{R}})$-normal form of t. □

Again, we could in fact prove strong normalization, but that requires a significantly stronger proof argument.

Theorem 4.2.4 (Cut-elimination for $\mathsf{PIL}(\to,\times,1,+,0)$)**.**
Each $\mathsf{PIL}(\to,\times,1,+,0)$ *formula provable by a sequent-calculus derivation has a cut-free sequent proof.*

Proof. To obtain a cut-free sequent proof, it suffices to convert the sequent derivation in a sequent-term of $\mathsf{SAC}(\to,\times,1,+,0)$, compute a cut-free $(\to_{\mathtt{R}})$-normal form (Theorem 4.2.3), and convert it back into a sequent-calculus derivation (by erasing the identity of variables). □

We could define (as is standard) cut-elimination on derivations directly. The two notions of cut-elimination could be put in correspondence: we have a Curry-Howard correspondence, this time for the sequent-calculus instead of natural deduction. It is much less striking than the previous correspondence, because our term calculus has been designed specifically to study the sequent calculus. It is more interesting to exhibit a correspondence between two separate formalisms that had been initially created for unrelated purposes.

4.2.2. Equi-provability of natural deduction and sequent calculus

We claimed that the natural deduction rules of Figure 1.2 and sequent calculus rules of Figure 4.1 capture the same logic, in the sense that the same judgments are provable in both systems. We now prove this, by showing that each rule of one system is admissible in the other, so that a proof in one system can always be translated into a proof in the other.

Because different readers will prefer different presentations, we include both the translation of inference rules and the translation of term syntaxes.

Lemma 4.2.5.
Each elimination rule of natural deduction is admissible in the sequent calculus.

Proof.

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash t\ u : B}\ \text{SEQ-IMPL-ELIM}$$

$$\stackrel{\text{def}}{=} \qquad \frac{\Gamma \vdash A \to B \qquad \dfrac{\Gamma \vdash A \qquad \overline{\Gamma, B \vdash B}}{\Gamma, A \to B \vdash B}\ \text{SEQ-IMPL-LEFT}}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = x\ u\ \mathtt{in}\ y : B}\ \text{SEQ-CUT}$$

$$\frac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash \pi_i\ t : A_i}\ \text{SEQ-CONJ-ELIM} \qquad \stackrel{\text{def}}{=}$$

$$\frac{\Gamma \vdash A_1 \times A_2 \qquad \dfrac{\overline{\Gamma, A_i \vdash A_i}}{\Gamma, A_1 \times A_2 \vdash A_i}\ \text{SEQ-CONJ-LEFT}}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = \pi_i\ x\ \mathtt{in}\ y : A_i}\ \text{SEQ-CUT}$$

$$\frac{\Gamma \vdash A_1 + A_2 \qquad \Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma \vdash \mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to u_1 \mid \sigma_2\ y_2 \to u_2\ C}\ \text{SEQ-DISJ-ELIM}$$

$$\stackrel{\text{def}}{=} \qquad \frac{\Gamma \vdash A_1 + A_2 \qquad \dfrac{\Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma, A_1 + A_2 \vdash C}\ \text{SEQ-DISJ-LEFT}}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to u_1 \mid \sigma_2\ y_2 \to u_2 : C}\ \text{SEQ-CUT}$$

$$\dfrac{\Gamma \vdash 0}{\Gamma \vdash \mathtt{absurd}(t) : A}\ \text{SEQ-FALSE-ELIM} \qquad \overset{\mathsf{def}}{=}$$

$$\dfrac{\Gamma \vdash 0 \qquad \dfrac{}{\Gamma, 0 \vdash A}\ \text{SEQ-FALSE-LEFT}}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{absurd}(x) : A}\ \text{SEQ-CUT}$$

Notice that the translation of the sum elimination is more direct; sum-elimination is closer to sequent calculus. □

Notation 4.2.1 Translation into sequent calculus proof terms**.**
If $\Gamma \vdash t : A$ is a well-typed λ-term, let us write $\Gamma \vdash [\![t]\!]_{\text{SEQ}} : A$ for the sequent term obtained by this translation.

Lemma 4.2.6.
The sequent calculus cut rule SEQ-CUT *is admissible in natural deduction.*

Proof. We in fact have already proved cut to be admissible in natural deduction: it is exactly the substitution meta-operation defined in §1.3.1. This highlights that while a cut rule is *necessary* in the sequent calculus for user convenience reason, it is also very important in natural deduction when we want to understand the dynamics of proofs.

$$\dfrac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : B}\ \text{ND-CUT} \qquad \overset{\mathsf{def}}{=} \qquad \dfrac{\Gamma, A \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash u[t/x] : B}\ \text{SUBST}$$

□

Lemma 4.2.7.
Each left-introduction rule of the sequent calculus is admissible in natural deduction.

Proof.

$$\dfrac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2 \vdash \mathtt{let}\ y = \pi_i\ x\ \mathtt{in}\ u : B}\ \text{ND-CONJ-LEFT} \qquad \overset{\mathsf{def}}{=}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A_1 \times A_2 \vdash A_1 \times A_2}}{\Gamma, A_1 \times A_2 \vdash A_i}\ \text{ND-CONJ-ELIM} \qquad \dfrac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2, A_i \vdash C}\ \text{WK}}{\Gamma, A_1 \times A_2 \vdash u[\pi_i\ x/y] : C}\ \text{SUBST}$$

$$\dfrac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash \mathtt{let}\ y = x\ t\ \mathtt{in}\ u : C}\ \text{ND-IMPL-LEFT} \qquad \overset{\mathsf{def}}{=}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A \to B \vdash A \to B} \qquad \dfrac{\Gamma \vdash A}{\Gamma, A \to B \vdash A}\ \text{WK}}{\Gamma, A \to B \vdash B}\ \text{ND-IMPL-ELIM} \qquad \dfrac{\Gamma, B \vdash C}{\Gamma, A \to B, B \vdash C}\ \text{WK}}{\Gamma, A \to B \vdash u[x\ t/y] : C}\ \text{SUBST}$$

$$\dfrac{\Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma, A_1 + A_2 \vdash \mathtt{match}\ x\ \mathtt{with}\ \begin{array}{|l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} : C}\ \text{ND-DISJ-LEFT} \qquad \overset{\mathsf{def}}{=}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A_1 + A_2 \vdash A_1 + A_2} \qquad \dfrac{\Gamma, A_1 \vdash C}{\Gamma, A_1 + A_2, A_1 \vdash C}\ \text{WK} \qquad \dfrac{\Gamma, A_2 \vdash C}{\Gamma, A_1 + A_2, A_2 \vdash C}\ \text{WK}}{\Gamma, A_1 + A_2 \vdash \mathtt{match}\ x\ \mathtt{with}\ \begin{array}{|l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} : C}\ \text{ND-DISJ-ELIM}$$

$$\dfrac{}{\Gamma, 0 \vdash \mathtt{absurd}(x) : C}\ \text{ND-FALSE-LEFT} \qquad \overset{\mathsf{def}}{=} \qquad \dfrac{\dfrac{}{\Gamma, 0 \vdash 0}}{\Gamma, 0 \vdash \mathtt{absurd}(x) : C}\ \text{ND-FALSE-ELIM}$$

□

Notation 4.2.2 Translation into natural deduction proof terms**.**
If $\Gamma \vdash t : A$ is a well-typed sequent term, let us write $\Gamma \vdash [\![t]\!]_{\mathrm{ND}} : A$ for the λ-term obtained by this translation.

Theorem 4.2.8.
Any proof in the sequent calculus for $\mathsf{PIL}(\to, \times, 1, +, 0)$ *is admissible in natural deduction, and conversely: the two systems of inference rules prove exactly the same judgments.*

Proof. This is a direct consequences of the previous lemmas: the (right)-introduction rules of both systems are the same, so we only need to translate the elimination rules of natural deduction (Lemma 4.2.5), and the left-introduction (Lemma 4.2.7) and cut rules (Lemma 4.2.6) of the sequent calculus. □

4.2.3. Non-canonicity of cut-free sequent proofs

Consider the transitivity of implication $A \to B, B \to C \vdash A \to C$. There is exactly one normal natural deduction proof of this judgment, but there exists *two* cut-free sequent proofs. Those three proofs are given below.

$$\dfrac{\dfrac{}{A \to B, B \to C, A \vdash B \to C} \qquad \dfrac{\dfrac{}{A \to B, B \to C, A \vdash A \to B} \qquad \dfrac{}{B \to C, A \vdash A}}{A \to B, B \to C, A \vdash B}}{\text{NATDED}\ \dfrac{A \to B, B \to C, A \vdash C}{A \to B, B \to C \vdash A \to C}}$$

$$\dfrac{\dfrac{}{A \to B, B \to C, A \vdash A} \qquad \dfrac{\dfrac{}{A \to B, B \to C, A, B \vdash B} \qquad \dfrac{}{A \to B, B \to C, A, B, C \vdash C}}{A \to B, B \to C, A, B \vdash C}}{\text{SEQ}\ \dfrac{A \to B, B \to C, A \vdash C}{A \to B, B \to C \vdash A \to C}}$$

$$\dfrac{\dfrac{\dfrac{}{A \to B, B \to C, A \vdash A} \qquad \dfrac{}{A \to B, B \to C, A, B \vdash B}}{A \to B, B \to C, A \vdash B} \qquad \dfrac{}{A \to B, B \to C, A, B, C \vdash C}}{\text{SEQ}\ \dfrac{A \to B, B \to C, A \vdash C}{A \to B, B \to C \vdash A \to C}}$$

To many the proof terms will be more informative:

$$f : A \to B, g : B \to C \vdash \lambda a.\, g\ (f\ a) : A \to C$$

$$f : A \to B, g : B \to C \vdash \lambda a.\, \mathtt{let}\ b = f\ a\ \mathtt{in}\ (\mathtt{let}\ c = g\ b\ \mathtt{in}\ c) : A \to C$$

$$f : A \to B, g : B \to C \vdash \lambda a.\, \mathtt{let}\ c = (\mathtt{let}\ b = f\ a\ \mathtt{in}\ b)\ \mathtt{in}\ c : A \to C$$

In term of proof search, the two sequent proofs correspond respectively to a goal-directed, backward reasoning (second proof: we need a C, let's bind $c : C$ first by applying $B \to C$) and to hypotheses-directed, forward reasoning (first proof: we have a A, let's build $b : B$ by applying $A \to B$).

It is easy to see that our translation of sequent calculus into natural deduction sends both sequent terms to the same result: the respective translations are $c[g\ b/c][f\ a/b]$ and $c[b[f\ a/b]/c]$, which are equal by commutation of substitutions.

In the case of the translation from natural deduction to sequent calculus, the two cut-free results are a consequence of the non-confluence of the reduction system we have presented. The translation of $g\ (f\ a)$ is $(\mathtt{let}\ x = (\mathtt{let}\ y = f\ a\ \mathtt{in}\ y)\ \mathtt{in}\ (\mathtt{let}\ z = g\ x\ \mathtt{in}\ z))$. Reducing the cut on x with a $(\triangleright_{\mathtt{RCl1}})$ rule gives the first cut-free term, using a $(\triangleright_{\mathtt{RCr1}})$ rule gives the second.

4.2.4. On canonical proof representations

Given the goal of this thesis (determining which types are inhabited by a unique program), it would seem that the natural deduction, being more canonical, is a better fit than the

sequent calculus. This would however be a hasty jump to conclusions: natural deduction is more canonical, but it is still not canonical: many equivalent programs have several distinct natural deduction proofs, for example:

$$\frac{}{x : A \to B \vdash x : A \to B} \qquad \frac{\dfrac{\dfrac{}{A \to B, A \vdash A \to B} \quad \dfrac{}{A \to B, A \vdash A}}{A \to B, A \vdash B}}{x : A \to B \vdash \lambda y.\, x\ y : A \to B}$$

To achieve canonicity, we need to go much beyond the spatial parallelism of natural deduction. In Chapter 7 (Focusing in sequent calculus) more powerful logical principles (in terms of the richer structure they impose to proof terms) that were, in fact, developed first and foremost for the sequent calculus, because it is more regular and thus more convenient for the logicians working on proof search, who developed these tools we now reap the benefits of.

4.2.5. Consistency (with sums) through the sequent calculus

Finally, we can use the cut-elimination result of sequent calculus to prove consistency of the full $\mathsf{PIL}(\to, \times, 1, +, 0)$ logic. The interesting aspect of this proof is its simplicity. In natural deduction, we were only able to prove consistency of the disjunction-free fragment $\mathsf{PIL}(\to, \times, 1, 0)$ at first (Theorem 1.4.4), and had to introduce commuting conversions to extend the result (Theorem 3.3.9). Sequent calculus is more appropriate for consistency proofs (in presence of sums).

Lemma 4.2.9.
There is no cut-free sequent proof of $\emptyset \vdash 0$.

Proof. The proof was simple in natural deduction, it is now trivial: there is no right-introduction rule for the succedent 0, and there is no left-introduction or axiom rule for the context $\emptyset$. □

Theorem 4.2.10 (Consistency of $\mathsf{PIL}(\to, \times, 1, +, 0)$ (sequent calculus))**.**
Propositional intuitionistic logic $\mathsf{PIL}(\to, \times, 1, +, 0)$ *is consistent: there is no proof of the false judgment* $\emptyset \vdash 0$.

Proof. Assume we have a sequent proof of $\emptyset \vdash 0$. By cut-elimination (Theorem 4.2.4 (Cut-elimination for $\mathsf{PIL}(\to, \times, 1, +, 0)$)), it has a cut-free proof. This is impossible by Lemma 4.2.9. □

4.3. Classical logic

As we have already mentioned, the logic $\mathsf{PIL}(\to, \times, 1, +, 0)$ is not the "classical logic" that most mathematicians use for their metatheory. Classical logic is boolean: for any formula we know that it is either true or false: $A + \neg A$ (excluded middle) is always true, and $\neg\neg A$ is equivalent to A (in particular $\neg\neg A \to A$ (double-negation elimination) is true).

4.3.1. Introducing the excluded middle

We could recover the full classical logic simply by adding axioms, for example a family of constants $\mathtt{EM}_A : A + \neg A$ providing the excluded-middle principle. However, uninterpreted axioms are unsatisfying for two different reasons:

1. We want to understand the *structure* of proofs in a logic, and realizing important aspects of it through arbitrary constants does not help. This is the same criticism we made of Hilbert-style systems in Section 1.2.4.

2. Axioms break the computational behavior of the logic, as we do not know how to reduce them. For those that are less concerned with computation itself, this loss

can be restated as a loss of "canonicity": the property that the normal forms of a given type has the shape that we expect. For example, we know that the only cut-free proof of type 1 in the empty context is $()$. With those extra axioms, we get additional cut-free proofs of the form $\texttt{match EM}_A\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x_1 \to t_1 \\ \sigma_2\ x_2 \to t_2\end{array}$. Our proof technique to show that the logic is consistent, by inspecting the normal forms of 0 in the empty context, would not work anymore, and we would have to resort to non-syntactic model arguments; the horror!

There are several traditional approaches to this question of giving meaning to axioms.

The first approach is to give a computational behavior to the axiom, under the form of reduction rule in the term syntax. For example, we can add an extra type $\mathbb{N}$ to $\mathsf{PIL}(\to, \times, 1, +, 0)$, two axioms $\texttt{Zero} : \mathbb{N}$ and $\texttt{Succ} : \mathbb{N} \to \mathbb{N}$, and a family of axioms $\texttt{Iter}_A : \mathbb{N} \to A \to (A \to A) \to A$. Then, it suffices to add the reduction rules $(\texttt{Iter}_A\ \mathbb{Z}\ x\ f) \triangleright_\beta x$ and $(\texttt{Iter}_A\ \mathbb{Z}\ n\ x\ f) \triangleright_\beta (\texttt{Iter}_A\ n\ (f\ x)\ f)$, and we have got a reasonable axiomatization of the natural numbers – and we can show that the normal natural numbers in the empty context are what you would expect. In the case of classical logic, we understand since Tim Griffin's remark [Griffin, 1989] that classical logic can be given an operational semantics through a "continuation capture" axiom, that captures the execution context in which it is invoked, and is able to jump back to it later. This has the seductive property of revealing an operation that some programmer communities were already familiar with (`call/cc`), but is, however, rather delicate to define, and even more delicate to reason about.

4.3.2. The multi-succedent sequent calculus is classical

The second approach is to change the rules of the logic, typically the structure of the judgments, to naturally realize the axioms – as derived rules. In a sense, this corresponds to finding a type system in which the extra computational behavior of the first approach can be validated; but this is often done by logicians that do not manipulate term syntaxes themselves and are not directly interested by the programming-language applications. In the case of classical logic, this was already done in Gentzen's original presentation of the sequent calculus. Instead of having the form $\Gamma \vdash A$, with a set of formulas Γ on the left and a single formula A on the right, Gentzen's sequents had the form $\Gamma \vdash \Delta$, with a set of formula Δ on the right, and the following rule for disjunction:

$$\text{CLASSIC-DISJ-RIGHT}\ \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta}$$

The context of intuitionistic sequents can be understood conjunctively: $A_1, A_2, A_3 \vdash B$ means that "if A_1 and A_2 and A_3 hold, then we can prove B"). This rule means that the succedents on the right should be understood disjunctively: $\Gamma \vdash B_1, B_2, B_3$ is understood as "if all formulas of Γ hold, then either B_1 or B_2 or B_3 can be proved". We can easily lift this other sequent structure to all other rules, just by leaving Δ unchanged. The rules for propositional classical logic $\mathsf{PCL}(\to, \times, 1, +, 0)$ in sequent style are given in Figure 4.4.

Note that we do not left-introduce conjunctions by projection, but by pattern-matching, adding both hypotheses at once – we remarked on this difference in §4.1.3. This preserves a pleasing symmetry between conjunction and disjunction, whose right-introduction rule adds both succedents at once.

Remark 4.3.1. We could get an even more symmetrical presentation by making the implication a derived connective, by having a primitive negation and defining $A \to B$ as $\neg A + B$. But the result would be harder to compare with intuitionistic logic. ∗

Figure 4.4.: propositional classical logic $\mathsf{PCL}(\to, \times, 1, +, 0)$ in multi-succedent sequent style

$$\textsc{classic-axiom}\ \frac{}{\Gamma, A \vdash A, \Delta}$$

$$\textsc{classic-cut}\ \frac{\Gamma \vdash \Delta, A \qquad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\textsc{classic-impl-left}\ \frac{\Gamma \vdash \Delta, A \qquad B, \Gamma \vdash \Delta}{\Gamma, A \to B \vdash \Delta} \qquad \textsc{classic-impl-right}\ \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

$$\textsc{classic-conj-left}\ \frac{\Gamma, A_1, A_2 \vdash \Delta}{\Gamma, A_1 \times A_2 \vdash \Delta} \qquad \textsc{classic-conj-right}\ \frac{\Gamma \vdash A_1, \Delta \qquad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \times A_2, \Delta}$$

$$\textsc{classic-disj-left}\ \frac{\Gamma, A_1 \vdash \Delta \qquad \Gamma, A_2 \vdash \Delta}{\Gamma, A_1 + A_2 \vdash \Delta} \qquad \textsc{classic-disj-right}\ \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta}$$

$$-$$

$$\textsc{classic-true-right}\ \frac{}{\Gamma \vdash 1, \Delta}$$

$$\textsc{classic-false-left}\ \frac{}{\Gamma, 0 \vdash \Delta} \qquad -$$

With theses rules we can easily prove the excluded middle in any context:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash A, 0}\ \textsc{classic-axiom}}{\Gamma \vdash A, A \to 0}\ \textsc{classic-impl-right}}{\Gamma \vdash A + (A \to 0)}\ \textsc{classic-disj-right}$$

The crucial effect of the proof, which cannot be realized in intuitionistic logic, is the transfer of the hypothesis A from one of the succedents, $A \to 0$, to the other succedent A. We promise to prove one of two things, but they "communicate" and we can use the hypotheses introduced by one to prove the other. With the intuitionistic presentation, because there is a single succedent, no communication happens on the right of the turnstile $\vdash$.

Remark 4.3.2. Our intuition of multi-succedent calculi is that the various formulas after the turnstile $\vdash$ correspond to types of "output doors", or "output ports". You can finish your proof by throwing a value (of the expected type) into any of those doors/ports; in intuitionistic logic, there is only one return door/port/point/adress, it is "the result".

The (positive) disjunction corresponds to splitting a door in two; you do not know which one you will take yet, and you will decide later. This is how the excluded middle can be made sense of. An axiom of type $A + \neg A$ does not guarantee that you will get, at its invocation time, either a proof of A or a proof of $\neg A$. What happens instead is that it gives you a choice of two doors to take in the future. Now the proof does something inherently classical, which is to claim to enter one of the doors $(A \to 0)$, introduce the A hypothesis in the process of giving a value to that door, but then it changes its mind to exit through the other door. $*$

We refrain from providing a term syntax for this logic, because it would be quite heavy: each right-introduction rule has to name explicitly the one of the succedents that is being worked on (using *co-variables*), and also provide binding occurrences for the new succedents added in its premises.[2]

Figure 4.5.: classical logic $\mathsf{PCL}(\to, \times, 1, +, 0)$ in multi-succedent natural deduction style

$$\textsc{classic-nd-axiom}\ \frac{}{\Gamma, A \vdash A, \Delta}$$

$$\textsc{classic-nd-impl-elim}\ \frac{\Gamma \vdash A \to B, \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta} \qquad \textsc{classic-nd-impl-intro}\ \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

$$\textsc{classic-nd-conj-elim}\ \frac{\Gamma \vdash \Delta, A_1 \times A_2 \qquad A_1, A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \qquad \textsc{classic-nd-conj-intro}\ \frac{\Gamma \vdash A_1, \Delta \qquad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \times A_2, \Delta}$$

$$\textsc{classic-nd-disj-elim}\ \frac{\Gamma \vdash \Delta, A_1 + A_2 \qquad A_1, \Gamma \vdash \Delta \qquad A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \qquad \textsc{classic-nd-disj-intro}\ \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta}$$

$$-$$

$$\textsc{classic-nd-true-intro}\ \frac{}{\Gamma \vdash 1, \Delta}$$

$$\textsc{classic-nd-false-elim}\ \frac{\Gamma \vdash \Delta, 0}{\Gamma \vdash \Delta} \qquad -$$

Gerhard Gentzen went through the sequent calculus to give a satisfying system of inference rules for classical logic, but retrospectively we can also present it as a natural deduction system – multi-succedents are really the key here, and the affected rule is a right rule (CLASSIC-DISJ-RIGHT), which can be kept unchanged in a natural deduction system. Figure 4.5 gives such a system.

4.3.3. Multi-succedent intuitionistic logic

It is a lesser-known fact that there is a multi-succedent presentation of intuitionistic logic, presented in Figure 4.6, in natural deduction style (sequents would work as well, but we will assume our readers still have more intuition with natural deduction rules).

This system has exactly one different with the classical natural deduction of Figure 4.5:

[2] I find it remarkable that, while inference rules and typed term syntax are equivalent formalisms, they have extremely different aesthetics. As a system of inference rules, sequent calculus is arguably more beautiful than natural deduction, but the term syntax we presented in Section 4.1.4 feels contrived – there are better syntaxes, but they are for variants of the proof system with non-trivial differences. The multi-succedent presentation is pleasing at a derivation level, but its term syntax is unpalatable to me – even in natural deduction. I think this comes from the fact that these two notations emphasize different computational phenomena: I read inference rules and I think of proof search (logic programming), I read proof terms and I think of reductions (functional programming), and few systems are good at both. When moving to term syntaxes for lower-level logics, it also helps to leave the pretense of a syntax inspired by the λ-calculus, recognize a lower-level computational phenomenon, and design the syntax accordingly: abstract machines, process calculi, etc.

Figure 4.6.: intuitionistic $\mathsf{PIL}(\to, \times, 1, +, 0)$ in multi-succedent natural deduction style

$$\textsc{ms-nd-axiom}\ \frac{}{\Gamma, A \vdash A, \Delta}$$

$$\textsc{ms-nd-impl-elim}\ \frac{\Gamma \vdash A \to B, \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta} \qquad \textsc{ms-nd-impl-intro}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B, \Delta}$$

$$\textsc{ms-nd-conj-elim}\ \frac{\Gamma \vdash \Delta, A_1 \times A_2 \qquad A_1, A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \qquad \textsc{ms-nd-conj-intro}\ \frac{\Gamma \vdash A_1, \Delta \qquad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \times A_2, \Delta}$$

$$\textsc{ms-nd-disj-elim}\ \frac{\Gamma \vdash \Delta, A_1 + A_2 \qquad A_1, \Gamma \vdash \Delta \qquad A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \qquad \textsc{ms-nd-disj-intro}\ \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta}$$

$$\emptyset \qquad \textsc{ms-nd-true-intro}\ \frac{}{\Gamma \vdash 1, \Delta}$$

$$\textsc{ms-nd-false-elim}\ \frac{\Gamma \vdash \Delta, 0}{\Gamma \vdash \Delta} \qquad \emptyset$$

the introduction/right rules for implication differ:

$$\textsc{classic-nd-impl-intro}\ \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \qquad \textsc{ms-nd-impl-intro}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B, \Delta}$$

We gave an intuition of multi-succedent systems where the hypotheses on the right correspond to "output doors" (expecting a value of the right type). Under this view, the intuitionistic rule has the following effect: whenever you enter an implication, you commit to only use its door, and the others are closed. This means that the trick we used to prove the excluded middle, namely changing our mind after choosing an implication door and going to another door instead, cannot work anymore in this new system.

Another way to see the difference with the classical calculus is to notice that we can prove $A \vdash \neg\neg A$ in the intuitionistic calculus, but that our proof of $\neg\neg A \vdash A$ crucially relies on the classical implication introduction rule.

Lemma 4.3.1 (Double-negation introduction)**.**
$A \vdash \neg\neg A$ *in (intuitionistic)* $\mathsf{PIL}(\to, \times, 1, +, 0)$

Proof.

$$\dfrac{\dfrac{\dfrac{}{A, A \to 0 \vdash A \to 0, 0} \qquad \dfrac{}{A, A \to 0 \vdash A, 0}}{A, A \to 0 \vdash 0}}{A \vdash (A \to 0) \to 0}$$

□

Lemma 4.3.2 (Double-negation elimination)**.**
$\neg\neg A \vdash A$ *in (classical)* $\mathsf{PCL}(\to, \times, 1, +, 0)$

Proof.

$$\dfrac{\dfrac{}{(A \to 0) \to 0 \vdash (A \to 0) \to 0, A, 0} \quad \dfrac{\dfrac{}{(A \to 0) \to 0, A \vdash A, 0, A \to 0}}{(A \to 0) \to 0 \vdash A, 0, A \to 0}}{\dfrac{(A \to 0) \to 0 \vdash A, 0}{(A \to 0) \to 0 \vdash A}}$$

□

To convince ourselves that this indeed captures intuitionistic logic as previously presented, we provide translation between the two presentations.

Proving that the single-succedent rules are valid for the multi-succedent rules is simple, as many of them are valid, unchanged, in the case where the right context is a singleton.

Lemma 4.3.3 (Right weakening)**.**
The following rule is admissible:

$$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash C, \Delta}\ \text{WK-RIGHT}$$

Proof. By induction on a complete proof of $\Gamma \vdash \Delta$. □

Lemma 4.3.4.
The single-succedent introduction rules for sum are admissible in multi-succedent natural deduction.

Proof.

$$\dfrac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_1 + A_2, \Delta} \quad \stackrel{\mathsf{def}}{=} \quad \dfrac{\dfrac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_1, A_2, \Delta}\ \text{WK-RIGHT}}{\Gamma \vdash A_1 + A_2, \Delta}$$

(We did the proof with a parameter Δ for generality, taking $\Delta \stackrel{\mathsf{def}}{=} \emptyset$ gives exactly the single-succedent rule.) □

The fact that the other introduction rules are admissible does not even need a proof: the single-succedent rules are instances of the multi-succedent rules in the case where $\Delta = \emptyset$.

Lemma 4.3.5.
The single-succedent elimination rules are admissible in the multi-succedent system.

Proof. Setting $\Delta \stackrel{\mathsf{def}}{=} \emptyset$ as for the introduction rules does not quite work, as it gives us a root judgment of the form $\Gamma \vdash \emptyset$ instead of the expected $\Gamma \vdash C$. To get the single-succedent rule, one should instantiate $\Delta \stackrel{\mathsf{def}}{=} \{C\}$, and then weaken the elimination premise.

$$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \quad \stackrel{\mathsf{def}}{=} \quad \dfrac{\dfrac{\Gamma \vdash A \to B}{\Gamma \vdash A \to B, B}\ \text{WK-RIGHT} \qquad \dfrac{\Gamma \vdash A}{\Gamma \vdash A, B}\ \text{WK-RIGHT}}{\Gamma \vdash B}$$

$$\dfrac{\Gamma \vdash A_1 \times A_2 \qquad \Gamma, A_1, A_2 \vdash C}{\Gamma \vdash B} \quad \stackrel{\mathsf{def}}{=}$$

$$\dfrac{\dfrac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash B, A_1 \times A_2}\ \text{WK-RIGHT} \qquad \Gamma, A_1, A_2 \vdash C}{\Gamma \vdash C}$$

$$\dfrac{\Gamma \vdash A_1 + A_2 \qquad A_1, \Gamma \vdash C \qquad A_2, \Gamma \vdash C}{\Gamma \vdash C}$$

$$\stackrel{\mathsf{def}}{=} \quad \dfrac{\dfrac{\Gamma \vdash A_1 + A_2}{\Gamma \vdash A_1 + A_2, C}\ \text{WK-RIGHT} \qquad A_1, \Gamma \vdash C \qquad A_2, \Gamma \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash 0}{\Gamma \vdash A} \quad \stackrel{\text{def}}{=} \quad \dfrac{\dfrac{\Gamma \vdash 0}{\Gamma \vdash 0, A}\ \text{WK-RIGHT}}{\Gamma \vdash A}$$

□

Theorem 4.3.6.
Any judgment provable in single-succedent intuitionistic natural deduction is provable in multi-succedent intuitionistic natural deduction.

The converse direction is more delicate. A first idea is to translate multi-succedent judgments of the form $\Gamma \vdash C_1, C_2, \ldots$ into single-succedent judgments of the form $\Gamma \vdash C_1 + C_2 + \ldots$. Translating each inference rule in this style would require to unpack and repack this big disjunction, adding a lot of bureaucracy to the translation.

The reason why a purely-local translation of each inference step is difficult is that the two introduction rules for disjunctions are fundamentally different: the multi-succedent rule leaves the choice of which side of the sum to take to future proof steps, while the single-succedent rule commits to one side. The idea to prove the equivalence is that we can in fact know which side will be taken by looking at the leafward sub-proofs. Because of the restriction on the introduction of implication MS-ND-IMPL-INTRO, we know that the context at the point where this choice is made will not have grown (it cannot happen after an implication has been introduced), so the choice can be "imported" back at the place of the disjunction introduction. This is what happens in the lemma below.

Note that $\Gamma \vdash \Delta$ and $\Gamma \vdash \Delta, 0$ are inter-derivable (by weakening and elimination of 0, respectively). In particular, for any succedent context Δ, we can assume that Δ is not the empty set; if it were empty we could always consider $\Delta, 0$ instead.

Lemma 4.3.7.
We can convert any multi-succedent proof $\Pi_\Delta :: \Gamma \vdash \Delta$ into a set of partial single-succedent proofs $\{\Pi_C :: \Gamma \vdash C \mid C \in \Delta\}$, such that at least one of the Π_C is a complete proof, provided that

- *Δ is not the empty set*
- *Π does not contain multi-succedent disjunction eliminations,*
- *and its only leaves are all single-succedent sequents.*

Proof. We do our proof by induction on Π_Δ. The rules without premises are easy to handle:

$$\frac{}{\Gamma, A \vdash A, \Delta} \quad \mapsto \quad \frac{}{\Gamma, A \vdash A}$$

$$\frac{}{\Gamma \vdash 1, \Delta} \quad \mapsto \quad \frac{}{\Gamma \vdash 1}$$

Now consider a rule with a premise, such as:

$$\frac{\Gamma \vdash 0, \Delta}{\Gamma \vdash \Delta}$$

Applying the induction hypothesis on the premise $\Gamma \vdash 0, \Delta$, there are two cases: either we get a valid proof of $\Gamma \vdash 0$, or we get a valid proof of $\Gamma \vdash C$ for some $C \in \Delta$. In both cases, we can conclude.

We will keep using the $\Pi \mapsto \Pi'$ notation for readability, with a specific notation to indicate which of the case we are considering: we will write $\Gamma \vdash \Delta \ni C$ in the premise of a multi-succedent rule, to indicate that we consider the case where the valid proof obtained by induction hypothesis is for the judgment $\Gamma \vdash C$. We can thus represent our two cases above as follows:

$$\frac{\Gamma \vdash (0, \Delta) \ni 0}{\Gamma \vdash \Delta} \quad \mapsto \quad \frac{\Gamma \vdash 0}{\Gamma \vdash B} \text{ for some } B \in \Delta$$

$$\frac{\Gamma \vdash 0, \Delta \ni C}{\Gamma \vdash \Delta} \quad \mapsto \quad \Gamma \vdash C$$

The first mapping reads as follows. If, by induction hypothesis, we get a proof of $\Gamma \vdash 0$ (a possible case because $(0, \Delta) \ni 0$), then we perform an elimination of false to get a proof of $\Gamma \vdash B$ for some $B \in \Delta$. If, by induction hypothesis, we get a proof of the judgment $\Gamma \vdash C$ for some arbitrary $C \in \Delta$, then we return this proof.

Having clarified the notation, we can use it to prove the remaining cases.

$$\frac{\Gamma \vdash A \to B, \Delta \ni C \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta} \quad \mapsto \quad \Gamma \vdash C$$

$$\frac{\Gamma \vdash A \to B, \Delta \qquad \Gamma \vdash A, \Delta \ni C}{\Gamma \vdash B, \Delta} \quad \mapsto \quad \Gamma \vdash C$$

$$\frac{\Gamma \vdash (A \to B, \Delta) \ni A \to B \qquad \Gamma \vdash (A, \Delta) \ni A}{\Gamma \vdash B, \Delta} \quad \mapsto \quad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B \ni B}{\Gamma \vdash A \to B, \Delta} \quad \mapsto \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\frac{\Gamma \vdash A_1 \times A_2, \Delta \ni C \qquad A_1, A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \quad \mapsto \quad \Gamma \vdash C$$

$$\frac{\Gamma \vdash (A_1 \times A_2, \Delta) \ni A_1 \times A_2 \qquad A_1, A_2, \Gamma \vdash \Delta \ni C}{\Gamma \vdash \Delta} \quad \mapsto$$

$$\frac{\Gamma \vdash A_1 \times A_2 \qquad A_1, A_2, \Gamma \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A_1, \Delta \ni C \qquad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \times A_2, \Delta} \quad \mapsto \quad \Gamma \vdash C$$

$$\frac{\Gamma \vdash A_1, \Delta \qquad \Gamma \vdash A_2, \Delta \ni C}{\Gamma \vdash A_1 \times A_2, \Delta} \quad \mapsto \quad \Gamma \vdash C$$

$$\frac{\Gamma \vdash (A_1, \Delta) \ni A_1 \qquad \Gamma \vdash (A_2, \Delta) \ni A_2}{\Gamma \vdash A_1 \times A_2, \Delta} \quad \mapsto \quad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$$

$$\frac{\Gamma \vdash (A_1 + A_2, B) \ni B \qquad A_1, \Gamma \vdash B \qquad A_2, \Gamma \vdash B}{\Gamma \vdash B} \qquad \mapsto \qquad \Gamma \vdash B$$

$$\frac{\Gamma \vdash (A_1 + A_2, B) \ni A_1 + A_2 \qquad A_1, \Gamma \vdash B \qquad A_2, \Gamma \vdash B}{\Gamma \vdash B}$$

$$\mapsto \qquad \frac{\Gamma \vdash A_1 + A_2 \qquad A_1, \Gamma \vdash B \qquad A_2, \Gamma \vdash B}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A_1, A_2, \Delta \ni C}{\Gamma \vdash A_1 + A_2, \Delta} \qquad \mapsto \qquad \Gamma \vdash C$$

$$\frac{\Gamma \vdash (A_1, A_2, \Delta) \ni A_i}{\Gamma \vdash A + B, \Delta} \qquad \mapsto \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}$$

□

To see the problem with multi-succedent disjunction elimination, consider the following case:

$$\frac{\Gamma \vdash (A_1 + A_2, \Delta) \ni A_1 + A_2 \qquad A_1, \Gamma \vdash \Delta \ni C \qquad A_2, \Gamma \vdash \Delta \ni C'}{\Gamma \vdash \Delta}$$

If C and C' are the same formula of Δ (which is always the case when Δ is a singleton), we can build a single-succedent proof. But if they are distinct formula, we are stuck. Recall that our informal explanation above explained that a choice of output door could be lifted back to the place of introduction, because the context in which the choice happens is identical to the context of the sum introduction. It is not the case for multi-succedent disjunction elimination – or disjunction left-introduction in sequent style – which has to be handled specially by the following lemma.

Lemma 4.3.8.
We can always permute the rules of a natural deduction proof so that the disjunction elimination only appear either at the root of the proof, or above an implication introduction, or above a disjunction elimination (as second or third premise).

Proof sketch. We have in fact already proved this in the context of the single-succedent natural deduction: this is the Theorem 3.3.4 (Standardization by extrusion) of Section 3.3 (Extrusion and commuting conversions). The proof in the multi-succedent case is identical: we only need to check that extrusion is possible in each case and give a valid proof. □

Theorem 4.3.9.
Every complete proof Π *of single-succedent sequent in the multi-succedent system can be rewritten in the single-succedent system.*

Proof. By Lemma 4.3.8, we can always permute disjunction rules in a proof so that they are all at the root of the proof, of above an implication introduction, or above a disjunction elimination (second or third premises). The premises of implication introductions are single-succedent sequents; by assumption, the root of Π is also a single-succedent sequent. This means that disjunction eliminations at the root or above implication are single-succedent as well and, transitively, so are the disjunction eliminations above their second and third premises. Our proof now only uses disjunction eliminations on single-succedent sequents.

We can then apply Lemma 4.3.7, which claims that proofs without multi-succedent can be rewritten into single-succedent proofs. Note that we use the assumption that our proof is complete (all leaves are closed by an axiom rule); this transformation is not modular with respect to open leaves, as we explore arbitrary far into the subproofs to resolve disjunction introductions. □

Note that the restriction that the root judgment be single-succedent does not removes generality from the result: a multi-succedent judgment $\Gamma \vdash B_1, B_2, \dots$ is provable if and only if the corresponding single-succedent judgment $\Gamma \vdash B_1 + B_2 + \dots$ is provable.

Going further We could, in fact, go even further that this multi-succedent system. The strategy of dropping alternative succedents when introducing an implication can be seen as a general case of a more general *labeled* presentation of logic, where labels track dependencies between hypotheses and succedents. See de Paiva and Pereira [2005] for more details in the case of intuitionistic logic, and some history of its multi-succedent presentations; this is a general construction that can be applied to many logics.

5. The bothersome equivalence of cut-free sequent proofs

The example in Section 4.2.3 (Non-canonicity of cut-free sequent proofs) of a cut-free natural deduction proof that translates to two distinct cut-free sequent proofs motivates us to look for an equivalence relation among cut-free sequent proofs – hopefully such that all translations of a natural deduction proof are equivalent.

In this chapter, we define a notion of equivalence for sequent proofs that has this property – yet remains sound with respect to program equivalence – using the term syntax of Section 4.1.4 (A term syntax for the intuitionistic sequent calculus). It is very similar to the analysis that we did in Section 3.3 (Extrusion and commuting conversions), splitting the strong η-expansion of sums in natural deduction into a weak η-expansion principle, and extrusion and commuting conversions.

For the sequent calculus, we first define the counterpart of extrusion and commuting conversions, called here the permutation equivalence; this suffices to recover a form of confluence of the reduction of cuts. Then we add weak η-expansion rules, and this recovers the full $\beta\eta$-equivalence of natural deduction.

Despite this good correspondence between the equivalences in the two system, we cannot help noticing that the permutation equivalence is a burden to define – the size of its definition has a quadratic growth in the number of connectives in the logic or type system.

5.1. Permutation equivalence

We define a *permutation equivalence* relation ($\approx_{\mathtt{scc}}$) as the congruent union of *permutation* rules that exchange the order of two introductions, *merging* rules that merge two consecutive introductions when they are equivalent, and *weakening* rules that removes unused left-introductions:

1. ($\approx_{\mathtt{scc:l/r}}$), the permutation of a left- and a right-introduction rule (Figure 5.1)
2. ($\approx_{\mathtt{scc:l/l}}$), the permutation of two left-introduction rules (Figures 5.2 and 5.3)
3. ($\approx_{\mathtt{scc:r}-\emptyset}$), the *erasing* of a rule by permutation with a premise-free right rule (Figure 5.1)
4. ($\approx_{\mathtt{scc:l}-\emptyset}$), the *erasing* of a rule by permutation with a premise-free left rule (Figures 5.2 and 5.3)
5. ($\approx_{\mathtt{scc:merge}}$), the merge of two equivalent consecutive (left)-rules. (Figure 5.4)
6. ($\approx_{\mathtt{scc:weak}}$), the *erasing* of unused (left)-rules. (Figure 5.5)

Remark 5.1.1. In all these figures, an implicit assumption of each equivalence is that it preserves scoping of variables and well-typing. Consider for example the following rules:

$$\lambda x.\, \mathtt{let}\ y = z\ t\ \mathtt{in}\ u \approx_{\mathtt{scc:l/r}} \mathtt{let}\ y = z\ t\ \mathtt{in}\ \lambda x.\, u \qquad () \approx_{\mathtt{scc:r}-\emptyset} \mathtt{absurd}(x)$$

The first rule is only well-typed if the following scoping conditions are respected. x and y being both bound in the terms, we can assume (by α-equivalence) that they are distinct, but other variable conflicts must be explicitly ruled out. The variable z bound under the $\lambda x.\_$ on the left-hand side must be distinct from x, otherwise it would become free in the

Figure 5.1.: Equivalence of cut-free sequent proofs: left/right rules

$$\lambda x.\, \texttt{let}\ y = z\ t\ \texttt{in}\ u \approx_{\texttt{scc:l/r}} \texttt{let}\ y = z\ t\ \texttt{in}\ \lambda x.\, u$$

$$\lambda x.\, \texttt{let}\ y = \pi_i\ z\ \texttt{in}\ u \approx_{\texttt{scc:l/r}} \texttt{let}\ y = \pi_i\ z\ \texttt{in}\ \lambda x.\, u$$

$$\lambda x.\, \texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to t_1 \\ \sigma_2\ z_2 \to t_2 \end{array} \right. \approx_{\texttt{scc:l/r}} \texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to \lambda x.\, t_1 \\ \sigma_2\ z_2 \to \lambda x.\, t_2 \end{array} \right.$$

$$\lambda x.\, \texttt{absurd}(y) \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(y)$$

$$(r, \texttt{let}\ y = x\ t\ \texttt{in}\ u) \approx_{\texttt{scc:l/r}} \texttt{let}\ y = x\ t\ \texttt{in}\ (r, u) \text{ (and symmetric)}$$

$$(r, \texttt{let}\ y = \pi_i\ z\ \texttt{in}\ u) \approx_{\texttt{scc:l/r}} \texttt{let}\ y = \pi_i\ z\ \texttt{in}\ (r, u) \text{ (and symmetric)}$$

$$\left(r, \texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to t_1 \\ \sigma_2\ z_2 \to t_2 \end{array} \right.\right) \approx_{\texttt{scc:l/r}}$$

$$\texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to (r, t_1) \\ \sigma_2\ z_2 \to (r, t_2) \end{array} \right. \text{ (and symmetric)}$$

$$(r, \texttt{absurd}(y)) \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(y) \text{ (and symmetric)}$$

$$\sigma_i\ (\texttt{let}\ y = z\ t\ \texttt{in}\ u) \approx_{\texttt{scc:l/r}} \texttt{let}\ y = z\ t\ \texttt{in}\ \sigma_i\ u$$

$$\sigma_i\ (\texttt{let}\ y = \pi_i\ z\ \texttt{in}\ u) \approx_{\texttt{scc:l/r}} \texttt{let}\ y = \pi_i\ z\ \texttt{in}\ \sigma_i\ u$$

$$\sigma_i \left(\texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to t_1 \\ \sigma_2\ z_2 \to t_2 \end{array} \right.\right) \approx_{\texttt{scc:l/r}} \texttt{match}\ y\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to \sigma_i\ t_1 \\ \sigma_2\ z_2 \to \sigma_i\ t_2 \end{array} \right.$$

$$\sigma_i\ \texttt{absurd}(y) \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(y)$$

$$() \approx_{\texttt{scc:r}-\emptyset} \texttt{let}\ x = y\ t\ \texttt{in}\ () \qquad () \approx_{\texttt{scc:r}-\emptyset} \texttt{let}\ x = \pi_i\ y\ \texttt{in}\ ()$$

$$() \approx_{\texttt{scc:r}-\emptyset} \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to () \\ \sigma_2\ y_2 \to () \end{array} \right. \qquad () \approx_{\texttt{scc:r}-\emptyset} \texttt{absurd}(x)$$

right-hand side. The variable x, which scopes over t in the left-hand side, must not appear in it ($x \notin t$), as these occurrences would become free in the right-hand side.

The second rule may at first appear very surprising but makes sense thanks to our typing assumptions. We can assume that $\texttt{absurd}(x)$ is well-typed, and thus that $x : 0$ is in the environment of both terms. We already know that, under a 0 assumption, all terms should be equated, so in particular $() \approx \texttt{absurd}(x)$ is semantically correct. $*$

Permutation rules Those rules permute two independent introduction rules. The rules of Figure 5.1 permute a left rule and a right rule, and the rules of Figures 5.2 and 5.3 permute two left rules together. Note that there are no cases of permutation of a right rule with a right rule, as this would never preserve the type of the goal.

Erasing by permutation rules A special cases of permutation rules is when one of the two permuted introduction rule has no premises: the other rule is erased by the permutation. A typical such rule is $() \approx_{\texttt{scc:r}-\emptyset} \texttt{let}\ x = \pi_i\ y\ \texttt{in}\ ()$.

One-branch and two-branches There is a discrepancy between left-left permutations and left-right permutations, when the left rule has more than one binding premise – only

Figure 5.2.: Equivalence of cut-free sequent proofs: left/left rules (part 1)

$$\texttt{let}\ x = y\ (\texttt{let}\ x' = y'\ t'\ \texttt{in}\ t)\ \texttt{in}\ u \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = y'\ t'\ \texttt{in}\ \texttt{let}\ x = y\ t\ \texttt{in}\ u$$

$$\texttt{let}\ x = y\ t\ \texttt{in}\ (\texttt{let}\ x' = y'\ t'\ \texttt{in}\ u) \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = y'\ t'\ \texttt{in}\ \texttt{let}\ x = y\ t\ \texttt{in}\ u$$

$$\texttt{let}\ x = \pi_i\ y\ \texttt{in}\ (\texttt{let}\ x' = y'\ t'\ \texttt{in}\ u) \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = y'\ t'\ \texttt{in}\ \texttt{let}\ x = \pi_i\ y\ \texttt{in}\ u$$

$$\begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to \texttt{let}\ x' = y'\ t'\ \texttt{in}\ u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = y'\ t'\ \texttt{in} \quad \begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \text{(and symmetric)}$$

$$\texttt{absurd}(x) \approx_{\texttt{scc:l}-\emptyset} \texttt{let}\ x' = y'\ t'\ \texttt{in}\ \texttt{absurd}(x)$$

$$\texttt{let}\ x = y\ (\texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ t)\ \texttt{in}\ u \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ \texttt{let}\ x = y\ t\ \texttt{in}\ u$$

$$\texttt{let}\ x = y\ t\ \texttt{in}\ (\texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ u) \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ \texttt{let}\ x = y\ t\ \texttt{in}\ u$$

$$\texttt{let}\ x = \pi_i\ y\ \texttt{in}\ (\texttt{let}\ x' = \pi_i\ y'\ \texttt{in}\ u) \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = \pi_i\ y'\ \texttt{in}\ \texttt{let}\ x = \pi_i\ y\ \texttt{in}\ u$$

$$\begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = \pi_i\ z\ \texttt{in} \quad \begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \text{(and symmetric)}$$

$$\texttt{absurd}(x) \approx_{\texttt{scc:l}-\emptyset} \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ \texttt{absurd}(x)$$

sum eliminations in our system.

$$\lambda x.\, \texttt{match}\ y\ \texttt{with} \mid \sigma_1\ z_1 \to t_1 \mid \sigma_2\ z_2 \to t_2 \quad \approx_{\texttt{scc:l/r}} \quad \texttt{match}\ y\ \texttt{with} \mid \sigma_1\ z_1 \to \lambda x.\, t_1 \mid \sigma_2\ z_2 \to \lambda x.\, t_2$$

$$\begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \approx_{\texttt{scc:l/l}} \texttt{let}\ x' = \pi_i\ z\ \texttt{in} \quad \begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to u_1\\ \mid\ \sigma_2\ y_2 \to u_2\end{array} \quad \text{(and symmetric)}$$

When read from right to left, these rules can be understood as extruding a syntactic construction (right λ or left π_i) out of the branches of a match. But note the difference: in the left-right case, we request that the right rule be present in *both* branches of the match, while in the left-left case we only request that it be present in one of the branches. Allowing the right rule to be present in one branch only would break typing preservation; restricting the left rule to be present in both branches would remove useful generality.

Merging rules The merging rules of Figure 5.4 can be justified by the discrepancy above. We also want the two-branches equivalence to hold for left/left permutations:

$$\begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y_1 \to \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ u_1\\ \mid\ \sigma_2\ y_2 \to \texttt{let}\ x' = \pi_i\ z\ \texttt{in}\ u_2\end{array} \quad \approx_{\texttt{scc}} \texttt{let}\ x' = \pi_i\ z\ \texttt{in} \quad \begin{array}{l}\texttt{match}\ x\ \texttt{with}\\ \mid\ \sigma_1\ y \to u_1\\ \mid\ \sigma_2\ y \to u_2\end{array}$$

To prove that this equivalence is derivable from the one-branch rule, we can extrude the left rule of each branch separately. But then we end up with two copies of the left rule

Figure 5.3.: Equivalence of cut-free sequent proofs: left/left rules (part 2)

$$\texttt{let } x = y \texttt{ in } \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to t_1 \\ \mid \sigma_2\ y'_2 \to t_2 \end{array} \quad u \approx_{\texttt{scc:l/l}} \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to \texttt{let } x = y\ t_1 \texttt{ in } u \\ \mid \sigma_2\ y'_2 \to \texttt{let } x = y\ t_2 \texttt{ in } u \end{array}$$

$$\texttt{let } x = y\ t \texttt{ in } \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to u_1 \\ \mid \sigma_2\ y'_2 \to u_2 \end{array} \approx_{\texttt{scc:l/l}} \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to \texttt{let } x = y\ t \texttt{ in } u_1 \\ \mid \sigma_2\ y'_2 \to \texttt{let } x = y\ t \texttt{ in } u_2 \end{array}$$

$$\texttt{let } x = \pi_i\ y \texttt{ in } \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to u_1 \\ \mid \sigma_2\ y'_2 \to u_2 \end{array} \approx_{\texttt{scc:l/l}} \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to \texttt{let } x = \pi_i\ y \texttt{ in } u_1 \\ \mid \sigma_2\ y'_2 \to \texttt{let } x = \pi_i\ y \texttt{ in } u_2 \end{array}$$

$$\begin{array}{|ll} \texttt{match } x \texttt{ with} & \\ \mid \sigma_1\ y_1 \to t & \\ \mid \sigma_2\ y_2 \to & \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to u_1 \\ \mid \sigma_2\ y'_2 \to u_2 \end{array} \end{array} \approx_{\texttt{scc:l/l}} \begin{array}{|ll} \texttt{match } x' \texttt{ with} & \\ \mid \sigma_1\ y'_1 \to & \begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y_1 \to t \\ \mid \sigma_2\ y_2 \to u_1 \end{array} \\ \mid \sigma_2\ y'_2 \to & \begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y_1 \to t \\ \mid \sigma_2\ y_2 \to u_2 \end{array} \end{array} \quad \text{(and symmetric)}$$

$$\texttt{absurd}(x) \approx_{\texttt{scc:l}-\emptyset} \begin{array}{|l} \texttt{match } x' \texttt{ with} \\ \mid \sigma_1\ y'_1 \to \texttt{absurd}(x) \\ \mid \sigma_2\ y'_2 \to \texttt{absurd}(x) \end{array}$$

$$\texttt{let } x = y\ \texttt{absurd}(x') \texttt{ in } u \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(x')$$

$$\begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y_1 \to t \\ \mid \sigma_2\ y_2 \to \texttt{absurd}(x') \end{array} \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(x') \quad \text{(and symmetric)}$$

$$\texttt{absurd}(x) \approx_{\texttt{scc:l}-\emptyset} \texttt{absurd}(x')$$

Figure 5.4.: Equivalence of cut-free sequent proofs: merge rules

$$\texttt{let } y = x\ t \texttt{ in let } y' = x\ t \texttt{ in } u \approx_{\texttt{scc:merge}} \texttt{let } y = x\ t \texttt{ in } u[y/y']$$

$$\texttt{let } y = \pi_i\ x \texttt{ in let } y' = \pi_i\ x \texttt{ in } u \approx_{\texttt{scc:merge}} \texttt{let } y = \pi_i\ x \texttt{ in } u[y/y']$$

$$\begin{array}{|ll} \texttt{match } x \texttt{ with} & \\ \mid \sigma_1\ y_1 \to t & \\ \mid \sigma_2\ y_2 \to & \begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y'_1 \to u_1 \\ \mid \sigma_2\ y'_2 \to u_2 \end{array} \end{array} \approx_{\texttt{scc:merge}} \begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y_1 \to t \\ \mid \sigma_2\ y_2 \to u_2[y_2/y'_2] \end{array} \quad \text{(and symmetric)}$$

before the match; we need the merge rule to conclude.

$$\begin{array}{ll} & \begin{array}{|l} \texttt{match } x \texttt{ with} \\ \mid \sigma_1\ y_1 \to \texttt{let } x' = \pi_i\ z \texttt{ in } u_1 \\ \mid \sigma_2\ y_2 \to \texttt{let } x' = \pi_i\ z \texttt{ in } u_2 \end{array} \\ \approx_{\texttt{scc:l/l}} & \texttt{let } x' = \pi_i\ z \texttt{ in let } x' = \pi_i\ z \texttt{ in match } x \texttt{ with} \begin{array}{|l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} \\ \approx_{\texttt{scc:merge}} & \texttt{let } x' = \pi_i\ z \texttt{ in match } x \texttt{ with} \begin{array}{|l} \sigma_1\ y_1 \to u_1 \\ \sigma_2\ y_2 \to u_2 \end{array} \end{array}$$

Figure 5.5.: Equivalence of cut-free sequent proofs: weakening rules

$$\texttt{let}\ y = x\ t\ \texttt{in}\ z \approx_{\texttt{scc:weak}} z \qquad \texttt{let}\ y = \pi_i\ x\ \texttt{in}\ z \approx_{\texttt{scc:weak}} z$$

$$\texttt{match}\ x\ \texttt{with} \mid \sigma_1\ y_1 \to z \mid \sigma_2\ y_2 \to z \approx_{\texttt{scc:weak}} z \qquad \texttt{absurd}(x) \approx_{\texttt{scc:weak}} z$$

This mode of use of the merge rule does not apply to sum elimination only, but to any term former with several subterm premises. For example we can similarly derive the following equalities – under some scoping assumptions:

$$\texttt{let}\ y = x\ t\ \texttt{in}\ \texttt{let}\ z = x'\ t'\ \texttt{in}\ u \approx_{\texttt{scc}}$$

$$\texttt{let}\ z = x'\ (\texttt{let}\ y = x\ t\ \texttt{in}\ t')\ \texttt{in}\ (\texttt{let}\ y = x\ t\ \texttt{in}\ u)$$

$$\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ (t_1, t_2) \approx_{\texttt{scc}} ((\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ t_1), (\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ t_2))$$

Weakening rules Merging rules let us (de)duplicate introduction rules. The weakening rules of Figure 5.5 let us remove or introduce new rules when they do not affect the semantics of the term. In general, if x does not appear in t, then any left-introduction of x before t may be added or removed, for example $\texttt{let}\ x = \pi_i\ y\ \texttt{in}\ t \approx_{\texttt{scc}} t$.

The rules given in Figure 5.5 are more restrictive than that, as they only apply when t is a variable, but the removal of introduction rules before arbitrary terms is directly derived from the other permutation rules:

- As long as t starts with an introduction rule, we use permutation rules.
- If t is a premise-free constructor, we use one of the erasing permutation rules.
- The weakening rules are used if t is a variable.

5.2. Bureaucracy panic: why are there so many rules?

Describing this permutation equivalence is inherently quadratic in the number of different constructions in our calculus; we basically have a rule for each possible combination of two introduction rules, and sometimes several rules per combination when one of the introduction rules has more than one term premise. Why inflict upon ourselves the writing and the reading of those rules?

One methodological reason is that the unsettling feeling of redundancy resulting from this definition is a strong motivation to further developments in logic aimed at giving better presentations. It is hard to really motivate more advanced approaches when the bureaucratic way has been omitted completely, or summarily discarded by a snarky remark.

Furthermore, this formal definition of permutation equivalence is important as it serves as a baseline to compare other, better formulations of sequent equivalence to.

Finally, one thing that is very clear when looking at those rules is that the merging and weakening rules are intimately linked to the permutation rules: merging is necessary to restore a certain symmetry between left-left and left-right permutation rules, and weakening is only one source of erased subterms, the other being permutation with premise-free introductions.

Remark 5.2.1. This last point may be obvious or folklore to the people having studied intuitionistic sequent calculus (or proof nets for exponential linear logic), but it took me some time to realize this so I would like to emphasize it.

When I first encountered the need for merging and weakening rules in Scherer [2015a], I was transposing work on canonical representations in linear logic that did not have them. My first impression was that the permutation rules were "motivated by the logic", while the merging and weakening rules (called at the time "redundancy elimination rules") were "motivated (only) by pure program equivalence": they were extra-logical in nature, and had to be added separately, *a posteriori*. One reason to doubt these rules and give them a status of second-class citizens is that they have an irritating non-local character: to know whether they can be applied, it does not suffice to look at subterms/subderivations up to a fixed depth. Arbitrary comparison between subterms may be required.

It is only during my collaboration with Guillaume Munch-Maccagnoni [Munch-Maccagnoni and Scherer, 2015] that I realized that these rules were of equal status to the permutation rules. The reason they do not appear in previous work is not a difference in focus between logically motivated permutations and program equivalence, but the use of linear or intuitionistic logic. Contraction and weakening are the signature moves of intuitionistic logic, and those equivalence rules are their term equivalence counterpart.

This first-class status is made entirely clear by the careful (yet very simple) presentation of permutation equivalence – sometimes it is good to do things the old way. ∗

5.3. Properties of permutation equivalence

Lemma 5.3.1 (Local confluence of sequent reduction)**.**
If t, u_1, u_2 are sequent terms – not necessarily cut-free – with $t \triangleright_{\mathtt{R}} u_1$ and $t \triangleright_{\mathtt{R}} u_2$, then there exists r_1, r_2 such that u_i $(\to^*_{\mathtt{R}})$-reduces to r_i in at most one step, and $r_1 \approx_{\mathtt{scc}} r_2$.

Proof. To prove this one need to consider all cases where a single sequent term may be the source of two distinct head-reduction rules – that is, all pairs of overlapping reduction rules. Such a term is necessarily a cut of the form $\mathtt{let}\ x = t\ \mathtt{in}\ u$.

We will not explicitly list all overlapping pairs; below are a few representative ones:

$$\begin{array}{rl} & \mathtt{let}\ x = y\ \mathtt{in}\ \lambda z.\, t \\ \triangleright_{\mathtt{RI}} & \lambda z.\, t[y/x] \end{array} \qquad \begin{array}{rl} & \mathtt{let}\ x = y\ \mathtt{in}\ \lambda z.\, t \\ \triangleright_{\mathtt{RCrr}} & \lambda z.\, \mathtt{let}\ x = y\ \mathtt{in}\ t \\ \triangleright_{\mathtt{RI}} & \lambda z.\, t[y/x] \end{array}$$

$$\begin{array}{rl} & \mathtt{let}\ x = \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ t\ \mathtt{in}\ \lambda z.\, u \\ \triangleright_{\mathtt{RCrr}} & \lambda z.\, \mathtt{let}\ x = \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ t\ \mathtt{in}\ u \\ \triangleright_{\mathtt{RCll}} & \lambda z.\, \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ u \end{array}$$

$$\begin{array}{rl} & \mathtt{let}\ x = \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ t\ \mathtt{in}\ \lambda z.\, u \\ \triangleright_{\mathtt{RCll}} & \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ \lambda z.\, u \\ \triangleright_{\mathtt{RCrr}} & \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ \lambda z.\, \mathtt{let}\ x = t\ \mathtt{in}\ u \end{array}$$

$$\lambda z.\, \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ u \approx_{\mathtt{scc}} \mathtt{let}\ y = \pi_i\ y'\ \mathtt{in}\ \lambda z.\, \mathtt{let}\ x = t\ \mathtt{in}\ u$$

$$\begin{array}{rl} & \mathtt{let}\ x = \mathtt{absurd}(y)\ \mathtt{in}\ () \\ \triangleright_{\mathtt{RCll}} & \mathtt{absurd}(y) \end{array} \qquad \begin{array}{rl} & \mathtt{let}\ x = \mathtt{absurd}(y)\ \mathtt{in}\ () \\ \triangleright_{\mathtt{RCrr}} & () \end{array}$$

$$\mathtt{absurd}(y) \approx_{\mathtt{scc}} ()$$

□

Lemma 5.3.2 (Soundness of permutation equivalence)**.**
If t, u are sequent terms with $t \approx_{\mathtt{scc}} u$, then the translations into natural deduction λ-terms are $\beta\eta$-equivalent: $[\![t]\!]_{\mathrm{ND}} \approx_{\mathtt{extr}} [\![u]\!]_{\mathrm{ND}}$, so in particular $[\![t]\!]_{\mathrm{ND}} \approx_{\beta\eta} [\![u]\!]_{\mathrm{ND}}$.

Proof. There is no permutation or merging involving right introductions only (right-right permutations are never well-typed). We reason by case analysis on the left introductions involved in the equivalence rule under consideration.

Soundness is immediate for left rules over implication and product, as they get translated to meta-level substitutions that are identical before and after permutation. Consider for example:

$$\lambda x.\, \texttt{let}\ y = z\ t\ \texttt{in}\ u \approx_{\texttt{scc:l/r}} \texttt{let}\ y = z\ t\ \texttt{in}\ \lambda x.\, u$$

The two translations $\lambda x.\, (u[z\ t/y])$ and $(\lambda x.\, u)[z\ t/y]$ are equal.

The rules involving a split on a sum or empty type are included, once seen as natural deduction equivalences, in the extrusion relation ($\triangleright_{\texttt{extr}}$) defined in Section 3.3 (Extrusion and commuting conversions), which is itself sound with respect to $\beta\eta$-equivalence – Lemma 3.3.1 (Soundness of ($\approx_{\texttt{extr}}$)). □

5.4. Cut-free sequent proofs are standard extruded forms

The translation from λ-term to sequent terms introduces arbitrary cuts. In the other direction, notice that the translation of cut-free sequent terms gives β-normal λ-terms. For functions for example, the application forms are all of the form $x\ t$, and the substitutions performed when translating cut-free terms can never be of the form $[\lambda y.\, \_u/x]$ in a cut-free term: only cuts may substitute introduction forms.

In this section, we show two more advanced results: that cut-free sequent terms correspond to standard extruded forms, and that the permutation equivalence of cut-free terms exactly corresponds to the extrusion relation of λ-terms.

Lemma 5.4.1.
If t is a cut-free sequent term, then $[\![t]\!]_{\text{ND}}$ is a standard extruded form.

Proof. In a sequent term, elimination forms only eliminate variables: $x\ t$, $\pi_i\ x$, $\texttt{absurd}(x)$, $(\texttt{match}\ x\ \texttt{with}\ \mid \sigma_1\ x \to t_1 \mid \sigma_2\ x \to t_2)$. Those variables may be substituted by λ-terms during the translation; cuts can create substitutions by arbitrary terms, but cut-free terms may only substitute those variables by elimination forms. In particular, a variable in elimination position can never become an elimination of sum or empty type. Those eliminations may only appear at the root, or at any subterm position corresponding to a full sequent term instead of a variable; those are exactly the non-eliminated subterms of a left-introduction form, or any subterm of a right-introduction form. This characterizes a standard extruded form. □

In sequent terms, let us write $L\,[\Box]$ for *linear binding contexts*, that is sequences of unary left-introduction rules, as described in Figure 5.6 (Linear binding contexts). Note that splits on sums or absurdity are not included.

Figure 5.6.: Linear binding contexts

$$\begin{array}{lll} L ::= & & \text{binding contexts} \\ \quad \mid \Box & & \\ \quad \mid \texttt{let}\ x = y\ t\ \texttt{in}\ L & & \\ \quad \mid \texttt{let}\ x = \pi_i\ y\ \texttt{in}\ L & & \end{array}$$

The translation $[\![L]\!]_{\text{ND}}$ of a linear binding context is the unique natural deduction substitution ρ such that $[\![L\,[t]]\!]_{\text{ND}} = [\![t]\!]_{\text{ND}}[\rho]$ for any t.

Conversely, linear binding contexts are the subset of cut-free sequent term contexts whose translation is a pure substitution. This let us reason without loss of generality on the inverse image of $[\![\_]\!]_{\text{ND}}$. If the head construction of the λ-term $[\![t]\!]_{\text{ND}}$ also exists as a sequent-term term former (everything but pair projection and function applications), then t must be of the form $L\,[t']$, where t' starts with this head construction. If it does not (it is a pair projection or a function application), then t is of the form $L\,[x]$, and a `let`-form inside L corresponds to it.

Theorem 5.4.2 (Translation into standard extruded form is surjective)**.**
Any extruded normal form t in standard extruded form is the translation of some cut-free sequent term t'.

Proof. By induction on t, we build a pair of a linear binding context L and a sequent term t' such that $[\![L\,[t']\,]\!]_{\text{ND}} =_\alpha t$.

In the variable case (t is x), we return the pair $(\Box, x)$.

If t starts with an introduction form, for example $\lambda x.\, u$ or (u_1, u_2), we inductively build sequent terms for its subterms and compose them using the same introduction form.

The interesting cases are the elimination forms, that must be translated into left-introduction rules in the sequent terms. For $\pi_i\ u$ for example, we can inductively build L and u' such that $[\![L\,[u']\,]\!]_{\text{ND}} = u$, but $L\,[\texttt{let}\ x = \pi_i\ u'\ \texttt{in}\ x]$ might not be a valid sequent term; it is only valid if u' is in fact a variable.

We thus strengthen our induction hypothesis as follows: if t is an elimination rule of negative type, we guarantee that in the pair L, t', t' is in fact a variable x'.

In the $\pi_i\ u$ case, remark that u cannot be an introduction form: by typing it would be a pair construction, but this would form a β-redex and we assume that t is a normal form. It cannot be a positive elimination either, as t is in standard normal form. It must be a negative elimination or a variable, and we can thus assume a decomposition into $(L\,[\Box], y')$ such that $[\![L\,[y']\,]\!]_{\text{ND}} = u$, and we return the pair $(L\,[\texttt{let}\ x' = \pi_i\ y'\ \texttt{in}\ \Box], x')$ that satisfies our goal

$$[\![L\,[\texttt{let}\ x' = \pi_i\ y'\ \texttt{in}\ x']\,]\!]_{\text{ND}} = \pi_i\ u$$

The reasoning is similar for other elimination forms, using both hypotheses of being in β-normal form and in standard extruded form. □

Lemma 5.4.3.
All cut-free sequent terms corresponding to the same λ-term are permutatively equivalent: if t, u are cut-free and $[\![t]\!]_{\text{ND}} =_\alpha [\![u]\!]_{\text{ND}}$ then $t \approx_{\texttt{scc}} u$.

Proof. We do the proof by well-founded simultaneous induction on t, u, performing a case analysis on the head construction of their (common) λ-term translation. We will detail one case of head-construction common to λ-terms and sequent terms (pair construction) and one case of term former of λ-term that becomes a linear `let`-binding in sequent terms (pair project); the other cases inside each category are similar.

Case (r_1, r_2) If $[\![t]\!]_{\text{ND}} = (r_1, r_2) = [\![u]\!]_{\text{ND}}$, then we know that t and u must be of the form $L\,[(t'_1, t'_2)]$ and $L'\,[(u'_1, u'_2)]$ with $[\![L\,[t'_i]\,]\!]_{\text{ND}} = r_i = [\![L'\,[u'_i]\,]\!]_{\text{ND}}$ for any $i \in \{1, 2\}$. Notice that $L\,[t'_i]$ and $L'\,[u'_i]$ are strictly smaller terms than t, u: by well-fouded induction we can conclude that $L\,[t'_i] \approx_{\texttt{scc}} L'\,[u'_i]$ for any i. We can then conclude with

$$\begin{array}{ll} & t \\ = & L\,[(t'_1, t'_2)] \\ \approx_{\texttt{scc:merge}} & (L\,[t'_1], L\,[t'_2]) \\ & \text{(hyp. ind.)} \\ \approx_{\texttt{scc}} & (L'\,[u'_1], L'\,[u'_2]) \\ \approx_{\texttt{scc:merge}} & L'\,[(u'_1, u'_2)] \\ = & u \end{array}$$

Case $\pi_i\ r$ If $[\![t]\!]_{\text{ND}} = \pi_i\ r = [\![u]\!]_{\text{ND}}$, then we know that t, u must be of the form $L\,[\texttt{let}\ x = \pi_i\ t'\ \texttt{in}\ L_0\,[x]]$ and $L'\,[\texttt{let}\ x = \pi_i\ u'\ \texttt{in}\ L'_0\,[x]]$ with $L\,[t'] = r = L'\,[u']$. By well-fouded induction we thus

have that $L\,[t'] \approx_{\mathtt{scc}} L'\,[u']$, and we can conclude with

$$\begin{array}{ll} & t \\ = & L\,[\mathtt{let}\ x = \pi_i\ t'\ \mathtt{in}\ L_0\,[x]] \\ \approx_{\mathtt{scc:weak}} & L\,[\mathtt{let}\ x = \pi_i\ t'\ \mathtt{in}\ x] \\ \approx_{\mathtt{scc:l/l}} & \mathtt{let}\ x = \pi_i\ L\,[t']\ \mathtt{in}\ x \\ & \text{(hyp. ind.)} \\ \approx_{\mathtt{scc}} & \mathtt{let}\ x = \pi_i\ L'\,[u']\ \mathtt{in}\ x \\ \approx_{\mathtt{scc:l/l}} & L'\,[\mathtt{let}\ x = \pi_i\ u'\ \mathtt{in}\ x] \\ \approx_{\mathtt{scc:weak}} & L'\,[\mathtt{let}\ x = \pi_i\ u'\ \mathtt{in}\ L'_0\,[x]] \\ = & u \end{array}$$

□

Theorem 5.4.4 (Permutation equivalence is complete for cut-free sequent terms)**.** *If t, u are cut-free sequent terms with $[\![t]\!]_{\mathrm{ND}} \approx_{\mathtt{extr}} [\![u]\!]_{\mathrm{ND}}$ then $t \approx_{\mathtt{scc}} u$.*

Proof. We can rephrase the goal as follows: if t', u' are λ-terms such that $t' \approx_{\mathtt{extr}} u'$ then for any cut-free sequent terms t, u such that $[\![t]\!]_{\mathrm{ND}} = t'$ and $[\![u]\!]_{\mathrm{ND}} = u'$ we have $t \approx_{\mathtt{scc}} u$.

From Lemma 5.4.3 we know that two cut-free sequent terms translating to the same natural deduction term are permutatively equivalent. Thus it suffices to prove our goal for *some* cut-free sequent terms t, u that translate to t', u', and it will hold for all others.

The proof is by case analysis on each extrusion rule. We will detail three representative cases.

An extrusion of sum elimination

$$\sigma_j \left(\mathtt{match}\ t'\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to u'_1 \\ \sigma_2\ x_2 \to u'_2 \end{array}\right) \quad \triangleright_{\mathtt{extr}} \quad \mathtt{match}\ t'\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to \sigma_j\ u'_1 \\ \sigma_2\ x_2 \to \sigma_j\ u'_2 \end{array}$$

Using the same reasoning as in the proof of Theorem 5.4.2 (Translation into standard extruded form is surjective), we may assume that t' is the translation of a term of the form $L\,[x]$, and each u'_i of an arbytrary sequent term u. We thus have, writing $[\![t']\!]^{-1}_{\mathrm{ND}}$ for the set of sequent terms that translate to t':

$$\begin{array}{ll} & \left[\!\!\left[\sigma_j \left(\mathtt{match}\ t'\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to u'_1 \\ \sigma_2\ x_2 \to u'_2 \end{array}\right)\right]\!\!\right]^{-1}_{\mathrm{ND}} \\ \ni & L\left[\sigma_j \left(\mathtt{match}\ x\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array}\right)\right] \\ \to_{\mathtt{scc:l/r}} & L\left[\mathtt{match}\ x\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to \sigma_j\ u_1 \\ \sigma_2\ x_2 \to \sigma_j\ u_2 \end{array}\right] \\ \in & \left[\!\!\left[\mathtt{match}\ t'\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x_1 \to \sigma_j\ u'_1 \\ \sigma_2\ x_2 \to \sigma_j\ u'_2 \end{array}\right]\!\!\right]^{-1}_{\mathrm{ND}} \end{array}$$

An extrusion of absurdity

$$t'\ \mathtt{absurd}(u') \quad \triangleright_{\mathtt{extr}} \quad \mathtt{absurd}(u')$$

We can assume pre-images for t' and u' of the form $L\,[x]$ and $L'\,[y]$, and then we have

$$\begin{array}{ll} & [\![t'\ \mathtt{absurd}(u')]\!]^{-1}_{\mathrm{ND}} \\ \ni & L\,[L'\,[\mathtt{let}\ z = x\ \mathtt{absurd}(y)\ \mathtt{in}\ z]] \\ \to_{\mathtt{scc:l-\emptyset}} & L\,[L'\,[\mathtt{absurd}(y)]] \\ \in & [\![\mathtt{absurd}(u)]\!]^{-1}_{\mathrm{ND}} \end{array}$$

A merging rule

$$\texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u' \\ \sigma_2\ y_2 \to \texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to r'_1 \\ \sigma_2\ z_2 \to r'_2 \end{array} \right. \end{array} \right.$$

$$\triangleright_{\texttt{extr}} \qquad \texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u' \\ \sigma_2\ y_2 \to r'_2[y_2/z_2] \end{array} \right.$$

We can assume pre-images for t', u' and the r'_i of the form $L\,[x]$, u and r_i. Then we have

$$\begin{array}{ll} & \left[\!\!\left[\texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u' \\ \sigma_2\ y_2 \to \texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to r'_1 \\ \sigma_2\ z_2 \to r'_2 \end{array} \right. \end{array} \right. \right]\!\!\right]^{-1}_{\text{ND}} \\ \ni & L \left[\texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u \\ \sigma_2\ y_2 \to \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ z_1 \to r_1 \\ \sigma_2\ z_2 \to r_2 \end{array} \right. \end{array} \right. \right] \\ \to^*_{\texttt{scc:merge}} & L \left[\texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u \\ \sigma_2\ y_2 \to r_2[y_2/z_2] \end{array} \right. \right] \\ \in & \left[\!\!\left[\texttt{match}\ t'\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to u' \\ \sigma_2\ y_2 \to r'_2[y_2/z_2] \end{array} \right. \right]\!\!\right]^{-1}_{\text{ND}} \end{array}$$

□

5.5. η-rules for the sequent calculus

We define in Figure 5.7 (η-expansion rules for the sequent calculus) η-equivalence rules ($\approx_{\texttt{s}\eta}$) for variables (not arbitrary terms) in our sequent calculus terms. As usual, the rules are restricted to the well-typed cases, and we will use extra type annotations for readability.

Figure 5.7.: η-expansion rules for the sequent calculus

$$(x : A \to B) \triangleright_{\texttt{s}\eta} \lambda(y : A).\,\texttt{let}\ (z : B) = x\ y\ \texttt{in}\ z$$

$$(x : A_1 \times A_2) \triangleright_{\texttt{s}\eta} ((\texttt{let}\ y_1 = \pi_1\ x\ \texttt{in}\ y_1), (\texttt{let}\ y_2 = \pi_2\ x\ \texttt{in}\ y_2)) \qquad (x : 1) \triangleright_{\texttt{s}\eta} ()$$

$$(x : A_1 + A_2) \triangleright_{\texttt{s}\eta} \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ y_1 \to \sigma_1\ y_1 \\ \sigma_2\ y_2 \to \sigma_2\ y_2 \end{array} \right. \qquad (x : 0) \triangleright_{\texttt{s}\eta} \texttt{absurd}(x)$$

The η-expansion rules are the direct counterpart of the *weak* η-expansion rules of natural deduction. Note that the variable occurrences $(x : A)$ on the left do not denote *any* occurrence of the variable x in a term, only the occurrences of this variable in *term* position (the axiom rules in the sequent derivation) – not occurrences in *variable* position, in left-introduction rules. For example, in the term expression $\texttt{let}\ z = x\ y\ \texttt{in}\ t$, the variable $(x : A \to B)$ cannot be η-expanded. That would not be syntactically correct anyway, as the post-expansion term cannot be used in this position.

Lemma 5.5.1 (Soundness of $\texttt{s}\eta$-expansion)**.**
If t and u are sequent terms with $t \approx_{\texttt{s}\eta} u$, then their translations into λ-terms are η-equivalent: $[\![t]\!]_{\text{ND}} \approx_{\texttt{weak}\,\eta} [\![u]\!]_{\text{ND}}$.

Proof. Immediate: the translations of the sequent η-expansions are exactly the weak η-equivalences of Figure 3.5. □

In the other direction, it is not quite true that $t \approx_\eta u$ implies $[\![t]\!]_{\text{SEQ}} \approx_{\texttt{s}\eta} [\![u]\!]_{\text{SEQ}}$, because the translation to the sequent calculus adds cuts. We would need an equivalence that

contains both $\approx_{\mathtt{s}\eta}$ and $\approx_{\mathtt{R}}$ to be able to reason on cuts, this is done in the next section. We can still prove an easy result on units.

Definition 5.5.1 ($\approx_{\mathtt{scc}\eta}$)**.**
We define the *strong* η-equivalence for sequent calculus terms ($\approx_{\mathtt{scc}\eta}$) as the congruent union of permutation equivalence ($\approx_{\mathtt{scc}}$) and η-equivalence ($\approx_{\mathtt{s}\eta}$).

Lemma 5.5.2 (Strong unit η-equivalence)**.**
If $\Gamma \vdash t : 1$ *and* t *is cut-free then* $t \approx_{\mathtt{scc}\eta} ()$*. If* $\Gamma, x : 0 \vdash t : A$ *and* t *is cut-free then* $t \approx_{\mathtt{scc}\eta} \mathtt{absurd}(x)$.

Proof. The two cases have the same proof, by induction on t's term structure. We show that the leaves of a term can always be rewritten under the desired form, $()$ or $\mathtt{absurd}(x)$. Then we can use permutation equivalences to push this desired form "up" the term, until we have proved the whole term equivalent to them.

The leaves of a sequent term are either a variable, a $()$ or some $\mathtt{absurd}(y)$. In the variable case, we use $\mathtt{s}\eta$-equivalence to rewrite it into the desired form. If it is an introduction or elimination of the other unit form, we use the left-right permutation $() \approx_{\mathtt{scc:r}-\emptyset} \mathtt{absurd}(y)$ to swap them.

In the non-leaf case, our term is cut-free so it starts with either a left-introduction or a right-introduction rule – in the 1 case it cannot be a right-introduction rule as $()$ is a leaf case and any other would be ill-typed. Consider for example the case $\mathtt{let}\ y = \pi_i\ z\ \mathtt{in}\ u$; by induction hypothesis we can rewrite u to the desired form $()$ or $\mathtt{absurd}(x)$, and then use the corresponding ($\approx_{\mathtt{scc:l}-\emptyset}$) permutation rule to rewrite the whole term under this form. □

5.6. Equivalence of equivalences

Definition 5.6.1 ($\approx_{\mathtt{scc}\beta\eta}$)**.**
We define the $\beta\eta$-equivalence for sequent calculus terms ($\approx_{\mathtt{scc}\beta\eta}$) as the congruent union of reduction equivalence ($\approx_{\mathtt{R}}$), permutation equivalence ($\approx_{\mathtt{scc}}$) and η-equivalence ($\approx_{\mathtt{s}\eta}$).

Lemma 5.6.1 (Cut commutation)**.**

$$\mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ r \quad \approx_{\mathtt{R}} \quad \mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ r$$

Proof. By simultaneous induction on u and r. Some representative cases are listed below.

$$\begin{array}{rll}
 & \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ \sigma_i\ r & \\
\to_{\mathtt{RCrr}} & \mathtt{let}\ x = t\ \mathtt{in}\ \sigma_i\ (\mathtt{let}\ y = u\ \mathtt{in}\ r) & \\
\triangleright_{\mathtt{RCrr}} & \sigma_i\ (\mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ r) & \\
\approx_{\mathtt{R}} & \sigma_i\ (\mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ r) & \text{(hyp. ind.)} \\
\leftarrow_{\mathtt{RCrr}} & \mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \sigma_i\ (\mathtt{let}\ x = t\ \mathtt{in}\ r) & \\
\triangleleft_{\mathtt{RCrr}} & \mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ \sigma_i\ r &
\end{array}$$

$$\begin{array}{rll}
 & \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = (\mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ u)\ \mathtt{in}\ r & \\
\to_{\mathtt{RCll}} & \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ r & \\
\to_{\mathtt{RCll}} & \mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y = u\ \mathtt{in}\ r & \\
\approx_{\mathtt{R}} & \mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ \mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ r & \text{(hyp. ind.)} \\
\leftarrow_{\mathtt{RCll}} & \mathtt{let}\ y = (\mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ r & \\
\triangleleft_{\mathtt{RCll}} & \mathtt{let}\ y = (\mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ y' = \pi_i\ z\ \mathtt{in}\ u)\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ r &
\end{array}$$

□

Lemma 5.6.2 (Substitution)**.**

$$\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u]\!]_{\mathrm{SEQ}} \quad \approx_{\mathtt{R}} \quad [\![u[t/x]]\!]_{\mathrm{SEQ}}$$

Proof. By induction on the λ-term u. Some representative cases are shown below.

$$\begin{array}{rll}
 & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![(u_1,u_2)]\!]_{\mathrm{SEQ}} & \\
= & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ ([\![u_1]\!]_{\mathrm{SEQ}}, [\![u_2]\!]_{\mathrm{SEQ}}) & \\
\triangleright_{\mathtt{R}} & (\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u_1]\!]_{\mathrm{SEQ}}, \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u_2]\!]_{\mathrm{SEQ}}) & \\
\approx_{\mathtt{R}} & ([\![u_1[t/x]]\!]_{\mathrm{SEQ}}, [\![u_2[t/x]]\!]_{\mathrm{SEQ}}) & \text{(hyp. ind.)}
\end{array}$$

$$\begin{array}{rll}
 & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![\pi_i\ u]\!]_{\mathrm{SEQ}} & \\
= & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ y = [\![u]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = \pi_i\ y\ \mathtt{in}\ z & \\
\approx_{\mathtt{R}} & \mathtt{let}\ y = (\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u]\!]_{\mathrm{SEQ}})\ \mathtt{in}\ \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = \pi_i\ y\ \mathtt{in}\ z & \text{(Lemma 5.6.1)} \\
\to_{\mathtt{R}} & \mathtt{let}\ y = (\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u]\!]_{\mathrm{SEQ}})\ \mathtt{in}\ \mathtt{let}\ z = \pi_i\ y\ \mathtt{in}\ \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ z & \\
\to_{\mathtt{R}} & \mathtt{let}\ y = (\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u]\!]_{\mathrm{SEQ}})\ \mathtt{in}\ \mathtt{let}\ z = \pi_i\ y\ \mathtt{in}\ z & \\
\to_{\mathtt{R}} & \mathtt{let}\ y = [\![u[t/x]]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = \pi_i\ y\ \mathtt{in}\ z & \text{(hyp. ind.)} \\
= & [\![\pi_i\ u[t/x]]\!]_{\mathrm{SEQ}} &
\end{array}$$

□

Corollary 5.6.3 (Cut merging)**.**

$$\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ x' = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u]\!]_{\mathrm{SEQ}} \quad \approx_{\mathtt{R}} \quad \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![u[x/x']]\!]_{\mathrm{SEQ}}$$

Proof. This is an immediate consequence of Lemma 5.6.2 (Substitution): both sides are $(\approx_{\mathtt{R}})$-related to $[\![u[t/x'][t/x]]\!]_{\mathrm{SEQ}}$. □

We could prove, by induction on t, the stronger result that

$$\mathtt{let}\ x = t\ \mathtt{in}\ \mathtt{let}\ x' = t\ \mathtt{in}\ u \quad \approx_{\mathtt{R}} \quad \mathtt{let}\ x = t\ \mathtt{in}\ u[x/x']$$

but in this section we only need the specific case where subterms are in the image of the translation, and this allows use to reuse the substitution proof directly.

Lemma 5.6.4.
If $t \triangleright_\beta u$, then we have $[\![t]\!]_{\mathrm{SEQ}} \approx_{\mathtt{R}} [\![u]\!]_{\mathrm{SEQ}}$

Proof. By case analysis on the head redex of t. For example:

$$\begin{array}{rll}
 & [\![(\lambda x.\, t)\ u]\!]_{\mathrm{SEQ}} & \\
= & \mathtt{let}\ y = \lambda x.\, [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = y\ [\![u]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ z & \\
\triangleright_{\mathtt{R}} & \mathtt{let}\ y = \lambda x.\, [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = (\mathtt{let}\ x = [\![u]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![t]\!]_{\mathrm{SEQ}})\ \mathtt{in}\ z & \\
\to_{\mathtt{R}} & \mathtt{let}\ y = \lambda x.\, [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ x = [\![u]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![t]\!]_{\mathrm{SEQ}} & \\
\approx_{\mathtt{R}} & \mathtt{let}\ y = \lambda x.\, [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ [\![t[u/x]]\!]_{\mathrm{SEQ}} & \text{(Lemma 5.6.2)} \\
\approx_{\mathtt{R}} & [\![t[u/x][\lambda x.\, t/y]]\!]_{\mathrm{SEQ}} & \text{(Lemma 5.6.2)} \\
= & [\![t[u/x]]\!]_{\mathrm{SEQ}} & (y \notin t, u)
\end{array}$$

□

Lemma 5.6.5 (Soundness of sequent reduction)**.**
If $t \triangleright_{\mathtt{RC}} u$ or $t \triangleright_{\mathtt{RI}} u$ then $[\![t]\!]_{\mathrm{ND}} = [\![u]\!]_{\mathrm{ND}}$. If $t \triangleright_{\mathtt{RP}} u$ then $[\![t]\!]_{\mathrm{ND}} \triangleright_\beta^* [\![u]\!]_{\mathrm{ND}}$.

Proof. Remark that one $\mathtt{R}$-reduction step is mapped to zero, one or several β-reduction steps: the sequent terms express more sharing than natural deduction λ-terms, and the translation can thus duplicate or erase redexes. For example:

$$\begin{array}{rl}
 & [\![\mathtt{let}\ x = (t_1,t_2)\ \mathtt{in}\ \mathtt{let}\ y = \pi_i\ x\ \mathtt{in}\ r]\!]_{\mathrm{ND}} \\
= & [\![r]\!]_{\mathrm{ND}}[\pi_i\ x/y][[\![(t_1,t_2)]\!]_{\mathrm{ND}}/x] \\
= & [\![r]\!]_{\mathrm{ND}}[\pi_i\ [\![(t_1,t_2)]\!]_{\mathrm{ND}}/y][[\![(t_1,t_2)]\!]_{\mathrm{ND}}/x] \\
= & [\![r]\!]_{\mathrm{ND}}[\pi_i\ ([\![t_1]\!]_{\mathrm{ND}}, [\![t_2]\!]_{\mathrm{ND}})/y][[\![(t_1,t_2)]\!]_{\mathrm{ND}}/x] \\
\approx_{\beta\eta} & [\![r]\!]_{\mathrm{ND}}[[\![t_i]\!]_{\mathrm{ND}}/y][[\![(t_1,t_2)]\!]_{\mathrm{ND}}/x] \\
= & [\![\mathtt{let}\ x = (t_1,t_2)\ \mathtt{in}\ \mathtt{let}\ y = t_i\ \mathtt{in}\ r]\!]_{\mathrm{ND}}
\end{array}$$

□

Theorem 5.6.6 (Equi-equivalence of sequent terms and λ-terms)**.**
If t and u are sequent terms with $t \approx_{\mathtt{scc}\beta\eta} u$, then $[\![t]\!]_{\mathrm{ND}} \approx_{\beta\eta} [\![u]\!]_{\mathrm{ND}}$.
Conversely, if t and u are λ-terms with $t \approx_{\beta\eta} u$, then $[\![t]\!]_{\mathrm{SEQ}} \approx_{\mathtt{scc}\beta\eta} [\![u]\!]_{\mathrm{SEQ}}$.

Proof. The first direction, proving sequent equivalence sound relatively to natural deduction equivalence, is immediately proved by the soundness results of sequent equivalence with respect to $\beta\eta$-equivalence:

- $(\approx_{\mathtt{scc}})$ was proved sound in Lemma 5.3.2 (Soundness of permutation equivalence).
- $(\approx_{\mathtt{s}\eta})$ was proved sound in Lemma 5.5.1 (Soundness of $\mathtt{s}\eta$-expansion).
- $(\approx_{\mathtt{R}})$ was proved sound in Lemma 5.6.5 (Soundness of sequent reduction).

The second direction, proving natural deduction equivalence sound relatively to sequent equivalence, requires more work. We have proved in Lemma 5.6.4 that $(\triangleright_\beta)$ is included in $(\approx_{\mathtt{R}})$, but it remains to show that strong η-equivalence $(\approx_\eta)$ is included in sequent equivalence $(\approx_{\mathtt{scc}\beta\eta})$ – it is not included in $(\approx_{\mathtt{s}\eta})$ or $(\approx_{\mathtt{scc}\eta})$ alone. We prove this by doing a case analysis on the η-expansion.

Function case The natural deduction expansion $(t : A \to B) \triangleright_\eta \lambda x.\, t\ x$ is proved sound as follows:

$$\begin{array}{rl}
 & [\![t]\!]_{\mathrm{SEQ}} \\
\lhd_{\mathtt{RI}} & \mathtt{let}\ y = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ (y : A \to B) \\
\to_{\mathtt{s}\eta} & \mathtt{let}\ y = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \lambda x.\, \mathtt{let}\ z = y\ x\ \mathtt{in}\ z \\
\triangleright_{\mathtt{RCrl}} & \lambda x.\, \mathtt{let}\ y = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ z = y\ x\ \mathtt{in}\ z \\
= & [\![\lambda x.\, t\ x]\!]_{\mathrm{SEQ}}
\end{array}$$

Pair case The natural deduction expansion $(t : A_1 \times A_2) \triangleright_\eta (\pi_1\ t, \pi_2\ t)$ is proved sound similarly:

$$\begin{array}{rl}
 & [\![t]\!]_{\mathrm{SEQ}} \\
\lhd_{\mathtt{RI}} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ (y : A_1 \times A_2) \\
\to_{\mathtt{s}\eta} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ ((\mathtt{let}\ y_1 = \pi_1\ x\ \mathtt{in}\ y_1), (\mathtt{let}\ y_2 = \pi_2\ x\ \mathtt{in}\ y_2)) \\
\triangleright_{\mathtt{RCrl}} & \left(\begin{array}{l} \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ y_1 = \pi_1\ x\ \mathtt{in}\ y_1 \\ , \\ \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{let}\ y_2 = \pi_2\ x\ \mathtt{in}\ y_2 \end{array}\right) \\
= & [\![(\pi_1\ t, \pi_2\ t)]\!]_{\mathrm{SEQ}}
\end{array}$$

Remark 5.6.1. Note that the proof would not work if we had made a slightly different choice of η-expansion of pairs, namely from $(x : A_1 \times A_2)$ to

$$\mathtt{let}\ y_1 = \pi_1\ x\ \mathtt{in}\ \mathtt{let}\ y_2 = \pi_2\ x\ \mathtt{in}\ (y_1, y_2)$$

In the proof it is important that the η-expansion of pairs starts with a right rule, under which we can push the cut, instead of the left rule.

This difference between those two choices is not arbitrary: it can be explained using the concepts of Chapter 7 (Focusing in sequent calculus). Our product is the so-called *negative* product, whose right rule is invertible and whose left rule is not. η-expansion corresponds to the "identity expansion" property in focused systems, which always expands invertible rules first. $*$

Unit cases The cases for 0 and 1 were already proved by Lemma 5.5.2 (Strong unit η-equivalence).

Sum case In the sum case, we use the decomposition of Section 3.3 (Extrusion and commuting conversions) of the strong η-rule for sums (in natural deduction) into the weak η-rule and extrusion and commutation rules.

The weak η-rule $t \triangleright_{\mathtt{weak}\,\eta} \mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ x_1 \to \sigma_1\ x_1 \mid \sigma_2\ x_2 \to \sigma_2\ x_2$ is included in ($\mathtt{scc}\beta\eta$) using a very direct argument.

$$
\begin{array}{rl}
 & [\![t]\!]_{\mathrm{SEQ}} \\
\triangleleft_{\mathtt{RI}} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ (x : A_1 + A_2) \\
\to_{\mathtt{s}\eta} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to \sigma_1\ y_1 \mid \sigma_2\ y_2 \to \sigma_2\ y_2 \\
= & \left[\!\!\left[\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to \sigma_1\ y_1 \mid \sigma_2\ y_2 \to \sigma_2\ y_2\right]\!\!\right]_{\mathrm{SEQ}}
\end{array}
$$

The extrusion rules ($\approx_{\mathtt{extr}}$) directly correspond to the permutation rule ($\approx_{\mathtt{scc}}$). Below are a few representative examples:

$$
\sigma_j\ (\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2)\ \triangleright_{\mathtt{extr}}\ \mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to \sigma_j\ r_1 \mid \sigma_2\ y_2 \to \sigma_j\ r_2
$$

$$
\begin{array}{rl}
 & \left[\!\!\left[\sigma_j\ (\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to r_1 \mid \sigma_2\ y_2 \to r_2)\right]\!\!\right]_{\mathrm{SEQ}} \\
= & \sigma_j\ (\mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to [\![r_1]\!]_{\mathrm{SEQ}} \mid \sigma_2\ y_2 \to [\![r_2]\!]_{\mathrm{SEQ}}) \\
\triangleleft_{\mathtt{RCrl}} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ \sigma_j\ (\mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to [\![r_1]\!]_{\mathrm{SEQ}} \mid \sigma_2\ y_2 \to [\![r_2]\!]_{\mathrm{SEQ}}) \\
\approx_{\mathtt{scc}} & \mathtt{let}\ x = [\![t]\!]_{\mathrm{SEQ}}\ \mathtt{in}\ (\mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to \sigma_j\ [\![r_1]\!]_{\mathrm{SEQ}} \mid \sigma_2\ y_2 \to \sigma_j\ [\![r_2]\!]_{\mathrm{SEQ}}) \\
= & \left[\!\!\left[\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to \sigma_j\ r_1 \mid \sigma_2\ y_2 \to \sigma_j\ r_2\right]\!\!\right]_{\mathrm{SEQ}}
\end{array}
$$

$$
\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to u \mid \sigma_2\ y_2 \to \mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ z_1 \to r_1 \mid \sigma_2\ z_2 \to r_2 \quad \triangleright_{\mathtt{extr}}
$$

$$
\mathtt{match}\ t\ \mathtt{with} \mid \sigma_1\ y_1 \to u \mid \sigma_2\ y_2 \to r_2[y_2/z_2]
$$

$$
\begin{array}{rl}
 & \left[\!\left[\texttt{match}\ t\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to u \\ \sigma_2\ y_2 \to \texttt{match}\ t\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ z_1 \to r_1 \\ \sigma_2\ z_2 \to r_2 \end{array} \end{array} \right] \right]_{\text{SEQ}} \\
= & \texttt{let}\ x = [\![t]\!]_{\text{SEQ}}\ \texttt{in}\ \texttt{match}\ x\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to [\![u]\!]_{\text{SEQ}} \\ \sigma_2\ y_2 \to \texttt{let}\ x' = [\![t]\!]_{\text{SEQ}}\ \texttt{in} \;\middle|\; \begin{array}{l} \texttt{match}\ x'\ \texttt{with} \\ \sigma_1\ z_1 \to [\![r_1]\!]_{\text{SEQ}} \\ \sigma_2\ z_2 \to [\![r_2]\!]_{\text{SEQ}} \end{array} \end{array} \\
\approx_{\texttt{scc}} & \texttt{let}\ x = [\![t]\!]_{\text{SEQ}}\ \texttt{in let}\ x' = [\![t]\!]_{\text{SEQ}}\ \texttt{in match}\ x\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to [\![u]\!]_{\text{SEQ}} \\ \sigma_2\ y_2 \to \;\middle|\; \begin{array}{l} \texttt{match}\ x'\ \texttt{with} \\ \sigma_1\ z_1 \to [\![r_1]\!]_{\text{SEQ}} \\ \sigma_2\ z_2 \to [\![r_2]\!]_{\text{SEQ}} \end{array} \end{array} \\
 & \qquad \text{(Corollary 5.6.3 (Cut merging))} \\
\approx_{\texttt{R}} & \texttt{let}\ x = [\![t]\!]_{\text{SEQ}}\ \texttt{in match}\ x\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to [\![u]\!]_{\text{SEQ}} \\ \sigma_2\ y_2 \to \;\middle|\; \begin{array}{l} \texttt{match}\ x\ \texttt{with} \\ \sigma_1\ z_1 \to [\![r_1]\!]_{\text{SEQ}} \\ \sigma_2\ z_2 \to [\![r_2]\!]_{\text{SEQ}} \end{array} \end{array} \\
\approx_{\texttt{scc:merge}} & \texttt{let}\ x = [\![t]\!]_{\text{SEQ}}\ \texttt{in match}\ x\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to [\![u]\!]_{\text{SEQ}} \\ \sigma_2\ y_2 \to [\![r_2]\!]_{\text{SEQ}}[y_2/z_2] \end{array} \\
= & \left[\!\left[\texttt{match}\ t\ \texttt{with} \;\middle|\; \begin{array}{l} \sigma_1\ y_1 \to u \\ \sigma_2\ y_2 \to r_2[y_2/z_2] \end{array} \right] \right]_{\text{SEQ}}
\end{array}
$$

□

We can strengthen this result slightly by using the following roundtrip lemmas.

Lemma 5.6.7 (Natural deduction roundtrip)**.**

$$[\![[\![t]\!]_{\text{SEQ}}]\!]_{\text{ND}} =_\alpha t$$

Proof. Immediate by induction on t. For example,

$$
\begin{array}{rl}
 & [\![[\![t\ u]\!]_{\text{SEQ}}]\!]_{\text{ND}} \\
= & [\![\texttt{let}\ x = [\![t]\!]_{\text{SEQ}}\ \texttt{in let}\ y = x\ [\![u]\!]_{\text{SEQ}}\ \texttt{in}\ y]\!]_{\text{ND}} \\
= & [\![[\![t]\!]_{\text{SEQ}}]\!]_{\text{ND}}[[\![x\ [\![u]\!]_{\text{SEQ}}]\!]_{\text{ND}}/x] \\
= & [\![[\![t]\!]_{\text{SEQ}}]\!]_{\text{ND}}\ [\![[\![u]\!]_{\text{SEQ}}]\!]_{\text{ND}} \\
\overset{\text{hyp. ind.}}{=_\alpha} & t\ u
\end{array}
$$

□

Lemma 5.6.8 (Sequent calculus roundtrip)**.**

$$[\![[\![t]\!]_{\text{ND}}]\!]_{\text{SEQ}} \approx_{\texttt{scc}\beta\eta} t$$

Proof. By induction on t. For example:

$$
\begin{array}{rl}
 & [\![[\![\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ u]\!]_{\text{ND}}]\!]_{\text{SEQ}} \\
= & [\![[\![u]\!]_{\text{ND}}[\pi_i\ x/y]]\!]_{\text{SEQ}} \\
 & \qquad \text{(Lemma 5.6.2 (Substitution))} \\
\approx_{\texttt{R}} & \texttt{let}\ y = [\![\pi_i\ x]\!]_{\text{SEQ}}\ \texttt{in}\ [\![[\![u]\!]_{\text{ND}}]\!]_{\text{SEQ}} \\
\approx_{\texttt{scc}} & \texttt{let}\ y = \pi_i\ x\ \texttt{in}\ [\![[\![u]\!]_{\text{ND}}]\!]_{\text{SEQ}} \\
\overset{\text{hyp.ind.}}{\approx_{\texttt{scc}\beta\eta}} & \texttt{let}\ y = \pi_i\ x\ \texttt{in}\ u
\end{array}
$$

□

Corollary 5.6.9 (Equi-equivalence of sequent terms and λ-terms)**.**

$$t \approx_{\texttt{scc}\beta\eta} u \qquad \Longleftrightarrow \qquad [\![t]\!]_{\text{ND}} \approx_{\beta\eta} [\![u]\!]_{\text{ND}}$$

$$[\![t]\!]_{\text{SEQ}} \approx_{\texttt{scc}\beta\eta} [\![u]\!]_{\text{SEQ}} \qquad \Longleftrightarrow \qquad t \approx_{\beta\eta} u$$

Proof. We already have from Theorem 5.6.6 (Equi-equivalence of sequent terms and λ-terms) that $t \approx_{\mathtt{scc}} u$ implies $[\![t]\!]_{\mathrm{ND}} \approx_{\beta\eta} [\![u]\!]_{\mathrm{ND}}$. Conversely, from $[\![t]\!]_{\mathrm{ND}} \approx_{\beta\eta} [\![u]\!]_{\mathrm{ND}}$ the theorem implies that $[\![[\![t]\!]_{\mathrm{ND}}]\!]_{\mathrm{SEQ}} \approx_{\mathtt{scc}\beta\eta} [\![[\![u]\!]_{\mathrm{ND}}]\!]_{\mathrm{SEQ}}$. We can conclude that $t \approx_{\mathtt{scc}\beta\eta} u$ by using the roundtrip lemma for sequent calculus on both sides of the (transitive) equivalence.

The other equality is proved similarly, using the roundtrip lemma for λ-terms. □

6. Proof and type systems, in general

So far we have discussed many *proofs systems* for two *logics* (intuitionistic and classical propositional logic), and some *type systems* for programming languages.

In this chapter, we will discuss a few notions that do not depend on a particular system (of proofs or types). They make sense in any system defined by a set of inference rules plus a set of restrictions on valid proofs (or programs), equipped with a notion of equivalence between proofs (or programs).

As an application, we use some of those generic concepts to prove decidability of provability in our logic – provide algorithms that decide whether any judgment is provable or not.

6.1. Notions of proof and type systems

We define general *systems* and some properties of them. We try to be precise but we are not *formal*, and in particular will not do formal proofs on those abstract notions. They are just here to help precise description of the various systems manipulated in this thesis.

Definition 6.1.1 System**.**
Given a language of *judgments*, a proof of type *system* $\mathcal{S}, \mathcal{T}$ is given by:

- A (finite) set of *inference rules* that determine a notion of *derivations* (trees of inference rules)
- A possible set of (decidable) *restrictions* on the valid derivations (for example, "have no elimination-introduction pairs")
- A notion of *equivalence* between valid derivations (for type systems, this is uniquely determined by the equivalence between well-typed programs).

Notation 6.1.1.
We write $\Pi ::_{\mathcal{S}} \mathcal{J}$ when the derivation Π of the judgment $\mathcal{J}$ is valid in the system $\mathcal{S}$.

Definition 6.1.2 Decidability**.**
A system $\mathcal{S}$ is *decidable* if there exists a decision algorithm that, given any judgment $\mathcal{J}$ for $\mathcal{S}$, tells in finite time whether there exists a derivation for $\mathcal{J}$ valid in $\mathcal{S}$.

Definition 6.1.3 Finiteness**.**
A system $\mathcal{S}$ is *finite* if, for any judgment $\mathcal{J}$, the set of partial derivations (with some open leaves) of conclusion $\mathcal{J}$ in $\mathcal{S}$ is finite.

Notice that finiteness implies decidability.

Definition 6.1.4 Canonicity**.**
A system is *canonical* if its equivalence is trivial: two derivations are equivalent exactly if they are the same derivation.

The notion *same derivation* is a bit imprecise; if the judgments themselves have a notion of equivalence, two derivations that are equal upto equivalence of judgments are considered the same. For example, in a type system, typing derivations for α-equivalent programs are the same if they have the same tree of inference rules modulo α-equivalence.

Subsystems

Definition 6.1.5 Subsystem**.**

A system $\mathcal{S}$ is a *subsystem* of $\mathcal{T}$ if there exists a mapping from the valid proofs of $\mathcal{S}$ to valid proofs of $\mathcal{T}$ that is injective for equivalence: two proofs are equivalent in $\mathcal{S}$ if and only if their mapping is equivalent in $\mathcal{T}$.

A common way to define a subsystem is to add a restriction to an existing system (the natural deduction proofs without elimination-introduction pairs), or to remove inference rules (the cut-free sequent calculus).

In general there may be several ways to build such a mapping between proof systems. The choice of the mapping $\phi : \mathcal{S} \to \mathcal{T}$ (which we call the *inclusion mapping*) is relevant: when we speak of the subsystem $\mathcal{S}$ of $\mathcal{T}$, we are actually speaking of the pair $(\mathcal{S}, \phi)$. We would give distinct names to two subsystems that are the same proof system mapped in different way to another system, and consider them different subsystems.

Remark 6.1.1. The erasure of well-typed $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ programs into valid $\mathsf{PIL}(\to, \times, 1, +, 0)$ proofs does *not* define a subsystem, as it is not injective; for example, $\lambda x.\, \lambda y.\, x$ and $\lambda x.\, \lambda y.\, y$ are distinct programs of $A \to A \to A$, but erase to the same proof.

We may speak of *system morphism* for mappings between systems that preserve equivalence but not non-equivalence; but we will not need this generality. ∗

Definition 6.1.6 Completeness for provability**.**
A subsystem $\mathcal{S}$ of $\mathcal{T}$ is *complete for provability*, or *provability complete*, if any judgment provable in $\mathcal{T}$ is provable in the subsystem $\mathcal{S}$.

In particular, if a system admits a decidable subsystem, then it is decidable.

Definition 6.1.7 Computational completeness**.**
A subsystem $\mathcal{S}$ of $\mathcal{T}$ is *complete for equivalence*, or *computationally complete*, if for any proof $\Pi ::_{\mathcal{T}} \mathcal{J}$, there is a proof $\Pi' ::_{\mathcal{S}} \mathcal{J}$ such that Π', seen as a proof of $\mathcal{T}$, is equivalent to Π.

Computational completeness is a stronger requirement than completeness for provability: we require not only that provable judgments are preserved, but also that the identity of their derivation is preserved.

Example 6.1.1. Natural deduction for $\mathsf{PIL}(\to, \times, 1, +, 0)$ may be seen as a subsystem of $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$. They correspond to an amusing syntactic restriction on well-typed programs where, among the variables available at some type A, only the most recently introduced one may be used. (Asking to use only the oldest one would also give a valid mapping.)

Under this view, $\mathsf{PIL}(\to, \times, 1, +, 0)$ is *complete for provability* with respect to $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ (inhabited types are provable formulas), but it is not *computationally complete*: no proof derivation is mapped to a program equivalent to $\lambda x.\, \lambda y.\, x : A \to A \to A$. ◊

Finally, recall that our main goal is to decide whether a type has a *unique inhabitant*. This calls for a notion of completeness that is weaker than computational completeness, but stronger than completeness for provability.

Definition 6.1.8 Completeness for unicity**.**
A subsystem $\mathcal{S}$ of $\mathcal{T}$ is *complete for unicity*, or *unicity complete* if, whenever there exists two distinct (non equivalent) derivations of $\mathcal{J}$ in $\mathcal{T}$, then there exists two distinct derivations of $\mathcal{J}$ in $\mathcal{S}$.

In Chapter 12 (From the logic to the algorithm: deciding unicity), we decide unique inhabitation of $\mathsf{PIL}(\to, \times, 1, +, 0)$ by providing a subsystem that is both complete for unicity and finite.

We will sometimes say that a system is *more canonical* than another. The intuitive idea is that the representation of proof is closer to a canonical representation: there are less pairs of proofs that are equivalent but syntactically distinct – but there may still be some of them, to it is not necessarily *canonical* in the sense of Definition 6.1.4. We can in fact define this notion formally, as done below.

Notice that if $\mathcal{S}$ is a subsystem of $\mathcal{T}$, the inclusion mapping from $\mathcal{S}$ to $\mathcal{T}$ can be defined

independently on each equivalence class of proofs in $\mathcal{S}$ (maximal set of proofs that are equivalent to each other): the condition of being injective for equivalence exactly means that the image of an equivalence class in $\mathcal{S}$ is included in an equivalence class in $\mathcal{T}$.

Definition 6.1.9 More canonical subsystem**.**
A subsystem $\mathcal{S}$ of $\mathcal{T}$ is *weakly more canonical* than $\mathcal{T}$ if the mapping from $\mathcal{S}$ to $\mathcal{T}$ is injective for strict equality of proofs (two syntactically distinct proofs of $\mathcal{S}$ are mapped to syntactically distinct proofs of $\mathcal{T}$).

A subsystem $\mathcal{S}$ of $\mathcal{T}$ is *(strictly) more canonical* if it is *weakly more canonical* and, furthermore, for some equivalence class in $\mathcal{S}$ the restricted mapping is injective but not surjective (some proofs in $\mathcal{T}$ have no counterpart in $\mathcal{S}$).

For example, the subsystem of $(\to_R)$-normal natural deduction proofs is more canonical than the space of all natural deduction proofs, because the set of normal proofs equivalent to some normal proof Π is strictly included in the set of non-normal proofs equivalent to Π. On the contrary, the subsystem of normal disjunction-free proofs is not more canonical than the system of normal proofs. The mapping from one to another is injective but not subjective (proofs of formulas with disjunctions are not reached by the mapping), but for each equivalence class in the image the mapping is surjective: all equivalent disjunction-free proofs are present in both systems.

6.2. Rudiments of proof search

In this section, we show that provability in propositional logics (intuitionistic or classical) is decidable and equivalently, that the inhabitation problem for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ (given a typing (Γ, A), does there exists a t such that $\Gamma \vdash t : A$?) is decidable.

These decidability results are obtained by providing a subsystem that is both complete for provability and finite.

6.2.1. The subformula property

Consider any introduction rule of a sequent calculus, for example:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}$$

A remarkable property of these introduction rules is that the formulas present in the premises are also present in the conclusion, or are "included" in one of the conclusion formulas (in the sense that A is included in $A \to B$). Let us make this observation precise.

Definition 6.2.1.
A formula A is a *subformula* of another formula B, written (A $\mathsf{subformula}$ B), if A appears (syntactically) inside B. For example, A is a subformula of A and of $B \to A \times C$.

Definition 6.2.2.
A proof $\Pi :: \mathcal{J}$ has the *subformula property* if all the formulas appearing inside Π are subformulas of the formulas of the conclusion judgment $\mathcal{J}$.

A proof system has the subformula property if all its valid proof derivations have the subformula property.

As sequent calculi are based on introduction rules (left and right), with the property that the premise formulas are subformulas of the conclusion formulas, it is very easy to reason on the subformulas of a sequent proof. The only exception is the cut rule:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$$

In this rule, the formula A is arbitrary, and in particular it may not be a subformula of any of the conclusion formulas $\Gamma \cup \Delta$. The other structural rule, namely the axiom rule, does not have this issue as it has no premises.

Theorem 6.2.1 (Subformula property for cut-free sequent calculi)**.**
The cut-free proofs of the single- or multi-succedent sequent calculus have the subformula property.

Proof. Immediate by inspection of the inference rules. □

The property that subformulas in the premises are subformulas of the formulas in the conclusion does not hold for the elimination rules of natural deduction:

$$\frac{\Gamma \vdash A \to B, \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta}$$

However, if you consider proofs without elimination-introduction pairs (the analogue of the cut-free proofs of sequent calculus), the subformula property may still hold. The intuition is the following: if you inspect the proof of the large formula $A \to B$ that appears in a premise of the rule above, you cannot find an introduction rule (that would make an elimination-introduction pair), so you will only find elimination rules, building larger and larger formulas ($C \times (A \to B)$ for example), and eventually an axiom rule. But the axiom rule means that this large formula (of which $A \to B$ is a subformula) is a hypothesis of the context Γ, and Γ is already in the conclusion of the judgment; so $A \to B$ is actually a subformula of Γ.

Lemma 6.2.2.
Normal, disjunction-free natural deduction proofs have the subformula property.

Proof. The proof goes exactly as explained above: we prove inductively, for any normal proof concluded by an elimination rule on a formula A in context Γ, the invariant that A is a subformula of Γ. □

We need to exclude disjunctions to get the property that formula proved by an elimination rule is a sub-formula of the formula eliminated by this rule. This is true of the implication-elimination rule above (B $\mathsf{subformula}$ $A \to B$), but false for general disjunction eliminations: C need not be a subformula of $A_1 + A_2$ in

$$\frac{\Gamma \vdash A_1 + A_2 \qquad \Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma \vdash C}$$

However, we have proved by studying commuting conversions the Theorem 3.3.4 (Standardization by extrusion), which let us rewrite any natural deduction proof into a proof where disjunction eliminations never appear above an elimination on another connective, or as first premise of another disjunction elimination. This defines a subsystem which has of natural deduction which has the subformula property.

Theorem 6.2.3 (Subformula property for natural deduction)**.**
For any normal natural deduction proof, there exists a sequence of commuting conversions giving a proof of the same judgment satisfying the subformula property.

Proof. By induction on the simplified proof, again with the invariant that eliminated premises are subformula of the context. □

Theorem 6.2.4.
For any provable judgment $\mathcal{J}$ of $\mathsf{PIL}(\to, \times, 1, +, 0)$ or $\mathsf{PCL}(\to, \times, 1, +, 0)$, there exists a proof (both in natural deduction and sequent calculus style) of $\mathcal{J}$ satisfying the subformula property.

Lemma 6.2.5.
The set of judgments formed of subformulas of some root judgment $\mathcal{J}$ is finite.

Proof. A judgment $\mathcal{J}$ has only finitely many subformulas, and in particular there are finitely many distinct sets of those formulas. This means that a judgment $\Gamma \vdash A$ or $\Gamma \vdash \Delta$ is formed of finitely many possible succedents and finitely many hypotheses. □

If a proof of some judgment $\mathcal{J}$ has the subformula property, the judgments appearing in the proof are formed of a finite number of possible formulas. This suggests that proof

search should be decidable, as the search space of the subformula judgments is finite. However, a given judgment may occur many times inside a proof, and we need to control this to bound the search space.

6.2.2. Recurrent ancestors in derivations

The idea that we formalize here is that a given judgment need not occur twice along a path from the root of a proof to any of its leaves. The reasoning depends only on the notion of derivation tree of a judgment: it is generic over the system used.

Definition 6.2.3 Path in a derivation**.**
If $\Pi :: \mathcal{J}$ has a sub-derivation $\Pi' :: \mathcal{J}'$, we call *path from* Π *to* Π' the ordered list of sub-derivations (Π first, Π' last) that occur along the path in the derivation tree from the root of Π to the sub-derivation Π', included.

We call *path in* Π a path from Π to one of its leaves. It is an *open* or *closed* path depending on whether the leaf is an open or closed leaf.

Notation 6.2.1 Path notations**.**
We use the names $\mathcal{P}, \mathcal{Q}$ for paths. We write $\mathcal{P} : \Pi \rightsquigarrow \Pi'$ when $\mathcal{P}$ is a path from Π to Π', and $\Pi'@\mathcal{P} :: \mathcal{J}'$ when a sub-derivation Π' is at the end of the path $\mathcal{P}$. Finally, we write $\mathcal{P} < \mathcal{Q}$ when $\mathcal{P}$ is a strict prefix of the path $\mathcal{Q}$, and $\mathcal{P} \leq \mathcal{Q}$ when it is a strict prefix or $\mathcal{Q}$.

Definition 6.2.4 Occurence substitution**.**
Given a proof $\Pi :: \mathcal{J}$ with sub-proof $\Pi_1@\mathcal{P} :: \mathcal{J}'$ and another proof of the sub-judgment $\Pi_2 :: \mathcal{J}'$, we write $\Pi[\mathcal{P} \leftarrow \Pi_2]$ for the proof of $\mathcal{J}$ where the sub-proof Π_1 is replaced by Π_2. In particular, we have $\mathcal{P} : \Pi[\mathcal{P} \leftarrow \Pi_2] \rightsquigarrow \Pi_2$, and all paths that are not suffixes of $\mathcal{P}$ are identical in Π and $\Pi[\mathcal{P} \leftarrow \Pi_2]$.

Definition 6.2.5 Recurrent ancestor**.**
In a proof Π, a *recurrent ancestor* of $\Pi'@\mathcal{P} :: \mathcal{J}$ is any sub-derivation of the same judgment $\Pi''@\mathcal{Q} :: \mathcal{J}$ strictly before Π' in its path from Π ($\mathcal{Q} < \mathcal{P}$).

Definition 6.2.6 Recurrence-free**.**
A proof Π is *recurrence-free* if no subproof has a recurrent ancestor.

Theorem 6.2.6 (Provability completeness of recurrence-free subsystems)**.**
If $\mathcal{J}$ is provable, then it has a recurrence-free proof.

Proof. Suppose we have a complete proof $\Pi :: \mathcal{J}$ where a subproof $\Pi_1@\mathcal{P}$ has a recurrent ancestor $\Pi_2@\mathcal{Q}$. We can rewrite Π into $\Pi' \stackrel{\text{def}}{=} \Pi[\mathcal{Q} \leftarrow \Pi_1]$. This is a valid proof of $\mathcal{J}$ and Π_1 has strictly less recurrent ancestors in Π' than in Π.

Π' is measurably strictly smaller than Π: for example, its (multi)set of subderivations is a strict sub(multi)set of the subderivations of Π. This means that iterating this rewrite process leads, after a finite number of steps, to a recurrence-free proof of $\mathcal{J}$. □

Remark 6.2.1. Recurrent-free subsystems are in general neither computationally complete nor unicity complete. For example, a type of church integers $A \to (A \to A) \to A$ is inhabited by infinitely many well-typed λ-terms, but it has only one recurrent-free proof: the only possible term is $\lambda z.\, \lambda s.\, z$, as trying to apply a successor function $s : A \to A$ generates a subgoal with the same judgment as the root judgment. ∗

6.2.3. Decidability of provability in propositional logics

Lemma 6.2.7.
Given any proof system, the subsystem of its proofs that are recurrence-free and have the subformula property is finite.

Proof sketch. A complete search procedure in this subsystem (enumerating all partial proofs in finite time) is the following: given a judgment to prove, look for any applicable inference rules whose premises respect the subformula property – there are finitely many possible such instances of each rule. For any such inference rule, recursively search for

proofs of the premises – or fail if no rule applies.

To make this algorithm terminating, it suffices to remember during the recursive search the list of judgments along the partial path upto the current search goal. Thanks to the occurrence-freeness assumption, we can cut the search on any subgoal that already appears in this list. By the subformula property, the set of possible judgments in this list is finite, so recursion always stops after a finite number of sub-calls. If each path along the tree of recursive calls is finite, then the tree itself is finite, and the algorithm terminates. ⊔

Theorem 6.2.8 (Propositional logic is decidable)**.**
Provability is decidable in $\mathsf{PIL}(\to, \times, 1, +, 0)$ *and* $\mathsf{PCL}(\to, \times, 1, +, 0)$.

Proof. Both the natural deduction or sequent calculus proof systems for those logics have a complete for provability subsystem with the subformula property (Theorem 6.2.4). The sub-sub-system of recurrence-free proofs with the subformula property is thus complete for provability (Theorem 6.2.6). This subsystem is also finite (Lemma 6.2.7): we can decide if a judgment $\mathcal{J}$ has any proof in this subsystem, and thus (by completeness) in the whole logic. □

Note that this proof of decidability is very simple because our logic is very simple: it is quantifier-free. In presence of quantifiers, the subformula property does not hold as stated here, and decidability may become much harder to prove or even false – provability in second-order logic (with System-F-style polymorphism) is undecidable.

6.2.4. Positive and negative positions in a formula

Using the subformula property and the very regular structure of the sequent calculus, this section gives a very simple sufficient criterion to determine that a formula is not provable.

The idea is to assign a *polarity* (a sign) to parts of formula, characterizing their role: *positive* positions correspond to sub-formulas that are "outputs" of the formula (they are provided by certain proofs of the formula), while *negative* positions correspond to sub-formulas that are "inputs" of the formula (they are assumed by certain proofs of the formula).[1] For example, in the formula $(A_1 \times A_2) \to (B_1 + B_2)$, the A_i are negative (they are assumptions of any proof of the formula), and the B_i are positive (a proof of a formula contains a proof of either one).

More precisely, we say that the parameters A_1, A_2 of disjunctions $A_1 + A_2$ and conjunctions $A_1 \times A_2$ are in *positive* position. For an implication $A \to B$, we say that A is in *negative* position, while B is in *positive* position. Finally, if a sub-formula is in negative position inside a connective, its *positive* positions in the subformula become *negative* positions in the whole formula, and conversely its negative positions become positive. If the sub-formula appears in positive position, its polarities are unchanged in the whole formula.

Formally, if we write p, q for polarities among $\{-1, +1\}$ (often simply written $\{-, +\}$, and $-p$ for the polarity opposite to p, we can define the following inference rules for a judgment $A \overset{p}{\mapsto} A'$:

$$\frac{}{X \overset{p}{\mapsto} X^p} \qquad \frac{A \overset{p}{\mapsto} A' \qquad B \overset{p}{\mapsto} B'}{A + B \overset{p}{\mapsto} (A' + B')^p} \qquad \frac{A \overset{p}{\mapsto} A' \qquad B \overset{p}{\mapsto} B'}{A \times B \overset{p}{\mapsto} (A' \times B')^p} \qquad \frac{A \overset{-p}{\mapsto} A' \qquad B \overset{p}{\mapsto} B'}{A \to B \overset{p}{\mapsto} (A' \to B')^p}$$

This judgment translates a formula A into a "polarity-annotated formula A'", that has a polarity p attached to each position in A. A position in A has polarity p if it is annotated with p in the translation A' uniquely defined by $A \overset{+1}{\mapsto} A'$.

We can extend our notion of polarity to judgments $\Gamma \vdash A$, by saying that A is in positive position in the judgment, while the formulas of Γ are in negative positions. This is consistent with our idea that negative means "input" while positive means "output".

[1] Note that the notion of polarity here, positive or negative positions in formulas, is orthogonal to the one used by focusing (Chapter 7 (Focusing in sequent calculus)), of positive or negative connectives.

What make polarities an interesting notion is that polarities are preserved by sequent calculus proofs, in the following sense.

Lemma 6.2.9 (Preservation of signs)**.**
For any rule

$$\frac{\mathcal{J}_1 \qquad \dots \qquad \mathcal{J}_n}{\mathcal{J}}$$

of the sequent calculus, if a subformula of $\mathcal{J}$ appears in one of the $\mathcal{J}_i$, then it appears in a position of the same polarity.

(Note that in general it is an abuse of notation to confuse subformulas and the positions at which they appear, as two distinct positions may contain the same subformula. In the case of inference *rules*, that are schemas using formulas as meta-variable, there is no confusion as the conclusion of sequent calculus rules only mention each formula once.)

Proof. By inspection of each inference rule of the sequent calculus. Consider for example (the most interesting cases):

$$\frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

In the left-introduction proof, the formula $A \to B$ appears in negative position in the conclusion judgment, and thus its subformula A appears positively, while B appears negatively. A appears positively in the left premise, and B appears negatively in the right premise. The polarities of Γ (negative) and C (positive) are similarly preserved.

In the right-introduction rule, Γ and A appear negatively and B appears positively, both in the conclusion and in the premise.

Note that this result also holds in presence of the cut rules (in contrast to many results proved first on cut-free proofs):

$$\frac{\Gamma \vdash B \qquad \Gamma, B \vdash A}{\Gamma \vdash A}$$

The formula B does not appear in the conclusion, but the formulas of the conclusion have their polarity preserved. □

Remark 6.2.2. Establishing a corresponding result for natural deduction appears more difficult. In the elimination of implication, A appears negatively in the premise $\Gamma \vdash A \to B$, and positively in the other premise $\Gamma \vdash A$. To have a robust notion of polarities, we would need to assign polarities to the positions of premises in each inference rule. The right premise of an implication elimination would be negatively polarized, as it is "consumed" by the proof to produce its conclusion. ∗

While it seems trivial, the result of Lemma 6.2.9 (Preservation of signs) has deep consequences when combined with the subformula property. Let us consider the polarities in the three rules without premises, that are necessarily at the leaves of complete proofs:

$$\frac{}{\Gamma, A \vdash A} \qquad\qquad \frac{}{\Gamma \vdash 1} \qquad\qquad \frac{}{\Gamma, 0 \vdash 1}$$

In the axiom rule, some formula occurs in both positive and negative positions. In the other rules we have a 0 in negative position or a 1 in positive position. Remark that the axiom formula A contains either an atomic formula $X, Y, Z \dots$, or a 0, or a 1. This means that in all three rules we have either a negative 0, or a positive 1, or an atom X appearing both negatively or positively.

Theorem 6.2.10.
Any provable judgment $\mathcal{J}$ necessarily has either a 1 in positive position, a 0 in negative position, or an atom X appearing both in positive and negative position.

Proof. This is immediately proved by combining the remark above (either case holds of the subformulas at each closed leaf of the proof), the subformula property (the leaf subformulas are among the conclusion subformulas), and the preservation of signs of subformulas. □

This very simple syntactic criterion directly rejects some formulas as unprovable – the polarity invariant embeds some partial information on all possible applications of rules. For example, $(X \to Y) \to X$ is not provable. Of course, this particular result would also be obtained by doing a direct analysis of all possible proofs of the formula; our generic result does nothing more than describe the structure of a family of cases where the same form of case analysis (corresponding to the polarity invariant) always succeeds.

On the other hand, we know nothing of the provability of $(X \to 0) \to 0$: it has a 0 in negative positions, so it may be provable. Indeed, the criterion applies to all the sequent calculi seen so far, intuitionistic or classical.

7. Focusing in sequent calculus

Focusing is a discipline to create a subsystem of any proof system by studying the invertibility properties of its connectives. In some restricted cases, the resulting subsystem is canonical, which makes focusing an interesting starting point for our question of unique inhabitation. In the general case, the focused subsystem is complete for provability, and in fact computationally complete as well.

7.1. Focused proofs as a subset of non-focused proofs

We here introduce the *focusing discipline* as a set of conditions that make a (sequent) proof a valid *focused proof*. In Section 7.2 (Structural presentations of focusing: a panorama of design choices), we will present different judgment structures that *structurally* enforce the focusing discipline.

7.1.1. Invertible rules

Definition 7.1.1 Invertible rule**.**

$$\frac{\mathcal{J}_1 \qquad \mathcal{J}_2 \qquad \dots \qquad \mathcal{J}_n}{\mathcal{J}}$$

is *invertible* when the following property holds: if $\mathcal{J}$ is provable, then all of the $\mathcal{J}_1, \dots \mathcal{J}_n$ are provable as well.

Example 7.1.1 (Invertible rule)**.**

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

is invertible, as witnessed by the following "inverse derivation":

$$\frac{\dfrac{\Gamma \vdash A \to B}{\Gamma, A \vdash A \to B} \qquad \dfrac{}{\Gamma, A \vdash A}}{\Gamma, A \vdash B}$$

. ◊

Example 7.1.2 (Non-invertible rule)**.**

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}\ i \in \{1, 2\}$$

is not invertible. For example, the judgment $A_1 + A_2 \vdash A_1 + A_2$ is provable, but none of the $A_1 + A_2 \vdash A_i$ are. ◊

Invertibility is an interesting notion for goal-directed proof search: by definition, the invertible rules are those can always be used without risk of "getting stuck". This suggests that we may study a sub-system of the proofs that always apply invertible rules whenever possible, and only try non-invertible rules once no invertible rule can be applied – focusing generalizes this idea.

On the contrary, applying non-invertible rules corresponds to making a choice: if the rule is wrongly applied, the proof attempt may fail whereas another rule would have led to a solution. In a sense, non-invertible rules are the "important" rules in a proof – and in fact, we could reconstruct a proof from only the tree of its non-invertible rules.

7.1.2. Focus

Definition 7.1.2 focus**.**
We define the *focus* of a non-invertible introduction rule to be the formula introduced by the rule – it is also often called the *principal formula* of the rule. To help readability, we often underline the foci in a proof:

$$\dfrac{\dfrac{A_j \vdash B_i}{\underline{A_1 \times A_2} \vdash B_i}}{A_1 \times A_2 \vdash \underline{B_1 + B_2}}$$

7.1.3. Positive and negative connectives

Given a proof system in sequent calculus style, a connective whose right-introduction rule is non-invertible ("important") is called *positive*, and a connective whose left-introduction rule is non-invertible is called *negative*.

(This naming is consistent with the positive and negative positions in a judgment (Section 6.2.4): a connective is positive if its important rules introduce the connective in positive position.)

In the single-succedent sequent calculus for intuitionistic logic given in Figure 4.1 (Section 4.1.3), the implication and the conjunction are negative connectives, and the disjunction is a positive connective:

$$\dfrac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, \underline{A \to B} \vdash C} \qquad \dfrac{\Gamma, A_i \vdash C}{\Gamma, \underline{A_1 \times A_2} \vdash C} \qquad \dfrac{\Gamma \vdash A_i}{\Gamma \vdash \underline{A_1 + A_2}}$$

It is immediate that the conjunction and disjunction rules are non-invertible: using them removes some information from the judgment to prove. For the left-introduction of implication, non-invertibility comes from the fact that $\Gamma \vdash A$ may not be provable, when C would have been provable by using another rule.

In some proof systems, both introduction rules may be invertible for some connective. This would be the case, for example, of the following single-succedent presentation of conjunction:

$$\dfrac{\Gamma, A_1, A_2 \vdash C}{\Gamma, A_1 \times A_2 \vdash C} \qquad \dfrac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$$

In this case, we may claim the connective to be of either polarity. Some choices are more reasonable than others; we claim that this product is *positive* because, first we already have rules for a negative product (the ones of Figure 4.1) and, second, those rules are strongly related to the rules of the positive product in linear logic, where the right-introduction rule is slightly different and not invertible anymore. We discuss this further in Section 7.2.2 (Connectives invertible on both sides).

The multi-succedent presentation of intuitionistic logic (Section 4.3.3) could be given in sequent style; in this case, the right-introduction rule of the disjunction becomes invertible (both sides are, but we say this is the negative disjunction), and the right-introduction rule of the implication is not invertible anymore, so we have a positive implication.

$$\dfrac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash A_1 + A_2, \Delta} \qquad \dfrac{\Gamma \vdash A, \Delta \qquad \Gamma, A \vdash B, \Delta}{\Gamma, A \to B \vdash \Delta} \qquad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash \underline{A \to B}, \Delta}$$

In the rest of this thesis, we will always refer to the polarities suggested by the *single-succedent* presentation of $\mathsf{PIL}(\to, \times, 1, +, 0)$: implication will always be negative, disjunction will always be positive, and the product will be either positive or negative depending on our needs – it would even make sense to have both as distinct but equiprovable

connectives, distinguished in the syntax of formulas. Note that, in particular, our presentations have the non-invertible rules be exactly the left-introduction of negatives and right-introduction of positives – this would not be the case with a product invertible on both sides.

7.1.4. Invertibility and side-conditions

One difficulty with the definition of invertibility given above is that it is sensitive to the way rules are presented. Consider these two different definitions of the axiom rule:

$$\frac{}{\Gamma, A \vdash A} \qquad\qquad \frac{A \in \Gamma}{\Gamma \vdash \underline{A}}$$

The rule on the left is premise-free, so in particular it should be invertible by the definition above: its conclusion is always provable, and its premises are always provable (there are none). The rule on the right is not invertible: it may be the case that $A \notin \Gamma$, yet that the conclusion be provable by a different rule – for example if $0 \in \Gamma$.

The problem with the rule on the left is the use of non-linear pattern-matching: we use A twice in the conclusion, and this hides an implicit side-condition. Invertibility makes perfect sense for the left- and right-introduction rules of logical connectives, which do not have such non-linear patterns: each meta-variable is used exactly once.

The same subtlety occurs in multi-succedent linear logic, where the definition of the positive right unit 1 requires the context and succedent to be empty.

$$\frac{}{\emptyset \vdash 1, \emptyset} \qquad\qquad \frac{\Gamma = \emptyset \qquad \Delta = \emptyset}{\Gamma \vdash \underline{1}, \Delta}$$

The solution to this subtlety is to always consider rules with their side-conditions (equality and emptiness checks) made explicit.

Note that the axiom rule for $\Gamma, A \vdash A$ could be reformulated in two different presentations with a simple conclusion pattern:

$$\frac{A = B}{\Gamma, \underline{A} \vdash B} \qquad\qquad \frac{A \in \Gamma}{\Gamma \vdash \underline{A}}$$

Both rules are non-invertible and equi-provable, but they do not have the same focus. For now, let us make an arbitrary choice and choose the formulation on the *right*: we consider that the focus of the axiom rule is the succedent occurrence of the formula. We revisit this choice using finer-grained rules that make both options useful and interesting in Section 7.1.7 (Polarized atoms).

Credits This clarification is the outcome of a discussion with Taus Brock-Nannestad, who vehemently disagreed with my definition of invertibility and pointed out that, with a direct reading, it would make the linear positive unit 1 invertible on the right. Zakaria Chihani and myself proposed the reformulation with explicit side-conditions as a way to reveal the inherent non-invertibility of the rule.

7.1.5. The focusing phase discipline

The focusing discipline relies on exposing a structure of consecutive *phases* of a proof, and verifying that they verify certain conditions.

Definition 7.1.3 phase**.**
Phases are sets of consecutive rules of the same polarity (invertible or non-invertible), defined as the maximal sets satisfying the following properties:

- two consecutive invertible rules are part of the same invertible phase

- two consecutive non-invertible rules are part of the same non-invertible phase *if* the focus of the leafward phase is a subformula occurrence of the focus of the rootward phase

When two consecutive non-invertible rules are in the same phase, we say that they have the same focus (the focused subformula of one is a subformula occurrence of the other).

For example, we have labeled each rule of the proof below with a phase indication, using different indices for distinct phases. The two most leafward non-invertible rules (in n_2) have the same focus, the third one is part of a distinct phase (n_1).

$$\dfrac{\dfrac{\dfrac{}{0 \vdash 0 + X}\,\mathrm{i} \quad \dfrac{\dfrac{\dfrac{}{X \vdash \underline{X}}\,\mathrm{n}_2}{X \vdash \underline{0 + X}}\,\mathrm{n}_2}{\underline{1 \times X} \vdash 0 + X}\,\mathrm{n}_1}{0 + (1 \times X) \vdash 0 + X}\,\mathrm{i}}{\vdash 0 + (1 \times X) \to 0 + X}\,\mathrm{i}$$

Remark 7.1.1. In the literature, invertible phases are called *negative* phases, and non-invertible phases *positive* phases; this comes from one-sided presentations of linear logic judgments $\vdash \Delta$ with only succedents and no hypotheses, in which the non-invertible phases always manipulate positive connectives and invertible phases always manipulate negative connectives.

The adjectives *synchronous* and *asynchronous* are also in common usage since Andreoli [1992b], but I never remember which is which. (One idea would be that asynchronous rules "have more freedom", they can be applied freely, they are the invertible rules.) $*$

Definition 7.1.4 Focusing conditions**.**
To be a valid focused proof, a sequent proof must respect the following conditions. In the rest of this section, we give several examples to explain and motivate those restrictions.

1. Invertible phases must be *as long as possible*: if the premise of a rule in an invertible phase matches the conclusion of an invertible rule, then it must be the conclusion of a rule in this invertible phase.

2. Non-invertible phases must be *as long as possible*: if the premise of a rule in a non-invertible phase matches a non-invertible rule of the same focus, then it must be the conclusion of a non-invertible rule in this phase.

Example 7.1.3 (Long invertible phases)**.** Consider the two following, equivalent proofs of $\vdash X \times Y \to 1 \times X$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X \vdash 1}\,\mathrm{i} \quad \dfrac{}{X \vdash \underline{X}}\,\mathrm{n}_2}{X \vdash 1 \times X}\,\mathrm{i}}{\underline{X \times Y} \vdash 1 \times X}\,\mathrm{n}_1}{\vdash X \times Y \to 1 \times X}\,\mathrm{i} \qquad \dfrac{\dfrac{}{X \times Y \vdash 1}\,\mathrm{i} \quad \dfrac{\dfrac{}{X \vdash \underline{X}}\,\mathrm{n}_2}{\underline{X \times Y} \vdash X}\,\mathrm{n}_1}{X \times Y \vdash 1 \times X}\,\mathrm{i}}{\vdash X \times Y \to 1 \times X}\,\mathrm{i}$$

The first proof breaks the focusing discipline: a (non-invertible) left-introduction of the pair $X \times Y$ happens at a place where an invertible rule could have been used – the right-introduction rule for the pair $1 \times X$. The second proof is a valid focused proof. $\Diamond$

This restriction allows us to reason on the polarity of connectives at the beginning of a non-invertible phase. In the particular proof system we chose, the invertible rules are exactly the left-introduction of positive connectives and right-introduction of a negative connective. At the start of a non-invertible phase, no invertible rule is applicable; this means that the formulas in the hypotheses are all negative or atomic, and the formula in succedent is positive or atomic.

Example 7.1.4 (Long non-invertible phases)**.** Consider the two following proofs of $1 \times (X + Y) \vdash 0 + (Y + X)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X \vdash \underline{X}}\,\mathrm{n}_4}{X \vdash \underline{Y + X}}\,\mathrm{n}_4 \quad \dfrac{\dfrac{}{Y \vdash \underline{Y}}\,\mathrm{n}_3}{Y \vdash \underline{Y + X}}\,\mathrm{n}_3}{X + Y \vdash Y + X}\,\mathrm{i}}{\underline{1 \times (X + Y)} \vdash Y + X}\,\mathrm{n}_2}{1 \times (X + Y) \vdash \underline{0 + (Y + X)}}\,\mathrm{n}_1$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X \vdash X}\,\mathrm{n}_3}{X \vdash \underline{Y + X}}\,\mathrm{n}_3}{X \vdash \underline{0 + (Y + X)}}\,\mathrm{n}_3 \quad \dfrac{\dfrac{\dfrac{}{Y \vdash Y}\,\mathrm{n}_2}{Y \vdash \underline{Y + X}}\,\mathrm{n}_2}{Y \vdash \underline{0 + (Y + X)}}\,\mathrm{n}_2}{X + Y \vdash 0 + (Y + X)}\,\mathrm{i}}{\underline{1 \times (X + Y)} \vdash 0 + (Y + X)}\,\mathrm{n}_1$$

The first proof starts with a non-invertible phase on the focused formula $0+(X+Y)$, but then stops to perform an invertible rule. But a non-invertible rule matches the introduced subformula $Y + X$, as it is a positive on the right of the sequent; the focusing discipline is not respected. To respect the focusing discipline, one would have to introduce either X or Y, but there is not enough information in the context to know which one is provable.

In the second proof, the corresponding non-invertible rule on the goal is applied later in the proof, after the formula $X+Y$ in the context has been decomposed. It is performed in the two branches of the case analysis, with either X or Y in context, and in each branch the focused phase is complete. This proof respects the focusing discipline. ◊

An important early result about the focusing discipline is that it is complete for provability: all provable judgments have a valid focused proof.

Theorem from previous works 1 (Liang and Miller [2007]). *The subsystem of propositional intuitionistic sequent proofs which respect the focusing discipline is complete for provability.*

This is a strong result. It is rather simple to see that imposing invertible rules to be applied as easy as possible is complete – this is essentially the definition of invertibility – but imposing that the non-invertible phases be as long as possible is a much stronger restriction, and it is not at all obvious that it is complete.

We do not provide a proof of this theorem here, but we later develop completeness proofs for other focused systems – Section 10.3 (Focusing completeness by big-step translation).

7.1.6. The atomic axiom rule

Among a given class of proofs (or programs) that are equivalent to each other, some will respect the focusing discipline above and some will not. Formally, a focused subsystem is more canonical than the original, non-focused system. This selectivity is an advantage of focusing: it brings us closer to the dream land of canonical proof systems.

A common source of non-canonicity in proofs is the axiom rule:

$$\dfrac{}{\Gamma, A \vdash A}$$

For example, there are two η-equivalent proofs of $\vdash (X \to Y) \to X \to Y$:

$$\dfrac{\dfrac{}{X \to Y \vdash \underline{X \to Y}}}{\vdash \lambda x.\, x : (X \to Y) \to X \to Y} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{X \to Y, X \vdash \underline{X}} \quad \dfrac{}{X \to Y, X, Y \vdash \underline{Y}}}{X \to Y, X \vdash Y}}{X \to Y \vdash X \to Y}}{\vdash \lambda x.\, \lambda y.\, \mathtt{let}\ z = x\ y\ \mathtt{in}\ z : (X \to Y) \to X \to Y}$$

However, notice that the left proof above is not a valid focused proof. Indeed, the axiom rule is non-invertible – see Section 7.1.4 (Invertibility and side-conditions). This non-invertible rule cannot be applied while invertible rules are still applicable, and in this proof $X \to Y$ can still be (invertibly) introduced on the right.

For the same reason, the axiom rule cannot be used when the formula A starts with a positive connective, as it is then its occurrence in the context that could be invertibly introduced.

In our logic, all connectives are either positive or negative. This means that the axiom rule can only be used for formulas that do not start with a head connective, that is with atoms. It is thus exactly equivalent, under the focusing discipline, to the following *atomic axiom rule*:

$$\frac{}{\Gamma, X \vdash X}$$

7.1.7. Polarized atoms

To understand that it is non-invertible, the atomic axiom rule above can be expressed using side-conditions in two different ways:

$$\frac{X = A}{\Gamma, \underline{X} \vdash A} \qquad \frac{X \in \Gamma}{\Gamma \vdash \underline{X}}$$

The rule on the left resembles a (non-invertible) left-introduction rule, and the rule on the right resembles a (non-invertible) right-introduction rule. We could informally say that the atom X is treated as a negative connective by the left rule, and as a positive connective by the right rule.

In Section 7.1.4 (Invertibility and side-conditions) we made the arbitrary choice of using the axiom rule only when an atom is in focus on the right – we have used *negative atoms*. It is more interesting, however, to consider both options. Let us assume a given *atom polarity* function mapping any atom to a sign $\{+, -\}$. We will write X^+ when X is mapped to the positive sign, and Y^- when Y is mapped to the negative sign. We can then refine the axiom rule in two *polarized* axiom rules as follows:

$$\frac{X^- = A}{\Gamma, \underline{X^-} \vdash A} \qquad \frac{X^+ \in \Gamma}{\Gamma \vdash \underline{X^+}}$$

The left rule is associated to the *negative* polarity as it resembles a non-inversion left-introduction for a (negative) connective.

Those choices of atom polarity do not endanger the completeness theorem.

Theorem from previous works 2. *Any choice of atom polarity function preserves completeness for provability of the focused sequent-calculus.*

This formulation is not generality for the sake of generality: changing the polarity function actually enforces interesting phenomena, in particular when studying the behavior of proof search in the focused system. In particular, forcing all atoms to be negatively polarized corresponds to *backward* search, while forcing all atoms to be positively polarized corresponds to *forward* search [Chaudhuri, Pfenning, and Price, 2008b]. To understand this, let us consider the proofs of the sequent $(X \to Y), (Y \to Z), X \vdash Z$.

There are two ways to start proving this sequent. We may start from our assumption X (forward search), decide to use the implication $X \to Y$, and then we have deduced the new assumption Y. Or we may start from the goal (backward search), decide to use the implication $Y \to Z$, and then it suffices to prove Y.

$$\dfrac{\dfrac{}{X \vdash X} \quad \dfrac{?}{X \to Y, Y \to Z, X, \underline{Y} \vdash Z}}{\underline{X \to Y}, Y \to Z, X \vdash Z} \qquad \dfrac{\dfrac{?}{X \to Y, Y \to Z, X \vdash \underline{Y}} \quad \dfrac{}{Z \vdash Z}}{X \to Y, \underline{Y \to Z}, X \vdash Z}$$

In both cases this first part of the proof finishes with Y under focus, and at this point no axiom rule can be used to discharge Y, so the proof can only proceed by ending the non-invertible phase and choosing a different focus. In the left case, Y is under focus on the left; if it was a negatively polarized axiom Y^-, then the focusing discipline would

not allow us to end the non-invertible phase while a negative formula is still under focus, and the proof attempt would fail. In other words, the forward-search approach can only succeed if Y is positively polarized Y^+. Conversely, the backward-search approach can only succeed with a negatively polarized Y^-.

More generally, when a non-invertible phase reaches a positive atom focused on the right (in the succedent), this atom must be in the context (have already been deduced) or the proof attempt fails. A positive atom must first be deduced from the assumptions in context, and only then proved in the goal. This is the essence of forward search; if all atoms are positive, then focused proofs are pure forward-search proofs.

Conversely, when a non-invertible phase reaches a negative atom focused on the left, (in the context), this atom must be in the succedent, so a deduction in the context can only start when it is the goal of the proof. If all atoms are negative, then focused proofs are pure backward-search (goal-directed, demand-driven) proofs.

Remark 7.1.2. In Section 4.2.3 (Non-canonicity of cut-free sequent proofs), we gave an example of cut-free natural deduction proof that corresponds to two distinct cut-free sequent proofs. This example relied in an essential way on the trace, in the sequent calculus proof structure, of a "backward" or "forward" search process.

In a focused sequent system with polarized atoms, only one of these two cut-free sequent proof is valid – for any choice of atom polarization. In particular, cut-free focused sequent proofs in the purely negative fragment (only negative connectives and negative atoms) correspond closely to cut-free natural deduction proofs, and this enabled Herbelin [1994] to propose a term syntax for the negative fragment of sequent calculus that is very close to the λ-calculus – although this result was not presented in terms of focusing at the time. $*$

7.2. Structural presentations of focusing: a panorama of design choices

7.2.1. A first structural presentation

The restrictions of Section 7.1.5 (The focusing phase discipline) define a focused subsystem of the sequent calculus for $\mathsf{PIL}(\to, \times, 1, +, 0)$.

In this section, we define an isomorphic subsystem, not as a subset of the valid sequent proofs, but by giving a new proof system that enforces the invariant. We call this a "structural" presentation of focusing as it relies on the structure of specific focused inference rules.

The key idea is to separate sequent judgments $\Gamma \vdash A$ into four distinct judgments:

- $\Gamma \vdash_{\mathsf{inv}} A$ proves $\Gamma \vdash A$ by starting with an invertible phase
- $\Gamma \vdash_{\mathsf{foc}} B$ proves $\Gamma \vdash B$ by starting with a non-invertible phase – it will choose to focus either on the left or on the right
- $\Gamma, [A] \vdash_{\mathsf{foc.l}} B$ proves $\Gamma, A \vdash B$ by focusing on A (on the left)
- $\Gamma \vdash_{\mathsf{foc.r}} [B]$ proves $\Gamma \vdash B$ by focusing on B (on the right)

The full rules are given in Figure 7.1. In Section 7.3.1 (Explicit shifts) we give a better, more regular system, so we do not give a name to the present system which is mostly for exposition purposes.

The rules allowing to transition between judgments use explicit requirements on the polarity of formulas to enforce phases to be as long as possible. We cannot transition from the invertible judgment to the non-invertible one (ending an invertible phase) if there remain a positive on the left or an atomic on the right, that is if we could apply an invertible rule. We cannot transition from the non-invertible to the invertible judgment (ending a non-invertible phase) if the formula under focus can still be non-invertibly introduced (positive on the right, or negative on the left).

Figure 7.1.: Focused sequent calculus (single-succedent, without shifts)

SEQ-INV-IMPL-RIGHT
$$\frac{\Gamma, A \vdash_{\mathsf{inv}} B}{\Gamma \vdash_{\mathsf{inv}} A \to B}$$

SEQ-INV-DISJ-LEFT
$$\frac{\Gamma, A_1 \vdash_{\mathsf{inv}} B \qquad \Gamma, A_2 \vdash_{\mathsf{inv}} B}{\Gamma, A_1 + A_2 \vdash_{\mathsf{inv}} B}$$

SEQ-INV-CONJ-RIGHT
$$\frac{\Gamma \vdash_{\mathsf{inv}} B_1 \qquad \Gamma \vdash_{\mathsf{inv}} B_2}{\Gamma \vdash_{\mathsf{inv}} B_1 \times B_2}$$

SEQ-INV-FALSE
$$\frac{}{\Gamma, 0 \vdash_{\mathsf{inv}} B}$$

SEQ-INV-TRUE
$$\frac{}{\Gamma \vdash_{\mathsf{inv}} 1}$$

SEQ-INV-FOC
$$\frac{\Gamma \text{ negative or atomic} \qquad \Gamma \vdash_{\mathsf{foc}} B \qquad B \text{ positive or atomic}}{\Gamma \vdash_{\mathsf{inv}} B}$$

SEQ-INV-FOC-RIGHT
$$\frac{\Gamma \vdash_{\mathsf{foc.r}} [B]}{\Gamma \vdash_{\mathsf{foc}} B}$$

SEQ-INV-FOC-LEFT
$$\frac{\Gamma, [A] \vdash_{\mathsf{foc.l}} B}{\Gamma, A \vdash_{\mathsf{foc}} B}$$

SEQ-FOC-DISJ-RIGHT
$$\frac{\Gamma \vdash_{\mathsf{foc.r}} [B_i]}{\Gamma \vdash_{\mathsf{foc.r}} [B_1 + B_2]}$$

SEQ-FOC-CONJ-LEFT
$$\frac{\Gamma, [A_i] \vdash_{\mathsf{foc.l}} B}{\Gamma, [A_1 \times A_2] \vdash_{\mathsf{foc.l}} B}$$

SEQ-FOC-IMPL-LEFT
$$\frac{\Gamma \vdash_{\mathsf{foc.r}} [B] \qquad \Gamma, [A] \vdash_{\mathsf{foc.l}} C}{\Gamma, [B \to A] \vdash_{\mathsf{foc.l}} C}$$

SEQ-FOC-AXIOM-LEFT
$$\frac{}{\Gamma, [X^-] \vdash_{\mathsf{foc.l}} X^-}$$

SEQ-FOC-AXIOM-RIGHT
$$\frac{}{\Gamma, X^+ \vdash_{\mathsf{foc.r}} [X^+]}$$

SEQ-FOC-INV-LEFT
$$\frac{A \text{ positive} \qquad \Gamma, A \vdash_{\mathsf{inv}} B}{\Gamma, [A] \vdash_{\mathsf{foc.l}} B}$$

SEQ-FOC-INV-RIGHT
$$\frac{B \text{ negative} \qquad \Gamma \vdash_{\mathsf{inv}} B}{\Gamma \vdash_{\mathsf{foc.r}} [B]}$$

As an example, consider the following sequent derivation that follows the focused discipline:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X_1, Y_1 \vdash \underline{X_1}} \quad \dfrac{}{X_1, Y_1 \vdash \underline{Y_1}}}{X_1, Y_1 \vdash X_1 \times Y_1}\,\mathrm{i}}{X_1, Y_1 \vdash \underline{0 + X_1 \times Y_1}}\,\mathrm{n}_3}{X_1, \underline{Y_1 \times Y_2} \vdash 0 + X_1 \times Y_1}\,\mathrm{n}_2}{\underline{X_1 \times X_2}, Y_1 \times Y_2 \vdash 0 + X_1 \times Y_1}\,\mathrm{n}_1}{\underline{(X_1 \times X_2) \times X_3}, Y_1 \times Y_2 \vdash 0 + (X_1 \times Y_1)}\,\mathrm{n}_1$$

It corresponds to the following proof in the structural presentation, with explicit rules corresponding to phase transitions.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X_1, Y_1 \vdash_{\mathsf{foc.r}} [X_1]}}{X_1, Y_1 \vdash_{\mathsf{foc}} X_1}}{X_1, Y_1 \vdash_{\mathsf{inv}} X_1} \qquad \dfrac{\dfrac{\dfrac{}{X_1, Y_1 \vdash_{\mathsf{foc.r}} [Y_1]}}{X_1, Y_1 \vdash_{\mathsf{foc}} Y_1}}{X_1, Y_1 \vdash_{\mathsf{inv}} Y_1}}{X_1, Y_1 \vdash_{\mathsf{inv}} X_1 \times Y_1}}{X_1, Y_1 \vdash_{\mathsf{foc.r}} [X_1 \times Y_1]}}{X_1, Y_1 \vdash_{\mathsf{foc.r}} [0 + X_1 \times Y_1]}}{X_1, Y_1 \vdash_{\mathsf{foc}} 0 + X_1 \times Y_1}}{X_1, Y_1 \vdash_{\mathsf{inv}} 0 + X_1 \times Y_1}}{X_1, [Y_1] \vdash_{\mathsf{foc.l}} 0 + X_1 \times Y_1}}{X_1, [Y_1 \times Y_2] \vdash_{\mathsf{foc.l}} 0 + X_1 \times Y_1}}{Y_1 \times Y_2, X_1 \vdash_{\mathsf{foc}} 0 + X_1 \times Y_1}}{Y_1 \times Y_2, X_1 \vdash_{\mathsf{inv}} 0 + X_1 \times Y_1}}{Y_1 \times Y_2, [X_1] \vdash_{\mathsf{foc.l}} 0 + X_1 \times Y_1}}{Y_1 \times Y_2, [X_1 \times X_2] \vdash_{\mathsf{foc.l}} 0 + X_1 \times Y_1}}{Y_1 \times Y_2, [(X_1 \times X_2) \times X_3] \vdash_{\mathsf{foc.l}} 0 + (X_1 \times Y_1)}}{Y_1 \times Y_2, (X_1 \times X_2) \times X_3 \vdash_{\mathsf{foc}} 0 + (X_1 \times Y_1)}
$$

It is possible to erase such structural proofs into sequent proofs in the restricted subsystem; the phase transition rules SEQ-INV-FOC, SEQ-INV-FOC-LEFT, SEQ-INV-FOC-RIGHT, SEQ-FOC-INV-LEFT, and SEQ-FOC-INV-RIGHT are erased in the process, but it is still a one-to-one mapping: the focusing structure, explicit in the structural presentation, is implicit in the restricted presentation: it is present in the justification of why a given proof is "valid", and a given proof is valid in a unique way.

In the rest of this chapter, we study several variations on the theme of focused subsystems, exploring various parts of the design space, before settling on a formulation we like and proving its completeness – the other formulations are also complete, but we do not care for re-doing the proofs each time.

On negative contraction We define contexts as *sets* of hypotheses, and the comma notation Γ, A as *non-disjoint union*: it does not prevent Γ from containing A, and its use in a conclusion of a rule in fact corresponds (when doing leafward proof search) to a non-deterministic choice: we may implicit keep A in Γ (implicit *contraction*) or not (disjoint union). In other words, the two rules below are equi-expressive:

$$
\text{SEQ-INV-FOC-LEFT}\ \dfrac{\Gamma, [A] \vdash_{\mathsf{foc.l}} B}{\Gamma, A \vdash_{\mathsf{foc}} B}
\qquad
\text{EQUIV-SEQ-INV-FOC-LEFT}\ \dfrac{\Gamma, A, [A] \vdash_{\mathsf{foc.l}} B}{\Gamma, A \vdash_{\mathsf{foc}} B}
$$

One remarkable property of the focusing discipline is that it gives a much finer-grained control on contraction. It is possible to define a computationally complete subsystem where the comma Γ, A always means a (contraction-free) disjoint union, except in the rule that introduces left focusing: this rule really *copies* a hypothesis from the usual context to the focused position. In particular, as only negative formulas can be put under left focus, focused proofs only ever use contractions on negatives.

7.2.2. Connectives invertible on both sides

In the structural presentation, the following rules would naturally correspond to what we called the *positive* product in Section 7.1.3, written here $A \otimes B$:

$$
\dfrac{\Gamma, A_1, A_2 \vdash_{\mathsf{inv}} B}{\Gamma, A_1 \otimes A_2 \vdash_{\mathsf{inv}} B}
\qquad
\dfrac{\Gamma \vdash_{\mathsf{foc.r}} [B_1] \qquad \Gamma \vdash_{\mathsf{foc.r}} [B_2]}{\Gamma \vdash_{\mathsf{foc.r}} [B_1 \otimes B_2]}
$$

In contrast to the *negative* product, this product connective has an invertible left-introduction rule, which is rightly part of the rules for the invertible judgment $\Gamma \vdash_{\mathsf{inv}} B$. It is more troubling that the right-introduction rule, whose erasure in the usual sequent-calculus is also invertible, is part of the right-focused non-invertible judgment, which was designed for non-invertible right-introduction rules.

In presence of these rules, there is a mismatch between the "restricted subsystem" presentation of focusing, which only depends on the invertibility of rules, and would thus allow right-introduction of $\otimes$ in invertible phases only, and the "structural subsystem" presentation of focusing, which, with these rules, makes them part of the negative connectives.

Which of the subsystems should we listen to? It depends on the application we have in mind. A nice thing with this structural presentation is that it is closer to the presentation of the positive product of linear logic (the tensor), and let us build an intuition of systems with both "negative" and "positive" products without otherwise leaving the familiar setting of intuitionistic logic.

Note that it would also be possible to present a substructural system with a connective with both rules in the invertible phase, if we are willing to abandon the invariant that connective always have a focused introduction rule (left or right):

$$\frac{\Gamma, A_1, A_2 \vdash_{\mathsf{inv}} B}{\Gamma, A_1 \otimes A_2 \vdash_{\mathsf{inv}} B} \qquad \frac{\Gamma \vdash_{\mathsf{inv}} B_1 \qquad \Gamma \vdash_{\mathsf{inv}} B_2}{\Gamma \vdash_{\mathsf{inv}} B_1 \otimes B_2}$$

This corresponds to abandoning the idea, coming from linear logic, that just a $\{+, -\}$ polarity suffices to determine the invertibility of both sides.

We stay clear of this subtlety by not having a positive product in our system.

7.2.3. Polarity invariants and explicit positive contexts

In any derivation Π of an invertible sequent $\Gamma \vdash_{\mathsf{inv}} A$ we have strong invariants on the polarity of the head connective of formulas in a focused proof. Indeed, a focused judgment $\Gamma, [A] \vdash_{\mathsf{foc.l}} B$ or $\Gamma \vdash_{\mathsf{foc.r}} [C]$ can only be introduced, from an invertible phase, by a rule that enforces that Γ is negative or atomic, and C (in the right-focus case) positive or atomic – otherwise the invertible phase could be continued. Those non-focused formulas are not touched by the non-invertible introduction rules of the focused judgment, so this invariant is preserved throughout the non-invertible phase.

In fact, we will only consider focused judgments where this invariant hold. This is a consequence of considering derivations starting with an invertible phase (the common case), but even when considering derivations starting with non-invertible phases we will always assume the non-focused context (Γ above) is negative or atomic, and that the non-focused conclusion (when it exists, C above) is positive or atomic.

With this restriction, when we get to the end of a left-focused non-invertible phase,

$$\frac{A \text{ positive} \qquad \Gamma, A \vdash_{\mathsf{inv}} B}{\Gamma, [A] \vdash_{\mathsf{foc.l}} B}$$

we know that Γ is negative or atomic, and B is negative or atomic. In particular, in the invertible phase that follows, leafward from $\Gamma, A \vdash_{\mathsf{inv}} B$, neither a formula of Γ nor B can be invertibly introduced. The only invertible introductions that can follow are from A (when it is positive).

It is common to express this invariant structurally in the syntax, by making the invertible judgment three-places: $\Gamma; \Delta \vdash_{\mathsf{inv}} B$, where Γ may contain only negative or atomic formulas, while Δ may contain any formula. The rules with invertible judgments would be changed as described in Figure 7.2.

This more complex presentation guarantees structurally, for example, that no left-introduction rule is present in an invertible phase at the leaf of a right-focusing phase.

Figure 7.2.: Focused rules with three-places invertible judgment ($\Gamma; \Delta \vdash_{\mathsf{inv}} B$)

$$\frac{\Gamma; \Delta, A \vdash_{\mathsf{inv}} B}{\Gamma; \Delta \vdash_{\mathsf{inv}} A \to B} \qquad \frac{\Gamma; \Delta, A_1 \vdash_{\mathsf{inv}} B \qquad \Gamma; \Delta, A_2 \vdash_{\mathsf{inv}} B}{\Gamma; \Delta, A_1 + A_2 \vdash_{\mathsf{inv}} B}$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{inv}} B_1 \qquad \Gamma; \Delta \vdash_{\mathsf{inv}} B_2}{\Gamma; \Delta \vdash_{\mathsf{inv}} B_1 \times B_2} \qquad \frac{}{\Gamma; 0, \Delta \vdash_{\mathsf{inv}} B} \qquad \frac{}{\Gamma; \Delta \vdash_{\mathsf{inv}} 1}$$

$$\frac{\Delta \text{ negative or atomic} \qquad \Gamma, \Delta \vdash_{\mathsf{foc}} B \qquad B \text{ positive or atomic}}{\Gamma; \Delta \vdash_{\mathsf{inv}} B}$$

$$\frac{A \text{ positive} \qquad \Gamma; A \vdash_{\mathsf{inv}} B}{\Gamma, [A] \vdash_{\mathsf{foc.l}} B} \qquad \frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} B \qquad B \text{ negative}}{\Gamma \vdash_{\mathsf{foc.r}} [B]}$$

This invariant was true of previous focused systems, but not apparent in the syntax of derivations.

Remark 7.2.1. My intuition with this presentation is that, in the judgment $\Gamma; \Delta \vdash_{\mathsf{inv}} B$, the general context Δ describes the "new stuff" that has been produced by the last non-invertible phase (rootward), and is being processed by applying invertible rules, while Γ is the "old stuff" that comes from before the last non-invertible phase, and is already known to be of the expected polarity (negative or atomic). $*$

Note that this is only a point in the design space. In particular, it would be a natural extension of this idea to also distinguish two succedent places $B \mid C$, with the invariant that the position B is either empty ($\emptyset$) or has an arbitrary formula, while C is either empty or has a positive or atomic formula; and that exactly one of B or C is non-empty. We provide such a system in Figure 7.3.

At the transition from the invertible to the focused judgment, we use the notation $(B \mid C)$ to describe the union of the optional formulas at these two places; as only one of them is non-empty, we know that this represents exactly one formula.

Figure 7.3.: Focused rules with four-places invertible judgment ($\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid C$)

$$\frac{\Gamma; \Delta, A \vdash_{\mathsf{inv}} B \mid \emptyset}{\Gamma; \Delta \vdash_{\mathsf{inv}} A \to B \mid \emptyset} \qquad \frac{\Gamma; \Delta, A_1 \vdash_{\mathsf{inv}} B \mid C \qquad \Gamma; \Delta, A_2 \vdash_{\mathsf{inv}} B \mid C}{\Gamma; \Delta, A_1 + A_2 \vdash_{\mathsf{inv}} B \mid C}$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{inv}} B_1 \mid \emptyset \qquad \Gamma; \Delta \vdash_{\mathsf{inv}} B_2 \mid \emptyset}{\Gamma; \Delta \vdash_{\mathsf{inv}} B_1 \times B_2 \mid \emptyset} \qquad \frac{}{\Gamma; 0, \Delta \vdash_{\mathsf{inv}} B \mid C} \qquad \frac{}{\Gamma; \Delta \vdash_{\mathsf{inv}} 1 \mid \emptyset}$$

$$\frac{\Delta \text{ negative or atomic} \qquad \Gamma, \Delta \vdash_{\mathsf{foc}} (B \mid C) \qquad B \text{ positive or atomic or empty}}{\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid C}$$

$$\frac{A \text{ positive} \qquad \Gamma; A \vdash_{\mathsf{inv}} \emptyset \mid C}{\Gamma, [A] \vdash_{\mathsf{foc.l}} C} \qquad \frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} B \mid \emptyset \qquad B \text{ negative}}{\Gamma \vdash_{\mathsf{foc.r}} [B]}$$

In the judgment $\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid C$, the place B is non-empty if we come (rootward) from a right-focused phase: it describes the "new goal" that may need to be processed by applying invertible right-introduction rule. On the other hand, if we come from a left-focused phase, the goal has already been processed, it is kept in the "old goal" place C.

Note that this discipline does not force you to conclude with an hypothesis matching the goal as soon as it appears in the new context. An atomic hypothesis can still be used

after being moved to the old context, even if the goal itself is old, by focusing on this atomic formula again.

Remark 7.2.2. This $(A \mid B)$ notation is admittedly not canonical and potentially confusing to newcomers – one-sided systems, or classical logics with a context of succedents do not have this issue. I used to write (A, B) instead of $(A \mid B)$, and it was even more confusing. Another option is to explicitly mark the formula-or-empty positions with a question mark, for example $(A^? \mid B^?)$, but I suspect that it makes the rule even harder to read for newcomers. Finally, one could always use two separate rules instead of a single rule, but I consider that this would obfuscate the real structure of the logic, and this unnecessary duplication, if it became an established design choice, would later creep into many other rules and definitions as well. ∗

7.2.4. Batch or incremental validation of non-polarized contexts

With any of the systems seen so far, the start of a non-invertible phase is conditioned over a polarity condition on a whole context:

$$\frac{\Delta \text{ negative or atomic} \qquad \Gamma, \Delta \vdash_{\mathsf{foc}} (B, C) \qquad B \text{ positive or atomic or empty}}{\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid C}$$

Another approach is to check the polarity of individual formulas of Δ, and move them incrementally to the known-polarity context Γ. Phase transition can then happen when the non-polarized context Δ becomes empty.

$$\frac{A \text{ negative or atomic} \qquad \Gamma, A; \Delta \vdash_{\mathsf{inv}} B \mid C}{\Gamma; A, \Delta \vdash_{\mathsf{inv}} B \mid C}\ \textsc{incremental-move-left} \qquad \frac{\Gamma; \Delta \vdash_{\mathsf{inv}} \emptyset \mid B \qquad B \text{ positive or atomic}}{\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid \emptyset}\ \textsc{incremental-move-right}$$

$$\frac{\Gamma \vdash_{\mathsf{foc}} B}{\Gamma; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid B}$$

Note that the rule INCREMENTAL-MOVE-LEFT does not make much sense when A, Δ denotes a non-disjoint union (if Δ still contains A). It is thus common in the focusing literature to use multisets and/or disjoint union for the context in this position – independently of the context structure of Γ.

Remark 7.2.3. I personally prefer the "batch validation" rule (the whole context at a time), because I like to preserve a close correspondence between derivations and a term syntax I would like to use. Note that it is possible to have a light term syntax for a system with incremental validation of contexts, just by not marking the use of this rule in the term syntax:

$$\frac{A \text{ negative or atomic} \qquad \Gamma, A; \Delta \vdash_{\mathsf{inv}} t : B}{\Gamma; A, \Delta \vdash_{\mathsf{inv}} t : B}\ \textsc{incremental-move-left}$$

This (rightly) assumes that the notion of equivalence we want for our proofs and proof terms quotients over where those specific INCREMENTAL-MOVE rules are placed in the proof derivation, and over their ordering. The derivation equivalence should not be finer-grained than the term syntax. ∗

7.2.5. Forced inversion ordering

An advantage of the incremental validation of the non-polarized context is that it gives us a structural way to enforce a particular application order for invertible rules. This is

done by restricting some rules to only be applicable when the non-polarized context Δ is empty – just as we only allow phase transition when this context is empty.

Consider for example the judgment $\Gamma; A_1 + A_2 \vdash_{\mathsf{inv}} B \to C \mid \emptyset$. We have a choice of two invertible rules, one on the right and one of the left. We can enforce left introduction to happen first by using the following rules:

$$\frac{\Gamma; \Delta, A_1 \vdash_{\mathsf{inv}} B \mid C \qquad \Gamma; \Delta, A_2 \vdash_{\mathsf{inv}} B \mid C}{\Gamma; \Delta, A_1 + A_2 \vdash_{\mathsf{inv}} B \mid C} \qquad \frac{\Gamma; A \vdash_{\mathsf{inv}} B \mid \emptyset}{\Gamma; \emptyset \vdash_{\mathsf{inv}} A \to B \mid \emptyset}$$

The rule on the left is the usual rule for left-introduction of disjunctions, but the rule on the right enforces that the non-polarized context be empty for right-introduction of implication to be allowed. In our example $\Gamma; A_1 + A_2 \vdash_{\mathsf{inv}} B \to C \mid \emptyset$, this forces us to introduce the sum, keep introducing the A_i until we get negative or atoms in the context, use INCREMENTAL-MOVE to put them in the negative context Γ, and only then introduce $A \to B$.

Conversely, we could enforce right-introductions to happen first using the following rules:

$$\frac{\Gamma; \Delta, A_1 \vdash_{\mathsf{inv}} \emptyset \mid B \qquad \Gamma; \Delta, A_2 \vdash_{\mathsf{inv}} \emptyset \mid B}{\Gamma; \Delta, A_1 + A_2 \vdash_{\mathsf{inv}} \emptyset \mid B} \qquad \frac{\Gamma; \Delta, A \vdash_{\mathsf{inv}} B \mid \emptyset}{\Gamma; \Delta \vdash_{\mathsf{inv}} A \to B \mid \emptyset}$$

Finally, we can even enforce the ordering of left-introduction rules by making Δ an *ordered list* (ordered multiset), where only the left-most assumption can be introduced (or moved to the negative or atomic context). We give in Figure 7.4 a description of invertible rules in this style, built on top of the four-places judgment of the system of Figure 7.3. To emphasize that Δ is an ordered list, we use $[]$ for the empty list and $A :: \Delta$ for adding a formula A to the left of a list Δ.

Figure 7.4.: Focused rules with unique invertible ordering

$$\frac{\Gamma; A_1 :: \Delta \vdash_{\mathsf{inv}} B \mid C \qquad \Gamma; A_2 :: \Delta \vdash_{\mathsf{inv}} B \mid C}{\Gamma; (A_1 + A_2) :: \Delta \vdash_{\mathsf{inv}} B \mid C} \qquad \frac{\Gamma; A \vdash_{\mathsf{inv}} B \mid \emptyset}{\Gamma; [] \vdash_{\mathsf{inv}} A \to B \mid \emptyset}$$

$$\frac{\Gamma; [] \vdash_{\mathsf{inv}} B_1 \mid \emptyset \qquad \Gamma; [] \vdash_{\mathsf{inv}} B_2 \mid \emptyset}{\Gamma; [] \vdash_{\mathsf{inv}} B_1 \times B_2 \mid \emptyset} \qquad \frac{}{\Gamma; 0 :: \Delta \vdash_{\mathsf{inv}} B \mid C} \qquad \frac{}{\Gamma; [] \vdash_{\mathsf{inv}} 1 \mid \emptyset}$$

$$\frac{A \text{ negative or atomic} \qquad \Gamma, A; \Delta \vdash_{\mathsf{inv}} B \mid C}{\Gamma; A :: \Delta \vdash_{\mathsf{inv}} B \mid C} \qquad \frac{\Gamma; [] \vdash_{\mathsf{inv}} \emptyset \mid C \qquad B \text{ positive or atomic}}{\Gamma; [] \vdash_{\mathsf{inv}} B \mid \emptyset}$$

$$\frac{\Gamma \vdash_{\mathsf{foc}} C}{\Gamma; [] \vdash_{\mathsf{inv}} \emptyset \mid C}$$

The system of Figure 7.4 has a strong left-to-right slant: left-introduction rules are always preferred to right-introduction rules, which are blocked until the non-polarized left context is emptied. In fact, the invertible rules of this system are syntax-directed, in the sense that for any invertible judgment there is exactly one applicable rule. This implies that each invertible phase is uniquely determined by its rootwardmost judgment.

This is a very interesting property because, in all the systems we are interested in, permuting the order of invertible rules preserves equivalence. In particular, a subsystem that structurally enforces a unique order can be computationally complete, and is strictly more canonical.

It is also convenient for a software implementation of focused proof search, as it interleaves the processing of invertibly-introducible formulas with the check that all such formulas have been introduced, instead of having to repeatedly traverse the new context. The software prototypes developed during this thesis used this approach.

However, I personally dislike the fact that arbitrary choices had to be made to give such a definition. There are many different ways to restrict the order of invertible rules to make it canonical, and no good reason to prefer one over another.

Remark 7.2.4. Arguably, the left-to-right order presented in Figure 7.4 would be natural in a dependently typed system (as the type of the right-hand side may depend on the shape of a value of the context, and left-introduction may thus expose new right-introduction opportunities). However there are deep, difficult obstacles to the combination of sequent calculus and dependent types in presence of connectives of both polarities [Herbelin, 2005]. Focusing Π-types is well-understood, see Lengrand, Dyckhoff, and McKinna [2011]. On the contrary, at the time of writing, no satisfactory sequent calculus with strong dependent Σ-types has been proposed. ∗

7.2.6. Invertible commuting conversions

The equivalence induced by permuting the ordering of invertible rules is a form of *commuting conversion* as we discussed in Section 3.3. Note that general commuting conversions allow other commutations, typically extruding a sum elimination (or elimination of absurdity) out of a non-invertible rule.

We call *invertible commuting conversions*, noted by the relation $(\approx_{\mathtt{icc}})$, the equivalence relation generated by reordering two consecutive invertible rules. Note that when a rule with premises is reordered was rootward of a rule without premises (0-left or 1-right), reordering it leafward makes it disappear. For the sequent calculus, it is generated by the equality schemes of Figure 7.5 (Invertible commuting conversions for the sequent calculus) (restricted to well-typed instances), which are a restriction of (the sequent-calculus equivalent of) the equivalence relation corresponding to the *extrusion* relation of Section 3.3 (Extrusion and commuting conversions).

Note that there are rules permuting invertible left-introduction and right-introduction rules, and rules that permute two left-introduction rules, but no rules permuting two right-introduction rules. There is no right-right permutation that would preserve typing; this is a conference of the fact that this presentation of (focused) intuitionistic logic is single-succedent, in a multi-succedent system we could have right-right permutations.

This equivalence relation, which is entirely local to invertible phases, captures the "don't care" non-determinism that is present in formulations of focusing that do not enforce an ordering on invertible rules, and is removed in forced-ordering presentations.

Fact 7.2.1.
All subsystems of focused sequent calculus (as defined in Figure 7.1 (Focused sequent calculus (single-succedent, without shifts))) with unique invertible ordering, such as the one defined in Figure 7.4 (Focused rules with unique invertible ordering), are isomorphic to the quotient of focused sequent calculus by the invertible commuting conversion relation $(\approx_{\mathtt{icc}})$ *defined in Figure 7.5 (Invertible commuting conversions for the sequent calculus).*

Remark 7.2.5. Some authors of focused systems criticize commuting conversions, or more precisely their use to prove completeness of focusing, for requiring definitions and proofs whose sizes are quadratic in the number of logical connective of the systems. Other completeness proof techniques [Chaudhuri, 2008, Simmons, 2011] do not rely directly on commuting invertible rules, and only manipulate objects and proofs that scale linearly with respect to the number of connectives of the system.

This is a good argument, but I still believe that describing commuting conversions is important. Proving completeness of a focused system with unique inversion ordering may be simpler, but it is also an arguably weaker result, as it does not explicitly account for the "don't care non-determinism" inherent in invertible rules.

∗

Figure 7.5.: Invertible commuting conversions for the sequent calculus

$$\lambda y.\, \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to u_1 \mid \sigma_2\ z_2 \to u_2 \quad \approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to \lambda y.\, u_1 \mid \sigma_2\ z_2 \to \lambda y.\, u_2$$

$$\left(t, \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to u_1 \mid \sigma_2\ z_2 \to u_2\right) \quad \approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to (t, u_1) \mid \sigma_2\ z_2 \to (t, u_2)$$

$$\left(\mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to u_1 \mid \sigma_2\ z_2 \to u_2\ , t\right) \quad \approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to (u_1, t) \mid \sigma_2\ z_2 \to (u_2, t)$$

$$() \quad \approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ z_1 \to () \mid \sigma_2\ z_2 \to ()$$

$$\lambda y.\, \mathtt{absurd}(x) \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

$$(t, \mathtt{absurd}(x)) \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

$$(\mathtt{absurd}(x), t) \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

$$() \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

$$\mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to u_1 \mid \sigma_2\ y_2 \to u_2 \mid \sigma_2\ z_2 \to t$$

$$\approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to \mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to u_1 \mid \sigma_2\ z_2 \to t \mid \sigma_2\ y_2 \to \mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to u_2 \mid \sigma_2\ z_2 \to t$$

$$\mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to t \mid \sigma_2\ z_2 \to \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to u_1 \mid \sigma_2\ y_2 \to u_2$$

$$\approx_{\mathtt{icc}} \quad \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ y_1 \to \mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to t \mid \sigma_2\ z_2 \to u_1 \mid \sigma_2\ y_2 \to \mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to t \mid \sigma_2\ z_2 \to u_2$$

$$\mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to t \mid \sigma_2\ z_2 \to \mathtt{absurd}(x) \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

$$\mathtt{match}\ x'\ \mathtt{with} \mid \sigma_1\ z_1 \to \mathtt{absurd}(x) \mid \sigma_2\ z_2 \to t \quad \approx_{\mathtt{icc}} \quad \mathtt{absurd}(x)$$

7.3. Polarized formulas

Once a choice of polarization has been made for atoms, all formulas are either positive or negative, and can thus be split in two grammatical categories. Recent presentations of focused systems often make the transitions from one grammatical category to another explicit by placing *shifts* in the syntax.

7.3.1. Explicit shifts

We write $\langle N \rangle^+$ for a negative formula embedded into the set of positive formulas, and conversely $\langle P \rangle^-$ for a positive formula seen as a negative formula. The complete grammar of those *polarized formulas* is defined in Figure 7.6.

Figure 7.6.: Polarized propositional formulas

P, Q	$::=$		positive formulas
		$\mid X^+$	positive atom
		$\mid P + Q$	sum
		$\mid 0$	false
		$\mid \langle N \rangle^+$	shift
N, M	$::=$		negative formulas
		$\mid X^-$	negative atom
		$\mid P \to N$	implication
		$\mid N \times M$	product
		$\mid 1$	true
		$\mid \langle P \rangle^-$	shift
$P^{\mathsf{at}}, Q^{\mathsf{at}}$	$::= P \mid X^-$		positive or atomic
$N^{\mathsf{at}}, M^{\mathsf{at}}$	$::= N \mid X^+$		negative or atomic
Σ	$::= \emptyset \mid \Sigma, P$		positive context
Γ^{at}	$::= \emptyset \mid \Gamma^{\mathsf{at}}, N^{\mathsf{at}}$		negative or atomic context

Notice that, while the (positive) sum expects positive subformulas and (negative) product expects negative subformula, the (negative) implication expects a positive subformula on its left-hand side. The polarity of this sub-formula gets reversed because it is in "negative position" in the sense of Section 6.2.4 (Positive and negative positions in a formula).

The rules for this calculus, readily adapted from the previous presentations, are given in Figure 7.7 (Focused sequent calculus for polarized propositional intuitionistic logic).

Remark 7.3.1. This notation for shifts is not standard in the focusing literature. The standard notation is to write $\downarrow N$ for $\langle N \rangle^+$, and $\uparrow P$ for $\langle P \rangle^-$. I strongly dislike this standard notation because I can never, ever remember which is which. (This is not quite true: an effective mnemonic is that $\downarrow$ looks somewhat like the bang (!) of linear logic, and bang is isomorphic to a tensor ($!A \simeq !A \otimes !A$) so it is positive, and thus $\downarrow$ returns a positive formula. I suffered months of head-scratching before I got this tip, and I still resent the notation and refuse to inflict it on innocent bystanders.). The notation used here emphasizes the polarity of the result of the shift: $\langle A \rangle^-$ is negative because the minus sign is outside the box, $\langle B \rangle^+$ is positive. In fact we could even use $\langle {}^+A \rangle^-$ and $\langle {}^-B \rangle^+$ to be heavily explicit on the inside-outside polarity shift. ∗

While this may seem a minor grammatical difference, adding explicit shifts is in fact a radical idea, because it let us make distinction between formulas that we couldn't distinguish before: a formula can be shifted to the other polarity, and then shifted back to its original polarity, and we obtain a different formula!

Example 7.3.1. The formulas $P \to Q \to N$ and $P \to \langle \langle Q \to N \rangle^+ \rangle^-$ have proofs that differ in very interesting ways: a focused proof of the first formula necessarily starts by invertibly introducing both P and Q in context; but for the second formula, the invertible phase stops after introducing P in context, as the formula $\langle Q \to N \rangle^+$ is positive and

Figure 7.7.: Focused sequent calculus for polarized propositional intuitionistic logic

$$\frac{\Gamma^{\mathsf{at}}, X^+; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; X^+, \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}, M; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \langle M \rangle^+, \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mid X^-}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} X^- \mid}$$

$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mid P}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \langle P \rangle^- \mid} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma, P \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} P \to N \mid} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \times N_2 \mid}$$

$$\frac{\Gamma^{\mathsf{at}}; \Sigma, Q_1 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, Q_1 + Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{}{\Gamma^{\mathsf{at}}; \Sigma, 0 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} 1 \mid} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \vdash_{\mathsf{inv}} \mid P^{\mathsf{at}}}$$

$$\frac{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, N \vdash_{\mathsf{foc}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P}$$

$$\frac{\Gamma^{\mathsf{at}}, [N_i] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [N_1 \times N_2] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [Q] \qquad \Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [Q \to N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [\langle Q \rangle^-] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}$$

$$\frac{}{\Gamma^{\mathsf{at}}, [X^-] \vdash_{\mathsf{foc.l}} X^-} \qquad \frac{}{\Gamma^{\mathsf{at}}, X^+ \vdash_{\mathsf{foc.r}} [X^+]} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P_i]}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P_1 + P_2]} \qquad \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^+]}$$

may thus not be invertibly introduced. A focused proof may thus perform arbitrary left-focusing phases on the context at this point, before focusing on this positive formula, unboxing its negative content, and then introducing the second function type. ◊

Remark 7.3.2. This is again an instance of the general trick that introducing finer-grained distinctions reveals interesting phenomena. This happened when moving from un-polarized to polarized axioms, and it is also an argument for viewing intuitionistic logic as a refinement of classical logic – through the double-negation translations. For a development of this argument, see Noam Zeilberger's lecture notes [Zeilberger, 2013]. ∗

All the proof and type systems so far were defined over the same grammar of formulas, the formulas of propositional logic. We are now going to define a proof system on the distinct grammar of formulas of *polarized* propositional logic. In particular, we should be careful about the relation between the proof systems so defined. For example, if I know that a formula is provable on one side, are there formulas that I know are provable on the other?

We study the direct relations between polarized and non-polarized formulas in Section 7.4 (Direct relations between focused and non-focused systems), and prove a stronger completeness result in Section 10.3 (Focusing completeness by big-step translation).

7.3.2. Batch validation of polarized contexts

In Section 7.2.4 (Batch or incremental validation of non-polarized contexts) we pointed out that, in focusing systems with no explicit shifts, we can choose between "batch" or "incremental" moving of decomposed formulas from the general invertible context to the negative-or-atomic context necessary for the non-invertible phase.

The explicit shift syntax favors incremental move rules. In the invertible judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$, first it is natural to separate two left contexts, one Γ^{at} for negative or atomic formulas that are the main context of non-invertible phases, and the other Σ for positive formulas. Then, it is again a natural choice to inspect each positive formula $P \in \Sigma$ in turn, and handle all possible cases: if it starts with a positive connective, we

apply an introduction rule, otherwise we move it into the context Γ^{at}. Same thing with the two succedent places on the right.

$$\frac{\Gamma^{\mathsf{at}}, X^+; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; X^+, \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}, M; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \langle M \rangle^+, \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}$$

$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mid X^-}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} X^- \mid} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mid P}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \langle P \rangle^- \mid}$$

$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid P^{\mathsf{at}}}$$

It is, however, still possible to present this system with a "batch" move rule. This makes derivation trees more concise, closer to the program terms we will eventually write, and also reflects in the system design the fact that our notion of identity will be oblivious to the placement and ordering of incremental move rules. (And it does not require manipulating a context mixing formulas of both polarities, which would have been an unpalatable design choice.)

To do this, we first define two partial shifting functions $\langle N^{\mathsf{at}} \rangle^{+\mathsf{at}}$ (respectively $\langle P^{\mathsf{at}} \rangle^{-\mathsf{at}}$) that take a negative or atomic (respectively positive or atomic) formula and turns it into a positive or atomic (respectively negative or atomic) formula.

$$\langle X^+ \rangle^{+\mathsf{at}} \stackrel{\mathsf{def}}{=} X^+ \qquad \langle N \rangle^{+\mathsf{at}} \stackrel{\mathsf{def}}{=} \langle N \rangle^+$$

$$\langle X^- \rangle^{-\mathsf{at}} \stackrel{\mathsf{def}}{=} X^- \qquad \langle P \rangle^{-\mathsf{at}} \stackrel{\mathsf{def}}{=} \langle P \rangle^-$$

Then, with a natural extension of this notation to whole contexts $\langle \Gamma^{\mathsf{at}} \rangle^{+\mathsf{at}}$, $\langle \Sigma^{\mathsf{at}} \rangle^{-\mathsf{at}}$, it is easy to capture the notion that a positive context only has "negative or atomic" formulas left: then it must be equal to $\langle \Gamma^{\mathsf{at}} \rangle^{+\mathsf{at}}$ for some negative or atomic Γ^{at}. We can remove incremental move rules and have a single rule transitioning from the invertible to the non-invertible judgment as follows:

$$\frac{\Sigma = \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} P^{\mathsf{at}} \qquad N = \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N}$$

Note that there is still considerable flexibility in the choice of presentation. Here we have chosen to have two places for contexts, but go back to a single-succedent presentation – which incurs a minor loss of information in the rules ending right focusing. We could keep the two-succedent presentation, or even have a single context on the left. A minor detail we will change is to use a more implicit presentation of the side-conditions:

$$\frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}}}$$

In our experience, the incremental move rules are a better choice when focusing on the logical aspects of the system (they are more systematic), and the batch presentation is more convenient when manipulating proof terms – which will come, in time, to focused systems.

In Figure 7.8 we give the full focused proof system with batch context validation, to reference it more easily in later sections.

Figure 7.8.: Focused sequent calculus with polarized formulas and batch context validation

$$\frac{\Gamma^{\mathsf{at}}; \Sigma, P \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} P \to N \mid} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \times N_2 \mid} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma, Q_1 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, Q_1 + Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}$$

$$\frac{}{\Gamma^{\mathsf{at}}; \Sigma, 0 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} 1 \mid} \qquad \frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

$$\frac{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, N \vdash_{\mathsf{foc}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P}$$

$$\frac{\Gamma^{\mathsf{at}}, [N_i] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [N_1 \times N_2] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [Q] \quad \Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [Q \to N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [\langle Q \rangle^{-}] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}$$

$$\frac{}{\Gamma^{\mathsf{at}}, [X^{-}] \vdash_{\mathsf{foc.l}} X^{-}} \qquad \frac{}{\Gamma^{\mathsf{at}}, X^{+} \vdash_{\mathsf{foc.r}} [X^{+}]} \qquad \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P_i]}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P_1 + P_2]} \qquad \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^{+}]}$$

7.4. Direct relations between focused and non-focused systems

7.4.1. Defocusing

When relating non-focused proof systems with focused systems with polarized formulas, the most direct result is that focused proofs are also valid non-focused proofs. This is self-evident when focused proofs are defined as the subset of non-focused proofs that satisfy the focusing discipline, but requires an erasure step for the structural presentations of focusing, in particular when using explicit shifts (that is, a different structure for formulas).

In Figure 7.9 (Polarity erasure), we define the polarity erasure operations $\lfloor P \rfloor_{\pm}$ and $\lfloor N \rfloor_{\pm}$ that return a formula without explicit shifts. It is readily extended to contexts.

Figure 7.9.: Polarity erasure

$$\begin{array}{lcl} \lfloor X^{+} \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & X \\ \lfloor P + Q \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & \lfloor P \rfloor_{\pm} + \lfloor Q \rfloor_{\pm} \\ \lfloor 0 \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & 0 \\ \lfloor \langle N \rangle^{+} \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & \lfloor N \rfloor_{\pm} \end{array} \qquad \begin{array}{lcl} \lfloor Y^{-} \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & Y \\ \lfloor P \to N \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & \lfloor P \rfloor_{\pm} \to \lfloor N \rfloor_{\pm} \\ \lfloor N \times M \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & \lfloor N \rfloor_{\pm} \times \lfloor M \rfloor_{\pm} \\ \lfloor 1 \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & 1 \\ \lfloor \langle P \rangle^{-} \rfloor_{\pm} & \stackrel{\mathsf{def}}{=} & \lfloor P \rfloor_{\pm} \end{array}$$

We can then erase any proof derivation from a focused proof to a non-focused proof.

Theorem 7.4.1 (Polarity erasure)**.**
The following provability implications hold:

$$\begin{array}{lcl} \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}} & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}, \lfloor \Sigma \rfloor_{\pm} \vdash \lfloor N \rfloor_{\pm}, \lfloor P^{\mathsf{at}} \rfloor_{\pm} \\ \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}} & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor P^{\mathsf{at}} \rfloor_{\pm} \\ \Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}} & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}, \lfloor N \rfloor_{\pm} \vdash \lfloor P^{\mathsf{at}} \rfloor_{\pm} \\ \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P] & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor P \rfloor_{\pm} \end{array}$$

Proof. Remember that we use the notation $\Pi :: \mathcal{J}$ to say that Π is a proof derivation for the judgment $\mathcal{J}$; for example, $\Pi :: \Gamma \vdash_{\mathsf{foc}} A$ means that Π is a derivation whose conclusion is the judgment $\Gamma \vdash_{\mathsf{foc}} A$.

The proof is by direct induction, by erasing the focusing information from the proof – the proof structure is unchanged. For example:

$$\left\lfloor \frac{\Pi_P :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P] \qquad \Pi_N :: \Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [P \to N] \vdash_{\mathsf{foc.l}} Q^{\mathsf{at}}} \right\rfloor_{\pm} \stackrel{\mathsf{def}}{=}$$

$$\frac{\lfloor \Pi_P \rfloor_{\pm} :: \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor P \rfloor_{\pm} \qquad \lfloor \Pi_N \rfloor_{\pm} :: \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}, \lfloor N \rfloor_{\pm} \vdash \lfloor Q^{\mathsf{at}} \rfloor_{\pm}}{\lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}, \lfloor P \to N \rfloor_{\pm} \vdash \lfloor Q^{\mathsf{at}} \rfloor_{\pm}}$$

Because $\lfloor P \to N \rfloor_{\pm}$ is equal (by definition) to $\lfloor P \rfloor_{\pm} \to \lfloor N \rfloor_{\pm}$, this is the (valid) left-introduction rule for implication in the non-focused sequent calculus.

The rules that are solely concerned with the focusing structure are erased in the process. For example:

$$\left\lfloor \frac{\Pi :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid P^{\mathsf{at}}} \right\rfloor_{\pm} \stackrel{\mathsf{def}}{=} \lfloor \Pi \rfloor_{\pm} :: \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor P^{\mathsf{at}} \rfloor_{\pm}$$

$$\left\lfloor \frac{\Pi :: \Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid \emptyset}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^{-}]} \right\rfloor_{\pm} \stackrel{\mathsf{def}}{=} \lfloor \Pi \rfloor_{\pm} :: \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor N \rfloor_{\pm}$$

□

In particular, this means that our structural focused system is *sound* with respect to propositional intuitionistic logic: the (defocused erasing of) formulas it proves are all provable in our reference proof system.

Furthermore, one can easily check that proofs obtained in this way remain valid focused proof – they are in the restricted subsystem defined by the focusing restrictions.

7.4.2. The minimal-shift translation

Conversely, valid focused proofs on non-polarized formulas correspond to valid focused proofs for polarized formulas obtained through the expected "minimal shift" translation described in Figure 7.10 (Minimal shift translation), that inserts shifts exactly at the boundary between positive and negative connectives. We also define this translation on formulas that are "positive or atomic" or "negative or atomic", preserving the atomic structure.

Theorem 7.4.2.
Proofs of a formula in the focused system without shifts of Figure 7.1 (Focused sequent calculus (single-succedent, without shifts)), when equipped with the incremental move rules of Section 7.2.4 (Batch or incremental validation of non-polarized contexts), are in one-to-one correspondence with the proofs of minimally-shifted formulas in the focused system with shifts of Figure 7.7 (Focused sequent calculus for polarized propositional intuitionistic logic).

$$\begin{array}{lcl}
\Gamma; \Delta \vdash_{\mathsf{inv}} A \mid B & \longleftrightarrow & (\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{min}}; (\!|\Delta|\!)^{+}_{\mathsf{min}} \vdash_{\mathsf{inv}} (\!|A|\!)^{-}_{\mathsf{min}} \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{min}} \\
\Gamma \vdash_{\mathsf{foc}} A & \longleftrightarrow & (\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{min}} \vdash_{\mathsf{foc}} (\!|A|\!)^{+\mathsf{at}}_{\mathsf{min}} \\
\Gamma, [A] \vdash_{\mathsf{foc.l}} B & \longleftrightarrow & (\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{min}}, [(\!|A|\!)^{-}_{\mathsf{min}}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{min}} \\
\Gamma \vdash_{\mathsf{foc.r}} [A] & \longleftrightarrow & (\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{min}} \vdash_{\mathsf{foc.r}} [(\!|A|\!)^{+}_{\mathsf{min}}]
\end{array}$$

Proof. The proof is again by direct induction on the derivations, exactly preserving the structure. The only notable cases are those where a shift appears in the rules with explicit shifts, which corresponds to a polarity test in rules without shifts.

Figure 7.10.: Minimal shift translation

$$
\begin{array}{lcl}
(\!|A \to B|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & \left\langle (\!|A|\!)^{+\mathsf{at}}_{\mathsf{min}} \to (\!|B|\!)^{-\mathsf{at}}_{\mathsf{min}} \right\rangle^{+} \\
(\!|A_1 \times A_2|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & \left\langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{min}} \times (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{min}} \right\rangle^{+} \\
(\!|1|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & \langle 1 \rangle^{+} \\
(\!|A_1 + A_2|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{min}} + (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{min}} \\
(\!|0|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & 0 \\
(\!|X|\!)^{+\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & X
\end{array}
$$

$$
\begin{array}{lcl}
(\!|A \to B|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & (\!|A|\!)^{+\mathsf{at}}_{\mathsf{min}} \to (\!|B|\!)^{-\mathsf{at}}_{\mathsf{min}} \\
(\!|A_1 \times A_2|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{min}} \times (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{min}} \\
(\!|1|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & 1 \\
(\!|A_1 + A_2|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & \left\langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{min}} + (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{min}} \right\rangle^{-} \\
(\!|0|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & \langle \rangle^{-} \\
(\!|X|\!)^{-\mathsf{at}}_{\mathsf{min}} & \stackrel{\mathsf{def}}{=} & X
\end{array}
$$

Consider for example:

$$
\frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} A \mid \qquad A \text{ negative}}{\Gamma \vdash_{\mathsf{foc.r}} [A]}
\qquad \longleftrightarrow \qquad
\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^{+}]}
$$

A formula A is negative if and only if there is a polarized negative formula N such that $(\!|A|\!)^{-}_{\mathsf{min}} = N$ and $A = \lfloor N \rfloor_{\pm}$. But then we have $(\!|A|\!)^{+}_{\mathsf{min}} = \langle N \rangle^{+}$. Similarly, a context Γ has only negative or atomic formulas if and only if there is a Γ^{at} such that $(\!|\Gamma|\!)^{+\mathsf{at}}_{\mathsf{min}} = \Gamma^{\mathsf{at}}$ and $\Gamma = \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}$. This gives the two-way correspondence:

$$
\frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} A \mid \qquad A \text{ negative}}{\Gamma \vdash_{\mathsf{foc.r}} [A]}
\quad \longrightarrow \quad
\frac{(\!|\Gamma|\!)^{+\mathsf{at}}_{\mathsf{min}}; \emptyset \vdash_{\mathsf{inv}} (\!|A|\!)^{-}_{\mathsf{min}} \mid}{(\!|\Gamma|\!)^{+\mathsf{at}}_{\mathsf{min}} \vdash_{\mathsf{foc.r}} [(\!|A|\!)^{+}_{\mathsf{min}}]}
\quad = \quad
\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^{+}]}
$$

$$
\frac{\Gamma; \emptyset \vdash_{\mathsf{inv}} A \mid \qquad A \text{ negative}}{\Gamma \vdash_{\mathsf{foc.r}} [A]}
\quad = \quad
\frac{\left\lfloor \Gamma^{\mathsf{at}} \right\rfloor_{\pm}; \emptyset \vdash_{\mathsf{inv}} \lfloor N \rfloor_{\pm} \mid}{\left\lfloor \Gamma^{\mathsf{at}} \right\rfloor_{\pm} \vdash_{\mathsf{foc.r}} [\left\lfloor \langle N \rangle^{+} \right\rfloor_{\pm}]}
\quad \longleftarrow
$$

$$
\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [\langle N \rangle^{+}]}
$$

"Negative or atomic" rules get translated in two different rules with explicit shifts. For example:

$$
\frac{A \text{ negative} \qquad \Gamma, A; \Delta \vdash_{\mathsf{inv}} B \mid C}{\Gamma; A, \Delta \vdash_{\mathsf{inv}} B \mid C}
\qquad \longleftrightarrow \qquad
\frac{\Gamma^{\mathsf{at}}, N; \Sigma \vdash_{\mathsf{inv}} M \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \langle N \rangle^{+}, \Sigma \vdash_{\mathsf{inv}} M \mid Q^{\mathsf{at}}}
$$

$$
\frac{A \text{ atomic} \qquad \Gamma, A; \Delta \vdash_{\mathsf{inv}} B \mid C}{\Gamma; A, \Delta \vdash_{\mathsf{inv}} B \mid C}
\qquad \longleftrightarrow \qquad
\frac{\Gamma^{\mathsf{at}}, X^{+}; \Sigma \vdash_{\mathsf{inv}} M \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; X^{+}, \Sigma \vdash_{\mathsf{inv}} M \mid Q^{\mathsf{at}}}
$$

Note that the same correspond holds between the versions of the two systems that use batch context validation instead of incremental move rules.

$$
\frac{\begin{array}{c}\Delta \text{ negative or atomic} \\ \Gamma, \Delta \vdash_{\mathsf{foc}} (B, C) \\ B \text{ positive or atomic or empty}\end{array}}{\Gamma; \Delta \vdash_{\mathsf{inv}} B \mid C}
\qquad \longleftrightarrow \qquad
\frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \left\langle P^{\mathsf{at}} \right\rangle^{-\mathsf{at}}}
$$

If Δ is positive or atomic, then its minimal shift is of the first $\langle \Gamma^{\mathsf{at}\prime} \rangle^{+\mathsf{at}}$. If B is positive or atomic, its minimal shift is of the form $\langle P^{\mathsf{at}} \rangle^{-\mathsf{at}}$, otherwise is empty and C is strictly positive. □

7.4.3. The double-shift translation

We have observed in Section 7.3.1 (Explicit shifts) that making transitions between polarities explicit as shifts allowed to consider double-shifted formula of the form $\langle \langle P \rangle^{-} \rangle^{+}$ and $\langle \langle N \rangle^{+} \rangle^{-}$, which give more flexibility to the proof by allowing to stop a focusing phase early.

This suggests a double-formula translation of non-polarized formulas into polarized formulas, that adds a double shift to each subformula of the translated formula. We define this translation in Figure 7.11 (Double shift translation); it follows the structure of Figure 7.10 (Minimal shift translation), but inserts more shifts: sub-formulas of the same polarity as the outer polarity are double-shifted, sub-formulas of the opposite polarity are simply shifted.[1]

Figure 7.11.: Double shift translation

$$\langle\!\langle P \rangle\!\rangle^{+} \stackrel{\mathsf{def}}{=} \langle \langle P \rangle^{-} \rangle^{+} \qquad \langle\!\langle N \rangle\!\rangle^{-} \stackrel{\mathsf{def}}{=} \langle \langle N \rangle^{+} \rangle^{-}$$

$$\begin{array}{lcl}
(\!| A \to B |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \left\langle \langle (\!| A |\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \to \langle (\!| B |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \right\rangle^{+} \\
(\!| A_1 \times A_2 |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \left\langle \langle (\!| A_1 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \times \langle (\!| A_2 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \right\rangle^{+} \\
(\!| 1 |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \langle 1 \rangle^{+} \\
(\!| A_1 + A_2 |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \langle\!\langle (\!| A_1 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+} + \langle\!\langle (\!| A_2 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+} \\
(\!| 0 |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & 0 \\
(\!| X |\!)^{+\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & X
\end{array}$$

$$\begin{array}{lcl}
(\!| A \to B |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \langle\!\langle (\!| A |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+} \to \langle\!\langle (\!| B |\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-} \\
(\!| A_1 \times A_2 |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \langle\!\langle (\!| A_1 |\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-} \times \langle\!\langle (\!| A_2 |\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-} \\
(\!| 1 |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & 1 \\
(\!| A_1 + A_2 |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \left\langle \langle (\!| A_1 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{+} + \langle (\!| A_2 |\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{+} \right\rangle^{-} \\
(\!| 0 |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & \langle \rangle^{-} \\
(\!| X |\!)^{-\mathsf{at}}_{\mathsf{double}} & \stackrel{\mathsf{def}}{=} & X
\end{array}$$

This translation has the key property that each argument of a logical connective starts with a shift: if this connective is chosen as focus, the focused phase will stop immediately after the first introduction rule.

Note that it is not unique in this regard, in particular this property is preserved by adding more double-shifts on any subformula. But the one-to-one correspondence result below depends on the fact that we did not add more shifts than necessary.

Theorem 7.4.3.
Proofs of a formula in the non-focused sequent calculus with atomic axioms are in one-to-one correspondence with the focused proofs of double-shifted formulas, modulo the ordering

[1] This is reminiscent of the realizability models that use single- and bi-orthogonal constructions to turn sets of "value witnesses" into general sets of realizability witnesses. See for example Munch-Maccagnoni [2009], Brunel [2014]. In terms of game semantics, the double-translation would ensure that we let our opponent play after each of our moves.

of incremental move rules.

$$\Gamma \vdash A \quad \longleftrightarrow \quad (\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}}$$

Proof. A complete derivation of $\Gamma \vdash A$ is characterized by the choice of a formula in Γ, A, an introduction or axiom rule applied to this formula, and complete derivations for its premises.

A complete derivation of $(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}}$ is characterized by a choice of focus, that is a formula in $(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}}$, a complete focused phase on this focus, the invertible phase following it, and complete derivations for the focused premises of this invertible phase.

To establish the one-to-one correspondence between non-focused derivations and focused double-shifted derivations, we show a one-to-one correspondence between each inference rule in the unfocused derivation, and each focused phase followed by its invertible phase in the focused double-shifted derivation.

For example, for the right-introduction rule for implication we have

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \quad \longleftrightarrow \quad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \langle (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \to \langle (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\langle \langle (\!|A|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \to \langle (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \rangle^{+}]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [(\!|A \to B|\!)^{+\mathsf{at}}_{\mathsf{double}}]}$$

The double bar in the right derivation indicates that this reasoning step is not the application of an inference rule, but merely the unfolding of a definition: the root and leaf judgments of the double bar are equal.

In the left-to-right direction, we have an easy provability result; in a sense we are showing that the unfocused rule is admissible in the focused double-shifted system. In the right-to-left direction, it is important to point out that the derivation is *uniquely* determined once the focus has been selected: we have not made choices – except on the ordering of incremental move rules, over which we quotiented explicitly. Any derivation focusing on a formula of the form $(\!|A \to B|\!)^{+\mathsf{at}}_{\mathsf{double}}$ will have this root prefix. This, plus the fact that there is no other case of focused judgment being mapped to the sequent $\Gamma \vdash A \to B$, establishes the one-to-one nature of the bidirectional correspondence.

For the right-introduction rule for conjunction we have:

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2} \quad \longleftrightarrow$$

$$\cfrac{\cfrac{\cfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset} \qquad \cfrac{\cfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \times \langle (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\langle \langle (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \times \langle (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \rangle^{+}]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [(\!|A_1 \times A_2|\!)^{+\mathsf{at}}_{\mathsf{double}}]}$$

For the right-introduction rule for disjunction we have:

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} \quad \longleftrightarrow \quad \frac{\dfrac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A_i|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|A_i|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A_i|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\left\langle \langle (\!|A_i|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \right\rangle^{+}]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\langle\!\langle (\!|A_1|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+} + \langle\!\langle (\!|A_2|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+}]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [(\!|A_1 + A_2|\!)^{+\mathsf{at}}_{\mathsf{double}}]}$$

For the right-introduction rule for truth we have:

$$\frac{}{\Gamma \vdash 1} \quad \longleftrightarrow \quad \frac{\dfrac{\dfrac{}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} 1 \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\langle 1 \rangle^{+}]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [(\!|1|\!)^{+\mathsf{at}}_{\mathsf{double}}]}$$

For the left-introduction rule for implication we have:

$$\frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \quad \longleftrightarrow$$

$$\frac{\dfrac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \langle (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \mid \emptyset}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [\left\langle \langle (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle^{-} \right\rangle^{+}]} \qquad \frac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|B|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|B|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|B|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\left\langle \langle (\!|B|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \right\rangle^{-}] \vdash_{\mathsf{foc.l}} (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\langle\!\langle (\!|A|\!)^{+\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{+} \to \langle\!\langle (\!|B|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-}] \vdash_{\mathsf{foc.l}} (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [(\!|A \to B|\!)^{-\mathsf{at}}_{\mathsf{double}}] \vdash_{\mathsf{foc.l}} (\!|C|\!)^{+\mathsf{at}}_{\mathsf{double}}}$$

For the left-introduction rule for conjunction we have:

$$\frac{\Gamma, A_i \vdash B}{\Gamma, A_1 \times A_2 \vdash B} \quad \longleftrightarrow \quad \frac{\dfrac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_i|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_i|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A_i|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\left\langle \langle (\!|A_i|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \right\rangle^{-}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\langle\!\langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-} \times \langle\!\langle (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle\!\rangle^{-}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [(\!|A_1 \times A_2|\!)^{-\mathsf{at}}_{\mathsf{double}}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}$$

For the left-introduction rule for disjunction we have:

$$\frac{\Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma, A_1 + A_2 \vdash B} \quad \longleftrightarrow$$

$$\frac{\dfrac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}} \qquad \frac{\dfrac{\dfrac{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}}; \emptyset \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; \langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} + \langle (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\left\langle \langle (\!|A_1|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} + \langle (\!|A_2|\!)^{-\mathsf{at}}_{\mathsf{double}} \rangle^{+} \right\rangle^{-}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [(\!|A_1 + A_2|\!)^{-\mathsf{at}}_{\mathsf{double}}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}$$

For the left-introduction rule for falsity we have:

$$\frac{}{\Gamma, 0 \vdash B} \quad \longleftrightarrow \quad \cfrac{\cfrac{\cfrac{}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}; 0 \vdash_{\mathsf{inv}} \emptyset \mid (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [\langle 0 \rangle^-] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [(\!|0|\!)^{-\mathsf{at}}_{\mathsf{double}}] \vdash_{\mathsf{foc.l}} (\!|B|\!)^{+\mathsf{at}}_{\mathsf{double}}}$$

Finally, for the atomic axiom rules we have:

$$\frac{}{\Gamma, X^- \vdash X^-} \quad \longleftrightarrow \quad \cfrac{\cfrac{}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [X^-] \vdash_{\mathsf{foc.l}} X^-}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, [(\!|X^-|\!)^{-\mathsf{at}}_{\mathsf{double}}] \vdash_{\mathsf{foc.l}} (\!|X^-|\!)^{+\mathsf{at}}_{\mathsf{double}}}$$

$$\frac{}{\Gamma, X^+ \vdash X^+} \quad \longleftrightarrow \quad \cfrac{\cfrac{}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, X^+ \vdash_{\mathsf{foc.r}} [X^+]}}{(\!|\Gamma|\!)^{-\mathsf{at}}_{\mathsf{double}}, (\!|X^+|\!)^{-\mathsf{at}}_{\mathsf{double}} \vdash_{\mathsf{foc.r}} [(\!|X^+|\!)^{+\mathsf{at}}_{\mathsf{double}}]}$$

□

An important consequence of this result is that we can reformulate any statement about the relation between focused and non-focused systems (in particular completeness for provability or computation) as a relation between a focused system and the double-shifted focused system. This justifies, after the fact, existing approaches to prove completeness of focusing that start from a fully-focused system and prove completeness "internally", by showing that the non-focused principles (identity expansion, cut-elimination, and non-focused introduction rules) are admissible in the focused system.

8. Semantics

In this chapter we give another justification for the choice of $(\approx_{\beta\eta})$ as our notion of program equivalence: it is sound with respect to observational equivalence. If two programs are $\beta\eta$-equivalent, then no context can distinguish them.

In fact, $\beta\eta$-equivalence is also complete for observational equivalence, but we need the tools of Chapter 11 (Saturation logic for canonicity) to prove it.

8.1. Strong normalization for $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$

By lack of time and space, we will omit the proof of strong normalization of our λ-calculus $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$. This is a classic result on which there is no doubt, but the details are actually interesting.

It is possible to prove strong normalization by embedding $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$ into a stronger known-normalizing calculus, in particular System F. There is a classic translation of sum types and the empty type in term of polymorphic function, which is thus a *negative* encoding:

$$A + B \quad \stackrel{\mathsf{def}}{=} \quad \forall\gamma.\, (A \to \gamma) \to (B \to \gamma) \to \gamma$$

$$0 \quad \stackrel{\mathsf{def}}{=} \quad \forall\gamma.\, \gamma$$

β-reductions are preserved by the translation, and strong normalization of the source language can thus be deduced from strong normalization of the target. Interestingly, the strong η-rule for the translation of sums or the empty type *cannot* be derived from $\beta\eta$-equivalence of functions in the translation domain. The internal equivalence between translated terms are weaker than those between source terms. To recover those equality principle, one need to use meta-theoretic results of *parametricity* instead.

Of course, using a very strong normalization result for System F to deduce strong normalization of a simply-typed calculus is somewhat disappointing. It is natural to look for a direct proof instead, using the reducibility method as is classic for simply-typed calculi. This amounts to defining a type-directed predicate "t normalizes at type A", by induction on A, that is stronger than just strong normalization at higher types, and allows an inductive proof of strong normalization to go through. For example, the classic definition of "t normalizes at type $A \to B$" is "for any u that normalizes at A, the application $t\ u$ normalizes at B" – notice that we use the definition of normalization at the subformulas A and B to define normalization at $A \to B$.

However, it is not immediate to apply this proof technique to $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$, which has positive types. The naive extension of this idea is to define "t normalizes at $A + B$" would be something like "for any C, the elimination $(\texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x \to u_1 \mid \sigma_2\ x \to u_2)$ normalizes at C assuming that the u_i normalize at C", but this is not well-founded – we use the definition of the normalization predicate for an arbitrary formula C that may be $A + B$ itself.

One interesting solution to this problem is the use of a different term syntax that has a more modular approach to reduction. In the unpublished note Munch-Maccagnoni [2012], Guillaume Munch-Maccagnoni uses a sequent-based term system that has just the right structural properties to let the definition of a normalization candidate go through. Interestingly, the reduction relation in this system does not correspond to β-reduction

alone, but to reduction modulo extrusion; we obtain a proof of strong normalization to a normal form modulo extrusion, which is an even stronger result.

A side-result of strong normalization that we will also assume is *confluence* of β-reduction: each term has a unique β-normal form.

8.2. Contextual equivalence for $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$

For a given programming language, the observational equivalence for this language relates two programs if they have the same observable behavior. Defining this relation requires to find, for each language, the appropriate notion of "observable behavior"

A typical notion of observation for programs at some type A is the following: a context $C\,[\Box]$ that expects a hole of type A and, when applied, returns a term of type $1+1$ in the empty context – the type of booleans, whose closed inhabitant are $\sigma_1\ ()$ and $\sigma_2\ ()$. If two terms t, u of type A are such that, for any such context C, $C\,[t]$ and $C\,[u]$ are always equal (to either $\sigma_1\ ()$ or $\sigma_2\ ()$), then they are equivalent in a very strong sense.

However, this definition is disappointing in presence of atomic types; for example, if $x \neq y$ are two distinct variables at some atomic type X, we would like to say that x and y are observably inequivalent. But in an empty environment, we do not have any operation available on this unknown type X, so there is no way to use an element of type X in a context. A closed context that takes a hole of type X and returns a boolean is always a constant context; the previous definition would thus suggest that x and y are observably equivalent, which is unsatisfying.

We thus propose the following strengthening of the definition. An atomic type X is "unknown", in the sense that we have not assumed anything about it; it could be replaced by any other type A, and the programs written using the type X would still type-check. We will say that programs of type X are equivalent if, for any possible replacement A of X, those two programs are still observably equivalent. For example, if we choose to replace X by the type of booleans $() + ()$, we can use the context $(\lambda x.\, \lambda y.\, \Box)\ (\sigma_1\ ())\ (\sigma_2\ ())$, which distinguishes the variables x and y.

Definition 8.2.1 ground type**.**
A *ground type* is a type that does not contain any atom.

Definition 8.2.2 model**.**
A *model* $\mathcal{M}$ is a mapping from atoms to ground types.

Notation 8.2.1 $\mathcal{M}(\_)$**.**
If x is some syntactic object containing types, we write $\mathcal{M}(x)$ for the result of replacing each atom in x by its image in the model $\mathcal{M}$. For example, $\mathcal{M}(A)$ is a ground type, $\mathcal{M}(\Gamma)$ is a context of ground types, and if $\Gamma \vdash t : A$ then we also have $\mathcal{M}(\Gamma) \vdash \mathcal{M}(t) : \mathcal{M}(A)$.

Definition 8.2.3 Contextual equivalence**.**
If $\Gamma \vdash t, u : A$ and for a given model $\mathcal{M}$, we say that t and u are *contextually equivalent in* $\mathcal{M}$, written $t \approx_{\mathtt{ctx}(\mathcal{M})} u$, if

$$\forall C, \qquad \emptyset \vdash C\,[\mathcal{M}(\Gamma) \vdash \Box : \mathcal{M}(A)] : () + () \qquad \Longrightarrow \qquad C\,[\mathcal{M}(t)] \approx_\beta C\,[\mathcal{M}(u)]$$

We say that t and u are *contextually equivalent*, written $t \approx_{\mathtt{ctx}} u$, if they are contextually equivalent in any model.

8.3. Semantic equivalence for $\mathsf{PIL}(\to, \times, 1, +, 0)$

Another natural way to give a meaning to program equivalence is to give a naive set-theoretic model of types and their inhabitants; equality of programs should then coincide with the mathematical equality of their interpretations.

In this section, we will provide such a definition, and show that it is equivalent to contextual equivalence. We will also show that $\beta\eta$-equivalence is sound with respect to

those new equivalence relation. This gives us a new way to prove that two terms are *not* $\beta\eta$-equivalent: it suffices, by contraposition, to provide a context that distinguishes them.

Definition 8.3.1 Semantics of types**.**
For a ground type A we define the set of *semantic values* of A, written $[\![A]\!]$, by induction on A as follows:

$$\begin{array}{lcl} [\![A \to B]\!] & \stackrel{\text{def}}{=} & \text{total functions from } [\![A]\!] \text{ to } [\![B]\!] \\ [\![A \times B]\!] & \stackrel{\text{def}}{=} & \{(v, w) \mid v \in [\![A]\!], w \in [\![B]\!]\} \\ [\![1]\!] & \stackrel{\text{def}}{=} & \{\star\} \\ [\![A + B]\!] & \stackrel{\text{def}}{=} & \{(1, v) \mid v \in [\![A]\!]\} \uplus \{(2, w) \mid w \in [\![B]\!]\} \\ [\![0]\!] & \stackrel{\text{def}}{=} & \emptyset \end{array}$$

Given a type A and a model $\mathcal{M}$, we define the set of *semantic values of A in $\mathcal{M}$*, written $[\![A]\!]_{\mathcal{M}}$, by

$$[\![A]\!]_{\mathcal{M}} \qquad \stackrel{\text{def}}{=} \qquad [\![\mathcal{M}(A)]\!]$$

Fact 8.3.1.
$[\![A]\!]_{\mathcal{M}}$ is always a finite type whose inhabitants can be (decidably) enumerated.

Definition 8.3.2 Semantics of environments**.**
For a typing environment Γ and a model $\mathcal{M}$, we define the set of *semantic valuations* of Γ in $\mathcal{M}$, written $[\![\Gamma]\!]_{\mathcal{M}}$, as the set of functions G, H from variables to semantic values such that, for any variable $x : P$ of Γ, $G(x)$ is a semantic value of A in $\mathcal{M}$.

$$[\![\Gamma]\!]_{\mathcal{M}} \qquad \stackrel{\text{def}}{=} \qquad \{G \;\mid\; \forall x : P \in \Gamma, \quad G(x) \in [\![A]\!]_{\mathcal{M}}\}$$

Definition 8.3.3 Semantics of typing judgments**.**
We write $[\![\Gamma \vdash A]\!]_{\mathcal{M}}$ for the set of function from semantic valuations of Γ to semantic values in A:

$$[\![\Gamma \vdash A]\!]_{\mathcal{M}} \qquad \stackrel{\text{def}}{=} \qquad [\![\Gamma]\!]_{\mathcal{M}} \quad \to \quad [\![A]\!]_{\mathcal{M}}$$

Definition 8.3.4 Semantics of term formers**.**

We define the following naive semantics for term formers:

$$\begin{array}{lcl}
\mathtt{var}_x & : & [\![\Gamma, x : A \vdash A]\!]_{\mathcal{M}} \\
\mathtt{var}_x(G) & \stackrel{\text{def}}{=} & G(x)
\end{array}$$

$$\begin{array}{lcl}
\mathtt{pair} & : & [\![\Gamma \vdash A_1]\!]_{\mathcal{M}} \times [\![\Gamma \vdash A_2]\!]_{\mathcal{M}} \to [\![\Gamma \vdash A_1 \times A_2]\!]_{\mathcal{M}} \\
\mathtt{pair}(f_1, f_2)(G) & \stackrel{\text{def}}{=} & (f_1(G), f_2(G))
\end{array}$$

$$\begin{array}{lcl}
\mathtt{proj}_i & : & [\![\Gamma \vdash A_1 \times A_2]\!]_{\mathcal{M}} \to [\![\Gamma \vdash A_i]\!]_{\mathcal{M}} \\
\mathtt{proj}_i(f)(G) & \stackrel{\text{def}}{=} & v_i \\
 & & \text{where } f(G) = (v_1, v_2)
\end{array}$$

$$\begin{array}{lcl}
\mathtt{lam} & : & [\![\Gamma, x : A \vdash B]\!]_{\mathcal{M}} \to [\![\Gamma \vdash A \to B]\!]_{\mathcal{M}} \\
\mathtt{lam}(f)(G) & \stackrel{\text{def}}{=} & (v \in [\![A]\!]_{\mathcal{M}}) \mapsto f(G, x \mapsto v)
\end{array}$$

$$\begin{array}{lcl}
\mathtt{app} & : & [\![\Gamma \vdash A \to B]\!]_{\mathcal{M}} \times [\![\Gamma \vdash A]\!]_{\mathcal{M}} \to [\![\Gamma \vdash B]\!]_{\mathcal{M}} \\
\mathtt{app}(f, g)(G) & \stackrel{\text{def}}{=} & f(G)(g(G))
\end{array}$$

$$\begin{array}{lcl}
\mathtt{unit} & : & [\![\Gamma \vdash 1]\!]_{\mathcal{M}} \\
\mathtt{unit}(G) & \stackrel{\text{def}}{=} & \star
\end{array}$$

$$\begin{array}{lcl}
\mathtt{inj}_i & : & [\![\Gamma \vdash A_i]\!]_{\mathcal{M}} \to [\![\Gamma \vdash A_1 + A_2]\!]_{\mathcal{M}} \\
\mathtt{inj}_i(f)(G) & \stackrel{\text{def}}{=} & (i, f(G))
\end{array}$$

$$\begin{array}{lcl}
\mathtt{match} & : & [\![\Gamma \vdash A_1 + A_2]\!]_{\mathcal{M}} \times [\![\Gamma, x : A_1 \vdash B]\!]_{\mathcal{M}} \times [\![\Gamma, x : A_2 \vdash B]\!]_{\mathcal{M}} \to [\![\Gamma \vdash B]\!]_{\mathcal{M}} \\
\mathtt{match}(f, g_1, g_2)(G) & \stackrel{\text{def}}{=} & g_i(G, x \mapsto v) \\
 & & \text{where } f(G) = (i, v)
\end{array}$$

$$\begin{array}{lcl}
\mathtt{absurd} & : & [\![\Gamma \vdash \emptyset]\!]_{\mathcal{M}} \to [\![\Gamma \vdash A]\!]_{\mathcal{M}} \\
\mathtt{absurd} & \stackrel{\text{def}}{=} & \emptyset
\end{array}$$

Definition 8.3.5 Semantics of terms and contexts**.**
By composing together the semantics of the term formers in the obvious way, we obtain semantics for terms t and one-hole contexts C:

$$[\![\Gamma \vdash t : A]\!]_{\mathcal{M}} \in [\![\Gamma \vdash A]\!]_{\mathcal{M}} \qquad [\![\Gamma \vdash C\left[\Gamma' \vdash \Box : A'\right] : A']\!]_{\mathcal{M}} \in [\![\Gamma, \Gamma' \vdash A']\!]_{\mathcal{M}} \to [\![\Gamma \vdash A]\!]_{\mathcal{M}}$$

For example we have $[\![(t_1, t_2)]\!]_{\mathcal{M}} = \mathtt{pair}([\![t_1]\!]_{\mathcal{M}}, [\![t_2]\!]_{\mathcal{M}})$ and $[\![(t, \Box)]\!]_{\mathcal{M}}(f) = \mathtt{pair}([\![t_1]\!]_{\mathcal{M}}, f)$. In particular, $[\![\Box]\!]_{\mathcal{M}}$ is the identity function.

The interest of this sophisticated definition – as opposed to a direct definition of the semantics of terms, and of the semantics of contexts is that it is obviously compositional. In particular we have the following compositional semantics of terms plugged into a context.

Fact 8.3.2 (Semantics of context plugging)**.**

$$[\![C\,[t]]\!]_{\mathcal{M}} \qquad = \qquad [\![C]\!]_{\mathcal{M}}([\![t]\!]_{\mathcal{M}})$$

Definition 8.3.6 Equality on semantics values**.**
If v, w are semantic values of the same semantic type, we write $v = w$ for the usual mathematical equality. For example, if v, v' are (total) functions from $[\![A]\!]_{\mathcal{M}}$ to $[\![B]\!]_{\mathcal{M}}$, they are equal if they are pointwise equal:

$$v = v' \in [\![A \to B]\!]_{\mathcal{M}} \qquad \Longleftrightarrow \qquad \forall w \in [\![A]\!]_{\mathcal{M}}, \quad v(w) = v'(w) \in [\![B]\!]_{\mathcal{M}}$$

Because the intepretation of types in each models are finite sets, equality of semantic values is always decidable – in particular, there is nothing fishy in assuming that either two semantic values are equal, or we have a tangible evidence of their difference. If we have $v \neq v' \in [\![A \to B]\!]_{\mathcal{M}}$, then we have a $w \in [\![A]\!]_{\mathcal{M}}$ such that $v(w) \neq v'(w)$.

Definition 8.3.7 Semantic equivalence**.**
For any terms $\Gamma \vdash t, t' : A$ and model $\mathcal{M}$, we say that t and t' are *semantically equivalent in* $\mathcal{M}$, written $t \approx_{\mathtt{sem}(\mathcal{M})} t'$, if their semantics are equal – pointwise, equal on any valuation.

$$t \approx_{\mathtt{sem}(\mathcal{M})} t' \qquad \stackrel{\mathsf{def}}{=} \qquad \forall G \in [\![\Gamma]\!]_{\mathcal{M}}, \quad [\![t]\!]_{\mathcal{M}}(G) = [\![u]\!]_{\mathcal{M}}(G) \in [\![A]\!]_{\mathcal{M}}$$

We say that t and u are *semantically equivalent*, written $t \approx_{\mathtt{sem}} u$, if they are semantically equivalent in any model $\mathcal{M}$.

8.4. $\beta\eta$ implies semantic implies contextual

Lemma 8.4.1 (Semantic equivalence is a congruence)**.**
For any terms t, t' and context Γ such that

$$\Gamma, \Gamma' \vdash t, t' : A \qquad \Gamma \vdash C\left[\Gamma' \vdash \Box : A'\right] : A \qquad t \approx_{\mathtt{sem}(\mathcal{M})} t'$$

we have $C[t] \approx_{\mathtt{sem}} C[t']$.

Proof. This is immediately proved by compositationality: for any model $\mathcal{M}$ we have

$$[\![C[t]]\!]_{\mathcal{M}} \quad = \quad [\![C]\!]_{\mathcal{M}}([\![t]\!]_{\mathcal{M}}) \quad = \quad [\![C]\!]_{\mathcal{M}}([\![t']\!]_{\mathcal{M}}) \quad = \quad [\![C[t']]\!]_{\mathcal{M}}$$

□

Lemma 8.4.2 (Semantic soundness of substitution)**.**

$$[\![t[u/x]]\!]_{\mathcal{M}}(G) \qquad = \qquad [\![t]\!]_{\mathcal{M}}(G, x \mapsto [\![u]\!]_{\mathcal{M}}(G))$$

Proof. By induction on t. □

Theorem 8.4.3 (Semantic soundness of $\beta\eta$-equivalence)**.**
If $t \approx_{\beta\eta} t'$ then $t \approx_{\mathtt{sem}} t'$.

Proof. We first remark that the semantic equivalence ($\approx_{\mathtt{sem}}$) is an equivalence relation. Reflexivity, transitivity and symmetry are immediate from the definition. It is also a congruence (it goes under any context), this is Fact 8.3.2 (Semantics of context plugging). To prove that ($\approx_{\beta\eta}$) is included in ($\approx_{\mathtt{sem}}$), it thus suffices to prove that each atomic β or η-step is included: the reflexivity, transitivity, symmetry, and congruence rules are included in any congruence.

β cases

$$\begin{aligned}
& [\![\lambda x.\, t\ u]\!]_{\mathcal{M}}(G) \\
= \ & \mathtt{app}(\mathtt{lam}([\![t]\!]_{\mathcal{M}}), [\![u]\!]_{\mathcal{M}})(G) \\
= \ & (v \mapsto [\![t]\!]_{\mathcal{M}}(G, x \mapsto v))\,([\![u]\!]_{\mathcal{M}}(G)) \\
= \ & [\![t]\!]_{\mathcal{M}}(G, x \mapsto [\![u]\!]_{\mathcal{M}}(G)) \\
= \ & \quad \text{(by Lemma 8.4.2 (Semantic soundness of substitution))} \\
& [\![t[u/x]]\!]_{\mathcal{M}}(G)
\end{aligned}$$

$$\begin{aligned}
& [\![\pi_i\ (t_1, t_2)]\!]_{\mathcal{M}}(G) \\
= \ & \mathtt{proj}_i(\mathtt{pair}([\![t_1]\!]_{\mathcal{M}}, [\![t_2]\!]_{\mathcal{M}}))(G) \\
= \ & [\![t_1]\!]_{\mathcal{M}}(G)
\end{aligned}$$

$$\begin{aligned}
& \left[\!\!\left[\mathtt{match}\ \sigma_i\ t\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x \to u_1 \\ \sigma_2\ x \to u_2 \end{array} \right]\!\!\right]_{\mathcal{M}}(G) \\
= \ & \mathtt{match}(\mathtt{inj}_i([\![t]\!]_{\mathcal{M}}), [\![u_1]\!]_{\mathcal{M}}, [\![u_2]\!]_{\mathcal{M}})(G) \\
= \ & [\![u_i]\!]_{\mathcal{M}}(G, x \mapsto [\![t]\!]_{\mathcal{M}}(G)) \\
= \ & \quad \text{(by Lemma 8.4.2 (Semantic soundness of substitution))} \\
& [\![u_i[t/x]]\!]_{\mathcal{M}}(G)
\end{aligned}$$

Negative η cases

$$
\begin{array}{lll}
 & [\![\lambda x.\, t\ x]\!]_{\mathcal{M}}(G) & \\
= & \mathtt{lam}(\mathtt{app}([\![t]\!]_{\mathcal{M}}, \mathtt{var}))(G) & \\
= & v \quad \mapsto \quad (\mathtt{app}([\![t]\!]_{\mathcal{M}}, \mathtt{var}_x))(G, x \mapsto v) & \\
= & v \quad \mapsto \quad [\![t]\!]_{\mathcal{M}}(G)(\mathtt{var}_x(G, x \mapsto v)) & \\
= & v \quad \mapsto \quad [\![t]\!]_{\mathcal{M}}(G)(v) & \\
= & [\![t]\!]_{\mathcal{M}} G &
\end{array}
$$

$$
\begin{array}{lll}
 & [\![(\pi_1\ t, \pi_2\ t)]\!]_{\mathcal{M}}(G) & \\
= & \mathtt{pair}(\mathtt{proj}_1[\![t]\!]_{\mathcal{M}}, \mathtt{proj}_2[\![t]\!]_{\mathcal{M}})(G) & \\
= & (v_1, v_2) & \\
 & \qquad \text{where } [\![t]\!]_{\mathcal{M}}(G) = (v_1, v_2) & \\
= & [\![t]\!]_{\mathcal{M}} G &
\end{array}
$$

$$
\begin{array}{lll}
 & [\![\Gamma \vdash t()]\!]_{\mathcal{M}}(G) & \\
= & \mathtt{unit}(G) & \\
= & \star & \\
= & [\![()]\!]_{\mathcal{M}}(G) &
\end{array}
$$

Positive η case: sum Suppose we have $\Gamma \vdash t : A_1 + A_2$ and $G \in [\![\Gamma]\!]_{\mathcal{M}}$ with $[\![t]\!]_{\mathcal{M}}(G) = (i, v)$.

Then for any $C\,[\emptyset \vdash \Box : A_1 + A_2]$ we have

$$
\begin{array}{lll}
 & \left[\!\!\left[\mathtt{match}\ t\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x \to C\,[\sigma_i\ x] \\ \sigma_2\ x \to C\,[\sigma_i\ x] \end{array} \right]\!\!\right]_{\mathcal{M}}(G) & \\
= & \mathtt{match}([\![t]\!]_{\mathcal{M}}, [\![C\,[\sigma_1\ x]]\!]_{\mathcal{M}}, [\![C\,[\sigma_2\ x]]\!]_{\mathcal{M}})(G) & \\
= & \qquad (\text{as } [\![t]\!]_{\mathcal{M}}(G) = (i, v)) & \\
 & [\![C\,[\sigma_i\ x]]\!]_{\mathcal{M}}(G, x \mapsto v) & \\
= & \qquad (\text{by Lemma 8.4.2 (Semantic soundness of substitution)}) & \\
 & [\![C\,[y]]\!]_{\mathcal{M}}(G, x \mapsto v, y \mapsto [\![\sigma_i\ x]\!]_{\mathcal{M}}(G, x \mapsto v)) & \\
= & [\![C\,[y]]\!]_{\mathcal{M}}(G, x \mapsto v, y \mapsto (i, v)) & \\
= & [\![C\,[y]]\!]_{\mathcal{M}}(G, x \mapsto v, y \mapsto [\![t]\!]_{\mathcal{M}}(G)) & \\
= & \qquad (\text{by Lemma 8.4.2 (Semantic soundness of substitution)}) & \\
 & [\![C\,[t]]\!]_{\mathcal{M}}(G) &
\end{array}
$$

Positive η case: empty Suppose we have $\Gamma \vdash t : 0$. We know that the set $[\![\Gamma \vdash 0]\!]_{\mathcal{M}}$ is inhabited by $[\![t]\!]_{\mathcal{M}}$; but this set is the set of functions from $[\![\Gamma]\!]_{\mathcal{M}}$ to the empty set $[\![0]\!]_{\mathcal{M}} = \emptyset$. It can only be inhabited if $[\![\Gamma]\!]_{\mathcal{M}}$ is also the empty set.

Then it is the case that

$$\forall G \in [\![\Gamma]\!]_{\mathcal{M}}, \qquad [\![u_1]\!]_{\mathcal{M}}(G) = [\![u_2]\!]_{\mathcal{M}} G$$

as no such G may exist. □

Theorem 8.4.4 (Semantic equivalence implies contextual equivalence)**.**
If t and t' are semantically equivalent, then they are contextually equivalent.

Proof. Suppose $t \approx_{\mathtt{sem}} t'$. For a given model $\mathcal{M}$ and boolean context C, we have to show that $C\,[t] \approx_\beta C\,[t']$.

As semantic equivalence is a congruence – Lemma 8.4.1 (Semantic equivalence is a congruence) – we have $C\,[t] \approx_{\mathtt{sem}} C\,[t']$. Now, suppose the closed β-normal form of $C\,[t]$ is $\sigma_i\ ()$ for some $i \in \{1, 2\}$, and the closed β-normal form of $C\,[t']$ is $\sigma_j\ ()$. Semantics is preserved by $\beta\eta$-equivalence – Theorem 8.4.3 (Semantic soundness of $\beta\eta$-equivalence) – so in particular by β-normalization. Thus we have $[\![\sigma_i\ ()]\!]_{\mathcal{M}} = [\![C\,[t]]\!]_{\mathcal{M}} = [\![C\,[t']]\!]_{\mathcal{M}} = [\![\sigma_j\ ()]\!]_{\mathcal{M}}$. It cannot be the case that $i \neq j$, as those semantics would then differ. We have proved that $C\,[t] \approx_\beta C\,[t']$. □

Corollary 8.4.5 ($\beta\eta$-equivalence implies contextual equivalence)**.**
If $t \approx_{\beta\eta} t'$*, then* $t \approx_{\mathtt{ctx}} t'$*.*

8.5. Contextual equivalence implies semantic equivalence

Our proof shall proceed by contraposition: given two terms with distinct semantics, we build a context that distinguishes them. A key ingredient to do this is a reification result: we need a way to build closed syntactic terms from semantic values. For example, two functions have distinct semantics if they differ on one input, which is a semantic value v; if we could obtain a syntactic term t corresponding to v, we could build the context $(\Box\ t)$ to distinguish the two functions.

Reification results of this kind are key to the proof technique known as "normalization by evaluation". It is fairly easy to prove in a purely negative type system, but the addition of sums makes it very difficult to prove in the general case – this is the purpose of the advanced techniques developed in Altenkirch, Dybjer, Hofmann, and Scott [2001], Balat, Di Cosmo, and Fiore [2004], and will also be made possible by the saturation technique presented in the later chapters of this thesis.

The reification of sum, product and the unit type are straightforward. The difficulties come from function types. Intuively, to reify a semantic function, it suffices to build a decision tree on its input type as a term. Such a decision tree is, again, straightforward to build if the input type is a sum, product or unit type; but what if we have a function? Building a decision tree on a function corresponds to tabulating this function, enumerating all its possible inputs and composing decision trees on each corresponding output. Again, enumerating all possible inputs of sum, product or unit type is straightforward, but what if the input is a function? Enumerating a function corresponds to building the composition of the enumeration of all its possible outputs on the leaves of a decision tree on its inputs.

Definifing the construction in this way is delicate – see Altenkirch and Uustalu [2004] for example. Giving a definition that respects the type structure and is thus correct by construction is even more challenging. This is done in Altenkirch, Dybjer, Hofmann, and Scott [2001] using more advanced semantic structures, in Balat, Di Cosmo, and Fiore [2004] by using control operators (to control the interleaving of enumeration and decision), and in Ahmad, Licata, and Harper [2010] using focusing.

In the present case, however, we can make use of the absence of atomic types to give a very easy solution to this challenge: if all types are finitely inhabited, we can actually get rid of the function types by converting them, through repeated application of type isomorphisms, to positive datatypes.

Remark 8.5.1. A similar idea is used in Ilik [2015], with a different form of type normalization that aims to remove sum types rather than function types. In presence of atomic or infinite types, neither sum nor function types can be fully removed. In absence of atoms, function types can be fully removed, but sum types cannot – there is no type isomorphic to $1 + 1$ in $\mathsf{PIL}(\to, \times, 1)$. ∗

In Figure 8.1 (Fun-less data types) we define a function $\lfloor \_ \rfloor$ from arbitrary ground types to ground types without functions. Because it is a recursive function whose well-foundedness is not immediate, we split it in one function $\lfloor \_ \rfloor$ that recurses structurally on its argument, and one function $\lVert \_ \to \_ \rVert$ that expects a function type whose type arguments contain no arrows, and recurses structurally on its left hand side argument.

In Figure 8.2 (Isomorphisms for fun-less types) we define isomorphism from and to these fun-less types, for closed terms and for semantic values: if v has type A, then $\lfloor v \rfloor_A$ has type $\lfloor A \rfloor$, and conversely if v has type $\lfloor A \rfloor$ then $\lceil v \rceil_A$ has type A.

Lemma 8.5.1 (Isomorphism)**.**

Figure 8.1.: Fun-less data types

$$
\begin{array}{lcl}
\lfloor A \to B \rfloor & \stackrel{\text{def}}{=} & \llbracket \lfloor A \rfloor \to \lfloor B \rfloor \rrbracket \\
\lfloor A_1 \times A_2 \rfloor & \stackrel{\text{def}}{=} & \lfloor A_1 \rfloor \times \lfloor A_2 \rfloor \\
\lfloor 1 \rfloor & \stackrel{\text{def}}{=} & 1 \\
\lfloor A_1 + A_2 \rfloor & \stackrel{\text{def}}{=} & \lfloor A_1 \rfloor + \lfloor A_2 \rfloor \\
\lfloor 0 \rfloor & \stackrel{\text{def}}{=} & 0
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket (A_1 \times A_2) \to C \rrbracket & \stackrel{\text{def}}{=} & \llbracket A_1 \to \llbracket A_2 \to C \rrbracket \rrbracket \\
\llbracket 1 \to B \rrbracket & \stackrel{\text{def}}{=} & B \\
\llbracket (A_1 + A_2) \to B \rrbracket & \stackrel{\text{def}}{=} & \llbracket A_1 \to B \rrbracket \times \llbracket A_2 \to B \rrbracket \\
\llbracket 0 \to B \rrbracket & \stackrel{\text{def}}{=} & 1
\end{array}
$$

$$
\begin{array}{lcl}
\lceil \lfloor v \rfloor_A \rceil_A & = & v \\
\lceil \lfloor t \rfloor_A \rceil_A & \approx_{\beta\eta} & t
\end{array}
$$

Proof. By case analysis. □

Lemma 8.5.2 (Commutation of isomorphisms)**.**

$$
\begin{array}{lcl}
\lfloor [\![t]\!]_{\mathcal{M}} \rfloor_A & = & [\![\lfloor t \rfloor_A]\!]_{\mathcal{M}} \\
\lceil [\![t]\!]_{\mathcal{M}} \rceil_A & = & [\![\lceil t \rceil_A]\!]_{\mathcal{M}}
\end{array}
$$

Proof. By case analysis. □

Theorem 8.5.3 (Reification)**.**
For each value v in $[\![A]\!]_{\mathcal{M}}$ we can define a term $\mathtt{reify}_{\mathcal{M}}(v)$ in $\mathcal{M}(A)$ such that

$$[\![\mathtt{reify}_{\mathcal{M}}(v)]\!]_{\mathcal{M}} = v$$

Proof. We define $\mathtt{reify}_{\mathcal{M}}(v)$ on general ground types as

$$\lceil \mathtt{reify'}_{\mathcal{M}}(\lfloor v \rfloor_A) \rceil_A$$

where $\mathtt{reify'}_{\mathcal{M}}(v)$ is only defined on types without functions. The definition of $\mathtt{reify'}_{\mathcal{M}}(v)$ is given below:

$$
\begin{array}{lcl}
\mathtt{reify'}_{\mathcal{M}}((v_1, v_2)) & \stackrel{\text{def}}{=} & (\mathtt{reify'}_{\mathcal{M}}(v_i), \mathtt{reify'}_{\mathcal{M}}(v_i)) \\
\mathtt{reify'}_{\mathcal{M}}((i, v)) & \stackrel{\text{def}}{=} & \sigma_i \; \mathtt{reify'}_{\mathcal{M}}(v) \\
\mathtt{reify'}_{\mathcal{M}}(\star) & \stackrel{\text{def}}{=} & ()
\end{array}
$$

and we can immediately check that $[\![\mathtt{reify'}_{\mathcal{M}}(v)]\!]_{\mathcal{M}} = v$.

It then remains to check that

$$[\![\mathtt{reify}_{\mathcal{M}}(v)]\!]_{\mathcal{M}} = v$$

which is proved as follows, using Lemma 8.5.1 (Isomorphism) and Lemma 8.5.2 (Commutation of isomorphisms).

$$
\begin{array}{ll}
 & [\![\mathtt{reify}_{\mathcal{M}}(v)]\!]_{\mathcal{M}} \\
= & [\![\lceil \mathtt{reify'}_{\mathcal{M}}(\lfloor v \rfloor_A) \rceil_A]\!]_{\mathcal{M}} \\
= & \lceil [\![\mathtt{reify'}_{\mathcal{M}}(\lfloor v \rfloor_A)]\!]_{\mathcal{M}} \rceil_A \\
= & \lceil \lfloor v \rfloor_A \rceil_A \\
= & v
\end{array}
$$

□

Figure 8.2.: Isomorphisms for fun-less types

$$
\begin{array}{llll}
\lfloor (v_1, v_2) \rfloor & \stackrel{\text{def}}{=} (\lfloor v_1 \rfloor, \lfloor v_2 \rfloor) & \lfloor (t_1, t_2) \rfloor & \stackrel{\text{def}}{=} (\lfloor t_1 \rfloor, \lfloor t_2 \rfloor) \\
\lfloor (i, v) \rfloor & \stackrel{\text{def}}{=} (i, \lfloor v \rfloor) & \lfloor \sigma_i\ t \rfloor & \stackrel{\text{def}}{=} \sigma_i\ \lfloor t \rfloor \\
\lfloor \star \rfloor & \stackrel{\text{def}}{=} \star & \lfloor () \rfloor & \stackrel{\text{def}}{=} () \\
\lfloor v \in [\![A \to B]\!]_{\mathcal{M}} \rfloor & \stackrel{\text{def}}{=} \llparenthesis w \mapsto \lfloor v(\lfloor w \rfloor) \rfloor \rrparenthesis & \lfloor t : A \to B \rfloor & \stackrel{\text{def}}{=} \llparenthesis \lambda x.\, \lfloor t\ \lfloor x \rfloor \rfloor \rrparenthesis \\
\\
\lceil (v_1, v_2) \rceil & \stackrel{\text{def}}{=} (\lceil v_1 \rceil, \lceil v_2 \rceil) & \lceil (t_1, t_2) \rceil & \stackrel{\text{def}}{=} (\lceil t_1 \rceil, \lceil t_2 \rceil) \\
\lceil (i, v) \rceil & \stackrel{\text{def}}{=} (i, \lceil v \rceil) & \lceil \sigma_i\ t \rceil & \stackrel{\text{def}}{=} \sigma_i\ \lceil t \rceil \\
\lceil \star \rceil & \stackrel{\text{def}}{=} \star & \lceil () \rceil & \stackrel{\text{def}}{=} () \\
\lceil v \rceil_{\llparenthesis \lfloor A \rfloor \to \lfloor B \rfloor \rrparenthesis} & \stackrel{\text{def}}{=} \rceil\!\rceil w \mapsto \lceil v(\lceil w \rceil) \rceil\!\rceil & \lceil t \rceil_{\lfloor A \rfloor \to \lfloor B \rfloor} & \stackrel{\text{def}}{=} \rceil\!\rceil \lambda x.\, \lceil t\ \lceil x \rceil \rceil\!\rceil
\end{array}
$$

$$
\begin{array}{lll}
\llparenthesis t \rrparenthesis_{A_1 \times A_2 \to B} & \stackrel{\text{def}}{=} & \llparenthesis \lambda x_1.\, \llparenthesis \lambda x_2.\, t\ (x_1, x_2) \rrparenthesis_{A_2 \to B} \rrparenthesis_{A_1 \to \llparenthesis A_2 \to B \rrparenthesis} \\
\llparenthesis t \rrparenthesis_{1 \to B} & \stackrel{\text{def}}{=} & t\ () \\
\llparenthesis t \rrparenthesis_{A_1 + A_2 \to B} & \stackrel{\text{def}}{=} & (\llparenthesis \lambda x.\, t\ (\sigma_1\ t) \rrparenthesis_{A_1 \to B}, \llparenthesis \lambda x.\, t\ (\sigma_2\ t) \rrparenthesis_{A_2 \to B}) \\
\llparenthesis t \rrparenthesis_{0 \to B} & \stackrel{\text{def}}{=} & ()
\end{array}
$$

$$
\begin{array}{lll}
\rceil\!\rceil t \lceil\!\rceil_{A_1 \times A_2 \to B} & \stackrel{\text{def}}{=} & \lambda x.\, \rceil\!\rceil \rceil\!\rceil t \lceil\!\rceil_{A_1 \to \llparenthesis A_2 \to B \rrparenthesis} (\pi_1\ x) \rceil\!\rceil_{A_2 \to B} (\pi_2\ x) \\
\rceil\!\rceil t \lceil\!\rceil_{1 \to B} & \stackrel{\text{def}}{=} & \lambda x.\, t \\
\rceil\!\rceil t \lceil\!\rceil_{A_1 + A_2 \to B} & \stackrel{\text{def}}{=} & \lambda x.\, \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to \rceil\!\rceil \pi_1\ t \lceil\!\rceil_{A_1 \to B}\ x_1 \\ \sigma_2\ x_2 \to \rceil\!\rceil \pi_2\ t \lceil\!\rceil_{A_2 \to B}\ x_2 \end{array} \\
\rceil\!\rceil t \lceil\!\rceil_{0 \to B} & \stackrel{\text{def}}{=} & \lambda x.\, \texttt{absurd}(x)
\end{array}
$$

$$
\begin{array}{lll}
\llparenthesis v \rrparenthesis_{A_1 \times A_2 \to B} & \stackrel{\text{def}}{=} & \llparenthesis w_1 \mapsto \llparenthesis w_2 \mapsto v((w_1, w_2)) \rrparenthesis_{A_2 \to B} \rrparenthesis_{A_1 \to \llparenthesis A_2 \to B \rrparenthesis} \\
\llparenthesis v \rrparenthesis_{1 \to B} & \stackrel{\text{def}}{=} & v(\star) \\
\llparenthesis v \rrparenthesis_{A_1 + A_2 \to B} & \stackrel{\text{def}}{=} & (\llparenthesis w \mapsto v(((1, v))) \rrparenthesis_{A_1 \to B}, \llparenthesis w \mapsto v(((2, v))) \rrparenthesis_{A_2 \to B}) \\
\llparenthesis v \rrparenthesis_{0 \to B} & \stackrel{\text{def}}{=} & \star
\end{array}
$$

$$
\begin{array}{lll}
\rceil\!\rceil v \lceil\!\rceil_{A_1 \times A_2 \to B} & \stackrel{\text{def}}{=} & (w_1, w_2) \mapsto \rceil\!\rceil \rceil\!\rceil v \lceil\!\rceil_{A_1 \to \llparenthesis A_2 \to B \rrparenthesis} (w_1) \rceil\!\rceil_{A_2 \to B} (w_2) \\
\rceil\!\rceil v \lceil\!\rceil_{1 \to B} & \stackrel{\text{def}}{=} & \star \mapsto v \\
\rceil\!\rceil (v_1, v_2) \lceil\!\rceil_{A_1 + A_2 \to B} & \stackrel{\text{def}}{=} & (i, w) \mapsto \rceil\!\rceil v_i \lceil\!\rceil_{A_i \to B} (w) \\
\rceil\!\rceil v \lceil\!\rceil_{0 \to B} & \stackrel{\text{def}}{=} & \emptyset
\end{array}
$$

Corollary 8.5.4 (Reification of typings)**.**

$$[\![\Gamma \vdash A]\!]_{\mathcal{M}} \neq \emptyset \qquad \Longrightarrow \qquad \mathcal{M}(\Gamma) \vdash \mathcal{M}(A)$$

Proof. If $[\![\Gamma \vdash A]\!]_{\mathcal{M}}$ is inhabited, then so is the isomorphic $[\![\Gamma \to A]\!]_{\mathcal{M}}$, where $\Gamma \to A$ is understood as a function type abstracting over all types of Γ. Let $v \in [\![\Gamma \to A]\!]_{\mathcal{M}}$; by reification we then have $\texttt{reify}_{\mathcal{M}}(v) : \mathcal{M}(\Gamma \to A)$, and thus

$$\mathcal{M}(\Gamma) \vdash \texttt{reify}_{\mathcal{M}}(v)\ x_1\ \ldots\ x_n : \mathcal{M}(A)$$

where $x_1, \ldots, x_n$ are the variables of Γ. □

Lemma 8.5.5.
Reification is an inverse modulo $\beta\eta$-equivalence For any closed term of ground type $\emptyset \vdash t : A$ we have

$$\texttt{reify}_{\mathcal{M}}([\![t]\!]_{\mathcal{M}}) \approx_{\beta\eta} t$$

Proof. We first check that, for types A without function types, $\mathtt{reify}'_{\mathcal{M}}(\_)$ is the inverse of $[\![\_]\!]_{\mathcal{M}}$ modulo β:

$$\mathtt{reify}'_{\mathcal{M}}([\![t]\!]_{\mathcal{M}}) \approx_\beta t$$

Let t' be the β-normal form of t. By Theorem 8.4.3 (Semantic soundness of $\beta\eta$-equivalence), we have $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$, and of course $t \approx_{\beta\eta} t'$. It thus suffice to check that $\mathtt{reify}'_{\mathcal{M}}([\![t']\!]_{\mathcal{M}}) \approx_{\beta\eta} t'$ holds for all β-normal forms t'. This is easily done by inversion on the possible well-typed normal forms in an empty context: a closed normal-form of type $A_1 + A_2$ must be of the form $\sigma_i\ t$, a closed normal-form of type $A_1 \times A_2$ must be of the form (t_1, t_2), and a closed normal-form of type 1 must be $()$. The fact that A contains no function type ensures that we never have to go under a binder, and thus that the context remains empty – preserving our induction hypothesis.

It then remains to check that

$$\mathtt{reify}_{\mathcal{M}}([\![t]\!]_{\mathcal{M}}) \approx_{\beta\eta} t$$

holds in the general case of a type A with function types, which is proved as follows, using Lemma 8.5.2 (Commutation of isomorphisms) and Lemma 8.5.1 (Isomorphism):

$$\begin{array}{ll} & \mathtt{reify}_{\mathcal{M}}([\![t]\!]_{\mathcal{M}}) \\ = & \lceil \mathtt{reify}'_{\mathcal{M}}(\lfloor [\![t]\!]_{\mathcal{M}} \rfloor_A) \rceil_A \\ = & \lceil \mathtt{reify}'_{\mathcal{M}}([\![\lfloor t \rfloor_A]\!]_{\mathcal{M}}) \rceil_A \\ = & \lceil \mathtt{reify}'_{\mathcal{M}}([\![\lfloor t \rfloor_A]\!]_{\mathcal{M}}) \rceil_A \\ \approx_\beta & \lceil \lfloor t \rfloor_A \rceil_A \\ \approx_{\beta\eta} & t \end{array}$$

□

Theorem 8.5.6 (Contextual equivalence implies semantic equivalence)**.**
If t and t' are contextually equivalent, then they are semantically equivalent.

Proof. By contraposition, let us assume that for $\Gamma \vdash t, t' : A$ we have a model $\mathcal{M}$ and a semantic valuation $G \in [\![\Gamma]\!]_{\mathcal{M}}$ such that $[\![t]\!]_{\mathcal{M}}(G) \neq [\![t']\!]_{\mathcal{M}}(G)$, and build a context $\emptyset \vdash C\,[\mathcal{M}(\Gamma) \vdash \Box : \mathcal{M}(A)] : 1 + 1$ such that $C\,[t] \not\approx_\beta C\,[t']$.

We reason by induction on A, with a slightly stronger induction hypothesis. For a fixed model $\mathcal{M}$, we assume a pair t, t' of terms typed in the ground types of $\mathcal{M}$, $\mathcal{M}(\Gamma) \vdash t, t' : \mathcal{M}(A)$, such that $[\![t]\!]_{\mathcal{M}} \neq [\![t']\!]_{\mathcal{M}}$, and we build a context C in $\mathcal{M}$ such that $C\,[t] \not\approx_\beta C\,[t']$.

This stronger induction hypothesis let us use "non-standard terms" to build our contexts, terms that are well-typed in $\mathcal{M}(A)$ but not in A.

If A is 1, the assumption $[\![t]\!]_{\mathcal{M}}(G) \neq [\![t']\!]_{\mathcal{M}}(G)$ is absurd, as those two values live in the same one-element set $\{\star\}$.

If A is $B \to C$, we have $w \in [\![B]\!]_{\mathcal{M}}$ such that $[\![t]\!]_{\mathcal{M}}(G)(w) \neq [\![t']\!]_{\mathcal{M}}(G)(w)$, that is, $[\![t\ \mathtt{reify}_{\mathcal{M}}(w)]\!]_{\mathcal{M}}(G) \neq [\![t'\ \mathtt{reify}_{\mathcal{M}}(w)]\!]_{\mathcal{M}}(G)$. By induction hypothesis on B, we thus have a closed context C such that $C\,[t\ \mathtt{reify}_{\mathcal{M}}(w)] \not\approx_\beta C\,[t'\ \mathtt{reify}_{\mathcal{M}}(w)]$ – notice the use of the non-standard term $\mathtt{reify}_{\mathcal{M}}(w)$ to invoke our induction hypothesis here. We can thus conclude with the context $C\,[\Box\ \mathtt{reify}_{\mathcal{M}}(w)]$.

If A is $A_1 \times A_2$, we know that the semantic value of t is some pair (v_1, v_2), similarly the one of t' is some (v'_1, v'_2). Our inequality assumption gives us a $i \in \{1, 2\}$ such that $v_i \neq v'_i$, in other words, $[\![\pi_i\ t]\!]_{\mathcal{M}}(G) \neq [\![\pi_i\ t']\!]_{\mathcal{M}}(G)$. By induction hypothesis on A_i we have some C that distinguishes $\pi_i\ t$ from $\pi_i\ t'$, and we can conclude with the context $C\,[\pi_i\ \Box]$.

If A is $A_1 + A_2$, the value $[\![t]\!]_{\mathcal{M}}(G)$ is a pair (i, v), and the value $[\![t']\!]_{\mathcal{M}}(G)$ a pair (j, v'). Our inequivalence assumption tells us that either $i \neq j$ or $v \neq v' \in [\![A_i]\!]_{\mathcal{M}}$.

In the first case, we can use the context $C \overset{\text{def}}{=} (\mathtt{match}\ \Box\ \mathtt{with}\ \mid \sigma_1\ x \to \sigma_1\ () \mid \sigma_2\ x \to \sigma_2\ ())$. We have $[\![C\,[t]]\!]_{\mathcal{M}}(G) = (i, \star)$ and $[\![C\,[t']]\!]_{\mathcal{M}}(G) = (j, \star)$, so in particular $C\,[t] \not\approx_{\mathtt{sem}} C\,[t']$; by the contrapositive of Theorem 8.4.3 (Semantic soundness of $\beta\eta$-equivalence) we can then deduce that $C\,[t] \not\approx_{\beta\eta} C\,[t']$, so a fortiori $C\,[t] \not\approx_\beta C\,[t']$.

In the second case, let us assume without loss of generality that $i = j = 1$; we have that $[\![\texttt{reify}_{\mathcal{M}}(v)]\!]_{\mathcal{M}} = v \neq v' = [\![\texttt{reify}_{\mathcal{M}}(v')]\!]_{\mathcal{M}}$, so by induction hypothesis on A_1 we have a context $C_1\,[\mathcal{M}(\Gamma) \vdash \Box : \mathcal{M}(A_1)]$ such that $C_1\,[\texttt{reify}_{\mathcal{M}}(v)] \not\approx_\beta C_1\,[\texttt{reify}_{\mathcal{M}}(v')]$. Let us define the context $C \stackrel{\text{def}}{=} (\texttt{match}\ \Box\ \texttt{with}\ \mid \sigma_1\ x \to C_1\,[x] \mid \sigma_2\ y \to \sigma_k\ ())$, for $k \in \{1, 2\}$ arbitrary. We have

$$\begin{array}{rl} & [\![C\,[t]]\!]_{\mathcal{M}}(G) \\ = & \texttt{match}([\![t]\!]_{\mathcal{M}}, [\![C_1\,[x]]\!]_{\mathcal{M}}, [\![\sigma_k\ ()]\!]_{\mathcal{M}}) \\ = & [\![C_1\,[x]]\!]_{\mathcal{M}}(G, x \mapsto v) \\ = & [\![C_1\,[\texttt{reify}_{\mathcal{M}}(v)]]\!]_{\mathcal{M}}(G) \end{array}$$

and likewise $[\![C\,[t']]\!]_{\mathcal{M}}(G) = [\![C_1\,[\texttt{reify}_{\mathcal{M}}(v')]]\!]_{\mathcal{M}}(G)$, so the two interpretations are distinct, and thus $C\,[t] \not\approx_\beta C\,[t']$. □

Part II.

Focusing for program equivalence and unique inhabitation

9. Counting terms and proofs

9.1. Introduction

In Section 3.1.2 (The Curry-Howard isomorphism, technically) we presented a correspondence between natural-deduction proofs of propositional intuitionistic logic, usually written as (logic) derivations for judgments of the form $\Gamma \vdash A$, and well-typed terms in the simply-typed lambda-calculus, with (typing) derivations for the judgment $\Gamma \vdash t : A$. This correspondence is not one-to-one. In typing judgments $\Gamma \vdash t : A$, the context Γ is a mapping from free variables to their type. In logic derivations, the context Γ is a set of hypotheses; there is no notion of variable, and at most one hypothesis of each type in the set. This means, for example, that the following logic derivation

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}}{A \vdash A \to A}}{\emptyset \vdash A \to A \to A}$$

corresponds to two *distinct* programs, namely $\lambda x.\, \lambda y.\, x$ and $\lambda x.\, \lambda y.\, y$. We say that those programs have the same *shape*, in the sense that the erasure of their typing derivation gives the same logic derivation – and they are the only programs of this shape.

Despite, or because, not being one-to-one, this correspondence is very helpful to answer questions about type systems. For example, the question of whether, in a given typing environment Γ, the type A is inhabited, can be answered by looking instead for a valid logic derivation of $\lfloor \Gamma \rfloor \vdash A$, where $\lfloor \Gamma \rfloor$ denotes the erasure of the mapping Γ into a set of hypotheses. In Section 6.2 (Rudiments of proof search) we have proved that only a finite number of different types need to be considered to find a valid proof (this is the case for propositional logic because of the *subformula property*). As a consequence, there are finitely many set-of-hypothesis Δ, and the search space of sequents $\Delta \vdash B$ to consider during proof search is finite. This property is key to the termination of proof search algorithm for propositional logic – Theorem 6.2.8 (Propositional logic is decidable). Note that it would not work if we searched typing derivations $\Gamma \vdash t : A$ directly: even if there are finitely many types of interest, the set of mappings from variables to such types is infinite.

In the present thesis, we are interested in a different problem. Instead of knowing whether there *exists* a term t such that $\Gamma \vdash t : A$, we want to know whether this term is *unique* – modulo a given notion of program equivalence. Intuitively, this can be formulated as a search problem where search does not stop at the first candidate, but tries to find whether a second one (that is nonequivalent as a program) exists. In this setting, the technique of searching for logic derivations $\lfloor \Gamma \rfloor \vdash A$ instead is not enough, because a unique logic derivation may correspond to several distinct programs of this shape: summarizing typing environments as set-of-hypotheses loses information about (non)-unicity, it is not complete for unicity.

To better preserve this information, one could keep track of the number of times a hypothesis has been added to the context, representing contexts as *multisets* of hypotheses; given a logic derivation annotated with such counts in the context, we can precisely compute the number of programs of this shape. However, even for a finite number of types/formulas, the space of such multisets is infinite; this breaks termination arguments. A natural idea is then to *approximate* multisets by labeling hypotheses with 0 (not available

in the context), 1 (added exactly once), or $\bar{2}$ (available two times *or more*); this two-or-more approximation has three possible states, and there are thus finitely many contexts annotated in this way.

The question we answer in this chapter is the following: is the two-or-more approximation correct? By correct, we mean that if the *precise* number of times a given hypothesis is available varies, but remains in the same approximation class, then the total number of programs of this shape may vary, but will itself remain in the same approximation class. A possible counter-example would be a logic derivation $\Delta \vdash B$ such that, if a given hypothesis $A \in \Delta$ is present exactly twice in the context (or has two free variables of this type), there is one possible program of this shape, but having three copies of this hypothesis would lead to several distinct programs.

Is this approximation correct? We found it surprisingly difficult to have an intuition on this question (guessing what the answer should be), and discussions with colleagues indicate that there is no obvious guess – people have contradictory intuitions on this. We show (Corollary 9.3.6 (Two-or-more approximation)) that this approximation is in fact correct.

9.2. Terms, types and derivations

We will manipulate several different systems of inference rules and discuss the relations between them: the type system, the logic, and inference systems annotated with counts (precise and approximated). To work uniformly over those various judgments, we will re-define their context structure as a mapping from types to some set. A set of hypothesis is now seen as a mapping from types to booleans, a multiset is a mapping to natural number, and typing judgment is a mapping from types to sets of free variables (we inverse the usual association order).

In this chapter, we shall write $\mathbb{T}$ for the set of formulas or types of $\mathsf{PIL}(\to, \times, 1, +, 0)$ defined in Figure 1.1 (Formulas of the propositional intuitionistic logic). Besides the set of types $\mathbb{T}$, we will write $\mathbb{V}$ for the set of term variables $x, y, \ldots$, $\mathbb{B}$ for the set of booleans $\{1, 0\}$, $\mathbb{N}$ for the (non-negative) natural numbers, and $\bar{2}$ for the set $\{0, 1, \bar{2}\}$ used by the two-or-more approximation – note the bar on $\bar{2}$ to indicate the extra element $\bar{2}$ and avoid confusion with other notations for the booleans.

We write $E \to F$ for the set of functions from the set E to the set F, and $\mathsf{cardinal}(E)$ for the cardinal of the set E.

To make our discussion of *shapes* (of propositional judgments) precise and notationally convenient, we give a syntax for them in Figure 9.1 (Syntax of propositional shapes), instead of manipulating derivation trees directly. A shape is a variable-less proof-term; we will manipulate *explicitly typed* shapes, where variables have been replaced with their typing information.

Figure 9.1.: Syntax of propositional shapes

S, T	$:=$		typed shapes
	$\mid$	A, B, C	axioms
	$\mid$	$\lambda A.\, S$	λ-abstraction
	$\mid$	$S\ T$	application
	$\mid$	(S, T)	pair
	$\mid$	$\pi_i\ S$	projection
	$\mid$	$()$	unit
	$\mid$	$\sigma_i\ S$	sum injection
	$\mid$	$\texttt{match}\ S\ \texttt{with} \mid \begin{array}{l}\sigma_1\ A_1 \to T_1 \\ \sigma_2\ A_2 \to T_2\end{array}$	sum destruction
	$\mid$	$\texttt{absurd}(S)$	absurdity

Shapes correspond to logic derivations, that is, proof term without variables. Instead of a variable $x : A$, we just use the shape A. Similarly, the term $\lambda x.\, t$, where the bound variable x has type A, becomes the shape $\lambda A.\, S$, where S is the shape of t.

There is an immediate mapping from valid derivations of the usual logic judgment $\Gamma \vdash A$ into shapes, which suggests reformulating the judgment as $S :: \Gamma \vdash A$. Valid judgments are then in direct one-to-one mapping with their valid derivations – a principle all our different judgments will satisfy. A gramatically correct shape S may be invalid, that is, not correspond to any valid logic derivation $S :: \Gamma \vdash A$ – for example $\pi_1\ (\lambda A.\, B)$ is an invalid shape. We will only consider valid shapes, classified by the provability judgment $\Gamma \vdash A$, in the rest of this document.

We will manipulate the following judgments, each annotated with a propositional shape S:

- the provability judgment $S :: \Gamma \vdash A$, where the context Γ is in $\mathbb{T} \to \mathbb{B}$ – isomorphic to sets of types;
- the typing judgment $S :: E \vdash t : A$, where the context E is in $\mathbb{T} \to \mathcal{P}(\mathbb{V})$ – isomorphic to mappings from term variables to types;
- various counting judgments of the form $S :: \Phi \vdash_K A : a$ for a set K, where Φ is in $\mathbb{T} \to K$ – mapping from types to a multiplicity in K – and a, in K, represents the output count of the derivation.

The context annotations of all those judgments each have a (commutative) monoid structure $((+_M), 0_M)$ of a binary operation and its unit/neutral element: $((\vee), 0)$ for $\mathbb{B}$ and $((\cup), \emptyset)$ for $\mathcal{P}(\mathbb{V})$. Our counting sets K will even have the stronger algebraic structure of a *semiring*, we detail this in Section 9.3 (Counting terms in semirings). This is used to define common notations as follows.

The binary operation of the monoid can be lifted to whole context, and we will write Γ, Δ for the addition of contexts: $(\Gamma, \Delta)(A) = \Gamma(A) +_M \Delta(A)$. We will also routinely specify a context as a *partial* mapping from types to annotations, for example the singleton mapping $[A \mapsto a]$ (for some a in the codomain of the mapping); by this, we mean that the value for any other element of the domain is the neutral element 0_M. In particular, the notation Γ, A on sets of hypotheses corresponds to the addition $\Gamma, [A \mapsto 1]$ in $\mathbb{T} \to \mathbb{B}$, and the notation $\Gamma, x : A$ on mapping from variables to types corresponds to the addition $\Gamma, [A \mapsto \{x\}]$ in $\mathbb{T} \to \mathcal{P}(\mathbb{V})$.

Finally, for any function $f : E \to F$, we will write $\lfloor \_ \rfloor_f : \mathbb{T} \to E \to \mathbb{T} \to F$ the pointwise lifting of f on contexts: $\lfloor \Phi \rfloor_f(A) \stackrel{\mathsf{def}}{=} f(\Phi(A))$. In particular, $\lfloor \_ \rfloor_{\neq \emptyset}$ erases typing environments $\mathbb{T} \to \mathcal{P}(\mathbb{V})$ into logic contexts $\mathbb{T} \to \mathbb{B}$, $\lfloor \_ \rfloor_{\neq 0}$ erases multiplicity-annotated contexts $\mathbb{T} \to \mathbb{N}$ into logic context $\mathbb{T} \to \mathbb{B}$, and $\lfloor \_ \rfloor_{\mathsf{cardinal}()}$ erases typing environments $\mathbb{T} \to \mathcal{P}(\mathbb{V})$ into multiplicity-annotated contexts $\mathbb{T} \to \mathbb{N}$.

The logic and typing judgments are defined in Figure 9.2 (Shaped provability judgment) and Figure 9.3 (Shaped typing judgment). In logic derivations we will simply write A for the singleton mapping $[A \mapsto 1]$. In typing derivations, we write $x : A$ for the singleton mapping $[A \mapsto \{x\}]$. Similarly, the variable freshness condition $x \notin E$ means $(\forall A \in \mathbb{T}, x \notin E(A))$.

Note that while changing the logic judgment from $\Gamma \vdash A$ to $S :: \Gamma \vdash A$ has the clear notational benefit of making valid judgments equivalent to derivations, this argument does not apply to changing the typing judgment from $E \vdash t : A$ to $S :: E \vdash t : A$, as the valid judgments $E \vdash t : A$ are already in one-to-one correspondence with their derivations; S adds some extra redundancy and could be computed from the triple (E, t, A) (or directly from t if we had used *explicitly typed* λ-terms). The benefit of $S :: E \vdash t : A$ is to let us talk very simply of the logical shape of a program, without having to define an additional erasure function from typing derivation to logical derivations: the set of programs of shape

Figure 9.2.: Shaped provability judgment

$$\frac{\Gamma(A) = 1}{A :: \Gamma \vdash A}$$

$$\frac{S :: \Gamma, A \vdash B}{\lambda A.\, S :: \Gamma \vdash A \to B} \qquad \frac{S :: \Gamma \vdash A \to B \qquad T :: \Gamma \vdash A}{S\ T :: \Gamma \vdash B}$$

$$\frac{S :: \Gamma \vdash A \qquad T :: \Gamma \vdash B}{(S, T) :: \Gamma \vdash A \times B} \qquad \frac{S :: \Gamma \vdash A_1 \times A_2}{\pi_i\ S :: \Gamma \vdash A_i}$$

$$\frac{S :: \Gamma \vdash A_i}{\sigma_i\ S :: \Gamma \vdash A_1 + A_2} \qquad \frac{S :: \Gamma \vdash A + B \qquad T_1 :: \Gamma, A_1 \vdash C \qquad T_2 :: \Gamma, A_2 \vdash C}{\texttt{match}\ S\ \texttt{with} \mid \begin{array}{l}\sigma_1\ A_1 \to T_1 \\ \sigma_2\ A_2 \to T_2\end{array} :: \Gamma \vdash C}$$

$$\frac{}{() :: \Gamma \vdash 1} \qquad \frac{S :: \Gamma \vdash 0}{\texttt{absurd}(S) :: \Gamma \vdash A}$$

Figure 9.3.: Shaped typing judgment

$$\frac{x \in E(A)}{A :: E \vdash x : A}$$

$$\frac{x \notin E \qquad S :: E, x : A \vdash t : B}{\lambda A.\, S :: E \vdash \lambda x.\, t : A \to B} \qquad \frac{S :: E \vdash t : A \to B \qquad T :: E \vdash u : A}{S\ T :: E \vdash t\ u : B}$$

$$\frac{S :: E \vdash t : A \qquad T :: E \vdash u : B}{(S, T) :: E \vdash (t, u) : A \times B} \qquad \frac{S :: E \vdash t : A_1 \times A_2}{\pi_i\ S :: E \vdash \pi_i\ t : A_i}$$

$$\frac{S :: E \vdash t : A_i}{\sigma_i\ S :: E \vdash \sigma_i\ t : A_1 + A_2}$$

$$\frac{S :: E \vdash t : A + B \quad x \notin E, y \notin E \quad T_1 :: E, x_1 : A_1 \vdash u_1 : C \quad T_2 :: E, x_2 : A_2 \vdash u_2 : C}{\texttt{match}\ S\ \texttt{with} \mid \begin{array}{l}\sigma_1\ A_1 \to T_1 \\ \sigma_2\ A_2 \to T_2\end{array} :: E \vdash \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_1\end{array} : C}$$

$$\frac{}{() :: E \vdash () : 1} \qquad \frac{S :: E \vdash t : 0}{\texttt{absurd}(S) :: E \vdash \texttt{absurd}(t) : A}$$

S and type A in the environment E is simply defined as:

$$\{t \mid S :: E \vdash t : A\}$$

9.3. Counting terms in semirings

We are trying to connect two distinct ways of "counting" things about a logic derivation $S :: \Gamma \vdash A$. One is precise, it counts the number of distinct programs of shape S, and the other is the two-or-more approximation.

We generalize those two ways of counting as instances of a generic counting scheme that works for any *semiring* $(K, 0_K, 1_K, +_K, \times_K)$. A semiring is defined as a two-operation

structure where $(0_K, +_K)$ and $(1_K, \times_K)$ are monoids, $(+_K)$ commutes and distributes over $(\times_K)$ (which may or may not commute), 0_K is a zero/absorbing element for $(\times_K)$, but $(+_K)$ and $(\times_K)$ need not have inverses[1]

The usual semiring is $(\mathbb{N}, 0, 1, +, *)$, and it will give the precise counting scheme. The 2-or-more semiring, which we will call $\bar{2}$, will correspond to the approximated scheme:

- its support is $\bar{2} = \{0, 1, \bar{2}\}$; 0_K is 0, 1_K is 1
- we define the addition by $1 +_K 1 = \bar{2}$ and $\bar{2} +_K 1 = \bar{2} +_K \bar{2} = \bar{2}$.
- we define the (commutative) multiplication by $\bar{2} \times_K \bar{2} = \bar{2}$.

Definition 9.3.1 Semiring notations**.**
Addition and multiplication can be lifted pointwise from K to $\mathbb{T} \to K$: for any $A \in \mathbb{T}$ we define $(\Phi +_K \Psi)(A) \stackrel{\text{def}}{=} \Phi(A) +_K \Psi(A)$ and $(\Phi \times_K \Psi)(A) \stackrel{\text{def}}{=} \Phi(A) \times_K \Psi(A)$.

Finally, we define a morphism from the semiring $\mathbb{N}$ to the semiring $\bar{2}$. Recall that $\varphi : K \to K'$ is a semiring morphism if $\varphi(0_K) = 0_{K'}$, $\varphi(1_K) = 1_{K'}$, $\varphi(a +_K b) = \varphi(a) +_{K'} \varphi(b)$ and $\varphi(a \times_K b) = \varphi(a) \times_{K'} \varphi(b)$.

Definition 9.3.2 The 2-or-more morphism $\varphi_{\bar{2}}$**.**
We define $\varphi_{\bar{2}} : \mathbb{N} \to \bar{2}$ as follows:

$$\begin{cases} \varphi_{\bar{2}}(0) = 0 \\ \varphi_{\bar{2}}(1) = 1 \\ \varphi_{\bar{2}}(n) = \bar{2} \quad \text{if } n \geq 2 \end{cases}$$

$\varphi_{\bar{2}}$ is a semiring morphism.

Note that $(\mathbb{B}, 0, 1, \vee, \wedge)$ is also a semiring. For any semiring K, the function $(\_ \neq 0_K) : K \to \mathbb{B}$ (which we may also write $(\neq 0)$) is a semiring morphism.

9.3.1. Semiring-annotated derivations

Given a semiring K, we now define derivations $S :: \Phi \vdash_K A : a$ where Φ is a set of types labeled with counts in K (that is, an element of the product $\mathbb{T} \to K$ for some set Γ), and a is itself in K.

We construct those inference rules such that, when K is instantiated with the semiring of natural numbers $\mathbb{N}$, they really count the different programs of the same shape. For example, consider a logic derivation $S :: \Gamma \vdash B$ starting with a function elimination rule

$$\frac{S_1 :: \Gamma \vdash A \to B \qquad S_2 :: \Gamma \vdash A}{S_1\ S_2 :: \Gamma \vdash B}$$

A program of this shape is of the form $t\ u$, at type B; it can be obtained by pairing any possible program t (of shape S_1) at type $A \to B$ with any possible program u at type A (of shape S_2), so the number of possible applications is the product of the number of possible functions and possible arguments. Formally, we have that, for any typing environment E, writing $\mathsf{cardinal}(S)$ for the cardinal of the set S:

$$\{t_0 \mid S_1\ S_2 :: E \vdash B\} = \left\{ (t\ u) \mid \begin{array}{l} S_1 :: E \vdash t : A \to B, \\ S_2 :: E \vdash u : A \end{array} \right\} \qquad \mathsf{cardinal}(\{t_0 \mid S_1\ S_2 :: E \vdash$$

$$B\}) = \mathsf{cardinal}(\{t \mid S_1 :: E \vdash t : A \to B\}) \times \mathsf{cardinal}(\{u \mid S_2 :: E \vdash u : A\})$$

This suggests the following semiring-annotated inference rule:

$$\frac{S_1 :: \Phi \vdash_K A \to B : a_1 \qquad S_2 :: \Phi \vdash_K B : a_2}{S_1\ S_2 :: \Phi \vdash_K B : a_1 \times_K a_2}$$

[1]For a *ring* $(K, 0_K, 1_K, +_K, \times_K)$, $(+_K)$ must be invertible, so $\mathbb{Z}$ is a ring while $\mathbb{N}$ is only a semiring.

The other rules are constructed in the same way, and the full inference system is given in Figure 9.4 (Shaped counting judgment). We write $A : a$ for the singleton mapping $[A \mapsto a]$.

Figure 9.4.: Shaped counting judgment

$$\frac{}{A :: \Phi \vdash_K A : \Phi(A)}$$

$$\frac{S :: \Phi, A : 1 \vdash_K B : a}{\lambda A.\, S :: \Phi \vdash_K A \to B : a} \qquad \frac{S_1 :: \Phi \vdash_K A \to B : a_1 \qquad S_2 :: \Phi \vdash_K A : a_2}{S_1\ S_2 :: \Phi \vdash_K B : a_1 \times a_2}$$

$$\frac{S_1 :: \Phi \vdash_K A : a_1 \qquad S_2 :: \Phi \vdash_K B : a_2}{(S_1, S_2) :: \Phi \vdash_K A \times B : a_1 \times a_2} \qquad \frac{S :: \Phi \vdash_K A_1 \times A_2 : a}{\pi_i\ S :: \Phi \vdash_K A_i : a}$$

$$\frac{S :: \Phi \vdash_K A_i : a}{\sigma_i\ S :: \Phi \vdash_K A_1 + A_2 : a}$$

$$\frac{S :: \Phi \vdash_K A_1 + A_2 : a_1 \qquad T_1 :: \Phi, A_1 : 1 \vdash_K C : a_2 \qquad T_2 :: \Phi, A_2 : 1 \vdash_K C : a_3}{\texttt{match}\ S\ \texttt{with} \mid \sigma_1\ A_1 \to T_1 \mid \sigma_2\ A_2 \to T_2 \ :: \Phi \vdash_K C : a_1 \times a_2 \times a_3}$$

$$\frac{}{() :: \Phi \vdash_K 1 : 1} \qquad \frac{S :: \Phi \vdash_K 0 : a}{\texttt{absurd}(S) :: \Phi \vdash_K A : a}$$

The identity rule says that if we have a different program variables of type A in our context, then using the variable rule of our typing judgment we can form a different programs. In particular, if A is absent from the context Φ, we have $A :: \Phi \vdash A : 0$. In the function-introduction rule, the number of programs of the form $\lambda x.\, t : A \to B$ is the number of programs $t : B$ in a context enriched with one extra variable of type A. The most complex rule is the sum elimination rule: the number of case-eliminations $(\texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2) : C$ is the product of the number of possible scrutinees $t : A + B$ and cases $u_1 : C$ and $u_2 : C$, with u_1 and u_2 built from one extra formal variable of type A or B accordingly.

We now precisely formulate the fact that the system $\vdash_{\mathbb{N}}$ really counts the number of programs of a given shape. Recall that $\lfloor \_ \rfloor_{\mathsf{cardinal}()} : (\mathbb{T} \to \mathcal{P}(\mathbb{V})) \to (\mathbb{T} \to \mathbb{N})$ erases a typing environment into a multiplicity-annotated context.

Lemma 9.3.1 (Cardinality count)**.**
For any typing environment $E \in \mathbb{T} \to \mathcal{P}(\mathbb{V})$, shape S and type A, the following is derivable:

$$S :: \lfloor E \rfloor_{\mathsf{cardinal}()} \vdash_{\mathbb{N}} A : \mathsf{cardinal}(\{t \mid S :: E \vdash t : A\})$$

Proof. By induction on the shape S, using the following equalities (obtained by inversion of the shape-directed typing judgment):

$$\{t_0 \mid A :: E \vdash t_0 : A\} = \{x \in E(A)\}$$

$$\{t_0 \mid \lambda A.\, S :: E \vdash t_0 : A \to B\} = \{\lambda x.\, t \mid S :: E, x : A \vdash t : A\}$$

$$\{t_0 \mid S\ T :: E \vdash t_0 : B\} = \left\{ t\ u \mid \begin{array}{l} S :: E \vdash t : A \to B \\ T :: E \vdash u : A \end{array} \right\}$$

$$\{t_0 \mid (S, T) :: E \vdash t_0 : A\} = \left\{ (t, u) \mid \begin{array}{l} S :: E \vdash t : A \\ T :: E \vdash u : B \end{array} \right\}$$

$$\{t_0 \mid \pi_i\ S :: E \vdash t_0 : A\} = \{\pi_i\ t \mid S :: E \vdash t : A\}$$

$$\{t_0 \mid \sigma_i\ S :: E \vdash t_0 : A\} = \{\sigma_i\ t \mid S :: E \vdash t : A\}$$

$$\{t_0 \mid \texttt{match}\ S\ \texttt{with} \mid \begin{array}{l}\sigma_1\ A_1 \to T_1 \\ \sigma_2\ A_2 \to T_2\end{array} :: E \vdash t_0 : C\}$$

$$= \left\{ \texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2\end{array} \mid \begin{array}{l} S :: E \vdash t : A_1 + A_2 \\ T_1 :: E, x_1 : A_1 \vdash u_1 : C \\ T_2 :: E, x_2 : A_2 \vdash u_2 : C \end{array} \right\}$$

$$\{t \mid () :: E \vdash t : 1\} = \{()\}$$

$$\{t \mid \texttt{absurd}(S) :: E \vdash \texttt{absurd}(t) : A\} = \{t \mid S :: E \vdash t : 0\}$$

□

While the inference system $\vdash_{\mathbb{N}}$ corresponds to counting programs of a given shape (we formally claim and prove it below), other semirings indeed correspond to counting schemes of interest. The system $\vdash_{\bar{2}}$ corresponds to the "two-or-more" approximation, as can be exemplified by the following derivations:

$$\dfrac{\dfrac{\dfrac{}{(A) :: A : \bar{2} \vdash_{\bar{2}} A : \bar{2}}}{(\lambda A.\, A) :: A : 1 \vdash_{\bar{2}} A \to A : \bar{2}}}{(\lambda A.\, \lambda A.\, A) :: \emptyset \vdash_{\bar{2}} A \to A \to A : \bar{2}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{(A) :: A : \bar{2} \vdash_{\bar{2}} A : \bar{2}}}{(\lambda A.\, A) :: A : \bar{2} \vdash_{\bar{2}} A \to A : \bar{2}}}{(\lambda A.\, \lambda A.\, A) :: A : 1 \vdash_{\bar{2}} A \to A \to A : \bar{2}}}{(\lambda A.\, \lambda A.\, \lambda A.\, A) :: \emptyset \vdash_{\bar{2}} A \to A \to A \to A : \bar{2}}$$

When adding the hypothesis in the context in the context, its count goes from 0 to 1 – we have $\emptyset(A) = 0$ by definition. When adding it the second time, its count goes from 1 to $\bar{2}$. But on the third addition on the right, the count remains $\bar{2}$, as in the semiring $\bar{2}$ we have $\bar{2} + 1 = \bar{2}$.

The $\vdash_{\mathbb{B}}$ system intuitively corresponds to a system where the two possible counts are "zero" and "one-or-more", that is, it only counts inhabitation. There is a precise correspondence between this system and the logic derivation we formulated: derivations of the form $S :: \Gamma \vdash A : 1$ are in one-to-one correspondence with valid logic derivations $S :: \Gamma \vdash A$, and derivations $S :: \Gamma \vdash A : 0$ correspond to *invalid* logic derivations, where the shape S is valid but the context Γ lacks some hypothesis used in S. In particular, $\emptyset \vdash A : 0$ is always provable by immediate application of the variable rule.

Lemma 9.3.2 (Provability count)**.**
There is a one-to-one correspondence between logic derivations of $S :: \Gamma \vdash A$ *and* $\mathbb{B}$*-counting derivations of* $S :: \Gamma \vdash_{\mathbb{B}} A : 1$.

Proof. Immediate by induction on the shape S. □

9.3.2. Semiring morphisms determine correct approximations

The key reason why the two-or-more approximation is correct is that the mapping from $\mathbb{N}$ to $\bar{2}$ is a semiring morphism and, as such, preserves the annotation structure of counting derivations.

Theorem 9.3.3 (Morphism of derivations)**.**
If $\varphi : K \to K'$ *is a semiring morphism and* $S :: \Phi \vdash A : a$ *holds, then* $S :: \lfloor \Phi \rfloor_\varphi \vdash A : \varphi(a)$ *also holds.*

Proof. By induction on S.

$$A :: \Phi \vdash_K A : \Phi(A) \qquad \Rightarrow \qquad A :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A : \varphi(\Phi(A))$$

$$\dfrac{S :: \Phi, A : 1_K \vdash_K B : a}{\lambda A.\, S :: \Phi \vdash_K A \to B : a} \qquad \Rightarrow \qquad \dfrac{S :: \lfloor \Phi \rfloor_\varphi, A : 1'_K \vdash_{K'} B : \varphi(a)}{\lambda A.\, S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A \to B : \varphi(a)}$$

To use our induction hypothesis, we needed the fact that $\lfloor \Phi \rfloor_\varphi, A : 1'_K$ is equal to $\lfloor \Phi, A : 1_K \rfloor_\varphi$; this comes from the fact that φ is a semiring morphism: $\varphi(1_K) = \varphi(1'_K)$ and $\varphi(a +_K b) = \varphi(a) +'_K \varphi(b)$, thus $\lfloor \Phi, \Psi \rfloor_\varphi = \lfloor \Phi \rfloor_\varphi, \lfloor \Psi \rfloor_\varphi$.

$$\frac{S_1 :: \Phi \vdash_K A \to B : a_1 \qquad S_2 :: \Phi \vdash_K A : a_2}{S_1\ S_2 :: \Phi \vdash_K B : a_1 \times a_2}$$

$$\Rightarrow \qquad \frac{S_1 :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A \to B : \varphi(a_1) \qquad S_2 :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A : \varphi(a_2)}{S_1\ S_2 :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} B : \varphi(a_1) \times \varphi(a_2)}$$

To conclude we then use the fact that $\varphi(a_1) \times \varphi(a_2) = \varphi(a_1 \times a_2)$.

$$\frac{S_1 :: \Phi \vdash_K A : a_1 \qquad S_2 :: \Phi \vdash_K B : a_2}{(S_1, S_2) :: \Phi \vdash_K A \times B : a_1 \times a_2}$$

$$\Rightarrow \qquad \frac{S_1 :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A : \varphi(a_1) \qquad S_2 :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} B : \varphi(a_2)}{(S_1, S_2) :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A \times B : \varphi(a_1) \times \varphi(a_2)}$$

$$\frac{S :: \Phi \vdash_K A_1 \times A_2 : a}{\pi_i\ S :: \Phi \vdash_K A_i : a} \quad \Rightarrow \quad \frac{S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A_1 \times A_2 : \varphi(a)}{\pi_i\ S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A_i : \varphi(a)}$$

$$\frac{S :: \Phi \vdash_K A_i : a}{\sigma_i\ S :: \Phi \vdash_K A_1 + A_2 : a} \quad \Rightarrow \quad \frac{S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A_i : \varphi(a)}{\sigma_i\ S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A_1 + A_2 : \varphi(a)}$$

$$\frac{S :: \Phi \vdash_K A + B : a_1 \qquad T_1 :: \Phi, A_1 : 1_K \vdash_K C : a_2 \qquad T_2 :: \Phi, A_2 : 1_K \vdash_K C : a_3}{\texttt{match}\ S\ \texttt{with} \mid \sigma_1\ A_1 \to T_1 \mid \sigma_2\ A_2 \to T_2 \ :: \Phi \vdash_K C : a_1 \times a_2 \times a_3}$$

$$\Rightarrow$$

$$\frac{S :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A + B : \varphi(a_1) \quad T_1 :: \lfloor \Phi \rfloor_\varphi, A_1 : 1'_K \vdash_{K'} C : \varphi(a_2) \quad T_2 :: \lfloor \Phi \rfloor_\varphi, A_2 : 1'_K \vdash_{K'} C : \varphi(a_3)}{\texttt{match}\ S\ \texttt{with} \mid \sigma_1\ A_1 \to T_1 \mid \sigma_2\ A_2 \to T_2 \ :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} C : \varphi(a_1) \times \varphi(a_2) \times \varphi(a_3)}$$

$$\frac{}{() :: \Phi \vdash_K 1 : 1_K} \quad \Rightarrow \quad \frac{}{() :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} 1 : 1'_K}$$

$$\frac{S :: \Phi \vdash_K 0 : a}{\texttt{absurd}(S) :: \Phi \vdash_K A : a} \quad \Rightarrow \quad \frac{S :: \Phi \vdash_{K'} 0 : \varphi(a)}{\texttt{absurd}(S) :: \lfloor \Phi \rfloor_\varphi \vdash_{K'} A : \varphi(a)}$$

□

From there, it remains to point out that the right-hand side count is uniquely determined by the context multiplicity.

Lemma 9.3.4 (Determinism)**.**
If $S :: \Phi \vdash_K A : a$ *and* $S :: \Phi \vdash_K A : b$ *then* $a = b$.

Proof. Immediate by induction on derivations. Note that the fact that the judgments are indexed by the same shape S is essential here. □

Corollary 9.3.5 (Relation under morphism)**.**
If $\varphi : K \to K'$ *is a semiring morphism and* $\lfloor \Phi_1 \rfloor_\varphi = \lfloor \Phi_2 \rfloor_\varphi$*, then* $S :: \Phi_1 \vdash_K A : a_1$ *and* $S :: \Phi_2 \vdash_K A : a_2$ *imply* $\varphi(a_1) = \varphi(a_2)$

Proof. By Theorem 9.3.3 (Morphism of derivations), we have $S :: \lfloor \Phi_1 \rfloor_\varphi \vdash_{K'} A : \varphi(a_1)$ and $S :: \lfloor \Phi_2 \rfloor_\varphi \vdash_{K'} A : \varphi(a_2)$. If $\lfloor \Phi_1 \rfloor_\varphi = \lfloor \Phi_2 \rfloor_\varphi$ we can conclude by Lemma 9.3.4 (Determinism) that $\varphi(a_1) = \varphi(a_2)$. □

Corollary 9.3.6 (Two-or-more approximation)**.**
The 2-or-more approximation is correct to decide unicity of inhabitants of a given shape S*. If* $\lfloor E_1 \rfloor_{\varphi_{\bar{2}} \cdot \mathsf{cardinal}()} = \lfloor E_2 \rfloor_{\varphi_{\bar{2}} \cdot \mathsf{cardinal}()}$*, then*

$$\varphi_{\bar{2}}(\mathsf{cardinal}(\{t \mid S :: E_1 \vdash t : A\})) = \varphi_{\bar{2}}(\mathsf{cardinal}(\{t \mid S :: E_2 \vdash t : A\}))$$

Proof. By Lemma 9.3.1 (Cardinality count), counting the inhabitants corresponds to the system $\vdash_{\mathbb{N}}$, so we have

$$S :: \lfloor E_1 \rfloor_{\mathsf{cardinal}()} \vdash_{\mathbb{N}} A : \mathsf{cardinal}(\{t \mid S :: E_1 \vdash t : A\})$$

$$S :: \lfloor E_2 \rfloor_{\mathsf{cardinal}()} \vdash_{\mathbb{N}} A : \mathsf{cardinal}(\{t \mid S :: E_2 \vdash t : A\})$$

The result then directly comes from the previous corollary, given that $\varphi_{\bar{2}}$ is a semiring morphism. □

9.4. n-or-more logics

The result can be extended to any "n-or-more" approximation scheme given by the semiring $\bar{n}$ and semiring morphism $\varphi_{\bar{n}} : \mathbb{N} \to \bar{n}$ defined as follows (assuming $n \geqslant 1$):

$$\bar{n} \stackrel{\mathsf{def}}{=} \{0, 1, \ldots, n-1, n\} \qquad 0_{\bar{n}} \stackrel{\mathsf{def}}{=} 0 \qquad 1_{\bar{n}} \stackrel{\mathsf{def}}{=} 1$$

$$(a +_{\bar{n}} b) \stackrel{\mathsf{def}}{=} \min(a +_{\mathbb{N}} b, n) \qquad (a \times_{\bar{n}} b) \stackrel{\mathsf{def}}{=} \min(a \times_{\mathbb{N}} b, n)$$

To check that $\varphi_{\bar{n}}$ is indeed a morphism, one needs to remark that having either $a \geqslant n$ or $b \geqslant n$ implies $(a +_{\mathbb{N}} b) \geqslant n$ and, if a and b are non-null, $(a *_{\mathbb{N}} b) \geqslant n$.

10. Focused λ-calculus

Term syntaxes for focused sequent calculi appear relatively exotic to the user of strongly-typed functional languages that is familiar with λ-calculus.

In this chapter we build this presentation of "focused λ-terms", which will be useful for the results of the latter chapters, when they are formulated in natural deduction style. We find interesting that it is possible to describe their syntax in a fairly non-invasive way, that should be familiar to people used to λ-calculi. It is mostly a refinement of the distinction, on β-normal forms, between constructors and neutral terms.

10.1. Intuitionistic natural deduction, focused

There are two paths to focused natural deduction.

- We could start from the usual natural deduction, convert non-invertible rules into rules with an explicit focus, and check that the resulting system has the properties we expect of a focusing system.
- We could start from focused sequent calculus, and apply the simple transformation of writing all elimination rules "upside down" to get a subsystem of natural deduction that is equivalent, by construction, to the focused sequent calculus we started from.

The good news is that those two paths bring us to the same end point, that we are going to present now.

The right-introduction rules of natural deduction and sequent calculus coincide, so we should expect that, in a structural presentation of focused natural deduction, the same right-introduction rule can be reused. The only change concerns the left-introduction and elimination rules.

10.1.1. Invertibility of elimination rules

Consider for example:

$$\textsc{seq-impl-left}\ \frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \qquad \textsc{nd-impl-elim}\ \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\textsc{seq-disj-left}\ \frac{\Gamma, A_1 \vdash C \quad \Gamma, A_2 \vdash C}{\Gamma, A_1 + A_2 \vdash C} \qquad \textsc{nd-disj-elim}\ \frac{\Gamma \vdash A_1 + A_2 \qquad \begin{array}{c}\Gamma, A_1 \vdash C \\ \Gamma, A_2 \vdash C\end{array}}{\Gamma \vdash C}$$

The definition of invertibility that we used for introduction rules (the conclusion is invertible if and only if all premises are) is not suited for elimination rules.

For ND-IMPL-ELIM for example, the first question is the status of the formula A appearing in the premises but not in the conclusion – this situation arises from the fact that elimination rules do not have the subformula property. It makes little sense to wonder whether $\Gamma \vdash A \to B$ is provable for *all* formulas A: it is almost never the case that $(\forall A,\ \Gamma \vdash A \to B)$, consider the case where B is 0. Another choice would be the existential quantification: is it the case that $(\exists A,\ (\Gamma \vdash A \to B) \land (\Gamma \vdash A))$ holds if and only

if ($\Gamma \vdash B$) holds? Unfortunately, this existentially-quantified proposition is trivially true (take $A \stackrel{\mathsf{def}}{=} 1$).

We do not quite know how to give an interesting interpretation of invertibility for this elimination rule for implications. Furthermore, we would expect any reasonable definition to make implication-elimination non-invertible, and disjunction-elimination non-invertible, and this seems incompatible with using (a variant of) the definition opposing conclusions and premises.

Instead, we will rely on the rootward reading of elimination rules: the elimination of implications let us deduce $\Gamma \vdash B$ from $\Gamma \vdash A \to B$ – whenever $\Gamma \vdash A$ is provable. This suggests that we could reason about the invertibility of this rule by starting from the eliminated premise, rather than from the conclusion. We propose the following notion of invertibility for elimination rules.

Definition 10.1.1 Invertible elimination rule**.**
An elimination rule is invertible if, whenever its eliminated premise is provable, then its conclusion is provable if and only if it is provable using this elimination rule.

The definition captures the intuition that an invertible rule "can always be applied", but in a situation where we do not decide which rule to apply by looking at the conclusion judgment, but by looking at the eliminated premise. Let us highlight the eliminated premise in both elimination rules:

$$\frac{\boxed{\Gamma \vdash A \to B} \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad\qquad \frac{\boxed{\Gamma \vdash A_1 + A_2} \qquad \Gamma, A_1 \vdash C \qquad \Gamma, A_2 \vdash C}{\Gamma \vdash C}$$

We can check that, with this definition, the elimination of implication is non-invertible and the elimination of disjunction is invertible, as expected. Invertibility fails when one of the non-eliminated premises is non-provable, while the conclusion would be – by applying another rule. For implication: if B is 1, both the eliminated premise and conclusion are provable but $\Gamma \vdash A$ may be non-provable. For disjunction: if $\Gamma \vdash C$ is provable, then we can build premises $\Gamma, A_i \vdash C$ by weakening, so the rule is applicable.

10.1.2. Intercalation syntax

For non-invertible rules, sequent calculus has a construction on each side of the sequent: a left-introduction judgment $\Gamma, [A] \vdash_{\mathsf{foc.l}} B$ and a right-introduction judgment $\Gamma \vdash_{\mathsf{foc.r}} [A]$. In natural deduction, both sorts of non-invertible rules (elimination or introduction) happen on the right. If we used a syntax such as $\Gamma \vdash_{\mathsf{foc.elim}} [A]$ for non-invertible elimination rules, there would thus be a risk of confusion with non-invertible introduction rules:

$$\text{NAT-FOC-ELIM-IMPL-BAD-NOTATION}\quad \frac{\Gamma \vdash_{\mathsf{foc.elim}} [A \to B] \qquad \Gamma \vdash_{\mathsf{foc.intro}} [A]}{\Gamma \vdash_{\mathsf{foc.elim}} [B]} \qquad \text{NAT-FOC-INTRO-DISJ-BAD-NOTATION}\quad \frac{\Gamma \vdash_{\mathsf{foc.intro}} [A_i]}{\Gamma \vdash_{\mathsf{foc.intro}} [A_1 + A_2]}$$

Instead, we use the following syntax from Brock-Nannestad and Schürmann [2010], itself inspired by the "intercalation calculus":

$$\text{NAT-FOC-ELIM-IMPL}\quad \frac{\Gamma \Downarrow A \to B \qquad \Gamma \Uparrow A}{\Gamma \Downarrow B} \qquad \text{NAT-FOC-INTRO-DISJ}\quad \frac{\Gamma \Uparrow A_i}{\Gamma \Uparrow A_1 + A_2}$$

This notation is a good notation: there is a good way to reconstruct which direction is associated with which rule. During proof search, elimination rules read rootward, they tell us how to go from the eliminated premise to its conclusion, so it is natural that they use the leafward arrow $\Downarrow$. Introduction rule read rootward, just as in the sequent calculus.

Remark 10.1.1. This notation also makes it visually obvious that implication elimination implies a "change of polarity" but remains focused. It is even better than the sequent calculus notation (touted for its visual symmetry) in this respect. ∗

10.1.3. Structural focusing for natural deduction

We give the full rules of our focused natural deduction in Figure 10.1 (Focused natural deduction, with explicit shifts), using explicit shifts in the style of Figure 7.7 (Focused sequent calculus for polarized propositional intuitionistic logic), but a batch move rule for invertible contexts as proposed in Section 7.3.2.

Figure 10.1.: Focused natural deduction, with explicit shifts

$$\textsc{nat-inv-impl-intro}\ \frac{\Gamma^{\mathsf{at}}; \Sigma, P \vdash_{\mathsf{inv}} N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} P \to N \mid} \qquad \textsc{nat-inv-conj-intro}\ \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N_1 \times N_2 \mid}$$

$$\textsc{nat-inv-disj-elim}\ \frac{\Gamma^{\mathsf{at}}; \Sigma, Q_1 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, Q_1 + Q_2 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}$$

$$\textsc{nat-inv-false-elim}\ \frac{}{\Gamma^{\mathsf{at}}; \Sigma, 0 \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}} \qquad \textsc{nat-inv-true-intro}\ \frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} 1 \mid}$$

$$\textsc{nat-inv-foc}\ \frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

$$\textsc{nat-foc-left-release-atom}\ \frac{\Gamma^{\mathsf{at}} \Downarrow X^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} X^-} \qquad \textsc{nat-foc-left-release-shift}\ \frac{\Gamma^{\mathsf{at}} \Downarrow \langle P \rangle^- \qquad \Gamma^{\mathsf{at}}; P \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}}$$

$$\textsc{nat-foc-contraction}\ \frac{}{\Gamma^{\mathsf{at}}, N \Downarrow N}$$

$$\textsc{nat-foc-right-release-atom}\ \frac{}{\Gamma^{\mathsf{at}}, X^+ \Uparrow X^+} \qquad \textsc{nat-foc-right-release-shift}\ \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} N \mid \emptyset}{\Gamma^{\mathsf{at}} \Uparrow \langle N \rangle^+} \qquad \textsc{nat-foc-right}\ \frac{\Gamma^{\mathsf{at}} \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P}$$

$$\textsc{nat-foc-conj-elim}\ \frac{\Gamma^{\mathsf{at}} \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \Downarrow N_i} \qquad \textsc{nat-foc-impl-elim}\ \frac{\Gamma^{\mathsf{at}} \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \Uparrow P}{\Gamma^{\mathsf{at}} \Downarrow N} \qquad \textsc{nat-foc-sum-intro}\ \frac{\Gamma^{\mathsf{at}} \Uparrow P_i}{\Gamma^{\mathsf{at}} \Uparrow P_1 + P_2}$$

The involved judgments are as follows:

- $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$, the invertible judgment, with the same structure as a sequent-calculus judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$
- $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}$, the judgment starting the focusing phase (a focus has not been chosen yet), with the same structure as the sequent-calculus judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}$
- $\Gamma^{\mathsf{at}} \Uparrow P$, the focused introduction judgment, corresponding to the right-focusing sequent judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]$
- $\Gamma^{\mathsf{at}} \Downarrow N$, the focused elimination judgment, corresponding to the left-focusing sequent judgment $\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}$.

The mapping between the various judgments is direct, except for the focused elimination judgment whose proofs, compared to left-focusing proofs, are *turned upside down*. For example, the "release" rules that explain how the focused phase stops (on an atom or a shift) are now the rootwardmost rule of the elimination phase. Conversely, the counterpart of the sequent rule that started a left-focusing phase $(\Gamma^{\mathsf{at}}, N), [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}$ now becomes the leaf rule concluding $\Gamma^{\mathsf{at}}, N \Downarrow N$.

Remark 10.1.2. Let us compare the rules of left-introduction and elimination focusing phases, in the case where they end up on a shifted positive formula – rather than a negative

atom.

$$\frac{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, N \vdash_{\mathsf{foc}} Q^{\mathsf{at}}}\ \text{SEQ} \qquad \frac{\Gamma^{\mathsf{at}}; P \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [\langle P \rangle^-] \vdash_{\mathsf{foc.l}} Q^{\mathsf{at}}}\ \text{SEQ}$$

$$\frac{\Gamma^{\mathsf{at}} \Downarrow \langle P \rangle^- \qquad \Gamma^{\mathsf{at}}; P \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}}\ \text{ND} \qquad \frac{}{\Gamma, N \Downarrow N}\ \text{ND}$$

In the sequent presentation, the left-focusing judgment remembers the goal Q^{at} that had to be proved at the beginning of the focusing phase. The focusing phase finishes at the leafwardmost rule of the left-focusing sequence; in the shift case, the final positive formula P is the "result" of this focused phase, and proof search starts again (with the result in context) on Q^{at}.

On the contrary, the elimination judgment $\Gamma^{\mathsf{at}} \Downarrow N$ does not depend on the current goal Q^{at} at all, and its leafwardmost rule is a leaf rule of conclusion $\Gamma^{\mathsf{at}}, N \Downarrow N$. The result formula P is not present in the leafwardmost rule, but in the rootwardmost rule (reversed order); it is as this level that a new premise is added that tries again to prove Q^{at} from the result P.

This difference captures the essence of why some elimination rules in sequent calculus are understood as a form of "continuation passing style". We can think of a subgoal as a recursive process called during proof search. The natural deduction rule that decides to focus on eliminations calls the subgoal $\Gamma^{\mathsf{at}} \Downarrow ?$ and inspects the results; when it is of the form $\langle P \rangle^-$, it then calls the invertible judgment. On the contrary, the sequent rule that decides left-focusing includes the current goal in the recursive call $\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} Q^{\mathsf{at}}$, and if the left-introduction phase succeeds it directly continues with this goal, without returning to its caller. $*$

10.1.4. Elimination or left-introduction rules for positives?

While we claim that the system of Figure 10.1 (Focused natural deduction, with explicit shifts) is in natural deduction style, one cannot help noticing that the invertible rules are actually sequent calculus rules; in particular, we have left-introduction rules for positives, rather than elimination rules as expected. Is this system really natural deduction? We have three different angles of answer.

First, we should point out that positive eliminations do not really fit natural deduction in the first place. Even though they do have a formulation that is different from the sequent calculus one, they stand out of the rest of the system and are the source of various difficulties when studying the meta-theory of the mixed-polarity system. We are making them more sequent-like than they were before, but the worm was already in the fruit. The negative elimination rules are the usual one, and this is what makes a system distinctively natural deduction in style.

Second, in a focused system, invertible rules do not really matter, because they are automatically applied in an irrelevant order. As this process is deterministic, they could in fact be removed from the term syntax, and reconstructed (in an irrelevant order) at type-checking time. Again, what really matters are the non-invertible elimination rules.

Third and finally, we tried to look for a formulation of the invertible positive rules that would be closer to the natural deduction rule, and didn't find any. In particular, it is interesting to see why the obvious idea does not work:

$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \langle P_1 + P_2 \rangle^- \mid \qquad \begin{array}{l}\Gamma^{\mathsf{at}}; \Sigma, P_1 \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}} \\ \Gamma^{\mathsf{at}}; \Sigma, P_2 \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}}} \qquad \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \langle 0 \rangle^- \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid}$$

Those would be sensible invertible rule if we could always choose them without having to make choices – choice is the privilege of non-invertible rules. They are not, because

we cannot know locally whether 0 would be provable (in the right rule), or even which $P_1 + P_2$ to attempt to prove (in the left rule). Furthermore, these invertible premises may incur arbitrary proof search, including non-invertible rules.

One idea would be to restrict this unbounded-search premise to a more specific judgment: instead of allowing any proofs of the positives to eliminate, could we allow only "simple" proofs? Using the focused elimination judgment $\Gamma^{\mathsf{at}} \Downarrow \langle P_1 + P_2 \rangle^-$ resembles the restriction on normal natural proofs (the eliminated premise cannot be a constructor, so it should start with an elimination or axiom rule), but it is still not invertible, as the focused elimination judgment has to make choice.

There remain an even simpler notion of "being provable": hypotheses are immediately provable if they are in the assumption context. Due to the polarity invariants, we know that positives are in Σ if they are in $\Gamma^{\mathsf{at}}, \Sigma$. This suggests the following restriction of those rules:

$$\frac{\Sigma \ni (P_1 + P_2) \qquad \begin{array}{c}\Gamma^{\mathsf{at}}; \Sigma, P_1 \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}} \\ \Gamma^{\mathsf{at}}; \Sigma, P_2 \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}}} \qquad \frac{0 \ni \Sigma}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid Q^{\mathsf{at}}}$$

These rules are (less convenient variants of) the sequent left-introduction rules that we use.

10.1.5. Equivalence with the focused sequent calculus

Comparing arbitrary natural deduction and sequent-calculus proofs is delicate, and in particular there is no one-to-one correspondence between cut-free proofs in either system. The restrictions of focusing give more structure to cut-free proofs, which allow to get a good correspondence.

Theorem 10.1.1 (Bijection between focused sequent calculus and focused natural deduction)**.**
There is a one-to-one correspondence between the cut-free focused sequent calculus proofs of Figure 10.1 (Focused natural deduction, with explicit shifts) and the cut-free focused natural deduction proofs of Figure 7.7 (Focused sequent calculus for polarized propositional intuitionistic logic).

Proof. The general idea of the proof is that the difference between the two focused systems is a stylistic choice of direction: elimination rules in natural deduction are written "rootward", while the corresponding left-introduction rules of sequent calculus are written "leafward". To translate between the two systems, it thus suffices to reverse the direction of these parts of the proof.

For example, consider the sequent proof:

$$\dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}}, [Z^-] \vdash_{\mathsf{foc.l}} Z^-}{\Gamma^{\mathsf{at}}, [Y \times Z^-] \vdash_{\mathsf{foc.l}} Z^-}}{\Gamma^{\mathsf{at}}, [X \times (Y \times Z^-)] \vdash_{\mathsf{foc.l}} Z^-}}{\Gamma^{\mathsf{at}} \ni X \times (Y \times Z^-) \vdash_{\mathsf{foc}} Z^-}$$

It corresponds to the following natural deduction proof, which is a direct reversal:

$$\dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}} \Downarrow X \times (Y \times Z^-)}{\Gamma^{\mathsf{at}} \Downarrow Y \times Z^-}}{\Gamma^{\mathsf{at}} \Downarrow Z^-}}{\Gamma^{\mathsf{at}} \ni X \times (Y \times Z^-) \vdash_{\mathsf{foc}} Z^-}$$

In the general case, remark that there is a direct correspondence between:

- invertible sequent judgments $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$ and invertible natural deduction judgments $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$

- right-focused sequent judgments $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]$ and introduction-focused natural deduction judgments $\Gamma^{\mathsf{at}} \Uparrow P$

To complete our correspondence, we give a one-to-one mapping between:

- choice-of-focusing sequent judgments $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}}$ and focused natural deduction judgments $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P$
- *partial* left-focused and elimination-focused phases, which is the reversal we described informally; it is a correspondence between partial proof derivations of the form

$$\dfrac{\dfrac{\Gamma^{\mathsf{at}}, [N'] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Pi}}{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \longleftrightarrow \qquad \dfrac{\dfrac{\Gamma^{\mathsf{at}} \Downarrow N}{\Pi'}}{\Gamma^{\mathsf{at}} \Downarrow N'}$$

  We write $\Pi \longleftrightarrow \Pi'$ when this correspondence holds.

The correspondence on the choice-of-focusing judgments is as follows:

$$\dfrac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P} \qquad \longleftrightarrow \qquad \dfrac{\Gamma^{\mathsf{at}} \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P}$$

$$\dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}}, [X^-] \vdash_{\mathsf{foc.l}} X^-}{\Pi}}{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} X^-}}{\Gamma^{\mathsf{at}} \ni N \vdash_{\mathsf{foc}} X^-} \qquad \longleftrightarrow \qquad \dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}} \Downarrow N}{\Pi'}}{\Gamma^{\mathsf{at}} \Downarrow X^-}}{\Gamma^{\mathsf{at}} \ni N \vdash_{\mathsf{foc}} X^-} \qquad \text{when} \qquad \Pi \longleftrightarrow \Pi'$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \emptyset \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [\langle Q \rangle^-] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}}{\Pi}}{\Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}}{\Gamma^{\mathsf{at}} \ni N \vdash_{\mathsf{foc}} P^{\mathsf{at}}} \qquad \longleftrightarrow \qquad \dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}} \Downarrow N}{\Pi'}}{\Gamma^{\mathsf{at}} \Downarrow \langle Q \rangle^-} \qquad \Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \emptyset \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \ni N \vdash_{\mathsf{foc}} P^{\mathsf{at}}} \qquad \text{when}$$

$$\Pi \longleftrightarrow \Pi'$$

The correspondence between the partial left-focused and elimination-focused phases is as follows. First, we describe the correspondence bet wen any inference rules (that is, partial proofs of height 2):

$$\dfrac{\Gamma^{\mathsf{at}}, [N_i] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [N\times] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \longleftrightarrow \qquad \dfrac{\Gamma^{\mathsf{at}} \Downarrow N\times}{\Gamma^{\mathsf{at}} \Downarrow N_i}$$

$$\dfrac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [Q] \qquad \Gamma^{\mathsf{at}}, [N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}, [Q \to N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \longleftrightarrow \qquad \dfrac{\Gamma^{\mathsf{at}} \Downarrow Q \to N \qquad \Gamma^{\mathsf{at}} \Uparrow Q}{\Gamma^{\mathsf{at}} \Downarrow N}$$

Then we can reverse longer proof by simply concatenating the reverses:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}}, [N_3] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\Pi_2}}{\Gamma^{\mathsf{at}}, [N_2] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}}{\Pi_1}}{\Gamma^{\mathsf{at}}, [N_1] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}} \qquad \longleftrightarrow \qquad \dfrac{\dfrac{\dfrac{\Gamma^{\mathsf{at}} \Downarrow N_1}{\Pi_1'}}{\Gamma^{\mathsf{at}} \Downarrow N_2}}{\Pi_2'}}{\Gamma^{\mathsf{at}} \Downarrow N_3} \qquad \text{when} \qquad \Pi_1 \leftrightarrow \Pi_1', \quad \Pi_2' \leftrightarrow \Pi_2'$$

□

10.2. A focused term syntax: focused λ-calculus

We propose a term syntax for this focused natural deduction that is as close as reasonably possible to the λ-calculus – as we did for our term syntax for the sequent calculus in Section 4.1.4 (A term syntax for the intuitionistic sequent calculus), we would like to think of it as mostly a subset of λ-terms with minor additions.

Looking at the four judgments of our focused system, we propose the following classes of terms:

- Terms for the invertible judgments $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}$ contain a mix of constructors and destructors and have subterms of arbitrary judgments; we will simply use the class of arbitrary (cut-free) terms, with meta-variable t. We will sometimes call these *invertible terms* to insist that they come from an invertible phase.

- Terms for the focused elimination judgment $\Gamma^{\mathsf{at}} \Downarrow N$ are variables, to which a series of elimination forms (function application or pair projection) are applied. This corresponds to the usual class (in the purely negative fragment) of *neutral terms*, often written with the meta-variable n. We call them *negative* neutral terms.

- Terms for the focused introduction judgment $\Gamma^{\mathsf{at}} \Uparrow P$ are series of introduction forms, eventually followed by an invertible proof term. We call them *positive* neutral terms, and use the meta-variable p.

- The choice-of-focusing judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$ has no interesting structure of its own, but it can become either an introduction-focused or elimination-focused phase, and we use the meta-variable f where this choice occurs. We call them *focusing terms*.

The grammar is described in Figure 10.2 (Term grammar for the focused λ-calculus), and the corresponding typing system (using mappings from term variables to types as contexts, instead of sets) is given in Figure 10.3 (Typing rules for the focused λ-calculus). We call this system the *focused λ-calculus*.

This grammar is designed to described well-typed terms, and we have used some typing annotations, which are not actually part of the term syntax, but describe the expected types of various subterms or variables – for the whole term to be well-typed. For example, the class p of positive neutral terms includes the whole class η of invertible terms (which itself includes f, in particular n and p), so as a grammar of untyped terms positive neutrals and invertible terms seem to be equivalent. However, because we will only allow the use of an invertible term inside a positive neutral at a negative type (and not in arbitrary positions), the two classes are very different for well-typed terms and expose interesting structure.

We could have a more explicit syntax, with term markers to indicate the various phase transitions that would remove the ambiguities, but we suspect that it would be much less pleasant to work with. In practice we will always manipulate *typed terms*, associated to their typing derivation, from which all necessary structural information can be obtained.

Remark 10.2.1. Our focused sequent calculus was cut-free in the literal sense of not having a cut rule. It is interesting to check that this focused natural deduction, and the focused λ-calculus, are also "cut-free" in the sense that the terms are irreducible. At first sight, this seems to come from the restriction on the elimination judgment $\Gamma \vdash n \Downarrow N$, that elimination forms are only applied to neutrals and thus never create redexes. But this omits an important subtlety of the system, namely the use of a `let`-binding to represent (some) left focusing phases, $\texttt{let}\ (x : P) = n\ \texttt{in}\ t$.

We think that this construction should not be considered as a cut; in particular, we remark that if you substitute away all those `let`-bindings, the substituted term remains irreducible: a variable $(x : P)$ of strictly positive type will always be matched-upon by the next invertible phase, but it will always be substituted with a neutral term n so the

Figure 10.2.: Term grammar for the focused λ-calculus

$$
\begin{array}{lll}
t, u, r ::= & & \text{(invertible) terms} \\
& \mid \lambda x.\, t & \lambda\text{-abstraction} \\
& \mid (t, u) & \text{pair} \\
& \mid \texttt{match}\ x\ \texttt{with}\ \mid \sigma_1\ x \to u_1 \mid \sigma_2\ x \to u_2 & \text{variable case split} \\
& \mid () & \text{trivial} \\
& \mid \texttt{absurd}(x) & \text{absurd variable} \\
& \mid (f : P^{\mathsf{at}}) & \text{focusing term} \\
\\
f, g \quad ::= & & \text{focusing terms} \\
& \mid (n : X^-) & \text{negative conclusion} \\
& \mid \texttt{let}\ (x : P) = n\ \texttt{in}\ t & \text{positive binding} \\
& \mid (p : P) & \text{positive conclusion} \\
\\
n, m \quad ::= & & \text{negative neutral terms} \\
& \mid \pi_i\ n & \text{projection} \\
& \mid n\ p & \text{application} \\
& \mid (x : N) & \text{negative head variable} \\
\\
p, q \quad ::= & & \text{positive neutral terms} \\
& \mid \sigma_i\ p & \text{injection} \\
& \mid (x : X^+) & \text{positive head variable} \\
& \mid (t : N) & \text{invertible conclusion}
\end{array}
$$

resulting elimination will never become a redex. One can talk of this `let`-binding as an "irreducible cut".

Another way to describe this would be to have a typing rule of the form

$$\frac{\Gamma^{\mathsf{at}} \vdash n \Downarrow \langle P \rangle^- \qquad \Gamma^{\mathsf{at}}; x : P \vdash_{\mathsf{inv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t[n/x] : Q^{\mathsf{at}}}\ \text{FOCLC-SUBST-POS}$$

so that proof terms are pure λ-terms (without `let`-bindings), but I personally dislike having a typing rule that cannot at all be reconstructed from the syntactic structure of its proof term. If necessary for precision and clarity, we should rather define an erasure function from our focused λ-terms to usual λ-terms (substituting the `let` away), and explicitly reason on the image of the erasure. $*$

10.2.1. Defocusing into non-focused λ-terms

We have glossed over the fact that focused λ-terms are not *quite* λ-terms as defined in Figure 3.1 (Full simply-typed lambda-calculus $\Lambda\mathsf{C}(\to, \times, 1, +, 0)$), because they use the $\texttt{let}\ x = t\ \texttt{in}\ u$ form that is not formally part of the syntax – it was only in our term system for the sequent calculus drafted in Figure 4.2 (Terms of the sequent-form λ-calculus $\mathsf{S}\Lambda\mathsf{C}(\to, \times, 1, +, 0)$).

In Figure 10.4 (Erasure of focusing $\lfloor t \rfloor_{\mathtt{foc}}$) we define the *erasure of focusing* operation $\lfloor \_ \rfloor_{\mathtt{foc}}$ that, for any focused λ-term t, gives its *erasure* as a simple λ-term $\lfloor t \rfloor_{\mathtt{foc}}$, obtained by replacing each $\texttt{let}\ x = t\ \texttt{in}\ u$ form by the substitution $u[t/x]$.

In Chapter 7 (Focusing in sequent calculus), Section 7.4 (Direct relations between focused and non-focused systems), we established a translation from each focused sequent proof of a judgment on polarized formulas A into a non-focused sequent proof of a judgment on the corresponding depolarized formulas $\lfloor A \rfloor_{\pm}$. We can now state a similar result for

Figure 10.3.: Typing rules for the focused λ-calculus

$$\textsc{foclc-lam}\ \frac{\Gamma^{\mathsf{at}}; \Sigma, x : P \vdash_{\mathsf{inv}} t : N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \lambda x.\, t : P \to N \mid} \qquad \textsc{foclc-pair}\ \frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t_1 : N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t_2 : N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} (t_1, t_2) : N_1 \times N_2 \mid}$$

$$\textsc{foclc-case}\ \frac{\Gamma^{\mathsf{at}}; \Sigma, x : Q_1 \vdash_{\mathsf{inv}} t_1 : N \mid P^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, x : Q_2 \vdash_{\mathsf{inv}} t_2 : N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, x : Q_1 + Q_2 \vdash_{\mathsf{inv}} \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to t_1 \mid \sigma_2\ x \to t_2 : N \mid P^{\mathsf{at}}}$$

$$\textsc{foclc-absurd}\ \frac{}{\Gamma^{\mathsf{at}}; x : \Sigma, 0 \vdash_{\mathsf{inv}} \mathtt{absurd}(x) : N \mid P^{\mathsf{at}}} \qquad \textsc{foclc-trivial}\ \frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} () : 1 \mid}$$

$$\textsc{foclc-inv-foc}\ \frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} f : (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} f : \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

$$\textsc{foclc-concl-neg}\ \frac{\Gamma^{\mathsf{at}} \vdash n \Downarrow X^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} n : X^-} \qquad \textsc{foclc-let-pos}\ \frac{\Gamma^{\mathsf{at}} \vdash n \Downarrow \langle P \rangle^- \qquad \Gamma^{\mathsf{at}}; x : P \vdash_{\mathsf{inv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} \mathtt{let}\ x = n\ \mathtt{in}\ t : Q^{\mathsf{at}}} \qquad \textsc{foclc-var-neg}\ \frac{}{\Gamma^{\mathsf{at}}, x : N \vdash x \Downarrow N}$$

$$\textsc{foclc-var-pos}\ \frac{}{\Gamma^{\mathsf{at}}, x : X^+ \vdash x \Uparrow X^+} \qquad \textsc{foclc-foc-inv}\ \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} t : N \mid}{\Gamma^{\mathsf{at}} \vdash t \Uparrow \langle N \rangle^+} \qquad \textsc{foclc-concl-pos}\ \frac{\Gamma^{\mathsf{at}} \vdash p \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} p : P}$$

$$\textsc{foclc-proj}\ \frac{\Gamma^{\mathsf{at}} \vdash n \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \vdash \pi_i\ n \Downarrow N_i} \qquad \textsc{foclc-app}\ \frac{\Gamma^{\mathsf{at}} \vdash n \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \vdash p \Uparrow P}{\Gamma^{\mathsf{at}} \vdash n\ p \Downarrow N} \qquad \textsc{foclc-inj}\ \frac{\Gamma^{\mathsf{at}} \vdash p \Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash \sigma_i\ p \Uparrow P_1 + P_2}$$

Figure 10.4.: Erasure of focusing $\lfloor t \rfloor_{\mathtt{foc}}$

$$\begin{aligned}
\lfloor \lambda x.\, t \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \lambda x.\, \lfloor t \rfloor_{\mathtt{foc}} \\
\lfloor (t_1, t_2) \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} (\lfloor t_1 \rfloor_{\mathtt{foc}}, \lfloor t_2 \rfloor_{\mathtt{foc}}) \\
\left\lfloor \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to u_1 \mid \sigma_2\ x \to u_2 \right\rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to \lfloor u_1 \rfloor_{\mathtt{foc}} \mid \sigma_2\ x \to \lfloor u_2 \rfloor_{\mathtt{foc}} \\
\lfloor () \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} () \\
\lfloor \mathtt{absurd}(x) \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \mathtt{absurd}(x)
\end{aligned}$$

$$\lfloor \mathtt{let}\ x = n\ \mathtt{in}\ t \rfloor_{\mathtt{foc}} \stackrel{\mathsf{def}}{=} \lfloor t \rfloor_{\mathtt{foc}} [\lfloor n \rfloor_{\mathtt{foc}} / x]$$

$$\begin{aligned}
\lfloor \pi_i\ t \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \pi_i\ \lfloor t \rfloor_{\mathtt{foc}} \\
\lfloor n\ p \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \lfloor n \rfloor_{\mathtt{foc}}\ \lfloor p \rfloor_{\mathtt{foc}} \\
\lfloor x \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} x \\
\lfloor \sigma_i\ t \rfloor_{\mathtt{foc}} &\stackrel{\mathsf{def}}{=} \sigma_i\ \lfloor t \rfloor_{\mathtt{foc}}
\end{aligned}$$

focused natural deduction, strengthened with a correspondence between the proof terms themselves.

Lemma 10.2.1 (Type soundness of defocusing)**.**

The following implications hold:

$$\begin{array}{lcl}
\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid P^{\mathsf{at}} & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm}, \lfloor \Sigma \rfloor_{\pm} \vdash \lfloor t \rfloor_{\mathtt{foc}} : (\lfloor N \rfloor_{\pm} \mid \lfloor P^{\mathsf{at}} \rfloor_{\pm}) \\
\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} f : P^{\mathsf{at}} & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor f \rfloor_{\mathtt{foc}} : \lfloor P^{\mathsf{at}} \rfloor_{\pm} \\
\Gamma^{\mathsf{at}} \vdash n \Downarrow N & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor n \rfloor_{\mathtt{foc}} : \lfloor N \rfloor_{\pm} \\
\Gamma^{\mathsf{at}} \vdash p \Uparrow P & \Longrightarrow & \lfloor \Gamma^{\mathsf{at}} \rfloor_{\pm} \vdash \lfloor p \rfloor_{\mathtt{foc}} : \lfloor P \rfloor_{\pm}
\end{array}$$

Proof. By direct mutual induction on the premises. □

The following technical lemma gives a specification of defocusing translations will be useful to establish later results.

Lemma 10.2.2 (Composability of defocusing)**.**
Any subterm of $\lfloor t \rfloor_{\mathtt{foc}}$ *is of the form*

$$\lfloor u \rfloor_{\mathtt{foc}} [\lfloor n_1 \rfloor_{\mathtt{foc}} / x_1] [\lfloor n_2 \rfloor_{\mathtt{foc}} / x_2] \dots [\lfloor n_n \rfloor_{\mathtt{foc}} / x_n]$$

where u *is a subterm of* t*, and the* $\mathtt{let}\ x_i = n_i$ *are the* `let`*-bindings in* t *that scope over* u*.*

Proof. By induction on (the subterms of) t. □

10.2.2. Correspondence with focused sequent terms

Another natural translation for focused λ-terms is to translate them into proof terms for the sequent calculus. We established a bijection between proof derivations in the two system in Theorem 10.1.1 (Bijection between focused sequent calculus and focused natural deduction), and this result naturally lifts into a bijection $t \longleftrightarrow t'$ between well-typed focused λ-terms and sequent terms well-typed in the focused sequent calculus.

We define in Figure 10.5 the translation $[\![t]\!]_{\mathsf{focseq}}$ from focused λ-terms to sequent terms – as defined in Section 4.1.4 (A term syntax for the intuitionistic sequent calculus). Negative neutrals n are translated into linear binding contexts (see Figure 5.6), that is sequences of `let`-bindings; more precisely, the translation of a negative neutral n binds some variable x in a body t', which is a sequent term. We write $[\![n]\!]^{x}_{\mathsf{focseq}}(t')$ for this translation.

In this document we use the notation $\Pi :: \Gamma \vdash A$ to say that Π is a valid derivation of the judgment $\Gamma \vdash A$. In the result below, we use the notation $t :: \mathcal{J}$, where t is a sequent term as defined in Section 4.1.4 (A term syntax for the intuitionistic sequent calculus), and $\mathcal{J}$ is a judgment of the focused sequent calculus of Figure 7.8 (Focused sequent calculus with polarized formulas and batch context validation), to say that t is a valid proof term for the judgment $\mathcal{J}$.

Lemma 10.2.3 (Type soundness of sequent translation)**.**
The following implications hold:

$$\begin{array}{lcl}
\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid P^{\mathsf{at}} & \Longrightarrow & [\![t]\!]_{\mathsf{focseq}} :: \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}} \\
\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} f : P^{\mathsf{at}} & \Longrightarrow & [\![f]\!]_{\mathsf{focseq}} :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}} \\
\Gamma^{\mathsf{at}} \vdash n \Downarrow N & \Longrightarrow & \forall x, t', P^{\mathsf{at}}, \quad t' :: \Gamma^{\mathsf{at}}, [x : N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}} \Longrightarrow [\![n]\!]^{x}_{\mathsf{focseq}}(t') :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P^{\mathsf{at}} \\
\Gamma^{\mathsf{at}} \vdash p \Uparrow P & \Longrightarrow & [\![t']\!]_{\mathsf{focseq}} :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P]
\end{array}$$

Proof. By induction on the typing derivation. □

Theorem 10.2.4 (Bijection between focused λ-terms and focused sequent terms)**.**
The translation $[\![t]\!]_{\mathsf{focseq}}$ *is bijective: it establishes a one-to-one correspondence (for* α*-equivalence) between well-typed focused* λ*-terms and well-typed focused sequent terms.*

Proof. This is proved by exhibiting an inverse function, from focused typing derivations for sequent terms to well-typed focused λ-terms, such that composing the two functions gives the identity in either domains.

Most typing rules for focused sequent calculus correspond to exactly one case in the definition of $[\![t]\!]_{\mathsf{focseq}}$, so the proof in these cases is immediate: there is exactly one possible shape of the inverse λ-term, and this is the only translation rule for this shape.

Figure 10.5.: Translating focused λ-terms into sequent term syntax

$$
\begin{array}{lcll}
[\![\lambda x.\, t\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & \lambda x.\, [\![t\,]\!]_{\mathsf{focseq}} & \\
[\![(t_1, t_2)\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & \big([\![t_1\,]\!]_{\mathsf{focseq}}, [\![t_2\,]\!]_{\mathsf{focseq}}\big) & \\
\left[\!\!\left[\begin{array}{l}\mathtt{match}\ x\ \mathtt{with}\\ \mid\ \sigma_1\ x \to u_1\\ \mid\ \sigma_2\ x \to u_2\end{array}\right]\!\!\right]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & \begin{array}{l}\mathtt{match}\ x\ \mathtt{with}\\ \mid\ \sigma_1\ x \to [\![u_1\,]\!]_{\mathsf{focseq}}\\ \mid\ \sigma_2\ x \to [\![u_2\,]\!]_{\mathsf{focseq}}\end{array} & \\
[\![()\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & () & \\
[\![\mathtt{absurd}(x)\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & \mathtt{absurd}(x) & \\
[\![(f : P^{\mathsf{at}})\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & [\![f\,]\!]_{\mathsf{focseq}} & \\
[\![(n : X^{-})\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & [\![n\,]\!]^{x}_{\mathsf{focseq}}(x) & \text{for } x \text{ fresh} \\
[\![\mathtt{let}\ x = n\ \mathtt{in}\ t\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & [\![n\,]\!]^{x}_{\mathsf{focseq}}([\![t\,]\!]_{\mathsf{focseq}}) & \\
[\![(p : P)\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & [\![p\,]\!]_{\mathsf{focseq}} & \\
[\![\pi_i\ n\,]\!]^{x}_{\mathsf{focseq}}(t') & \stackrel{\mathsf{def}}{=} & [\![n\,]\!]^{y}_{\mathsf{focseq}}(\mathtt{let}\ x = \pi_i\ y\ \mathtt{in}\ t') & \text{for } y \text{ fresh} \\
[\![n\ p\,]\!]^{x}_{\mathsf{focseq}}(t') & \stackrel{\mathsf{def}}{=} & [\![n\,]\!]^{y}_{\mathsf{focseq}}(\mathtt{let}\ x = y\ p\ \mathtt{in}\ t') & \text{for } y \text{ fresh} \\
[\![(y : N)\,]\!]^{x}_{\mathsf{focseq}}(t') & \stackrel{\mathsf{def}}{=} & t'[y/x] & \\
[\![\sigma_i\ p\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & \sigma_i\ [\![p\,]\!]_{\mathsf{focseq}} & \\
[\![(x : X^{+})\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & x & \\
[\![(t : N)\,]\!]_{\mathsf{focseq}} & \stackrel{\mathsf{def}}{=} & [\![t\,]\!]_{\mathsf{focseq}} &
\end{array}
$$

The only exception, of course, concerns negative neutrals. When we have a sequent derivation of the general form

$$\frac{u' :: \Gamma^{\mathsf{at}}, [y : N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{u' :: \Gamma^{\mathsf{at}} \ni y : N \vdash_{\mathsf{foc}} P^{\mathsf{at}}}$$

we do not know whether a term of the form $(n : X^{-})$ or $(\mathtt{let}\ x = n\ \mathtt{in}\ t)$ should be chosen for the inverse translation.

This is solved by proving a stronger recursion hypothesis for the left-focusing case. We present it as a nested lemma below. The proof of the theorem and the lemma are done by mutual induction, but we prove them separately for readability.

Lemma 10.2.5 (Decomposition of left-focused phases)**.**
For any well-typed left-focused sequent term u' of the general form

$$u' :: \Gamma^{\mathsf{at}}, [y : N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}$$

and a variable x, there is a unique pair of a negative neutral n and a subterm t' of u'

$$\Gamma^{\mathsf{at}} \vdash n \Downarrow N \qquad\qquad t' :: \Gamma^{\mathsf{at}}, [x : \langle Q^{\mathsf{at}} \rangle^{-\mathsf{at}}] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}$$

such that

$$u' =_{\alpha} [\![n\,]\!]^{x}_{\mathsf{focseq}}(t')$$

Note that the uniqueness of the pair crucially depends on the hypothesis that x has a shifted positive or atomic type $\langle Q^{\mathsf{at}} \rangle^{-\mathsf{at}}$, and thus corresponds to the end of the focusing

phase. Without this constraint, if x could have any negative type M, then for example the pair (x, u') would also be a valid choice.

This lemma suffices to conclude the proof of the theorem: if the unique pair is of the form (n, x), then the inverse λ-term is n, if it is of the form (n, t') where t' is not a variable, then it is of the form $\texttt{let}\ x = n\ \texttt{in}\ t$, where t is the inverse of t' – assuming inductively that t' has an inverse is correct as we know that t' is a subterm of the term u' we are currently inverting. □

Proof (Lemma 10.2.5 (Decomposition of left-focused phases)). By induction on u'.

If it is of the form

$$x :: \Gamma^{\mathsf{at}}, [x : X^-] \vdash_{\mathsf{foc.l}} X^-$$

then the pair is just (x, x).

If it is of the form

$$\frac{t' :: \Gamma^{\mathsf{at}}; x : Q \vdash_{\mathsf{inv}} P^{\mathsf{at}} \mid}{t' :: \Gamma^{\mathsf{at}}, [x : \langle Q \rangle^-] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}$$

then the pair is (x, t').

In the hereditary cases, we assume that u' is built by applying a left-focused inference rule to some left-focused term r', which is itself (by induction hypothesis) uniquely decomposed through the variable y into a pair (n, t'). There are two such cases. If we have

$$\frac{r' :: \Gamma^{\mathsf{at}}, [y : N_i] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\texttt{let}\ y = \pi_i\ x\ \texttt{in}\ r' :: \Gamma^{\mathsf{at}}, [x : N_1 \times N_2] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}$$

then the pair is $(\pi_i\ n, t')$, and if we have

$$\frac{p' :: \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc.r}} [P] \qquad r' :: \Gamma^{\mathsf{at}}, [y : N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}{\texttt{let}\ y = x\ p'\ \texttt{in}\ r' :: \Gamma^{\mathsf{at}}, [x : P \to N] \vdash_{\mathsf{foc.l}} P^{\mathsf{at}}}$$

then the pair is $(n\ p, t')$, where p is the unique inverse image of p': $[\![p]\!]_{\mathsf{focseq}} =_\alpha p'$. □

After translating a focused λ-term into a focused sequent term, we could forget about the focusing structure of this sequent term, and apply the general (not one-to-one) translation $[\![\_]\!]_{\mathrm{ND}}$ of Lemma 4.2.7, from non-focused sequent terms to non-focused λ-terms. This is in fact equivalent to the defocusing translation of Figure 10.4 (Erasure of focusing $\lfloor t \rfloor_{\mathtt{foc}}$).

Lemma 10.2.6 (Translation commutation).

$$\left[\!\left[[\![t]\!]_{\mathsf{focseq}} \right]\!\right]_{\mathrm{ND}} =_\alpha \lfloor t \rfloor_{\mathtt{foc}}$$

Proof. By induction on t. For neutral decomposition $[\![t]\!]^x_{\mathsf{focseq}}(t')$ we prove, by induction on n,

$$\forall n, t, t', \qquad [\![t']\!]_{\mathrm{ND}} =_\alpha \lfloor t \rfloor_{\mathtt{foc}} \qquad \Longrightarrow \qquad \left[\!\left[[\![n]\!]^x_{\mathsf{focseq}}(t') \right]\!\right]_{\mathrm{ND}} =_\alpha \lfloor n \rfloor_{\mathtt{foc}} [\lfloor t \rfloor_{\mathtt{foc}} / x]$$

□

10.3. Focusing completeness by big-step translation

Reference *The present section is extracted from Scherer and Rémy [2015].*

Theorem 10.3.1 (Completeness of focusing).
The focused intuitionistic logic is complete with respect to intuitionistic logic. It is also computationally complete: any well-typed lambda-term is $\beta\eta$-equivalent to (the `let`*-substitution of) a proof witness of the focused logic.*

Proof (Logical completeness)**.** This system naturally embeds into the system LJF of Liang and Miller [2007] (by polarizing the products negatively), which is proved sound, and complete for any polarization choice. □

Computational completeness could be argued to be folklore, or a direct adaptation of previous work on completeness of focusing: a careful reading of the elegant presentation of Simmons [2011] (or Laurent [2004] for linear logic) would show that its logical completeness argument in fact proves computational correctness. Without sums, it exactly corresponds to the fact that β-short η-long normal forms are computable for well-typed lambda-terms of the simply-typed calculus.

We introduce an explicit η-expanding, `let`-introducing transformation from β-normal forms to valid focused proofs for our system. Detailing this transformation also serves by building intuition for the computational completeness proof of the *saturating* focused logic in Figure 11.2 (Saturation translation), Section 11.4 (Canonicity of saturated proofs).

Proof (Computational completeness)**.** Let us remark that simply-typed lambda-calculus without fixpoints is strongly normalizing (Section 8.1 (Strong normalization for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$)), and write $\mathsf{NF}_\beta(t)$ for the (full) β-normal form of t.

We define in Figure 10.6 an expansion relation $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ that turns any well-typed β-normal form $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Sigma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$ into a valid *focused* derivation $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t' : N \mid Q^{\mathsf{at}}$.

We use four mutually recursive judgments, one for each judgment in the focused λ-calculus of Figure 10.3 (Typing rules for the focused λ-calculus): the invertible and focusing translations $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ and $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t \rightsquigarrow t' : Q^{\mathsf{at}}$, and the negative and positive neutral translations $\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow N$ and $\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow P$. For the two first judgments, the inputs are the context(s), source term, and translation type, and the output is the translated term. For the neutral judgments the translation type is an output – this reversal follows the usual bidirectional typing of normal forms.

Three distinct aspects of the translation need to be discussed:

1. Finiteness. It is not obvious that a translation derivation $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ *exists* for any $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Sigma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$, because subderivations of invertible rules perform β-normalization of their source term, which may a priori make it grow without bounds. It could be the case that for certain source terms, there does not exist any finite derivation.

2. Partiality. As the rules are neither type- nor syntax-directed, it is not obvious that any input term, for example $\texttt{match}\ t_1\ t_2\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2$, has a matching translation rule.

3. Non-determinism. The invertible rules are not quite typed-directed, and the REW-FOC-ELIM rule is deeply non-deterministic, as it applies for any neutral subterm of the term being translated – that is valid in the current typing environment. This non-determinism allows the translation to accept *any* valid focused derivation for an input term, reflecting the large choice space of when to apply the FOC-ELIM rule in backward focused proof search.

Totality The use of β-normalization inside subderivations precisely corresponds to the "unfocused admissibility rules" of Simmons [2011]. To control the growth of subterms in the premises of rules, we will use as a measure (or accessibility relation) the three following structures, from the less to the more important in lexicographic order:

- The (measure of the) types in the context(s) of the rewriting relation. This measure is strictly decreasing in the invertible elimination rule for sums, but increasing for the arrow introduction rule.

- The (measure of the) type of the goal of the rewriting relation. This measure is

Figure 10.6.: Translation into focused terms

REW-INV-SUM
$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P_1 \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t[\sigma_1\ x/x]) \rightsquigarrow t'_1 : N \mid Q^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, x : P_2 \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t[\sigma_2\ x/x]) \rightsquigarrow t'_2 : N \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, x : A_1 + A_2 \vdash_{\mathsf{inv}} t \rightsquigarrow \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to t'_1 \mid \sigma_2\ x \to t'_2 : N \mid Q^{\mathsf{at}}}$$

REW-INV-ARROW
$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t\ x) \rightsquigarrow u' : N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow \lambda x.\, u' : P \to N \mid}$$

REW-INV-PROD
$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(\pi_1\ t) \rightsquigarrow u'_1 : N_1 \mid \qquad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(\pi_2\ t) \rightsquigarrow u'_2 : N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow (u'_1, u'_2) : N_1 \times N_2 \mid}$$

REW-INV-FOC
$$\frac{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{foc}} t \rightsquigarrow t' : (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} t \rightsquigarrow t' : \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

REW-FOC-ATOM
$$\frac{\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow X^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} n \rightsquigarrow n' : X^-}$$

REW-FOC-INTRO
$$\frac{\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t \rightsquigarrow t' : P}$$

REW-FOC-ELIM
$$\frac{\Gamma^{\mathsf{at}}, x : P \vdash C[x] : Q^{\mathsf{at}} \qquad \Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow \langle P \rangle^- \qquad \Gamma^{\mathsf{at}}; x : P \vdash_{\mathsf{inv}} C[x] \rightsquigarrow t' : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} C[n] \rightsquigarrow \mathtt{let}\ x = n'\ \mathtt{in}\ t' : Q^{\mathsf{at}}}$$

REW-UP-SUM
$$\frac{\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash \sigma_i\ t \rightsquigarrow \sigma_i\ t' \Uparrow P_1 + P_2}$$

REW-UP-INV
$$\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid \emptyset}{\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow \langle N \rangle^+}$$

REW-UP-VAR
$$\frac{(x : X^+) \in X^+}{\Gamma^{\mathsf{at}} \vdash x \rightsquigarrow x \Uparrow X^+}$$

REW-DOWN-VAR
$$\frac{(x : N) \in \Gamma^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash x \rightsquigarrow x \Downarrow N}$$

REW-DOWN-PAIR
$$\frac{\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \vdash \pi_i\ n \rightsquigarrow \pi_i\ n' \Downarrow N_i}$$

REW-DOWN-ARROW
$$\frac{\Gamma^{\mathsf{at}} \vdash t : P \qquad \Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow P}{\Gamma^{\mathsf{at}} \vdash n\ t \rightsquigarrow n'\ t' \Downarrow N}$$

strictly decreasing in the introduction rules for arrow, products and sums, but increasing in REW-FOC-ELIM or neutral rules.

- The set of (measures of) translation judgments $\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow N$ for well-typed neutral subterms n of the translated term whose type N is of maximal mesaure.

  Note that while that complexity seems to increase in the premises of the judgment $\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow N$, this judgment should be read top-down: all the sub-neutrals of n already appear as subterms of the source t in the REW-FOC-ELIM application $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t \rightsquigarrow ? : Q^{\mathsf{at}}$ that called $\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow ? \Downarrow N$.

  This measure is non-increasing in all non-neutral rules other than REW-FOC-ELIM, in particular the rules that require re-normalization (β-reduction or η-reduction may at best duplicate the occurrences of the neutral of maximal type, but not create new neutrals at higher types). In the sum-elimination rule, the neutral x of type $P_1 + P_2$ is shadowed by another neutral x of smaller type (P_1 or P_2). In the arrow rule, a new neutral $t\ x$ is introduced if t is already neutral, but then $t\ x : N$ is at a strictly smaller type than $t : P \to N$. In the product rule, new neutral $\pi_i\ t : N_i$ are introduced if $t : N_1 \times N_2$ is neutral, but again at strictly smaller types.

Finally, this measure is strictly decreasing when applying REW-FOC-ELIM. Note that by typing we know that n, of shifted positive type $\langle P \rangle^-$, is not the whole term t, of positive or atomic type Q^{at} – ruling this case out is an advantage of using explicit shifts, compared to the presentation of Scherer and Rémy [2015].

This three-fold measures proves termination of $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow ? : N \mid Q^{\mathsf{at}}$ seen as an algorithm: we have proved that there are no infinite derivations for the translation judgments.

Partiality The invertible translation rules are type-directed; the neutral translation rules are directed by the syntax of the neutral source term. But the focusing translation rules are neither type- nor source-directed. We have to prove that one of those three rule applies for any term – assuming that the context is negative or atomic, and the goal type positive or atomic.

The term t either starts with a constructor (introduction form), a destructor (elimination form), or it is a variable; a constructor may be neither a λ or a pair, as we assumed the type is positive or atomic. It starts with a non-empty series of sum injections, followed by a negative or atomic term, we can use REW-FOC-INTRO. Otherwise it contains (possibly after some sum injections) a positive subterm that does not start with a constructor.

If it starts with an elimination form or a variable, it may or may not be a neutral term. If it is neutral, then one of the rules REW-FOC-ATOM (if the goal is atomic) or REW-FOC-INTRO (if the goal is strictly positive) applies. If it is not neutral (in particular not a variable), it has an elimination form applied to a subterm of the form $\texttt{match}\ t\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2\end{array}$; but then (recursively) either t is a (strictly positive) neutral, or of the same form, and the rule REW-FOC-ELIM is eventually applicable.

We have proved that for any well-typed $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Sigma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$, there exists a translation derivation $\Gamma^{\mathsf{at}}; \Gamma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ for some t'.

Non-determinism The invertible rules may be applied in any order; this means that for any t' such that $\Gamma^{\mathsf{at}}; \Gamma \vdash t \rightsquigarrow t' : A$, for any $t'' =_{\mathtt{icc}} t'$ we also have $\Gamma^{\mathsf{at}}; \Gamma \vdash t \rightsquigarrow t'' : A$: a non-focused term translates to a full equivalence class of commutative conversions.

The rule REW-FOC-ELIM may be applied at will (as soon as the `let`-extruded neutral n is well-typed in the current context). Applying this rule eagerly would give a valid saturated focused deduction. Not enforcing its eager application allows (but we need not formally prove it) *any* $\beta\eta$-equivalent focused proof to be a target of the translation.

Validity We prove by immediate (mutual) induction that, if $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Gamma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$ holds, then the focusing translations are type-preserving:

- if $\Gamma^{\mathsf{at}}; \Sigma; t \vdash_{\mathsf{inv}} t' \rightsquigarrow N : Q^{\mathsf{at}} \mid$ then $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t' : N \mid Q^{\mathsf{at}}$
- if $\Gamma = \emptyset$ and $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t \rightsquigarrow t' : Q^{\mathsf{at}}$ then $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} t' : Q^{\mathsf{at}}$
- if $\Gamma = \emptyset$ and $\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow N$ then $\Gamma^{\mathsf{at}} \vdash n' \Downarrow N$
- if $\Gamma = \emptyset$ and $\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow P$ then $\Gamma^{\mathsf{at}} \vdash t' \Uparrow P$

Soundness Finally, we prove that the translation preserves $\beta\eta$-equivalence. If $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Sigma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$ and $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$, then $t \approx_{\beta\eta} t'$, that is, $t \approx_{\beta\eta} \lfloor t' \rfloor_{\mathtt{foc}}$.

As for validity, this is proved by mutual induction on all judgments. The interesting cases are the invertible rules and the focusing elimination rule; all other cases are discarded by immediate induction.

The invertible rules correspond to an η-expansion step. For REW-INV-PROD, we have that $t \approx_\eta (\pi_1\ t, \pi_2\ t)$, and can thus deduce by induction hypothesis that $t \approx_{\beta\eta} (u'_1, u'_2)$. For

REW-INV-ARROW, we have that $t \approx_\eta \lambda x.\, t$, and can thus deduce by induction hypothesis that $t \approx_{\beta\eta} \lambda x.\, t'$. For REW-INV-SUM, let us write t as $C[x]$ with $x \notin C$, we have that

$$\begin{array}{rcll} t & = & C\,[x : A + B] & \\ & \approx_\eta & \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to C\,[\sigma_1\ x] \\ \sigma_2\ x \to C\,[\sigma_2\ x] \end{array} & \\ & = & \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to t[\sigma_1\ x/x] \\ \sigma_2\ x \to t[\sigma_2\ x/x] \end{array} & \\ & \approx_{\beta\eta} & \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to {t'}_1 \\ \sigma_2\ x \to {t'}_2 \end{array} & \text{(by induction hypothesis)} \end{array}$$

In the case of the rule REW-FOC-ELIM, the fundamental transformation is the `let`-binding that preserves $\beta\eta$-equivalence.

$$\begin{array}{rcll} t & = & t[x/n][n/x] & \\ & \approx_{\beta\eta} & \texttt{let}\ x = n\ \texttt{in}\ t[x/n] & \\ & \approx_{\beta\eta} & \texttt{let}\ x = n'\ \texttt{in}\ t' & \text{(by induction hypothesis)} \end{array}$$

Conclusion We have proved computational completeness of the focused logic: for any $\lfloor \Gamma^{\mathsf{at}} \rfloor_\pm, \lfloor \Gamma \rfloor_\pm \vdash t : \lfloor (N \mid Q^{\mathsf{at}}) \rfloor_\pm$, there exists some $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t' : N \mid Q^{\mathsf{at}}$, such that $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t) \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$, with $t \approx_{\beta\eta} \lfloor t' \rfloor_{\mathtt{foc}}$. □

10.4. Focused phases are focused contexts

Now that we have a term syntax, we can use term contexts as a convenient concept and notation to capture whole focused phases.

10.4.1. Invertible multi-contexts

Consider an invertible term for the invertible judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$. We know that t starts with some invertible rules decomposing the formulas of Σ, and N if it is non-empty. Those rules form a prefix of the term, after which may be found zero, one or several focused terms $(f_i)^{i \in I}$. The purely invertible part of t can thus be represented as a context into which the focused terms $(f_i)^{i \in I}$ are plugged. We will write E_{IN} for such "invertible contexts".

Those invertible contexts have a family of holes $(\Box_i)^{i \in I}$, so they may also be written $E_{\text{IN}}\,[\Box_i]^{i \in I}$. Each hole $\Box_i$ occurs in the typing environment that characterizes the end of the invertible phase: it is of the form $\Gamma^{\mathsf{at}}; {\Gamma^{\mathsf{at}}}'_i \vdash_{\mathsf{inv}} \Box_i : {P^{\mathsf{at}}}'_i \mid P^{\mathsf{at}}$ for some context ${\Gamma^{\mathsf{at}}}'_i$ and optional type ${P^{\mathsf{at}}}'_i$.

Notice that such invertible contexts do not use any variable of the negative context Γ^{at} (none of the invertible term formers use a variable from the context), nor do they depend on the formula P^{at} if it exists: those are only relevant to the focused terms f_i plugged in the holes. We can thus define a new typing judgment for invertible contexts, giving only the necessary information: the types to decompose at the start of the invertible phase, and for each hole the post-decomposition types at the end of the invertible phase. We will use the dense notation $\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}}\,[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}}]^{i \in I} : N$ for this; the typing rules for this judgment are directly derived from the focused system, but we repeated them in Figure 10.7 (Typing rules for focused invertible contexts) for explicitness. This is a multi-hole but linear context: each hole $\Box_i$ should appear exactly once in the term – we use disjoint union $\uplus$ to combine the families of indices that the holes of two sub-contexts range over, rather than the usual non-disjoint set union.

The validity of this judgment is characterized by two easy lemmas, formalizing the decomposition and recomposition of invertible contexts E_{IN} from invertible terms t.

Figure 10.7.: Typing rules for focused invertible contexts

FOC-CTX-HOLE-FOC

$$\overline{\left\langle \Gamma^{\mathsf{at}} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \Box \left[\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box : P^{\mathsf{at}} \right] : \left\langle P^{\mathsf{at}} \right\rangle^{-\mathsf{at}}}$$

FOC-CTX-ABSURD

$$\overline{\Sigma, x : 0 \vdash_{\mathsf{inv}} \mathtt{absurd}(x) : N}$$

FOC-CTX-TRIVIAL

$$\overline{\Sigma \vdash_{\mathsf{inv}} () : 1}$$

FOC-CTX-LAM

$$\frac{\Sigma, x : P \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I} : N}{\Sigma \vdash_{\mathsf{inv}} (\lambda x.\, E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I}) : P \to N}$$

FOC-CTX-PAIR

$$\frac{\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I} : N \qquad \Sigma \vdash_{\mathsf{inv}} F_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in J} : M}{\Sigma \vdash_{\mathsf{inv}} (E_{\text{IN}}, F_{\text{IN}}) \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I \uplus J} : N \times M}$$

FOC-CTX-CASE

$$\frac{\begin{array}{c} \Sigma, x : Q_1 \vdash_{\mathsf{inv}} E_{\text{IN}1} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I_1} : N \\ \Sigma, x : Q_2 \vdash_{\mathsf{inv}} E_{\text{IN}2} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I_2} : N \end{array}}{\Sigma, x : Q_1 + Q_2 \vdash_{\mathsf{inv}} \left(\mathtt{match}\ x\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ x \to E_{\text{IN}1} \\ \sigma_2\ x \to E_{\text{IN}2} \end{array} \right) \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I_1 \uplus I_2} : N}$$

Lemma 10.4.1 (Plugging invertible contexts)**.**
If we have

$$\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I} : N$$

with N possibly empty and, for all $i \in I$, we have

$$\Gamma^{\mathsf{at}}, \Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} f_i : (P_i^{\mathsf{at}} \mid Q^{\mathsf{at}})$$

then plugging the $(f_i)^{i \in I}$ in the invertible context E_{IN} produces a valid invertible derivation

$$\Gamma^{\mathsf{at}} \vdash_{\mathsf{inv}} \Sigma : E_{\text{IN}} \left[f_i \right]^{i \in I} N Q^{\mathsf{at}}$$

Proof sketch. By construction of the invertible context judgment. ⊔

Lemma 10.4.2 (Unique decomposition of invertible terms)**.**
For any invertible term t such that

$$\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$$

there is a unique pair of a context E_{IN} and a family of focused terms $(f_i)^{i \in I}$ such that we have families $(\Gamma_i^{\mathsf{at}}, P_i^{\mathsf{at}})^{i \in I}$ such that

$$t = E_{\text{IN}} \left[f_i \right]^{i \in I}$$

$$\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box_i : P_i^{\mathsf{at}} \right]^{i \in I} : N$$

$$\forall i \in I, \quad \Gamma^{\mathsf{at}}, \Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} f_i : (P_i^{\mathsf{at}} \mid Q^{\mathsf{at}})$$

Proof sketch. The decomposition is immediate by following the invertible rules until the end of the invertible phase. The unicity condition comes from the fact that the hole-typing rule FOC-CTX-HOLE-FOC only accepts shifted contexts and formula $\langle \Sigma^{\mathsf{at}} \rangle^{+\mathsf{at}}, \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}}$, forcing the holes to be placed only at the very end of the invertible phases – which are as long as possible. ⊔

10.4.2. Non-invertible multi-contexts

Similarly, we can decompose focused terms $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} f : P^{\mathsf{at}}$ into a non-invertible context $E_{\mathrm{NI}}\,[\Box_i]^{i\in I}$, containing only non-invertible rules, and a family of invertible terms $(t_i)^{i\in I}$. We decompose this in three judgments, given in Figure 10.8 (Typing rules for focused non-invertible contexts), corresponding to the syntactic categories of focused terms f, positive neutrals p and negative neutrals n:

- $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} E_{\mathrm{NI}}\,[\Sigma_i \vdash_{\mathsf{inv}} \Box_i : N_i \mid Q_i^{\mathsf{at}}]^{i\in I} : P^{\mathsf{at}}$
- $\Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \Box_i : N_i]^{i\in I} \Uparrow P$
- $\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_i : N_i]^{i\in I} \Downarrow N$

Figure 10.8.: Typing rules for focused non-invertible contexts

FOC-CTX-FOC-LEFT
$$\frac{\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Downarrow X^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} N\,[\emptyset \vdash_{\mathsf{inv}} \Box_j : N_j \mid \emptyset]^{j\in J} : X^-}$$

FOC-CTX-FOC-RIGHT
$$\frac{\Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P\,[\emptyset \vdash_{\mathsf{inv}} \Box_j : N_j \mid \emptyset]^{j\in J} : P}$$

FOC-CTX-LET
$$\frac{\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Downarrow \langle P\rangle^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} (\mathtt{let}\ x = N\ \mathtt{in}\ \Box)\,[\emptyset \vdash_{\mathsf{inv}} \Box_j : N_j \mid \emptyset]^{j\in J}\,\left[x : P \vdash_{\mathsf{inv}} \Box : \emptyset \mid Q^{\mathsf{at}}\right] : Q^{\mathsf{at}}}$$

FOC-CTX-APP
$$\frac{\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J_1} \Downarrow P \to M \qquad \Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J_2} \Uparrow P}{\Gamma^{\mathsf{at}} \vdash (N\ P)\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J_1 \uplus J_2} \Downarrow M}$$

FOC-CTX-VAR-NEG
$$\frac{}{\Gamma^{\mathsf{at}}, x : N \vdash x \Downarrow N}$$

FOC-CTX-PROJ
$$\frac{\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \vdash (\pi_i\ N)\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Downarrow N_i}$$

FOC-CTX-INJ
$$\frac{\Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash (\sigma_i\ P)\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Uparrow P_1 + P_2}$$

FOC-CTX-VAR-POS
$$\frac{}{\Gamma^{\mathsf{at}}, x : X^+ \vdash x \Uparrow X^+}$$

FOC-CTX-HOLE-INV
$$\frac{}{\Gamma^{\mathsf{at}} \vdash \Box\,[\vdash_{\mathsf{inv}} \Box : N] \Uparrow \langle N\rangle^-}$$

The holes of the judgments for neutral terms (positive or negative) are not typed by a context: there is no context that grows during the application of the non-invertible rules that would be available to the terms in the holes. On the contrary, the holes of the multi-focusing judgment do have a context that varies: the holes inside the left-focusing phases have an empty context (no additional variable is added in scope), but the hole corresponding to the right-hand side of a *let* binding is in the scope of the formula resulting from the left foci. More precisely, in the rule

FOC-CTX-LET
$$\frac{\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \Box_j : N_j]^{j\in J} \Downarrow \langle P\rangle^-}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} (\mathtt{let}\ x = N\ \mathtt{in}\ \Box)\,[\emptyset \vdash_{\mathsf{inv}} \Box_j : N_j \mid \emptyset]^{j\in J}\,\left[x : P \vdash_{\mathsf{inv}} \Box : \emptyset \mid Q^{\mathsf{at}}\right] : Q^{\mathsf{at}}}$$

The non-invertible context $(\mathtt{let}\ x = N\ \mathtt{in}\ \Box)$ has a family of holes $((\Box_j)^{j\in J}, \Box)$ ranging over the indices $J + 1$: all the holes of N, which do not have any extra variable in scope, plus the right-hand side hole which lives in a context extended with the binding $\{x : P\}$.

Finally, the ambient context Γ^{at} is necessary to type-check variables occurring in neutral terms.

Lemma 10.4.3 (Plugging non-invertible contexts)**.**
If we have

$$\Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \square_i : N_i]^{i\in I} \Uparrow P \qquad \forall i \in I, \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t_i : N_i \mid \emptyset$$

then we have $\Gamma^{\mathsf{at}} \vdash \Sigma \Uparrow P\,[t_i]^{i\in I} P$.
If we have

$$\Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \square_i : N_i]^{i\in I} \Downarrow M \qquad \forall i \in I, \quad \Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{inv}} t_i : N_i \mid \emptyset$$

then we have $\Gamma^{\mathsf{at}} \vdash N\,[t_i]^{i\in I} \Downarrow M$.
If we have

$$\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} E_{\text{NI}}\left[\Sigma_i \vdash_{\mathsf{inv}} \square_i : N_i \mid Q_i^{\mathsf{at}}\right]^{i\in I} : P^{\mathsf{at}} \qquad \forall i \in I, \quad \Gamma^{\mathsf{at}}; \Sigma_i \vdash_{\mathsf{inv}} t_i : N_i \mid Q_i^{\mathsf{at}}$$

then we have $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} E_{\text{NI}}\,[t_i]^{i\in I} : P^{\mathsf{at}}$

Proof sketch. By construction of the non-invertible context judgments. ⊔

Lemma 10.4.4 (Unique decomposition of non-invertible terms)**.**
For any positive neutral p*, negative neutral* n *or, respectively, multi-focused term* f *such that*

$$\Gamma^{\mathsf{at}} \vdash p \Uparrow P \qquad \Gamma^{\mathsf{at}} \vdash n \Downarrow N \qquad \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} f : P^{\mathsf{at}}$$

there is a unique pair of a context P*,* N *or* E_{NI} *respectively and a family of invertible terms* $(t_i)^{i\in I}$ *such that we have* $\Gamma^{\mathsf{at}} \vdash_{\mathsf{inv}} \Sigma_i : t_i N_i Q_i^{\mathsf{at}}$ *for any* $i \in I$ *and*

$$p = P\,[t_i]^{i\in I} \qquad \Gamma^{\mathsf{at}} \vdash P\,[\vdash_{\mathsf{inv}} \square_i : N_i]^{i\in I} \Uparrow P \qquad \Sigma_i = \emptyset \qquad Q_i^{\mathsf{at}} = \emptyset$$

or

$$n = N\,[t_i]^{i\in I} \qquad \Gamma^{\mathsf{at}} \vdash N\,[\vdash_{\mathsf{inv}} \square_i : N_i]^{i\in I} \Downarrow N \qquad \Sigma_i = \emptyset \qquad Q_i^{\mathsf{at}} = \emptyset$$

or

$$f = E_{\text{NI}}\,[t_i]^{i\in I} \qquad \Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} E_{\text{NI}}\,[\vdash_{\mathsf{inv}} \square_i : N_i]^{i\in I} : P^{\mathsf{at}}$$

respectively.

Proof sketch. This is the same principle as invertible decomposition – Lemma 10.4.2. Unicity comes from the fact that the only rule adding new holes to the derivation, FOC-CTX-HOLE-INV (end of positive phase) and FOC-CTX-MULTI-LET-HOLE, only apply when the non-invertible phases are as long as possible. ⊔

10.5. Strong positive phases

Informally, it is interesting to contrast three different ways to prove a formula A in a given context Γ:

- The most general judgment is the unfocused notion of proof $\Gamma \vdash A$; in a focused system, it would correspond to first performing an invertible phase (on the positives of Γ and A if negative), then looking for an arbitrary focused proof.
- The elimination judgment $\Gamma \Downarrow A$ corresponds to a kind of "simple proof", or a single "deduction step"; we see its derivations *direct deductions*. We make progress by taking a variable from the context and using it to deduce a formula A. But this is more restrictive than the general notion of provability $\Gamma \vdash A$; informally, the general case corresponds to being able to consecutively perform many single deduction steps as desired – thanks to the left-focusing rule.

- One can think of the introduction judgment $\Gamma \Uparrow A$ as an even simpler notion of proof, a construction that was already "in" the context. If we think of proving as the discovery of new fact, we could describe $\Gamma \Downarrow A$ as the atomic step of deducing a new fact, while $\Gamma \Uparrow A$ is the even simpler step of retrieving a fact already known by the system.

Consider the left-focusing rule and right-focusing rules in focused natural deduction:

$$\text{NAT-FOC-LEFT-RELEASE-SHIFT}\ \frac{\Gamma^{\mathsf{at}} \Downarrow \langle Q \rangle^{-} \qquad \Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \emptyset \mid P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P} \qquad \text{NAT-FOC-RIGHT}\ \frac{\Gamma^{\mathsf{at}} \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P}$$

One can think of these rules as expressing the idea that, to do a general proof $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} P$, we can perform an arbitrary sequence of direct deductions of the form $\Gamma^{\mathsf{at}} \Downarrow \langle Q \rangle^{+}$ with NAT-FOC-LEFT-RELEASE-SHIFT; eventually the desired goal has been deduced, and we can end by retrieving it from the context with NAT-FOC-RIGHT – or by proving 0 in the invertible phase, or with a left-focusing on a negative atom.

In particular, if we replaced the premise $\Gamma^{\mathsf{at}} \Downarrow \langle Q \rangle^{-}$ of the left-focusing rule by $\Gamma^{\mathsf{at}} \Uparrow Q$, this would radically stifle the expressivity of the logic – we would lose completeness. Instead of performing a new direct deduction and building upon it, this rule would only allow to build on facts already retrievable from the context. In fact, we can prove – Lemma 10.5.1 (Strong positive neutral substitution) – that this weaker rule is derivable without using the left-focusing rule.

In fact, the judgment $\Gamma^{\mathsf{at}} \Uparrow P$ is still a bit too flexible to capture our notion of "context retrieval", as it may end the positive phase and continue with an invertible phase followed by arbitrary proof search:

$$\text{SAT-UP-SINV}\ \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sinv}} t : N \mid \emptyset}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} t \Uparrow \langle N \rangle^{+}}$$

In Figure 10.9 (Strong positive neutral judgment $\Gamma^{\mathsf{at}} \vdash p \Uparrow\!\!\Uparrow P$), we define a restricted $\Gamma^{\mathsf{at}} \vdash p \Uparrow\!\!\Uparrow P$ that is less expressive: shifted negatives have to be found in the context directly, satisfying our intuition of context retrieval. Note that the grammar of those "strong positive neutrals" p is thus slightly different from the usual positive neutrals p, as it may contain variables of negative type – we will reuse the notation p.

Figure 10.9.: Strong positive neutral judgment $\Gamma^{\mathsf{at}} \vdash p \Uparrow\!\!\Uparrow P$

$$\text{SAT-STRONG-UP-NEG}\ \frac{}{\Gamma^{\mathsf{at}}, x : N \vdash x \Uparrow\!\!\Uparrow \langle N \rangle^{-}} \qquad \text{SAT-STRONG-UP-ATOM}\ \frac{}{\Gamma^{\mathsf{at}}, x : X^{+} \vdash x \Uparrow\!\!\Uparrow X^{+}} \qquad \text{SAT-STRONG-UP-INJ}\ \frac{\Gamma^{\mathsf{at}} \vdash p \Uparrow\!\!\Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash \sigma_i\, p \Uparrow\!\!\Uparrow P_1 + P_2}$$

Remark 10.5.1. The two rules SAT-STRONG-UP-NEG and SAT-STRONG-UP-ATOM could be compressed in a single rule

$$\frac{}{\Gamma^{\mathsf{at}}, x : N^{\mathsf{at}} \vdash x \Uparrow\!\!\Uparrow \langle N^{\mathsf{at}} \rangle^{+\mathsf{at}}}$$

be we preferred having two rules for easier comparison with the usual $\Gamma^{\mathsf{at}} \vdash p \Uparrow P$ judgment. ∗

Besides capturing our intuition of "context retrieval", this restricted judgment has interesting provability properties.

Lemma 10.5.1 (Strong positive neutral substitution)**.**
The following rule is admissible

$$\frac{\Gamma^{\mathsf{at}} \Uparrow\!\!\Uparrow P \qquad \Gamma^{\mathsf{at}}; P \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}}\ \text{SUBST-EXPAND}$$

Furthermore, the proof derivation returned by the admissibility procedure is a focused subderivation of the derivation of $\Gamma^{\mathsf{at}}; P \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}$.

Proof. By induction on the proof of $\Gamma^{\mathsf{at}} \Uparrow P$. This is immediate in the two variable cases. In the sum case, we have

$$\frac{\Gamma^{\mathsf{at}} \Uparrow P_i}{\Gamma^{\mathsf{at}} \Uparrow P_1 + P_2} \qquad \frac{\begin{array}{l}\Pi_1 :: \Gamma^{\mathsf{at}}; P_1 \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}} \\ \Pi_2 :: \Gamma^{\mathsf{at}}; P_2 \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; P_1 + P_2 \vdash_{\mathsf{inv}} \emptyset \mid Q^{\mathsf{at}}}$$

and we conclude by induction hypothesis on Π_i. □

The name of the following theorem is a reference to a difference presentation of invertible phase in so-called "higher-order" focused systems, as found in Zeilberger [2009].

Lemma 10.5.2 (Higher-order invertible phase)**.**
If we have

$$\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_j^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_j : Q_i^{\mathsf{at}}\right]^{j \in J} : Q^{\mathsf{at}}$$

then the $\left(\Gamma_j^{\mathsf{at}}\right)^{j \in J}$ *are exactly the contexts such that*

$$\forall P \in \Sigma, \qquad \Gamma_j^{\mathsf{at}} \Uparrow P$$

Proof. By induction on E_{IN}.

Sum case

FOC-CTX-CASE

$$\frac{\begin{array}{c}\Sigma, x : Q_1 \vdash_{\mathsf{inv}} E_{\text{IN}1} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I_1} : N \\ \Sigma, x : Q_2 \vdash_{\mathsf{inv}} E_{\text{IN}2} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I_2} : N\end{array}}{\Sigma, x : Q_1 + Q_2 \vdash_{\mathsf{inv}} \left(\texttt{match}\ x\ \texttt{with} \;\middle|\; \begin{array}{l}\sigma_1\ x \to E_{\text{IN}1} \\ \sigma_2\ x \to E_{\text{IN}2}\end{array}\right) \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I_1 \uplus I_2} : N}$$

Suppose we have Γ_i^{at} with $i \in I_k$ for $k \in \{1, 2\}$. For $P \in \Sigma$ we have $\Gamma_i^{\mathsf{at}} \Uparrow P$ by induction hypothesis; if P is $Q_1 + Q_2$, then we have $\Gamma_i^{\mathsf{at}} \Uparrow Q_k$ by induction hypothesis, and thus

$$\frac{\Gamma_i^{\mathsf{at}} \Uparrow Q_k}{\Gamma_i^{\mathsf{at}} \Uparrow Q_1 + Q_2}$$

Empty or unit cases

FOC-CTX-ABSURD

$$\frac{}{\Sigma, x : 0 \vdash_{\mathsf{inv}} \texttt{absurd}(x) : N}$$

FOC-CTX-TRIVIAL

$$\frac{}{\Sigma \vdash_{\mathsf{inv}} () : 1}$$

We have $I = \emptyset$: there is no Γ_i^{at} so the property is vacuously true.

Function or product cases

FOC-CTX-LAM

$$\frac{\Sigma, x : P \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I} : N}{\Sigma \vdash_{\mathsf{inv}} (\lambda x.\, E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I}) : P \to N}$$

FOC-CTX-PAIR

$$\frac{\Sigma \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I} : N \qquad \Sigma \vdash_{\mathsf{inv}} F_{\text{IN}} \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in J} : M}{\Sigma \vdash_{\mathsf{inv}} (E_{\text{IN}}, F_{\text{IN}}) \left[\Gamma_i^{\mathsf{at}} \vdash_{\mathsf{foc}} \square_i : P_i^{\mathsf{at}}\right]^{i \in I \uplus J} : N \times M}$$

The invertible context is equal or larger in the premises, so the result is immediate by induction hypothesis.

Release case

FOC-CTX-HOLE-FOC

$$\overline{\left\langle \Gamma^{\mathsf{at}} \right\rangle^{+\mathsf{at}} \vdash_{\mathsf{inv}} \Box \left[\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} \Box : P^{\mathsf{at}} \right] : \left\langle P^{\mathsf{at}} \right\rangle^{-\mathsf{at}}}$$

There is exactly one Γ_i^{at}, and it is Γ^{at}, which is exactly the context which proves all $\Gamma^{\mathsf{at}} \Uparrow N^{\mathsf{at}}$ for $N^{\mathsf{at}} \in \Gamma^{\mathsf{at}}$. □

Remark 10.5.2. The fact that $\Gamma^{\mathsf{at}} \Uparrow P$ implies $\Gamma^{\mathsf{at}} \uparrow P$ is true, but not trivial to prove: in the release case, we need to build a focused proof of $\Gamma^{\mathsf{at}}, x : N; ? \vdash_{\mathsf{sinv}} N : \emptyset \mid$. This property is direct in logics with arbitrary axiom rules, but not in focused logics with atomic axioms, where it is known as *axiom expansion*. It is a consequence of the focusing completeness result – Theorem 10.3.1 (Completeness of focusing). ∗

10.6. (Non)-canonicity of focused λ-terms

Is focusing enough to capture the commuting conversions in general? Are focused systems canonical with respect to η-equivalence? The answer to this question is *no*, as soon as we are in a type system that mixes connectives of different polarities. In the more often studied purely-negative fragment (with just functions and products), then focusing does capture satisfying $\beta\eta$-normal forms; same for the purely-positive fragment. However, as we will show by discussing a counter-example, focused proofs are not canonical in our systems with both (negative) functions and (positive) sums.

The erasure operation $\lfloor t \rfloor_{\mathtt{foc}}$ of Section 10.2.1 let us talk about canonicity. A subsystem of focused proofs is canonical with respect to the λ-calculus if, for any t, u in the subsystem, we had $\lfloor t \rfloor_{\mathtt{foc}} \approx_{\beta\eta} \lfloor u \rfloor_{\mathtt{foc}}$ if and only if t and u are the same proof in the subsystem; because we quotient over the ordering of invertible phases ($\approx_{\mathtt{icc}}$), this means $t \approx_{\mathtt{icc}} u$. If, furthermore, we have that for any non-focused term t there is a focused term u in the subsystem such that $t \approx_{\beta\eta} \lfloor u \rfloor_{\mathtt{foc}}$, then the subsystem is computationally complete.

10.6.1. Equivalence of focused λ-terms

Given that focused λ-terms are *not*, in the general case, a canonical representation, it is interesting to study their equivalence classes.

In Section 10.1.5 (Equivalence with the focused sequent calculus) we have shown that, unlike in the non-focused setting, there is a one-to-one correspondence between proofs in the sequent calculus and focused natural deduction.

We may thus consider two natural notions of equivalence are their equivalence as λ-terms (through a defocusing transformation), as sequent-calculus derivations). In fact, those notions coincide.

Lemma 10.6.1.
If t, u are focused λ-terms for the same judgment, then

$$\llbracket t \rrbracket_{\mathsf{focseq}} \approx_{\mathtt{scc}\beta\eta} \llbracket u \rrbracket_{\mathsf{focseq}} \qquad \Longleftrightarrow \qquad \lfloor t \rfloor_{\mathtt{foc}} \approx_{\beta\eta} \lfloor u \rfloor_{\mathtt{foc}}$$

Proof. Going from left to right is a direct consequence of Lemma 10.2.6 (Translation commutation), giving $\lfloor t \rfloor_{\mathtt{foc}} = \left\llbracket \llbracket t \rrbracket_{\mathsf{focseq}} \right\rrbracket_{\mathrm{ND}}$, followed by Lemma 5.3.2 (Soundness of permutation equivalence), which let us deduce $\llbracket t' \rrbracket_{\mathrm{ND}} \approx_{\beta\eta} \llbracket u' \rrbracket_{\mathrm{ND}}$ from $t' \approx_{\mathtt{scc}} u'$.

In the other direction, we have that $\left\llbracket \llbracket t \rrbracket_{\mathsf{focseq}} \right\rrbracket_{\mathrm{ND}} \approx_{\beta\eta} \left\llbracket \llbracket u \rrbracket_{\mathsf{focseq}} \right\rrbracket_{\mathrm{ND}}$, which by Corollary 5.6.9 (Equi-equivalence of sequent terms and λ-terms) implies that $\llbracket t \rrbracket_{\mathsf{focseq}} \approx_{\mathtt{scc}\beta\eta} \llbracket u \rrbracket_{\mathsf{focseq}}$. □

10.6.2. Focused terms are β-short normal forms

As a first step towards understanding the (non)-canonicity of focused λ-terms, we can easily prove that focused λ-terms are normal forms for reduction/computation relations

– either when seen as their corresponding sequent terms, or erased to usual λ-terms. Remember that (R) is the reduction relation for sequent terms defined in Section 4.2.1 (Normal sequent proofs: cut-elimination).

Fact 10.6.2 (Focused λ-terms are R-normal forms)**.**
If t is a well-typed focused λ-term, then its sequent-equivalent $[\![t]\!]_{\mathsf{focseq}}$ (defined in Figure 10.5) is cut-free, a R*-normal form.*

Theorem 10.6.3 (Focused λ-terms are β-normal forms)**.**
If t is a well-typed focused λ-term, then $\lfloor t \rfloor_{\mathtt{foc}}$ is a β-normal form.

Proof. For each type connective, we consider each elimination form in a term of the foc $\lfloor t \rfloor_{\mathtt{foc}}$, and prove that its eliminated subterm is not a constructor of the corresponding type.

In the function case, consider the subterms of $\lfloor t \rfloor_{\mathtt{foc}}$ that are applications. By Lemma 10.2.2 (Composability of defocusing), we know that any such terms are the translation of a subterm of t, to which a series of substitutions is applied. The only subterms of t that may translate to applications are the neutral subterms $n\ p$. Their translation is of the form $(\lfloor n \rfloor_{\mathtt{foc}}\ \lfloor p \rfloor_{\mathtt{foc}})[\rho]$, that is $(\lfloor n \rfloor_{\mathtt{foc}}[\rho])\ (\lfloor p \rfloor_{\mathtt{foc}}[\rho])$ where ρ is a sequence of substitutions of the form $[\lfloor m_i \rfloor_{\mathtt{foc}}/x_i]$, for bindings $\mathtt{let}\ x_i = m_i$ appearing in t. This could only be a β-redex if $\lfloor n \rfloor_{\mathtt{foc}}[\rho]$ was of the form $\lambda x.\ \_$, which is impossible as:

- if n is not a variable, it starts (and its substitution starts) with an elimination form for a negative type, not a λ-abstraction
- if n is a variable x, it cannot be transformed into a λ-abstraction by substitution: by the focusing discipline, the substituted $\lfloor n_i \rfloor_{\mathtt{foc}}$ come from a neutral subterm n_i of strictly positive type, so it cannot start with a λ-abstraction.

The exact same reasoning applies to the product case: in the translation of a $\pi_i\ n$, n cannot be a pair construction $(\_, \_)$.

For sums, the elimination forms in $\lfloor t \rfloor_{\mathtt{foc}}$ come from the translation of an invertible step $\mathtt{match}\ x\ \mathtt{with} \mid \begin{array}{l}\sigma_1\ x \to u_i \\ \sigma_2\ x \to u_i\end{array}$. Again, x cannot be substituted into an introduction form $\sigma_i\ \_$, as negative neutrals only ever start with elimination forms. □

Note in particular that performing invertible commuting conversions on a focused term preserves the focusing discipline, and thus the fact that its defocused form is β-normal. This is not true of arbitrary λ-terms, where extruding a sum elimination may reveal new β-redexes.

10.6.3. Focused terms are weak η-long forms

When we say that a λ-term is in β-short η-long normal form, we mean that it is in β-normal form, but not that it is in η-normal form. Indeed, η-expansion can be performed indefinitely, it is not a terminating reduction: if y is of type $X \to Y$, we have

$$y \quad \triangleright_\eta \quad \lambda x_1.\, y\ x_1 \quad \triangleright_\eta \quad \lambda x_2.\, (\lambda x_1.\, y\ x_1)\ x_2 \quad \triangleright_\eta \quad \dots$$

The term η-long refers to the fact that additional η-expansions are possible, but that they are in a sense "useless": performing an additional η-expansion results in a term that is β-equivalent to the previous one. This is what happens in our example above: the first expansion from y to $\lambda x_1.\, y\ x_1$ is useful, in the sense that the terms are not β-equivalent. The second expansion is useless, as $\lambda x_2.\, (\lambda x_1.\, y\ x_1)\ x_2$ reduces to $\lambda x_2.\, y\ x_2$, which is α-equivalent to the term $\lambda x_1.\, y\ x_1$ we started from.

Definition 10.6.1 Head weak η-long form**.**
A λ-term t is in *head* weak *η-long form* if we have

$$\forall u, \qquad t \triangleright_{\mathtt{weak}\,\eta} u \qquad \Longrightarrow \qquad u \to_\beta t$$

However, in presence of sums, this notion of $\mathtt{weak}\,\eta$-long or $\mathtt{weak}\,\eta$-long normal form is not sufficient. Indeed, $\mathtt{weak}\,\eta$-expansion on sums generates infinite sequences whose elements are not β-reducible to each other: if y is of type $X + Y$, we have

$$y \quad \triangleright_{\mathtt{weak}\,\eta} \quad \mathtt{match}\ y\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_1 \to \sigma_1\ x_1 \\ \sigma_2\ x_1 \to \sigma_2\ x_1 \end{array} \quad \triangleright_{\mathtt{weak}\,\eta}$$

$$\mathtt{match} \left(\mathtt{match}\ y\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_1 \to \sigma_1\ x_1 \\ \sigma_2\ x_1 \to \sigma_2\ x_1 \end{array} \right) \mathtt{with} \begin{array}{|l} \sigma_1\ x_2 \to \sigma_1\ x_2 \\ \sigma_2\ x_2 \to \sigma_2\ x_2 \end{array} \quad \triangleright_{\mathtt{weak}\,\eta} \quad \dots$$

While the third term does not β-reduce to the second one, note that extruding the inner matching would give a term that does:

$$\mathtt{match} \left(\mathtt{match}\ y\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_1 \to \sigma_1\ x_1 \\ \sigma_2\ x_1 \to \sigma_2\ x_1 \end{array} \right) \mathtt{with} \begin{array}{|l} \sigma_1\ x_2 \to \sigma_1\ x_2 \\ \sigma_2\ x_2 \to \sigma_2\ x_2 \end{array}$$

$$\approx_{\mathtt{extr}} \quad \mathtt{match}\ y\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_1 \to \mathtt{match}\ \sigma_1\ x_1\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_2 \to \sigma_1\ x_2 \\ \sigma_2\ x_2 \to \sigma_2\ x_2 \end{array} \\ \sigma_2\ x_1 \to \mathtt{match}\ \sigma_2\ x_1\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_2 \to \sigma_1\ x_2 \\ \sigma_2\ x_2 \to \sigma_2\ x_2 \end{array} \end{array}$$

$$\to_\beta \quad \mathtt{match}\ y\ \mathtt{with} \begin{array}{|l} \sigma_1\ x_1 \to \sigma_1\ x_1 \\ \sigma_2\ x_1 \to \sigma_2\ x_1 \end{array}$$

Focused λ-terms are $\mathtt{weak}\,\eta$-long in this sense: if we perform weak η-expansions on them, we obtain terms that are β-equivalent, modulo invertible commuting conversion.

This suggests the notion of $\mathtt{weak}\,\eta$-long normal form modulo extrusions.

Definition 10.6.2 Head $\mathtt{weak}\,\eta$-long form modulo extrusion**.**
A λ-term t is in *head* $\mathtt{weak}\,\eta$*-long form modulo extrusion* if we have

$$\forall u, \quad t \triangleright_{\mathtt{weak}\,\eta} u \quad \Longrightarrow \quad \exists r, \quad u \approx_{\mathtt{extr}} r \to_\beta t$$

Definition 10.6.3 $\mathtt{weak}\,\eta$-long form modulo extrusion**.**
A term t is in $\mathtt{weak}\,\eta$-long form modulo extrusion if any of its subterms is in head $\mathtt{weak}\,\eta$-long form modulo extrusion, or equivalently if

$$\forall u, \quad t \to_{\mathtt{weak}\,\eta} u \quad \Longrightarrow \quad \exists r, \quad u \approx_{\mathtt{extr}} r \to_\beta t$$

Theorem 10.6.4 (Focused λ-terms are $\mathtt{weak}\,\eta$-long forms modulo extrusion)**.**
If t is a well-typed focused λ-term, then it is in $\mathtt{weak}\,\eta$-long form modulo extrusion.

Proof. We consider each connective in turn.

In the function case, we consider a subterm $u : A \to B$ of t and the expansion $u \triangleright_\eta \lambda x.\, u\ x$. If u is proved by the invertible judgment, then it is equal, modulo a commuting conversion, to a term of the form $\lambda y.\, u'$, and $\lambda x.\, \lambda y.\, u'\ x$ is β-reducible. If u is proved by the non-invertible judgment, then it is inside an elimination form for functions $\square\ p$, and $(\lambda x.\, u\ x)\ p$ is β-reducible.

The product case is similar – simpler.

In the sum case, we consider a subterm $u : A_1 + A_2$ of t and the expansion

$$u \triangleright_{\mathtt{weak}\,\eta} \mathtt{match}\ u\ \mathtt{with} \begin{array}{|l} \sigma_1\ x \to \sigma_1\ x \\ \sigma_2\ x \to \sigma_2\ x \end{array}$$

a focused term of sum type can only appear as a variable y in the invertible judgment or

as a constructor $\sigma_i\ u'$ in the non-invertible judgment. In the first case, we have

$$\begin{array}{rl}
 & \texttt{match}\ y\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y \to r_1 \\ \sigma_2\ y \to r_2\end{array} \\
\to_\eta & \texttt{match}\ \left(\texttt{match}\ y\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x \to \sigma_1\ x \\ \sigma_2\ x \to \sigma_2\ x\end{array}\right)\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y \to r_1 \\ \sigma_2\ y \to r_2\end{array} \\
\approx_{\texttt{extr}} & \texttt{match}\ y\ \texttt{with} \mid \begin{array}{ll}\sigma_1\ x \to & \texttt{match}\ \sigma_1\ x\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y \to r_1 \\ \sigma_2\ y \to r_2\end{array} \\ \sigma_2\ x \to & \texttt{match}\ \sigma_2\ x\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y \to r_1 \\ \sigma_2\ y \to r_2\end{array}\end{array} \\
\to_\beta^* & \texttt{match}\ y\ \texttt{with} \mid \begin{array}{l}\sigma_1\ y \to r_1 \\ \sigma_2\ y \to r_2\end{array}
\end{array}$$

and in the second case, we simply have

$$\sigma_i\ u' \quad \triangleright_\eta \quad \texttt{match}\ \sigma_i\ u'\ \texttt{with} \mid \begin{array}{l}\sigma_1\ x \to \sigma_1\ x \\ \sigma_2\ x \to \sigma_2\ x\end{array} \quad \triangleright_\beta \quad \sigma_i\ u'$$

□

10.6.4. Non-canonicity of the full focused system

Consider the following judgment:

$$x : 1 \to (X + X) \vdash_{\mathsf{inv}} ? : 0 + (X \times X)$$

There are two distinct focused λ-terms for this judgment. The first is

$$t_1 \stackrel{\mathsf{def}}{=} \texttt{let}\ y = x\ ()\ \texttt{in}\ \texttt{match}\ y\ \texttt{with} \mid \begin{array}{l}\sigma_1\ z \to \sigma_2\ (z, z) \\ \sigma_2\ z \to \sigma_2\ (z, z)\end{array}$$

and the second is

$$t_2 \stackrel{\mathsf{def}}{=} \sigma_2 \left(\begin{array}{l} \texttt{let}\ y_1 = x\ ()\ \texttt{in}\ \texttt{match}\ y_1\ \texttt{with} \mid \begin{array}{l}\sigma_1\ z \to z \\ \sigma_2\ z \to z\end{array} \\ , \\ \texttt{let}\ y_2 = x\ ()\ \texttt{in}\ \texttt{match}\ y_2\ \texttt{with} \mid \begin{array}{l}\sigma_1\ z \to z \\ \sigma_2\ z \to z\end{array} \end{array} \right)$$

These two terms t_1, t_2 are $\beta\eta$-equivalent. To see it, let us define the (non-focused) context

$$E\,[\square] \stackrel{\mathsf{def}}{=} \sigma_2 \left(\begin{array}{l} \texttt{match}\ \square\ \texttt{with} \mid \begin{array}{l}\sigma_1\ z \to z \\ \sigma_2\ z \to z\end{array} \\ , \\ \texttt{match}\ \square\ \texttt{with} \mid \begin{array}{l}\sigma_1\ z \to z \\ \sigma_2\ z \to z\end{array} \end{array} \right)$$

Then we have

$$
\begin{array}{lll}
\lfloor t_1 \rfloor_{\mathtt{foc}} & = & E\,[x\ ()] \\
& \triangleright_\eta & \left(\begin{array}{ll} \mathtt{match}\ x\ ()\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ y \to E\,[\sigma_1\ y] \\ \sigma_2\ y \to E\,[\sigma_2\ y] \end{array}\right. \end{array}\right) \\
& = & \left(\begin{array}{ll} \mathtt{match}\ x\ ()\ \mathtt{with} & \left|\begin{array}{ll} \sigma_1\ y \to \sigma_2 & \left(\begin{array}{ll} \mathtt{match}\ \sigma_1\ y\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ z \to z \\ \sigma_2\ z \to z \end{array}\right. \\ , & \\ \mathtt{match}\ \sigma_1\ y\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ z \to z \\ \sigma_2\ z \to z \end{array}\right. \end{array}\right) \\ \sigma_2\ y \to \sigma_2 & \left(\begin{array}{ll} \mathtt{match}\ \sigma_2\ y\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ z \to z \\ \sigma_2\ z \to z \end{array}\right. \\ , & \\ \mathtt{match}\ \sigma_2\ y\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ z \to z \\ \sigma_2\ z \to z \end{array}\right. \end{array}\right) \end{array}\right. \end{array}\right) \\
& \triangleright_\beta^* & \left(\begin{array}{ll} \mathtt{match}\ x\ ()\ \mathtt{with} & \left|\begin{array}{l} \sigma_1\ y \to \sigma_2\ (y, y) \\ \sigma_2\ y \to \sigma_2\ (y, y) \end{array}\right. \end{array}\right) \\
& = & \lfloor t_2 \rfloor_{\mathtt{foc}}
\end{array}
$$

We can see in this example that non-canonicity is caused by a redundant choice of ordering of the non-invertible phase. In t_1, we decided to perform the right-focused phase first $\sigma_2\ \_$, and the left-focused phases $\mathtt{let}\ y = x\ ()\ \mathtt{in}\ \_$ later. In t_2, we decided to perform a left-focusing phase first, and the right-focused phases later. This choice introduces a redundancy because both options are $\beta\eta$-equivalent: informally, those non-invertible phases are *independent*, they do not depend on each other and could have one before the other, or even simultaneously.

To get a more canonical calculus, we will introduce in Chapter 11 (Saturation logic for canonicity) an extension of focusing that focuses on several independent non-invertible phases in parallel, which removes this source of redundancy.

11. Saturation logic for canonicity

Reference *The work that resulted in this chapter has been previously presented in the article Scherer and Rémy [2015], which gives a more compact presentation of the main result. In this chapter, we have improved the presentations in the following ways:*

- *We use an explicitly focused syntax for types/formulas, which gives a system that is hopefully closer to what focusing experts expect. Note that non-experts need not be frightened by the notational overhead: it is possible to ignore the shifts and polarities and the content should still make sense, at a simpler level of reading. In fact, we may ourselves use the non-polarized syntax in some examples, assuming minimal shift insertion.*
- *We explain the construction of the saturation rule, giving examples for each of the potential difficulties, in more details than a conference article allows.*

A notable non-improvement is that this chapter does not handle the empty type 0*. While the saturation algorithm presented here seems to perfectly handle it, we spent considerable effort trying to adapt the proofs to this setting and failed.*

Equipped with the understanding of program equivalence acquired through our study of focusing (Chapter 10), it is now time to go back to our original question: which types have a unique inhabitant? In this chapter, we provide a decision algorithm for this question in the context of simply-typed lambda-calculus with products and unit types, sums and empty types.

With the technical ideas that we have built throughout this document, the idea can be described in a concise way: we define a variant of focusing for intuitionistic natural deduction that is canonical and has a structural presentation which makes goal-directed proof search possible in this subsystem. The key idea is to use *saturation* in non-invertible phases, that is a complete forward search for left focused phases, until reaching a saturated state (all deducible strict positives have been deduced), then doing right focus and continuing with goal-directed (backward) search. We also need a precise notion of saturation to ensure both completeness and termination.

However, we also wish this chapter to be accessible to readers jumping to it in isolation, with little background on ((maximal) multi-)focusing and more programming-driven intuitions. We will thus start in Section 11.1 (Introduction to saturation for unique inhabitation) by a motivation with programming examples and an informal introduction of the saturation process.

In Section 11.2 (A saturating focused type system), we will present the typing rule of our *saturated* focused type system, which can be understood as a variant of multi-focused λ-calculus. This system serves as a declarative *specification* of saturation, but it does not suffice to obtain an algorithm as its goal-directed proof search process is not always terminating. It Chapter 12 (From the logic to the algorithm: deciding unicity) we introduce an algorithmic restriction of the system in which proof search is terminating and gives a deduction procedure for unicity – and prove its correctness.

Re-introduction to canonical and complete type systems

Our approach to decide unique inhabitation is to design a generic term enumeration procedure that only enumerates distinct terms (no duplicates) and enumerate distinct terms lazily. Given such a procedure, it suffices to enumerate at most two term to decide unicity.

How can we enumerate all distinct values of type $(A_1 \times A_2)$? Well, we know from the η-equivalence of products $(t : A_1 \times A_2) = (\pi_1\ t, \pi_2\ t)$ that any term of type $A_1 \times A_2$ is equivalent to some pair (t, u) of some $t : A_1$ and $u : B_2$, and it thus suffices to enumerate all distinct values of A_1, of A_2, and take their (lazily enumerated) cartesian product. Similarly, to enumerate all distinct values of type $(A \to B)$, it suffices to enumerate B in an environment extended with a formal variable $x : A$, and return $\lambda x.\, t$ for each distinct t in B.

A similar reasoning cannot be applied to enumerate the terms of some atomic type X, on which no constructor form is known; we then need to look at all the ways to produce a X by combining variables from the current environment (and those are the only way to obtain a X). In the purely negative fragment of the λ-calculus (no sums or empty type), this corresponds to the (negative) neutrals n, defined as series of pair projections or function applications applied to a head variable of the context.

By now, the reader familiar with focusing, and in particular the focused λ-calculus as described in Section 10.2 (A focused term syntax: focused λ-calculus), may have recognized that this is an informal description of a process enumerating the values in the focused lambda-calculus, restricted to negative types. We decide to generalize this idea to any type system (for example in presence of sums and empty types, where simple focusing is not enough). Enumerating all distinct values at a type should be formulated as a *proof search* problem, of enumerating all the proofs accepted by a well-chosen proof system, or equivalently all the programs accepted by a well-chosen type system, classifying some strong notion of "normal form". Instead of an enumeration procedure that may be defined in arbitrary ways, we are restricting the scope to the search processes following a specific structure, namely goal-directed search in a type system defined as a set of inference rules.

Given a type system $\Gamma \vdash t : A$, and a notion of program equivalence $\Gamma \vdash t \approx u : A$, we are looking for a type sub-system for "normal forms" $\Gamma \vdash_{\mathsf{nf}} v : A$ that is:

- canonical: If $\Gamma \vdash_{\mathsf{nf}} v : A$ and $\Gamma \vdash_{\mathsf{nf}} w : B$, then v and w are syntactically equal ($v =_\alpha w$ if and only if they are semantically equivalent ($\Gamma \vdash v \approx w : A$). This guarantees that proof search does not enumerate duplicates.

- computationally complete: If a program t is well-typed in the original type system ($\Gamma \vdash t : A$), then there exists an equivalent normal-form v ($\Gamma \vdash_{\mathsf{nf}} v : A$ and $\Gamma \vdash t \approx v : A$). This guarantees that we do not count strictly less distinct terms, in our enumerations, than there are programs at this type in the original system. In the limit case, a sub-system of normal forms that would reject all programs as invalid normal forms would be canonical, but quite incomplete.

Furthermore, we also need goal-directed search $\Gamma \vdash_{\mathsf{nf}} ? : A$ to be feasible in practice. We do not have a formal definition of this last criterion, but we can make two remarks about it.

First, sub-systems defined by refinement (all the proofs of the original system that satisfy condition P) can have no natural goal-directed search process, if the condition P is highly non-local and thus prevents working with partial proofs with missing leaves. Reformulating such a sub-system into a structural presentation with no validity condition – that is, doing the work of expressing the validity condition as local invariants that can be encoded structurally in the various judgments and their transitions – makes it easier to define a goal-directed search process.

Second, even purely structural system may have no good search implementation, if finding a valid proof may not terminate. (A proof enumeration process may not terminate because it enumerates infinitely many distinct proofs, but we need it to be productive in the sense that the next proof is always found after a finite number of search steps.) Having the subformula property can help build a termination argument, but is neither a necessary nor sufficient condition for termination.

The subformula property (see Section 6.2.1 (The subformula property)) guarantees that the formulas appearing in a cut-free proof are all subformulas present in the judgment we are trying to prove; in particular, there is a finite number of possible formulas. Thus, in a logic where contexts are *sets* of formulas, there is only a finite number of possible contexts in a proof, hence a finite number of judgments. In a type system where contexts are mappings from program variables to types, that is *multi-sets* of formulas, the finiteness argument goes away: we could have arbitrarily many distinct formal variables at the same type. Using the results of Chapter 9 (Counting terms and proofs) we show that, in order to decide unicity, it suffices to consider contexts with at most two variables of each type.

In the purely negative fragment of λ-calculus $\mathsf{\Lambda C}(\to, \times, 1)$, focused values $\Gamma \vdash_{\mathsf{inv}} t : A$ form a canonical, computationally complete sub-system. It also has productive goal-directed proof search, but proving this requires some work. Instead of doing our termination proof for the purely negative fragment now, and extending to support sums and empty types later, we directly work on the full type system; but our full system has the nice property that, when used on formulas that are in fact in the purely negative fragment, then it degrades to what is easily recognized as simply focused proof search. The termination arguments are given in Chapter 12 (From the logic to the algorithm: deciding unicity).

11.1. Introduction to saturation for unique inhabitation

The rules of program equivalence for the full, pure simply-typed λ-calculus $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$ are given in Figure 3.4 (Typed program equivalence for $\mathsf{\Lambda C}(\to, \times, 1, +, 0)$). Of particular interest is the distinction between the *weak eta*-rule ($\approx_{\mathtt{weak}\,\eta}$) and the *strong* η-rule ($\approx_\eta$) for sums

$$(t : A_1 + A_2) \;\triangleright_{\mathtt{weak}\,\eta}\; \mathtt{match}\; t\; \mathtt{with} \;\Big|\; \begin{array}{l} \sigma_1\; y_1 \to \sigma_1\; y_1 \\ \sigma_2\; y_2 \to \sigma_2\; y_2 \end{array}$$

$$\forall C\,[x], \quad C\,[t : A_1 + A_2] \;\triangleright_\eta\; \mathtt{match}\; t\; \mathtt{with} \;\Big|\; \begin{array}{l} \sigma_1\; y_1 \to C\,[\sigma_1\; y_1] \\ \sigma_2\; y_2 \to C\,[\sigma_2\; y_2] \end{array}$$

Another source of difficulty, which we discuss in Section 11.1.5 (Saturation and the empty type), is the equivalence rule for the empty type (everything is equivalent under an inconsistent context):

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

11.1.1. Non-canonicity of simple focusing: splitting points

Simple focusing, as described in Chapter 10 (Focused λ-calculus), classifies terms that are also called β-short η-long normal forms, but they in fact correspond to *weak* β-short $\mathtt{weak}\,\eta$-long normal forms. In the purely negative fragment (no sums and empty types), the weak and strong η-rules coincide, so focusing captures the right notion of normal form and is a canonical system.

Focusing fails to be canonical when positives are added; consider for example the following goal:

$$f : 1 \to X^+ + X^+, g : X^+ \to Y^- \vdash ? : 0 + (Y^- \times Y^-)$$

The three following programs are equivalent, yet are syntactically distinct valid focused terms (normal forms). Note that there are other possible ways to write a well-typed β-short $\mathtt{weak}\,\eta$-long normal form at this type – but they are all equivalent.

$$\sigma_1 \left(\begin{array}{l} \texttt{let } y = f\ () \texttt{ in match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\ z_1 \to g\ z_1 \\ \sigma_2\ z_2 \to g\ z_2 \end{array} \right. \\ , \\ \texttt{let } y = f\ () \texttt{ in match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\ z_1 \to g\ z_1 \\ \sigma_2\ z_2 \to g\ z_2 \end{array} \right. \end{array} \right)$$

$$\texttt{let } y = f\ () \texttt{ in match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\ z_1 \to \sigma_1\ (g\ z_1, g\ z_1) \\ \sigma_2\ z_2 \to \sigma_1\ (g\ z_2, g\ z_2) \end{array} \right.$$

$$\texttt{let } y = f\ () \texttt{ in match } y \texttt{ with } \left| \begin{array}{l} \sigma_1\ z_1 \to \sigma_1\ (g\ z_1, g\ z_1) \\ \sigma_2\ z_2 \to \texttt{let } y' = f\ () \texttt{ in } \left(\texttt{match } y' \texttt{ with } \left| \begin{array}{l} \sigma_1\ z' \to (g\ z_2, g\ z'_1) \\ \sigma_2\ z' \to (g\ z_2, g\ z'_2) \end{array} \right. \right) \end{array} \right.$$

We can prove that these three terms are $\beta\eta$-equivalent, but an informal explanation also helps following these examples.

The first two terms perform the same splitting (binding then pattern-matching) of $f\ ()$, but one does it once before building the pair, and the other does it separately in each branch of the pair. Because we assume that f is a pure function[1], it must returns the same thing in each element of the pair, and the final results are thus identical. To prove that the two terms are identical, it suffices to extrude the binding $\texttt{let } y = f\ () \texttt{ in } ?$ from the pair elements in the second case, and extrude the pattern-matching as well. (In terms of focusing, we are suggesting to permute two independent non-invertible phases, the let binding and the sum injection; pair construction and variable case-split are implicitly moved around as well, being the invertible phases that systematically follow each non-invertible phase.)

The third term is slightly different, as instead of performing two splits in two parallel branches (as the first term), it performs two splits in sequence, with the second split being in scope of the (right branch of) the first split. The reasoning to informally justify the equivalence with the second term is that, at the time when y' (that is $f\ ()$) is matched over, we *already know* the value of $f\ ()$: if this branch has been taken, it is because $f\ ()$ is equal to $\sigma_2\ z_2$ for some z_2 that is currently in scope. We can thus replace y' with $\sigma_2\ z_2$ in the nested pattern-matching. Performing a β-reduction step then gives exactly the second term.

Remark 11.1.1. Note that this example of non-canonicity of the focusing discipline would break if we replaced the context hypothesis $f : 1 \to X^+ + X^+$ by a mere sum $x_0 : X^+ + X^+$. Indeed, the focusing discipline would recognize it as a positive in context, to be split before the start of the first non-invertible phase, and this would give a unique focused derivation.

By wrapping this positive under a (negative) function type, we make it out of reach from the simple focusing discipline[2]. In a system with explicit shifts (here we assumed minimal shifts), see Section 7.3.1 (Explicit shifts), we could also simply put the sum under a double-shift delay. $*$

[1] Note that non-termination plus lazy pairs would already allow to observe a difference between those two terms.

[2] Another, more positive way of understanding this is that simple focusing is agnostic of the effectfulness of the calculus. It does not allow to perform reordering which may be invalid in presence of effects. This is only partly convincing, however, given that the idea of always performing invertible steps first may change the observational behavior of effectful terms under weak evaluation strategies. Typically, η-expanding t into $\lambda x.\, t\ x$ may delay (and duplicate) side-effects from definition-site to call-site. To discuss effects and purity, we recommend looking at program terms directly, using System L or CPBV for example.

11.1.2. Canonicity for term equivalence: extrusion

These examples allow to understand where non-canonicity comes from. We have (focused) terms that are syntactically distinct but semantically equivalent. They differ by the place, and the number of times, on which a particular subterm (here $g\ ()$) of sum type is bound and matched over. We need to quotient over this source of difference, by imposing a unique place at which those subterms should be bound and matched, that can be decided during goal-directed proof search.

Definition 11.1.1 Splitting**.**
Splitting a (sub)term is pattern-matching over it, possibly after having bound it to a variable name. We call *splitting point* the place where the term is bound and pattern-matched.

In the work on deciding equivalence of λ-terms with sums, the solution is to move each subterm of sum type as high/early as possible in the term, to split them there – and merge equal subterms that end up being split at the same place. This is clearly visible in the rewriting-based work of Ghani [1995b] and Lindley [2007], but it is also perceptible in the normalization-by-evaluation work [Balat, Di Cosmo, and Fiore, 2004, Altenkirch, Dybjer, Hofmann, and Scott, 2001]. For example, Balat, Di Cosmo, and Fiore [2004] define a notion of quasi-normal form for terms with sums, with a side-condition (Condition (B), page 5) says that a split term must become ill-typed if we move it before the latest series of variable bindings (in fact, the latest invertible phase). This is a way to guarantee that subterms of sum type are split as early as possible in the term.

We have shown in Scherer [2015a] that this "as early as possible" split criterion can be logically justified as maximal multi-focusing, and that the normalization procedures are turning an arbitrary term into its canonical maximally multi-focused equivalent. Those procedures proceed by moving subterms (invertible and non-invertible phases) around, so in particular they rely on the presence of one initial term to normalize, or two initial terms to compare: it makes sense to search, for example, for all neutral subterms n of the initial term that are valid at some possible splitting point (the start of a non-invertible phase) and extrude them.

11.1.3. Canonicity for term enumeration: saturation

On the contrary, the problem of unique inhabitation requires enumerating proof terms out of the blue, without starting from a pre-existing proof term to transform. When reaching a potential splitting point (the start of a non-invertible phase) during term enumeration (goal-directed proof search), there are no subterms to collect and extrude, only recursive sub-goals that have not yet be filled. This crucial difference leads us to taking a quite different (yet related) approach.

Another way to see the situation of term normalization or equivalence is that the initial term serves as an oracle to answer the following question: "which terms should we split now, that will be *useful* to the rest of the proof?". Useful sub-terms are those that it is necessary to bind now to build a term equivalent (computationally) to the initial term. We can also see them as an over-approximation of a set of terms that we must split to find a proof at all; we use the initial term as a base of "hints" (its subterms) to find a proof of the desired judgment – a proof with the particular property of being equivalent to the initial term we started from.

To move from term normalization or comparison to term enumeration, our idea is to drop the usefulness criterion. We cannot know in advance, at this stage of the proof search, without having searched for the sub-goals, which terms of positive types will actually be used by the proof(s) that we will find, but we can split *all of them*. Then we start again enumerating terms of the desired type, in a context extended with (the decomposition of) all those freshly split sums. Some splits will prove useful to build all terms of our enumeration, some will only be used by some of those distinct terms, and some will not

be used at all. This is the idea of *saturation*.

What exactly do we mean by "all terms of positive types"? It is easy to see that, even in the empty context, we can build infinitely many terms of sum types: $\sigma_1\ ()$, $\sigma_2\ ()$, $\sigma_1\ (\sigma_1\ ())$, etc. But pattern-matching on those would be silly, as we already know their value. We are only interested in the terms of positive type whose value is unknown, because they come from the (unknown) formal variables in the typing context of the search. Those are the neutral terms n, m that are obtained by taking a variable x of the context, and applying pair projections $\pi_i\ n$ or function applications $n\ p$ on it until we reach a result of sum type. One can think of a neutral term $n : A_1 + A_2$ as a specific "observation" of the richly-typed value of its head variable x; saturation, which splits all those neutral terms, is the process of learning everything we can learn from our context by these observations, before continuing the proof search.

Saturation should come before any committing choice. If we delay these observations, and first perform a non-invertible introduction step, we can get in a dead search branch, because we do not have enough information at hand to know which choice to make (consider again the proofs of $f : () \to X + Y \vdash ? : Y + X$). This justifies performing saturation "as early as possible", or at least before making any mistake, that is before the start of each non-invertible (right) introduction phase.

In the rest of this chapter, we will see

- A structural presentation of a focused *saturating* type system, which encapsulates this idea of saturation as a typing rule.
- A simple mechanism to avoid splitting the same neutral of positive type during two successive saturation phases, to preserve canonicity; Section 11.2 (A saturating focused type system).
- Various methods to avoid saturating on infinitely many distinct neutrals, or repeating saturation infinitely long before reaching a stable state, to preserve termination; Chapter 12 (From the logic to the algorithm: deciding unicity).

11.1.4. An example of saturation

Let us consider our previous example showing that focusing alone is not canonical:

$$f : 1 \to X^+ + X^+, g : X^+ \to Y^- \vdash ? : 0 + (Y^- \times Y^-)$$

The context Γ^{at} is negative or atomic, and the goal is positive. In our focused logic, we would start by looking for all n of positive type such that $\Gamma^{\mathsf{at}} \vdash n \Downarrow P$. There is exactly one such $(n : P)$ in this context, it is $(f\ () : X^+ + X^+)$. Saturation would thus start with the following phase:

$$\texttt{let}\ x = f\ ()\ \texttt{in}\ ?$$

and the following invertible phase would be

$$\texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to ? \\ \sigma_2\ x \to ? \end{array}$$

leaving us with two subgoals, each with a context of the form

$$f : 1 \to X^+ + X^+, g : X^+ \to Y^-, x : X^+$$

As the two goals are identical, we will focus here on one of them, the other proceeds in the exact same way.

At this point, a new focusing phase begins, looking for all negative neutrals with a positive type. But the addition of a X^+ in the context did not give us any way to deduce

a new neutral: we can still build $f\ ()$, but we have already saturated over it. At this point, saturation stops, and our algorithm tries all possible (non-invertible) rules to prove our goal.

In our case the goal is a strict positive (rather than a negative atom) so we look for all possible positive neutrals p at this type. Proof search will thus attempt to use a term of the form $\sigma_1\ ?$, and prove the remaining goal 0, and also to use a term of the form $\sigma_2\ ?$ and prove the remaining goal $\langle Y^- \times Y^- \rangle^+$. In the first case 0, search fails immediately as there is no strictly positive neutral of this type. In the second case, $\langle Y^- \times Y^- \rangle^+$, the focused introduction phase stops at the shift, and a new invertible phase starts.

The invertible phase for $Y^- \times Y^-$ creates two identical goals, so we can focus on any of them, trying to prove Y^-. A new saturation phase start, but there is still no new negative neutral of positive type in sight. The search algorithm then tries to prove the goal, and because we have a negative atom it looks for a negative neutral at this type. All negative neutrals of type Y^- in this context are of the form $g\ ?$, with a subgoal of type X^+, to be filled by a positive neutral; there is exactly one positive neutral at this type, namely x; because there is only one choice, we know that this goal has a unique inhabitant.

This leaves us with a unique program of this type, namely

$$\texttt{let}\ x = f\ ()\ \texttt{in}\ \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ x \to (g\ x, g\ x) \\ \sigma_2\ x \to (g\ x, g\ x) \end{array} \right.$$

Positive variant We could also consider a variant of this goal with a different choice of atom polarities – there are other possible choices but this one is interesting.

$$f : 1 \to X^+ + X^+, g : X^+ \to Y^+ \vdash ? : 0 + (Y^+ \times Y^+)$$

As before, the first saturation step has exactly one neutral to introduce, $\texttt{let}\ x = f\ ()\ \texttt{in}\ ?$, with $x : X^+ + X^+$. But, after the following invertible phase $\texttt{match}\ x\ \texttt{with}\ \mid \sigma_1\ x \to ? \mid \sigma_2\ x \to ?$, saturated proof search differs from the previous one as a new positive becomes provable, $(g\ x : Y^+)$. This judgment is still uniquely inhabited, but with a different saturated proof term:

$$\texttt{let}\ x = f\ ()\ \texttt{in}\ \texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ x \to \texttt{let}\ y = g\ x\ \texttt{in}\ (y, y) \\ \sigma_2\ x \to \texttt{let}\ y = g\ x\ \texttt{in}\ (y, y) \end{array} \right.$$

11.1.5. Saturation and the empty type

This idea of canonical enumeration through saturation extends seamlessly to the empty type . Recall the equivalence rule for the empty type:

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

To integrate this rule in a saturating focused type system, it suffices to split, during the saturation phase, all neutral terms n of *positive type*, including 0, instead of just sum types. The saturation process will then, in particular, look for any possible way to obtain a proof of 0 from the variables in the context. If the context is inconsistent, a proof of 0 will be bound and eliminated, cutting the proof search: a single term of the form $\texttt{absurd}(\_)$ will be returned, as all possible terms under this inconsistent context are equivalent by rule above.

In particular, out of the proof that this saturating focused type system is canonical, we can in fact extract an equivalence algorithm that decides equivalence of terms in presence of sums *and* empty types.

Remark 11.1.2. Equivalence in presence of empty types was previously perceived to be a delicate problem, while it here falls of as a simple consequence of our work on unique

inhabitation. It is interesting to look at the difference between this and past approaches to proof equivalence that makes it simpler in our setting.

Among the existing work on program equivalence, the more algorithmically-flavored proposal work by moving around portions of the terms to normalize or compare, typically extruding certain subterms them out of certain contexts – a notable exception is the type-directed partial evaluation approach of Balat, Di Cosmo, and Fiore [2004].

On the contrary, checking equivalence in presence of the empty type requires looking for *arbitrary* terms that may prove the current typing context inconsistent, but may be entirely unrelated to the terms being compared – the term $t : 0$ in the rule above, to compare to the inputs of the algorithms $u_1, u_2 : A$. In particular, t may be unreachable by just combining subterms of the terms to compare, or otherwise reasoning on their syntactic shape. In other words, adding 0 requires to have a part of the algorithm that performs arbitrary proof search, and this is given for free by saturation.

Of course, this only works in type systems in which testing for inhabitation of 0 is decidable (a test implicitly done by our saturation process), which is the case in the simply-lambda calculus, as it corresponds to propositional (quantifier-free) intuitionistic logic.

The idea that checking equivalence in presence of 0 requires arbitrary proof search is not new. It was already suggested, informally, in Neil Ghani's 1995 PhD thesis [Ghani, 1995a], that is in the first work giving a positive answer for decidability in equivalence in presence of sums.[3] However, going after this intuition would have required a potentially invasive change to the structure of the equivalence algorithms, which we suppose is probably why the problem remained open for so long. The contribution of our saturating focused system is not the idea of introducing proof search in equivalence algorithms, but the creation of a setting where this behavior occurs naturally. ∗

11.2. A saturating focused type system

As for the focused λ-calculus, the types in our system make an explicit distinction between "positive" types P, Q (we have to make choices to *build* their values: sums) and "negative" types N, M (we have to make choices to *use* their values: functions and products). For example, a function type $P \to N$ expects a positive type on the left and a negative type on the right. If you want the function to return a positive type such as $X^+ + Y^+$ it has to be wrapped in an explicit marker $\langle \_ \rangle^-$, converting it into a negative type. The full type would be, for example, $Z^+ \to \langle X^+ + Y^+ \rangle^-$.

We introduced these explicit shifts in Chapter 7 (Focusing in sequent calculus), Section 7.3.1 (Explicit shifts), but a reader not familiar with focusing could just read Figure 7.6 (Polarized propositional formulas) to know the grammar, and just ignore the plusses and minusses from now on. In the article version of this chapter [Scherer and Rémy, 2015], we used the usual (non-polarized) syntax of simple types, and the saturating type system is essentially the same – using polarized types is not essential, it just gives more structure to the presentation.

In Figure 11.1 (Cut-free saturating focused type system (in natural deduction style)) we give the full typing rules for our *saturating* focused λ-calculus. They share many similarities with the focused λ-calculus of Chapter 10 (Focused λ-calculus), with several changes that we will describe in detail. The calculus is described by four mutually recursive judgments, whose role we will detail in this section.

- The invertible judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t : N \mid Q^{\mathsf{at}}$, which is very close to the invertible judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$ of the focused λ-calculus.
- The saturating judgment $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$ is where most of the novelty lies, in particular the SAT rule that enforces saturating. It is inspired by the "choice of

[3] The discussion of eventual extension to the empty type is at pages 99, 100 and 101 of the manuscript version I could find.

focusing" judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} f : Q^{\mathsf{at}}$ of the simple focused λ-calculus, but behaves in a different way.

- The focused introduction and elimination judgments $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P$ and $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow N$, which are identical to the corresponding judgments of the focused λ-calculus.

In addition, the type system is parametrized by a family of selection functions $\mathtt{Select}_{\Gamma^{\mathsf{at}}}(\_)$; for any negative or atomic context Γ^{at} and positive or atomic goal type P^{at}, it takes as input a (potentially infinite) set of neutrals of positive type (n, P) and returns a finite subset of its input. This parameter represents choices that can be made by an algorithm derived from this logic. We do impose a requirement on the possible choice of selection function: it has to satisfy the requirement of SELECT-SPECIF, which we will explain in this section.

Figure 11.1.: Cut-free saturating focused type system (in natural deduction style)

SINV-LAM
$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P \vdash_{\mathit{sinv}} t : N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} \lambda x.\, t : P \to N \mid}$$

SINV-PAIR
$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t_1 : N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t_2 : N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} (t_1, t_2) : N_1 \times N_2 \mid}$$

SINV-TRIVIAL
$$\frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} () : 1 \mid}$$

SINV-ABSURD
$$\frac{}{\Gamma^{\mathsf{at}}; \Sigma, x : 0 \vdash_{\mathit{sinv}} \mathtt{absurd}(x) : N \mid Q^{\mathsf{at}}}$$

SINV-CASE
$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P_1 \vdash_{\mathit{sinv}} t_1 : N \mid Q^{\mathsf{at}} \quad \Gamma^{\mathsf{at}}; \Sigma, x : P_2 \vdash_{\mathit{sinv}} t_2 : N \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, x : P_1 + P_2 \vdash_{\mathit{sinv}} \mathtt{match}\ x\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x \to t_1 \\ \sigma_2\ x \to t_2 \end{array} : N \mid Q^{\mathsf{at}}}$$

SINV-SAT
$$\frac{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathit{sinv}} f : \langle P^{\mathsf{at}} \rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

SAT
$$\frac{(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^{-}) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}) \quad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

SAT-UP
$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} p : \langle P \rangle^{-}}$$

SAT-DOWN
$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow X^{-}}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} n : X^{-}}$$

SAT-UP-SINV
$$\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathit{sinv}} t : N \mid \emptyset}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} t \Uparrow \langle N \rangle^{+}}$$

SAT-UP-ATOM
$$\frac{}{\Gamma^{\mathsf{at}}, x : X^{+} \vdash_{\mathsf{s}} x \Uparrow X^{+}}$$

SAT-DOWN-VAR
$$\frac{}{\Gamma^{\mathsf{at}}, x : N \vdash_{\mathsf{s}} x \Downarrow N}$$

SAT-DOWN-PROJ
$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} \pi_i\ n \Downarrow N_i}$$

SAT-DOWN-APP
$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n\ p \Downarrow N}$$

SAT-UP-INJ
$$\frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} \sigma_i\ p \Uparrow P_1 + P_2}$$

SELECT-SPECIF
$$\frac{\forall P, \quad n : P \in S \quad \Longrightarrow \quad \Gamma^{\mathsf{at}} \Uparrow P \ \vee\ \exists n', n' : P \in \mathtt{Select}_{\Gamma^{\mathsf{at}}}(S)}{\mathtt{Select}\_(\_) \text{ is a valid selection function}}$$

11.2.1. Invertible phase

The invertible judgment $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t : N \mid Q^{\mathsf{at}}$ corresponds to the reasoning that we used in Chapter 11 (Re-introduction to canonical and complete type systems) to informally describe enumeration of distinct terms, at type that have a "generic" constructor: to enumerate $A \to B$, it suffices to look for terms of the form $\lambda x.\, \_$. In term of focusing, we say that the λ-introduction rule is "invertible", which means here that we can always assume terms of function types are built using it, without losing any generality. Same things for product $-$ and unit, obviously.

A novelty of the focusing-based point of view is that this "without loss of generality" reasoning not only applies to terms with an invertible *constructor* (the negative types), but also terms that can be *destructed* without any loss of generality (the positive types). If we have a variable of sum type in the context, any possible well-typed term can be rewritten to begin with a case-split on this variable.

SINV-CASE

$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P_1 \vdash_{\mathit{sinv}} t_1 : N \mid Q^{\mathsf{at}} \qquad \Gamma^{\mathsf{at}}; \Sigma, x : P_2 \vdash_{\mathit{sinv}} t_2 : N \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, x : P_1 + P_2 \vdash_{\mathit{sinv}} \mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to t_1 \mid \sigma_2\ x \to t_2 \ : N \mid Q^{\mathsf{at}}}$$

When reading this rule, one should first read the rule without the terms, or with the terms replaced by not-yet-filled holes, and think of the goal-directed search process: whenever we want to enumerate all terms at this typing judgment, it suffices to enumerate the possible terms t_1 and t_2 in the premises, and for each pair of such terms (in the cartesian product of the enumeration) return the term $(\mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to t_1 \mid \sigma_2\ x \to t_2)$. In other words, all distinct terms are (equivalent to a term) of the shape $(\mathtt{match}\ x\ \mathtt{with} \mid \sigma_1\ x \to ?_1 \mid \sigma_2\ x \to ?_2)$ with the holes $?_i$ filled as per the premise judgments.

Remark 11.2.1. This arguably distinguishes focusing from other approaches such as bidirectional type-checking, which are essentially identical on the purely negative fragment. Focusing is justified in a general enough setting to easily extend to sum types. It predicts that some type-directed transformations should be guided by the typing context, rather than the goal type. ∗

The negative types are those whose construction (introduction) rule is invertible, and the positive types are those whose destruction (elimination) rule is invertible. This means that while the goal is negative, or while there remains a negative in the context, an invertible rule can be applied. The structure of our judgments forces us to apply these invertible rules as long as possible; we only leave the invertible judgment in the transition rule SINV-SAT, which is only available when the context has only negative or atomic formulas, and the goal is positive or atomic:

SINV-SAT

$$\frac{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathit{sinv}} t : \left\langle P^{\mathsf{at}} \right\rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

More precisely, the polarity constraint is enforced by the fact that the function $\langle \_ \rangle^{+\mathsf{at}}$ takes a negative formula, and returns a positive formula (by shifting) or a negative atom (atoms are preserved); so the judgment context is in the image of this function only if all its formulas are shifted positive formulas or negative atoms. Same thing for $\langle \_ \rangle^{-\mathsf{at}}$ in the goal.

On the goal side, let us recall that a convention of the $(A \mid B)$ notation is that exactly one of the sides is empty, and the other is a formula. The invertible judgment maintains two different formula positions,

Finally, let us comment on the role of the two contexts Γ^{at} and $\Gamma^{\mathsf{at}\prime}$ appearing in this rule, and in general Γ^{at} (a context of negative or atomic formulas) and the second context

Σ (a context of positive formulas). Γ^{at} never evolves when applying rules of the invertible phase: it is the "old" context, in which the invertible phase started, unchanged. On the contrary, Σ is the context of formulas that are added to the context during the phase (by introducing a λ-abstraction, or by decomposing a formula already in Σ). It contains the "new" formulas that were unknown at the beginning of the invertible phase.

11.2.2. Saturation phase – a first look

The saturation phase only starts where all possible invertible rules have been applied. Any rule we can apply now is *non-invertible*: it requires making a choice, and it may be the wrong choice – going to a dead end.

There are two kinds of non-invertible rules: the ones that try to use variables from the context (for example choosing to call a function from the context, which may fail if we can't build a value of the argument's type), and the ones that try to construct values at the goal type (if the goal is a sum $A_1 + A_2$, it would be an injection constructor $\sigma_i\ \_$, representing the choice to either build a A_1 or a A_2). In the (asymmetric) intuitionistic logic, using the context is better choice, as failure there does not require backtracking (at worst we do not manage to call the function, and we continue the proof with something else); thus, we try to deduce everything we can from the context first, and do a choice on the goal type only later – this is saturation, done by the SAT rule.

$$\textsc{sat}\ \frac{(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}},\Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P\rangle^{-}) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}) \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

The SAT rule is the central and most complex rule of our saturated calculus. I do not know how to explain it in one go – the current definition evolved by refinement. Instead of trying to dissect it now, we will use a two-step approach: first describe informally what it does, *assume* that it does it correctly to understand the rest of the rules and the big picture of how the whole type system works, and then go back to its definition once the general mechanics is in place.

What the SAT rule does is the following: it looks for all the way that a positive formula can be *deduced* from the context, that is, proved by a neutral term $n : \langle P\rangle^{-}$. It adds all these deductions to the current context, and goes to the invertible judgments again – where these positive formulas are decomposed by the invertible rules, before starting another step of saturation. Note that the goal type is not changed by saturation, it is still positive or atomic, and is thus not decomposed by the following invertible phase. Only the types just deduced by saturation change during inversion.

With this description, it looks like the saturation process would never stop. This is where the separation, in the invertible judgment, between the "old" context and the "new" context come in. Eventually, it will become the case that all positive formulas deducible from the context have been deduced, and the next saturation phase will not split on any new formula. The invertible phase will start, but stop immediately after (no positive formula from the context to decompose), and call the saturation judgment again with $\Gamma^{\mathsf{at}\prime}$ being the empty set $\emptyset$. When the "new" context is empty, we know that saturation has reached a stable state, and we allow saturation to stop: instead of the SAT rule, the proof may continue with either SAT-UP or SAT-DOWN, that escape the saturation judgment by finally trying to construct a term/proof of the goal type. At this point, we have done all possible deductions from the context, so we can make arbitrary choices (in fact, try all those choices), as there is nothing more to learn to help us making those choices.

The rules SAT-UP and SAT-DOWN do not overlap, only one of them is usable depending on the goal type. If it is a positive formula, we try to prove by a series of introduction rules (in terms of focusing, this is a right focusing phase). If it is a negative atom, we try to

prove it by a series of elimination rules (in terms of focusing, this is a left focusing phase that ends on a negative atom).

$$\textsc{sat-up}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} p : \langle P \rangle^-} \qquad \textsc{sat-down}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow X^-}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} n : X^-}$$

11.2.3. Focused introduction and elimination phases

The judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P$, entered from the SAT-UP rule, tries to prove a positive formula by a series of introduction rules, by building a term out of value constructors. At each step of this judgment we need to make a non-invertible choice; to enumerate all possible proofs, we just backtrace on each of those choices. When we reach a (shifted) negative formula $\langle N \rangle^+$ in the rule SAT-UP-SINV, there are no non-invertible constructors to apply anymore, so we revert to the invertible judgment.

$$\textsc{sat-up-inj}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P_i}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} \sigma_i\ p \Uparrow P_1 + P_2} \qquad \textsc{sat-up-sinv}\ \frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathit{sinv}} t : N \mid \emptyset}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} t \Uparrow \langle N \rangle^+}$$

The judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow N$, entered from the SAT-DOWN rule, describes a series of elimination steps (function application or pair projection) applied to a head variable taken in the context. Unlike all other judgments of natural deduction or sequent calculus, the rules of this judgment should be read from leaf to root. A proof start from a variable chosen from the context, in the rule SAT-DOWN-VAR, that is of negative type, and applies a series of non-invertible elimination rules, passing an argument (if the negative type is a function) or projecting one component (if the negative type is a product).

$$\textsc{sat-down-var}\ \frac{}{\Gamma^{\mathsf{at}}, x : N \vdash_{\mathsf{s}} x \Downarrow N} \qquad \textsc{sat-down-app}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n\ p \Downarrow N} \qquad \textsc{sat-down-proj}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow N_1 \times N_2}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} \pi_i\ n \Downarrow N_i}$$

Notice that the input type of function is a positive type, and that we look for an argument as a positive neutral term p by typing it with the non-invertible introduction judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P$. Some proof systems are "less focused", in that they allow function arguments to start with a more general invertible phase.

The ending rule of the introduction judgment $\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} p \Uparrow P$ enforces the fact that an introduction phase ends only when the formula becomes negative (or, in the SAT-UP-ATOM rule, when we reach a positive axiom). The elimination judgment goes in the other direction, so it is the "caller" of this judgment (the rule who has the elimination judgment as a premise) that decides when it can end. In the SAT-DOWN rule, we only consider elimination phases that end on a negative atom, and in the SAT rule we only consider elimination phases that end of a (shifted) positive formula.

$$\textsc{sat}\ \frac{(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\boxed{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^-}) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}) \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

$$\textsc{sat-down}\ \frac{\Gamma^{\mathsf{at}} \vdash_{\mathsf{s}} n \Downarrow X^-}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} n : X^-}$$

11.2.4. The saturation rule – a deeper look

A naive attempt at defining the SAT rule would look as follows:

$$\textsc{sat-1}\ \frac{(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^-)\} \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

This definition looks for all ways to deduce (by a neutral proof term) a positive from the current context, adds it to the context, and continues with an invertible phase that will decompose those positives. It has two independent problems:

1. A single neutral term n will be introduced many times, by all saturation steps where it is typable – by monotonicity, subsequent saturation steps will introduce all the proofs of the previous iteration steps, plus some more. This breaks canonicity, which relies on the fact that each possible neutral (each possible observation of the formal context) is given a *unique* name. Consider for example the judgment

$$x : 1 \to \langle X^+ \rangle^- ; \emptyset \vdash_{\mathsf{sinv}} ? : \emptyset \mid X^+$$

   The first saturation phase will deduce X^+ by introducing the proof $y_1 \stackrel{\mathsf{def}}{=} x\ ()$ of type X^-. It is followed by an invertible phase that will stop immediately, as there is no connective to decompose in the context or the goal. Then a new saturation phase starts; because there is no provision in SAT-1 against performing the same deduction again, the term could introduce $y_2 \stackrel{\mathsf{def}}{=} x\ ()$ of type X^+. This could go on indefinitely, but forgetting about the termination aspect for a moment, we have a canonicity problem: it now appear that there are two distinct ways to build the goal X^+, using either y_1 or y_2 – formal variables in the context are considered distinct.

   We need a way to remember which neutrals have been introduced in previous saturation step, not to re-introduce them again; not doing so would break canonicity of the proof system, and thus soundness of the unicity-deciding algorithm.

2. The present definition introduces, at each saturation steps, *all* the neutrals of positive types. Even without taking the previously introduced ones into account, there may be too much new neutrals, leading saturated proof search into an infinite loop. There are two different sources of non-termination:

   - A single saturation step may, with this definition, introduce infinitely many positives. Consider for example a variable in context $x : \mathbb{N} \to P$, where $\mathbb{N}$ is a type of natural numbers, defined as $\mathbb{N} \stackrel{\mathsf{def}}{=} (X^- \to X^-) \to X^- \to X^-$ for example, and P is some positive type. With such a variable x in the context $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}$, the set

$$\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^-)\}$$

     is infinite (it contains $x\ 0$, $x\ 1$, $x\ 2$, etc., with the definition of natural constants of Section 2.1.7). Even if we extended our syntax to accommodate infinitely-wide let-bindings $\texttt{let}\ \bar{x} = \bar{n}\ \texttt{in}\ \_$, the following invertible phase would have to deconstruct infinitely many copies of the type P in context, so there would be no finite proof (term) in this type system for any goal with $x : \mathbb{N} \to P$ in context.

   - Even if each saturation step is finite, saturation may keep going on indefinitely if each step introduces a new variable to use. Consider for example that for some "stream state" type X^+ we have in the typing environment a state value $x_0 : X^+$ and a "next" function $y : X^+ \to 1 + X^+$ that returns the next state if it exists, or 1 if there is no next state – we reached the end of the stream. The first saturation step can use x_0 to deduce a new value $y\ x_0$ of positive type $1 + X^+$; the invertible phase will pattern-match on this new value, and in the right branch we will have a new variable $x_1 : X^+$ in context. The second saturation phase can deduce a new value $y\ x_1$ of positive type, and the second invertible phase will decompose it and (in the right case) bind a new variable $x_2 : X^+$ in context. This saturation process can continue indefinitely, even though each saturation step only introduces finitely many positives. This corresponds to the

incremental construction of an infinite term spine, matching over an unbounded stream:

$$\texttt{let } x_1 = y\ x_0 \texttt{ in } \begin{array}{|l} \texttt{match } x_1 \texttt{ with} \\ \mid \sigma_1\ x_1 \to \dots \\ \\ \mid \sigma_2\ x_1 \to \texttt{let } x_2 = y\ x_1 \texttt{ in } \end{array} \begin{array}{|l} \\ \\ \texttt{match } x_2 \texttt{ with} \\ \mid \sigma_1\ x_2 \to \dots \\ \mid \sigma_2\ x_2 \to \texttt{let } x_3 = y\ x_2 \texttt{ in } \dots \end{array}$$

In particular, the "new context" $\Gamma^{\mathsf{at}'}$ will always contain at least one new variable x_n of type X^+; it will never be empty, and the rules exiting the saturation cycle, SAT-UP and SAT-DOWN, will never be applicable. No matter what the goal type is (as long as it is positive, that is there is at least one saturation step), a system using the rule SAT-1 would have no (finite) proof term as soon as those "state" and "next" variables are in context.

Those are not canonicity issues (we are not enumerating duplicates), but termination and completeness issues. If some judgments that should be provable have no finite proofs, it means that our system is incomplete (even for provability), and also that proof search and enumeration will not terminate. To prevent this, we must somehow allow the logic to "drop" some new variables produced by saturation (when it is correct to do so), so that no single saturation step binds infinitely many variables, and so that repeated saturation steps eventually reach a stable state with an empty "new" context. This is done by keeping at most two variables of each type, using the Corollary 9.3.6 (Two-or-more approximation) of Chapter 9 (Counting terms and proofs).

Avoiding redundant splits An idea to solve the first problem (not splitting on the same neutral terms in several saturation processes) is to simply index all judgments will the set of all neutrals split so far, and to remove those neutrals from any following saturation step. This is, in fact, not necessary, thanks to our structural separation of the context between an "old" context Γ^{at} and a "new" context $\Gamma^{\mathsf{at}'}$. The new context contains exactly the variables that were split by the last invertible phase, and the old context the older ones, that were already available during the previous saturation step.

There is thus a very simple characterization of which neutrals n were already split in a previous saturation step, and should not be split again. They are the neutrals that are already typable in the old context Γ^{at}, or conversely the neutrals that do not use any variable from the new context $\Gamma^{\mathsf{at}'}$. This is the simplification that justifies keeping the static separation between the old and new context in the invertible rules.

An improved (but still unsatisfying) reformulation of the preliminary SAT-1 rule, that avoids redundant splits, is as follows:

$$\text{SAT-2}\quad \frac{\begin{array}{c}(\bar n, \bar P) \overset{\mathsf{def}}{=} \{(n,P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}'} \vdash_{\mathsf{s}} n \Downarrow \langle P\rangle^-) \text{ and } (\exists x \in \Gamma^{\mathsf{at}'}, x \in n)\} \\ \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}'}; \bar x : \bar P \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}'} \vdash_{\mathsf{sat}} \texttt{let } \bar x = \bar n \texttt{ in } t : Q^{\mathsf{at}}}$$

This new rule forces us to introduce only (and all) the terms that are "new", in the sense that they use the new context $\Gamma^{\mathsf{at}'}$ – this is checked by the condition $(\exists x \in \Gamma^{\mathsf{at}'}, x \in t)$.

Finite saturation proofs As we have seen with a few examples, some contexts have saturation processes that split infinitely many new neutrals, either during a single step or through infinitely many steps never reaching a fixpoint. This is not surprising or wrong: some types are inhabited by infinitely many distinct programs. However, while we expect that enumerating all those programs would require infinitely many steps, we would like

to be able to have finite proofs for each of those programs, which our current saturation rules does not allow.

To have finite proofs even during an infinite saturation process, it suffices to allow some proofs to use only *a subset* of the split subterms. Instead of SAT-2, consider the following rule, which only replaces the $(\overset{\mathsf{def}}{=})$ in the first premise by a $(\subseteq)$:

$$\textsc{sat-2-sub}\quad \frac{(\bar{n}, \bar{P}) \subseteq \{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^{-}) \text{ and } (\exists x \in \Gamma^{\mathsf{at}\prime}, x \in n)\} \quad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

It may seem that this definition of the saturation rule allow the goal-directed proof enumeration process to stop the saturation earlier than it should (in particular if we select $\bar{n} \overset{\mathsf{def}}{=} \emptyset$, then saturation stops) and thus make the search incomplete. But as the enumeration process is looking for all possible proof terms of the judgment, it may consider all possible subsets, and thus not miss a single term; note that each finite term uses only a finite subset of the split neutrals, so we can always assume $\bar{n}$ finite.

Unfortunately, this weaker condition also causes a loss of canonicity: two proof terms may be essentially the same, but differ by the fact that one saturates on a few additional neutrals – otherwise unused.

Canonicity by deterministic restriction This gets us to the final version of our rule:

$$\textsc{sat}\quad \frac{(\bar{n}, \bar{P}) \overset{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^{-}) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}) \quad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

In this version, the choice of which subset of neutrals to saturate on is fixed once and for all by the saturation function $\mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\_)$. For a given choice of saturation function, all proof search processes for a given judgment will select the same set of neutrals. This avoids the previous canonicity issue: two terms cannot differ merely by the choice of which neutrals to saturate over.

Comparison with the previous approach of Scherer and Rémy [2015] In the previous presentation of Scherer and Rémy [2015], we did not use a fixed saturation-selection function; instead, the saturation rule had one extra requirement that all the neutrals introduced by saturation where "useful" in some sense.

$$\textsc{sat}\quad \frac{(\bar{n}, \bar{P}) \subseteq \{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^{-}) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\} \quad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t : \emptyset \mid Q^{\mathsf{at}} \qquad \forall x \in \bar{x}, t \ \mathsf{uses}\ x}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

The $(t \ \mathsf{uses}\ x)$ judgment, which we have not defined here, corresponds to the fact that the introduced variable x is used after the first invertible phase.

This condition did not affect proof search, as it is expressed on the proof term t that is only known after the search for this recursive subgoal has taken place. The set of useful neutrals was obtained by filtering the saturating neutrals after the fact. This mean that each possible outcome of the proof search (the term t) was uniquely associated with a "minimal" saturating set, avoiding any canonicity issue.

Unfortunately, this simplification would not scale to a richer setting with the empty

type 0.[4] It may be that there are, in a particular typing environment, two syntactically distinct neutrals of empty type. Saturation will find these two distinct neutrals, but it will then make of choice of eliminating the empty type on either one of those, creating an artifical choice of which of the two is really "useful".

For example under the typing environment $\{x : 1 \to 0, y : 1 \to 0\}$, we can prove 0 by introducing either $x\ ()$ or $y\ ()$. This gives two proof terms, $\texttt{let}\ z_1 = x\ ()\ \texttt{in}\ \texttt{absurd}(z_1)$ and $\texttt{let}\ z_2 = y\ ()\ \texttt{in}\ \texttt{absurd}(z_2)$ that are both minimal for the refinement relation, syntactically distinct from each other, yet semantically equivalent. The formula of the saturation rule using this usefulness condition would accept both as separate proof terms, losing canonicity.

Note that the search algorithm of Scherer and Rémy [2015] gave the correct answer on this example, or in fact any query involving the empty type: upon deducing either or both of these ways to deduce 0, it would correctly conclude during the following invertible phase that the goal is uniquely inhabited. The implementation was correct, but the logic in which it was specified did not scale to the empty type.

Our more flexible current definition allows the $\texttt{Select}_{\Gamma^{\mathsf{at}},\Gamma^{\mathsf{at}\prime}}(\_)$ filter to select either one, or both of these proofs of 0; in any case, all proof search in this context will use the same saturating set, so canonicity is not lost for this reason. This is also closer to what the unicity algorithm actually does, as the choice of the saturating set is made one and for all, before the t is recursively searched for.

On the other hand, we would be in trouble if the saturation selection function picked none of the possible proofs of 0 that can be made in the context – or, in general, missed a positive that reveals an absurdity after decomposition, for example $0+0$. In absence of the empty type, "not deducing enough" can only lead to a loss of completeness; in presence of the empty type, missing an incoherence deduction could lead an algorithm to falsely believe that there are several terms, while they are in fact all equivalent.

Completeness condition on the selection function As we noted, canonicity would be endangered by a selection function that is too incomplete. Consider for example the goal

$$\emptyset; f : \langle 1 \rangle^+ \to \langle 0 \rangle^-, x : X^-, y : X^- \vdash_{\mathsf{sat}} ? : X^-$$

If our selection function $\texttt{Select}\_(\_)$ always returned the empty set (selecting none of the potential neutrals), saturation would stop at the next phase and we would have two distinct proof terms for this goal, namely x and y. This is wrong, as saturating more would have let us discover the proof $(f\ () : 0)$ that the context is inconsistent, and that only one proof $(\texttt{let}\ x = f\ ()\ \texttt{in}\ \texttt{absurd}(x))$ is possible.

To guarantee that all possible proofs of 0 are found, it suffices to demand that the selection function drops no provable type: if some $(n : P)$ belongs to the set S of new neutrals deducible at this step, then there must exists some proof of P in the returned selection:

$$\exists n', \qquad n' : P \in \texttt{Select}_{\Gamma^{\mathsf{at}}}(S)$$

Note that while we may have infinitely many new neutral terms n, the subformula property guarantees that there are only finitely many new types P deducible by elimination rules. In particular, a selection function can respect this requirement and still return finite sets.

This requirement is correct, but it is a bit too strong. Intuitively, it is not necessary to force P to be part of the returned set if has already been added to the context Γ^{at} by a previous saturation phase. Of course, P is a strictly positive formula, while Γ^{at} is a context of positive or atomic types, so it does not make sense to check $P \in \Gamma^{\mathsf{at}}$ in general;

[4] Note that we are not claiming that the presentation *does* scale to handling 0, as we have not been able to prove so, but we would at least rather use one that is not known to be broken in presence of 0.

but we can instead check for whether the formula P can be retrieved from the types in Γ^{at}, if $\Gamma^{\mathsf{at}} \Uparrow P$ holds – we use the strong positive phase introduced in Section 10.5 (Strong positive phases). In this case, we need not require P to be part of the selected types.

This gives us the full criterion given in SELECT-SPECIF:

$$\frac{\text{SELECT-SPECIF}\quad \forall P,\quad n : P \in S \quad\Longrightarrow\quad \Gamma^{\mathsf{at}} \Uparrow P \ \vee\ \exists n',\, n' : P \in \mathtt{Select}_{\Gamma^{\mathsf{at}}}(S)}{\mathtt{Select}\_(\_) \text{ is a valid selection function}}$$

11.2.5. The roles of forward and backward search in a saturated logic

Focusing is a fruitful theoretical tool to propose a more logical understanding of proof search strategies – see for example Chaudhuri, Pfenning, and Price [2008b], Chaudhuri [2010], Farooque, Graham-Lengrand, and Mahboubi [2013]. This flexibility is built out of two components whose interaction can be subtle. On one hand, the way formulas are polarized prevents or enforce certain shapes of proof terms, for example forward- or backward-chaining, as we detailed in Section 7.1.7 (Polarized atoms). On the other hand, there are several distinct strategies for proof search, notably the rather natural judgment-directed or goal-directed backward search, and the inverse method, a form saturation-based forward search. The strength of focusing is to move a lot of the sophistication from the search strategy into the logic itself: a lot of subtle operational ideas on good proof strategies can be obtained by using one of those two simple strategies with a subtle logic or polarisation of formulas.

To prove a judgment of the form $\Delta \vdash A$, the natural intuition for goal-directed search procedure is to look at A and search for all possible ways to introduce its head connective. A focused system has a richer behavior, in that it will also decompose the positives of Δ, but this reliance on the context remains "superficial" in the sense that only the first positive layer of those formulas will be peeled of by the invertible phase. The "real" work happens at the end of the invertible phase, where choices must be made, and typically various attempts will be made, with a backtracking discipline to roll back the wrong choices, for example the right introduction on a sum that happened too early.

On the contrary, on a judgment of the form $\Delta \vdash A$, an inverse method will, in rough terms, look as the subformulas of Δ, A as the "search space" of facts to prove. It will try to build proofs in a leafward-rootward fashion, from elementary deduction in this search space to more elaborated facts, until maybe a deduction implying the original goal $\Delta \vdash A$: happens.

I would now like to discuss the operational search behavior of this saturated logic when using a simple judgment-directed backward search implementation.

Goal-directed proof search in our saturated logic starts in a state where all of the context is "new", it has not been saturated over: $\emptyset; \Delta \vdash A$. During the invertible phase, it behaves like others goal-directed procedures, and extract a negative or atomic context Γ^{at} of "new" formulas, and a refined goal Q^{at}, and start the saturation phase $\emptyset; \Gamma^{\mathsf{at}} \vdash_{\mathsf{sat}} ? : Q^{\mathsf{at}}$.

The saturation phase does not behave like a goal-directed backward procedure, it is a phase of forward search. However, there is an important difference with the inverse method or other approaches that are "full" forward search: the "search space" of the saturation is not the complete goal $\Gamma^{\mathsf{at}} \vdash Q^{\mathsf{at}}$, it is only Γ^{at}. We are not trying to discover arbitrary facts that will help us in eventually proving our goal Q^{at}, we are restricting the set of deductions to subformulas of the context Γ^{at}. So it is a forward search phase, but it is "localized" by the use of only a part of the judgment.

After this local saturation phase ends, goal-directed search starts over with non-invertible steps attempting to prove the goal formula, and the corresponding backtracking behavior of backward search. The right rules that happen during this right focusing phase will change the goal formula to a negative subformula of Q^{at}. This creates opportunities for the following invertible to move parts of the goal into the context, expanding the "horizon" of the following saturation phases.

This would be my understanding of the operation behavior of saturated proof search. There is an alternation of backward and forward search phases. The forward search is bounded by the context, while the backward search is directed by the goal formula, and transmits new hypothesis to the context, expanding the reach of the subsequent forward phases.

Interestingly, this mixture of backward and forward search exists in some seemingly unrelated work on logic programming, in particular in Lollimon López, Pfenning, Polakow, and Watkins [2005]; we give a detailed comparison in Section 13.3.2 (Lollimon: backward and forward search together).

11.3. Saturation theorem

In Chapter 10 (Focused λ-calculus), Section 10.5 (Strong positive phases), we proposed a judgment $\Gamma^{\mathsf{at}} \vdash p \Uparrow P$ that corresponds to the intuition of "retrieving" a positive P from the context Γ^{at}. In this section, we build upon this intuition to state and prove a formal statement that captures the essence of the saturation process.

This informal view of the different ways to deduce a positive formula give a specification of what saturation is doing. From a high-level or big-step point of view, saturation is trying all possible new deductions iteratively, until all positives deductible from the context have been added to it. The following characterization is more fine-grained, as it describes the state of an intermediary saturation judgment $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : P^{\mathsf{at}}$, and makes precise what we mean by "old context" (Γ^{at}) and "new context" ($\Gamma^{\mathsf{at}\prime}$).

The characterization is as follows: any formula that can be "simply deduced" from the old context Γ^{at} is "retrievable" in the larger context $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}$. In other words, if $\Gamma^{\mathsf{at}} \Downarrow \langle P \rangle^-$, then $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \Uparrow P$; in other words, either it has already been saturated over in Γ^{at}, or it is part of the new deductions $\Gamma^{\mathsf{at}\prime}$. This gives a precise meaning to the intuition that Γ^{at} is "old"; what we mean by "new" can be deduced negatively: it is the part of the context that is still fresh, its deductions have not been stored in the knowledge base yet.

Remark 11.3.1. This notion of the context as a "knowledge base", which is useful when thinking of saturation, is fairly specific to our setting of intuitionistic logic, where all facts are duplicable and the context grows monotonically. It is unclear whether a form of saturation would work for linear logic. *

Theorem 11.3.1 (Saturation)**.**
If a saturated proof starts from a judgment of the form

$$\emptyset; \Gamma_0^{\mathsf{at}} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}} \qquad \textit{or} \qquad \emptyset; \Sigma_0 \vdash_{\mathsf{sinv}} t : N \mid Q^{\mathsf{at}}$$

then for any sub-derivation of the form

$$\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$$

we have the following property:

$$\forall P, \qquad \Gamma^{\mathsf{at}} \Downarrow \langle P \rangle^- \qquad \Longrightarrow \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \Uparrow P$$

Proof. By induction on the derivation.

This is immediately true in the initial case of a judgment of the form $\emptyset; \Gamma_0^{\mathsf{at}} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$ or $\emptyset; \Sigma_0 \vdash_{\mathsf{sinv}} t : N \mid Q^{\mathsf{at}}$, as no direct deductions can be made from the empty set: $\emptyset \not\Downarrow N$ for any N.

The induction case is the saturation rule

$$\textsc{sat}\ \frac{(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P \rangle^-) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}) \qquad \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathsf{sinv}} t : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t : Q^{\mathsf{at}}}$$

Let us assume that we have

$$t \approx_{\mathtt{icc}} E_{\text{IN}} \left[f_j\right]^{j\in J} \qquad \bar{P} \vdash_{\mathsf{inv}} E_{\text{IN}} \left[\Gamma^{\mathsf{at}\prime\prime}_j \vdash_{\mathsf{foc}} \Box_j : \emptyset\right]^{j\in J} : \emptyset$$

$$\left(\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \Gamma^{\mathsf{at}\prime\prime}_j \vdash_{\mathsf{sat}} f_j : Q^{\mathsf{at}}\right)^{j\in J}$$

We have to prove that for any P such that $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \Downarrow P$, we have $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}, \Gamma^{\mathsf{at}\prime\prime}_j \Uparrow P$ for all $j \in J$.

If $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \Downarrow P$ holds, it may be the case that $\Gamma^{\mathsf{at}} \Downarrow P$ already holds. In this case we have $\Gamma^{\mathsf{at}} \Uparrow P$ by induction hypothesis, as desired. In the other case, P is not provable without using variables from the new context $\Gamma^{\mathsf{at}\prime}$; so there must be a proof of it in the set

$$\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{s}} n \Downarrow \langle P\rangle^-) \wedge \exists x \in \Gamma^{\mathsf{at}\prime}, x \in n\}$$

By the condition on selection functions SELECT-SPECIF, we thus know that either $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \Uparrow P$, as desired, or there is a proof $n : P$ among the selected bindings $\bar{n} : \bar{P}$. By Lemma 10.5.2 (Higher-order invertible phase), we can then deduce that $\Gamma^{\mathsf{at}\prime\prime}_j \Uparrow P$, as desired. □

11.3.1. Saturated contexts

From this characterization of arbitrary saturation judgments $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$, we can easily deduce a characteristic property of the environments appearing at the end of the saturation phase, that is in judgments of the form $\Gamma^{\mathsf{at}\prime\prime}; \emptyset \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$.

Definition 11.3.1 Saturated environment**.**
We say that Γ^{at} is *saturated* if $\Gamma^{\mathsf{at}} \Downarrow \langle P\rangle^-$ implies $\Gamma^{\mathsf{at}} \Uparrow P$.

Corollary 11.3.2 (Saturation)**.**
If a saturated proof starts from a judgment of the form

$$\emptyset; \Gamma^{\mathsf{at}}_0 \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}} \qquad \textit{or} \qquad \emptyset; \Sigma_0 \vdash_{\mathit{sinv}} t : N \mid Q^{\mathsf{at}}$$

then for any sub-derivation of the form

$$\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$$

the environment Γ^{at} *is saturated.*

11.3.2. Saturated consistency

In particular, Theorem 11.3.1 (Saturation) lets us deduce that the context at the end of a saturation phase are always consistent: if there was a proof of 0 deducible from the initial context, it would have been found during saturation, and the proof would have ended on the following invertible phase.

Lemma 11.3.3 (Saturated consistency)**.**
If Γ^{at} *is saturated, then* $\Gamma^{\mathsf{at}} \nvdash 0$.

Proof. Let us consider the shape of contradiction proofs $\Gamma^{\mathsf{at}} \vdash 0$ – this context has only negative or atomic formulas. By completeness of focusing, if a proof of 0 exists, then a focused proof exist. We call such proofs *contradiction proofs*. Focused proofs Figure 10.1 (Focused natural deduction, with explicit shifts) of 0 have a particular structure: an inversion phase on this typing stops immediately, then no right-focused phase is possible (0 has no constructor), so the only possible proof step is of the form

$$\text{NAT-FOC-LEFT-RELEASE-SHIFT}\ \frac{\Gamma^{\mathsf{at}} \Downarrow \langle Q\rangle^- \qquad \Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \emptyset \mid 0}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} 0}$$

We know that the following invertible proof, in the right premise, may have zero, one or more non-invertible subproofs, and those will all be of the form $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}'} \vdash_{\mathsf{foc}} 0$ for some new context $\Gamma^{\mathsf{at}'}$. In particular, all immediate non-invertible subproofs on the right-hand side are themselves proofs of 0.

From our hypothesis that Γ^{at} is saturated we know that $\Gamma^{\mathsf{at}} \Downarrow \langle Q^{\mathsf{at}} \rangle^{-}$ implies $\Gamma^{\mathsf{at}} \Uparrow \langle Q^{\mathsf{at}} \rangle^{-}$. By Lemma 10.5.1 (Strong positive neutral substitution), we know that $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} 0$ is admissible, and that one of the focused subproofs of our derivation of $\Gamma^{\mathsf{at}}; Q \vdash_{\mathsf{inv}} \emptyset \mid 0$ proves this statement. In other words, any focused proof of $\Gamma^{\mathsf{at}'} \vdash_{\mathsf{foc}} 0 :$ can be reduced to a strictly smaller subproof, also of the form $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} 0 :$. There cannot exist a proof of $\Gamma^{\mathsf{at}'} \vdash_{\mathsf{foc}} 0 :$. □

In particular, we can prove that the saturated logic is canonical for inconsistent contexts.

Theorem 11.3.4 (Inconsistent canonicity)**.**
If $\Gamma^{\mathsf{at}} \vdash 0$*, then for any* f, f' *such that* $\emptyset; \Gamma^{\mathsf{at}} \vdash_{\mathsf{sat}} f, f' : Q^{\mathsf{at}}$ *we have* $f \approx_{\mathtt{icc}} f'$*.*

Proof. This result is simply proved by simultaneous induction on f and f'.

The invertible rules are purely type-directed, so we can assume modulo ($\approx_{\mathtt{icc}}$) that both terms start with the same invertible constructors.

The saturation step is purely type-directed: two terms under the same contexts will saturate on the exact same set of neutrals.

The only rules on which the head of f, f' may differ syntactically are SAT-UP and SAT-DOWN, but those cannot be used in those terms as the assumption that Γ^{at} is inconsistent would contradict Lemma 11.3.3 (Saturated consistency). □

11.4. Canonicity of saturated proofs

To prove the main theorems on saturating focused logic, we describe how to convert a focused λ-term into a valid saturated proof derivation. This can be done either as a small-step rewrite process, or as a big-step transformation. The small-step rewrite would be very similar to the *preemptive rewriting* relation of Scherer [2015a]; we will here use a big-step transformation, as in Scherer and Rémy [2015], by defining in Figure 11.2 a type-preserving translation judgments of the form $\Gamma; \Sigma \vdash_{\mathit{sinv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$, which turns a focused term t into a valid *saturating* focused term t'.

Backward search for saturated proofs corresponds to enumerating the canonical inhabitants of a given type. Our translation can be seen as a restriction of this proof search process, searching inside the $\beta\eta$-equivalence class of t. Because saturating proof terms are canonical (to be shown), the restricted search is deterministic – modulo invertible commuting conversions.

Compared to the focusing translation of Figure 10.6 used to prove completeness of focusing with respect to the non-focused λ-calculus in Section 10.3 (Focusing completeness by big-step translation), this rewriting is simpler as it starts from an already-focused proof whose overall structure is not modified. The only real change is moving from the left-focusing rule REW-FOC-ELIM to the saturating rule REW-SAT. Instead of allowing to cut on any neutral subterm, we enforce a maximal cut on exactly all the neutrals of t that can be typed in the current environment. Because we know that "old" neutrals have already been cut and replaced with free variables earlier in the translation, this is fact respects the saturation condition.

Compared to the focusing translation, the termination of this translation is immedate induction: thanks to the focused structure of the input, every recursive call happens on a strictly smaller term. In the REW-SAT rule, the recusive call is on $t[\bar{x}/\bar{n}]$, which is not be strictly smaller if the $\bar{n}$ are variables, which can happen for $x : \langle P \rangle^{-}$. But this case is only possible when x is in the "new" context, as this neutral uses no other variable that could be in the new context; and this variable gets replaced by a variable in the post-saturation new context at the strictly smaller type P, so it can only happen finitely many times.

Figure 11.2.: Saturation translation

REW-SINV-LAM
$$\frac{\Gamma^{\mathsf{at}}; \Sigma, x : P \vdash_{\mathit{sinv}} t \rightsquigarrow t' : N \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} \lambda x.\, t \rightsquigarrow \lambda x.\, t' : P \to A \mid}$$

REW-SINV-PAIR
$$\frac{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t_1 \rightsquigarrow t'_1 : N_1 \mid \quad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t_2 \rightsquigarrow t'_2 : N_2 \mid}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} (t_1, t_2) \rightsquigarrow (t'_1, t'_2) : N_1 \times N_2 \mid}$$

REW-SINV-CASE
$$\frac{\begin{array}{c}\Gamma^{\mathsf{at}}; \Gamma, x : P_1 \vdash_{\mathit{sinv}} t_1 \rightsquigarrow t'_1 : N \mid Q^{\mathsf{at}} \\ \Gamma^{\mathsf{at}}; \Gamma, x : P_2 \vdash_{\mathit{sinv}} t_2 \rightsquigarrow t'_2 : N \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; \Gamma, x : P_1 + P_2 \vdash_{\mathit{sinv}} \mathtt{match}\ x\ \mathtt{with} \left|\begin{array}{l}\sigma_1\ x \to t_1 \\ \sigma_2\ x \to t_2\end{array}\right. \rightsquigarrow \mathtt{match}\ x\ \mathtt{with} \left|\begin{array}{l}\sigma_1\ x \to t'_1 \\ \sigma_2\ x \to t'_2\end{array}\right. : N \mid Q^{\mathsf{at}}}$$

REW-SINV-TRIVIAL
$$\frac{}{\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} () \rightsquigarrow () : 1 \mid}$$

REW-SINV-ABSURD
$$\frac{}{\Gamma^{\mathsf{at}}; \Sigma, x : 0 \vdash_{\mathit{sinv}} \mathtt{absurd}(x) \rightsquigarrow \mathtt{absurd}(x) : N \mid Q^{\mathsf{at}}}$$

REW-SINV-SAT
$$\frac{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} f \rightsquigarrow f' : (P^{\mathsf{at}} \mid Q^{\mathsf{at}})}{\Gamma^{\mathsf{at}}; \left\langle \Gamma^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}} \vdash_{\mathit{sinv}} f \rightsquigarrow f' : \left\langle P^{\mathsf{at}} \right\rangle^{-\mathsf{at}} \mid Q^{\mathsf{at}}}$$

REW-SAT-INTRO
$$\frac{\Gamma^{\mathsf{at}} \vdash p \rightsquigarrow p' \Uparrow P}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} p \rightsquigarrow p' : P}$$

REW-SAT-ATOM
$$\frac{\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow X}{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} n \rightsquigarrow n' : X}$$

REW-SAT
$$\frac{\begin{array}{c}(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash n \Downarrow P)\}) \\ \forall n \in t,\ (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash n \Downarrow P) \implies n \in \bar{n} \\ \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t[\bar{x}/\bar{n}] \rightsquigarrow t' : Q^{\mathsf{at}} \mid\end{array}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} t \rightsquigarrow \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t' : Q^{\mathsf{at}}}$$

REW-SINTRO-SUM
$$\frac{\Gamma^{\mathsf{at}} \vdash p \rightsquigarrow p' \Uparrow A_i}{\Gamma^{\mathsf{at}} \vdash \sigma_i\ p \rightsquigarrow \sigma_i\ p' \Uparrow A_1 + A_2}$$

REW-SINTRO-END
$$\frac{\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathit{sinv}} t \rightsquigarrow t' : N \mid}{\Gamma^{\mathsf{at}} \vdash t \rightsquigarrow t' \Uparrow \langle N \rangle^{+}}$$

REW-SINTRO-AXIOM
$$\frac{(x : X^{+}) \in \Gamma^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash x \rightsquigarrow x \Uparrow X^{+}}$$

REW-SELIM-PAIR
$$\frac{\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow A_1 \times A_2}{\Gamma^{\mathsf{at}} \vdash \pi_i\ n \rightsquigarrow \pi_i\ n' \Downarrow A_i}$$

REW-SELIM-ARR
$$\frac{\Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow P \to N \qquad \Gamma^{\mathsf{at}} \vdash p \rightsquigarrow p' \Uparrow P}{\Gamma^{\mathsf{at}} \vdash n\ p \rightsquigarrow n'\ p' \Downarrow N}$$

REW-SELIM-START
$$\frac{(x : N) \in \Gamma^{\mathsf{at}}}{\Gamma^{\mathsf{at}} \vdash x \rightsquigarrow x \Downarrow N}$$

$$(\mathtt{let}\ x = n\ \mathtt{in}\ t)[y/n] \stackrel{\mathsf{def}}{=} t[y/x][y/n]$$

Assumptions on the selection function The REW-SAT rule makes an interesting assumption on the selection function:

REW-SAT
$$\frac{\begin{array}{c}(\bar{n}, \bar{P}) \stackrel{\mathsf{def}}{=} \mathtt{Select}_{\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}}(\{(n, P) \mid (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash n \Downarrow P)\}) \\ \forall n \in t,\ (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime} \vdash n \Downarrow P) \implies n \in \bar{n} \\ \Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}; \bar{x} : \bar{P} \vdash_{\mathit{sinv}} t[\bar{x}/\bar{n}] \rightsquigarrow t' : Q^{\mathsf{at}} \mid\end{array}}{\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} t \rightsquigarrow \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t' : Q^{\mathsf{at}}}$$

This rule can only be applied if all the neutrals of the translated n that are typeable in the present context happen to be part of the neutrals selected for saturation. This is a requirement that most selection functions will *not* meet: for any choice of selection

functions there are many t such that no valid derivation of the form $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ exist for any t'.

However, for any t we can construct some – and in fact many – valuation functions for which such a $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ exists for some t'. If we start from an arbitrary selection function satisfying SELECT-SPECIF, we can build another selection function that meets this requirement by simply adding all the neutral subterms that happen during this translation. As we are only adding new neutrals, the resulting selection still satisfies SELECT-SPECIF. Any finite derivation of the translation judgment will only add finitely many new neutrals this way, which means that the returned selection function still returns finite sets of neutrals for each context.

We say that a selection function is *adequate* for some term $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$ if it does select all neutrals of t, in the sense that there exists a derivation $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ for some t'. Note that different adequate selection functions will result in different translations t'. In general we will implictly assume that the selection function is adequate for the terms considered.

Lemma 11.4.1 (Translation soundness)**.**
If $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$ *and* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ *then* $t \approx_{\beta\eta} t'$.

Proof. By immediate induction. □

Lemma 11.4.2 (Translation validity)**.**
Suppose that $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$ *holds in the focused logic, and that* t *has no "old" neutral: for no* $n \in t$ *do we have* $\Gamma^{\mathsf{at}} \vdash n \Downarrow \langle P \rangle^{-}$. *Then,* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$ *implies that* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t' : N \mid Q^{\mathsf{at}}$ *in the saturated focusing logic.*

Proof. The restriction on "old" neutrals is necessary because the REW-SAT rule would not know what to do on such old neutrals – it assumes that they were all substituted away for fresh variable in previous inference steps.

With this additional invariant the proof goes by immediate induction. In the REW-SAT rule, this invariant tells us that the bindings satisfy the freshness condition of the SAT rule of saturated logic, and because we select *all* such fresh bindings we preserve the property that the extended context $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}$ has no old neutrals either. □

Lemma 11.4.3 (Translation determinism)**.**
If the selection function is adequate for $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$, *then there exists a* unique t' *such that* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$.

Proof. By immediate induction. □

Note that the indeterminacy of invertible step ordering is still present in saturating focused logic: a *non-focused* term t may have several saturated translations that only equal upto commuting conversions ($\approx_{\mathtt{icc}}$). However, there is no more variability than in the focused proof of the non-saturating focused logic; because we translate from those, we can respect the ordering choices that are made, and the *translation* is thus fully deterministic.

Theorem 11.4.4 (Computational completeness of saturating focused logic)**.**
If we have $\emptyset; \Sigma \vdash_{\mathsf{inv}} t : N \mid Q^{\mathsf{at}}$ *in the non-saturating focused logic, then for an adequate saturation function and some* $t' \approx_{\beta\eta} t$ *we have* $\emptyset; \Sigma \vdash_{\mathsf{sinv}} t' : N \mid Q^{\mathsf{at}}$ *in the saturating focused logic.*

Proof. This is an immediate corollary of the previous results. For an adequate selection function, there is a unique t' such that $\emptyset; \Sigma \vdash_{\mathsf{sinv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$. By Lemma 11.4.2 (Translation validity) we have that $\emptyset; \Sigma \vdash_{\mathsf{sinv}} t' : N \mid Q^{\mathsf{at}}$ in the saturating focused calculus – the condition that there be no old neutrals is trivially true for the empty context $\emptyset$. Finally, by Lemma 11.4.1 (Translation soundness) we have that $\lfloor t \rfloor_{\mathtt{foc}} \approx_{\beta\eta} \lfloor u \rfloor_{\mathtt{foc}}$. □

Lemma 11.4.5 (Determinacy of saturated translation)**.**
For any u_1, u_2, *if we have* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow u_1 : N \mid Q^{\mathsf{at}}$ *and* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow u_2 : N \mid Q^{\mathsf{at}}$ *then we have* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} u_1 \rightsquigarrow r_1 : N \mid Q^{\mathsf{at}}$ *and* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} u_2 \rightsquigarrow r_2 : N \mid Q^{\mathsf{at}}$ *with* $r_1 \approx_{\mathtt{icc}} r_2$.

Proof sketch. There are only two sources of non-determinism in the focused translation:

- an arbitrary choice of the order in which to apply the invertible rules

- a neutral `let`-extrusion may happen at any point between the first scope where it is well-defined to the lowest common ancestors of all uses of the neutral in the term.

The first source of non-determinism gives $(\approx_{\mathtt{icc}})$-equivalent derivations. The second disappears when doing the saturating translation, which enforces a unique placement of `let`-extrusions at the first scope where the strictly positive neutrals are well-defined.

As a result, two focused translations of the same term may differ in both aspect, but their saturated translations differ at most by $(\approx_{\mathtt{icc}})$. ⊔

Definition 11.4.1 Normalization by saturation**.**
For a well-typed (non-focused) λ-term $\lfloor\Gamma^{\mathsf{at}}\rfloor_{\pm}, \lfloor\Sigma\rfloor_{\pm} \vdash t : \lfloor(N \mid Q^{\mathsf{at}})\rfloor_{\pm}$, we write $\mathsf{NF}_{\mathsf{sat}}(t)$ for any saturated term t'' such that

$$\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_{\beta}(t) \rightsquigarrow t' : N \mid Q^{\mathsf{at}} \qquad\qquad \Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathit{sinv}} t' \rightsquigarrow t'' : N \mid Q^{\mathsf{at}}$$

Note that all possible t'' are equal modulo $(\approx_{\mathtt{icc}})$, by Lemma 11.4.5 (Determinacy of saturated translation).

Lemma 11.4.6 (Saturation congruence)**.**
For any context $C[\Box]$ and term t we have

$$\mathsf{NF}_{\mathsf{sat}}(C[t]) \approx_{\mathtt{icc}} \mathsf{NF}_{\mathsf{sat}}(C[\mathsf{NF}_{\mathsf{sat}}(t)])$$

Proof. We reason by induction on $C[\Box]$. Without loss of generality we will assume $C[\Box]$ atomic. It is either a redex-forming context

$$\Box\ u \qquad\qquad \pi_k\ \Box \qquad\qquad \mathtt{match}\ \Box\ \mathtt{with} \mid \begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}$$

or a non-redex forming context

$$u\ \Box \qquad\qquad \sigma_i\ \Box$$

$$(u, \Box) \qquad\qquad (\Box, u)$$

$$\mathtt{match}\ u\ \mathtt{with} \mid \begin{array}{l}\sigma_1\ x \to \Box\\ \sigma_2\ x \to u_2\end{array} \qquad\qquad \mathtt{match}\ u\ \mathtt{with} \mid \begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to \Box\end{array}$$

If it is a non-context-forming redex, then we have $\mathsf{NF}_{\beta}(C[t]) = C[\mathsf{NF}_{\beta}(t)]$. The focused and saturated translations then work over $C[\mathsf{NF}_{\beta}(t)]]$ just as they work with $\mathsf{NF}_{\beta}(t)$, possibly adding bindings before $C[\Box]$ instead of directly on the (translations of) $\mathsf{NF}_{\beta}(t)$. The results are in the $(\approx_{\mathtt{icc}})$ relation.

The interesting case is when $C[\Box]$ is a redex-forming context: a reduction may overlap the frontier between $C[\Box]$ and the plugged term. In that case, we will reason on the saturated normal form $\mathsf{NF}_{\mathsf{sat}}(t)$. Thanks to the strongly restricted structure of focused and saturated normal form, we have precise control over the possible reductions.

Application case $C[\Box] \stackrel{\mathsf{def}}{=} \Box\ u$**.** We prove that there exist t' such that $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : P \to N \mid$, and a r such that both $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t\ u \rightsquigarrow r : N \mid$ and $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t'\ u \rightsquigarrow r : N \mid$ hold. This implies the desired result – after translation of r into a saturated term. The proof proceeds by induction on the derivation $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t\ u \rightsquigarrow r : N \mid$ (we know that all possible such translations have finite derivations).

To make the proof easier to follow, we introduce the notation $\mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma \vdash t)$ to denote a focused translation t' of $\mathsf{NF}_{\beta}(t)$ (that is, $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow t' : N \mid Q^{\mathsf{at}}$, where N, Q^{at} are uniquely defined by $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} t' : N \mid Q^{\mathsf{at}}$)). This notation should be used with care because it is not well-determined: there are many such possible translations. Statements using the notation should be interpreted existentially: $P(\mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma \vdash t))$ means that

there exists a translation t' of t such that $P(t')$ holds. The current goal (whose statement took the full previous paragraph) can be rephrased as follows:

$$\mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma \vdash t\ u) = \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma \vdash \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma \vdash t)\ u)$$

We will simply write $\mathsf{NF}_{\mathsf{foc}}(t)$ when the typing environment of the translation is clear from the context.

If Σ contains a sum type, it is of the form $(\Sigma', x : C_1 + C_2)$ and we can get by induction hypothesis that

$$\mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma', x : C_i \vdash t\ u) = \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Sigma', x : C_i \vdash \mathsf{NF}_{\mathsf{foc}}(t)\ u)$$

for i in $\{1, 2\}$, from which we can conclude with

$$\begin{array}{rl}
& \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Gamma', x : C_1 + C_2 \vdash t\ u) \\
= & \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Gamma', x : C_1 \vdash t\ u) \\ \sigma_2\ x \to \ldots C_2 \ldots \end{array} \\
= & \texttt{match}\ x\ \texttt{with} \mid \begin{array}{l} \sigma_1\ x \to \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Gamma', x : C_1 \vdash \mathsf{NF}_{\mathsf{foc}}(t)\ u) \\ \sigma_2\ x \to \ldots C_2 \ldots \end{array} \\
= & \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Gamma', x : C_1 + C_2 \vdash \mathsf{NF}_{\mathsf{foc}}(t)\ u)
\end{array}$$

If Σ contains the empty type $x : 0$ then, for any t, $\mathsf{NF}_{\mathsf{foc}}(X^-; \Sigma \ni x : 0 \vdash t)$ is equal (modulo $(\approx_{\mathtt{icc}})$) to $\texttt{absurd}(x)$ and the result is immediate.

Otherwise Σ is of the form $\langle \Gamma^{\mathsf{at}'} \rangle^{+\mathsf{at}}$.

Any focused translation of t at type $N \to P$ is thus necessarily of the form $\lambda x.\, \mathsf{NF}_{\mathsf{foc}}(t\ x)$. In particular, any $\mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t)\ u)$, that is, any $\mathsf{NF}_{\mathsf{foc}}((\lambda x.\, \mathsf{NF}_{\mathsf{foc}}(t\ x))\ u)$, is equal by stability of the translation to β-reduction to a term of the form $\mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t\ x)[u/x])$. On the other hand, $\mathsf{NF}_{\mathsf{foc}}(t\ u)$ can be of several different forms.

Note that $t\ u$ is translated at the same type as $t\ x$. In particular, if this is a negative type, they both begin with a suitable η-expansion (of a product or function type); in the product case for example, we have $\mathsf{NF}_{\mathsf{foc}}(t\ u) = (\mathsf{NF}_{\mathsf{foc}}(\pi_1\ (t\ u)), \mathsf{NF}_{\mathsf{foc}}(\pi_2\ (t\ u)))$, and similarly $\mathsf{NF}_{\mathsf{foc}}(t\ x) = (\mathsf{NF}_{\mathsf{foc}}(\pi_1\ (t\ x)), \mathsf{NF}_{\mathsf{foc}}(\pi_2\ (t\ x)))$: we can then conclude by induction hypothesis on those smaller pairs of terms $\pi_i\ (t\ u)$ and $\pi_i\ (t\ x)$ for i in $\{1, 2\}$. We can thus assume that $t\ u$ is of positive or atomic type, and will reason by case analysis on the β-normal form of t.

If $\mathsf{NF}_\beta(t)$ is of the form $\lambda x.\, t'$ for some t', then $\mathsf{NF}_{\mathsf{foc}}(t\ u)$ is equal to $\mathsf{NF}_{\mathsf{foc}}((\lambda x.\, t')\ u)$, that is, $\mathsf{NF}_{\mathsf{foc}}(t'[u/x])$. Finally, we have $\mathsf{NF}_{\mathsf{foc}}(t\ x) = \mathsf{NF}_{\mathsf{foc}}((\lambda x.\, t')\ x) = \mathsf{NF}_{\mathsf{foc}}(t')$, which let us conclude from our assertion that $\mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t)\ u)$ is equal to $\mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t\ x)[u/x])$.

If $\mathsf{NF}_\beta(t)$ contains a strictly positive neutral subterm $n : P$ (this is in particular always the case when it is of the form $\texttt{match}\ t'\ \texttt{with}\ \ldots$, we can `let`-extrude it to get

$$\mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}'} \vdash t) \qquad = \texttt{let}\ x = \mathsf{NF}_{\mathsf{foc}}(n)\ \texttt{in}\ \mathsf{NF}_{\mathsf{foc}}(\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}'}; x : P \vdash t[x/n])$$

But then $\mathsf{NF}_{\mathsf{foc}}(n) : P$ is also a strictly positive neutral subterm of $(\texttt{let}\ x = \mathsf{NF}_{\mathsf{foc}}(n)\ \texttt{in}\ \ldots)$, so we have

$$\begin{array}{rl}
& \mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t)\ u) \\
= & \mathsf{NF}_{\mathsf{foc}}((\texttt{let}\ x = \mathsf{NF}_{\mathsf{foc}}(n)\ \texttt{in}\ \mathsf{NF}_{\mathsf{foc}}(t[x/n]))\ u) \\
= & \texttt{let}\ x = \mathsf{NF}_{\mathsf{foc}}(n)\ \texttt{in}\ \mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t[x/n])\ u[x/n]) \\
= & \texttt{let}\ x = \mathsf{NF}_{\mathsf{foc}}(n)\ \texttt{in}\ \mathsf{NF}_{\mathsf{foc}}((t\ u)[x/n]) \\
= & \mathsf{NF}_{\mathsf{foc}}(t\ u)
\end{array}$$

Finally, if $\mathsf{NF}_\beta(t)$ contains no strictly positive neutral subterm, the rule REW-UP-ARROW applies: $\mathsf{NF}_{\mathsf{foc}}(t\ u)$ is of the form $n\ \mathsf{NF}_{\mathsf{foc}}(u)$, where $n \stackrel{\mathsf{def}}{=} \mathsf{NF}_{\mathsf{foc}}(t)$. In this case we also have $\mathsf{NF}_{\mathsf{foc}}(t\ x) = n\ x$, and thus

$$\begin{array}{rl}
& \mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t)x\ ) \\
= & \mathsf{NF}_{\mathsf{foc}}(\mathsf{NF}_{\mathsf{foc}}(t\ x)[u/x]) \\
= & \mathsf{NF}_{\mathsf{foc}}(n\ u) \\
= & \mathsf{NF}_{\mathsf{foc}}(t\ u)
\end{array}$$

Projection case $C[\square] \stackrel{\mathsf{def}}{=} \pi_i\ \square$ This case is proved in the same way as the application case: after some sum eliminations, the translation of t is an η-expansion of the product, which is related to the translations $\mathsf{NF}_{\mathsf{foc}}(\pi_i\ t)$, which either reduce the product or build a neutral term $\pi_i\ n$ after introducing some `let`-bindings.

Sum elimination case Reusing the notations of the application case, show that

$$\mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ t\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.) = \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \mathsf{NF}_{\mathsf{foc}}(t)\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)$$

In the case of the function application or pair projection, the congruence proof uses the fact that the translation of t (of function or product type) necessarily starts with a λ-abstraction or pair construction – in fact, we follow the incremental construction of the first invertible phase, in particular we start by eliminating sums from the context.

In the case of the sum elimination, we must follow the translation into focused form further: we know the first invertible phase of $\mathsf{NF}_{\mathsf{foc}}(t)$ may only have sum-eliminations (pair or function introductions would be ill-typed as t has a sum type $A + B$).

As in the application case, we can then extrude neutrals from t, and the extrusion can be mirrored in both $\mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ t\ \texttt{with}\ \ldots)$ and $\mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \mathsf{NF}_{\mathsf{foc}}(t)\ \texttt{with}\ \ldots)$. Finally, we reason by case analysis on $\mathsf{NF}_\beta(t)$.

If $\mathsf{NF}_\beta(t)$ is of the form $\sigma_i\ t'$, then we have

$$\begin{array}{rl} & \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \mathsf{NF}_{\mathsf{foc}}(t)\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)\\ = & \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \sigma_i\ \mathsf{NF}_{\mathsf{foc}}(t')\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)\\ = & \mathsf{NF}_{\mathsf{foc}}(u_i[\mathsf{NF}_{\mathsf{foc}}(t')/x]) \end{array}$$

and

$$\begin{array}{rl} & \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ t\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)\\ = & \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \mathsf{NF}_\beta(t)\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)\\ = & \mathsf{NF}_{\mathsf{foc}}(\texttt{match}\ \sigma_i\ t'\ \texttt{with}\ \left|\begin{array}{l}\sigma_1\ x \to u_1\\ \sigma_2\ x \to u_2\end{array}\right.)\\ = & \mathsf{NF}_{\mathsf{foc}}(u_i[t'/x]) \end{array}$$

What is left to prove is that $\mathsf{NF}_{\mathsf{foc}}(u_i[\mathsf{NF}_{\mathsf{foc}}(t')/x]) = \mathsf{NF}_{\mathsf{foc}}(u_i[t'/x])$ but that is equivalent (by stability of the focusing translation by β-reduction) to $\mathsf{NF}_{\mathsf{foc}}((\lambda x.\, u_i)\ \mathsf{NF}_{\mathsf{foc}}(t')) = \mathsf{NF}_{\mathsf{foc}}((\lambda x.\, u_i)\ t')$, which is exactly the application case proved previously.

This is in fact the only possible case: when all strictly positive neutrals have been extruded, then $\mathsf{NF}_\beta(t)$ is necessarily an injection $\sigma_i\ t'$ (already handled) or a variable x (this corresponds to the case where t itself reduces to a strictly positive neutral), but this variable would be in the context and of strictly positive type, so this case is already handled as well.

Absurd case $\texttt{absurd}(x)$ The normal-form of $(t : 0)$ cannot start with a constructor, as there are none at this type. After neutral extrusion, it is thus necessarily a variable $x : 0$; both sides are thus immediately equated with $\texttt{absurd}(x)$ during the invertible translation phase following normalization. □

Theorem 11.4.7 (Canonicity of saturating focused logic)**.**
If we have $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} t : N \mid Q^{\mathsf{at}}$ *and* $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{sinv}} u : N \mid Q^{\mathsf{at}}$ *in saturating focused logic with* $t \not\approx_{\mathtt{icc}} u$*, then* $t \not\approx_{\beta\eta} u$*.*

Proof. By contrapositive: if $t \approx_{\beta\eta} u$ (that is, if $\lfloor t \rfloor_{\mathtt{foc}} \approx_{\beta\eta} \lfloor u \rfloor_{\mathtt{foc}}$) then $t \approx_{\mathtt{icc}} u$.

The difficulty to prove this statement is that $\beta\eta$-equivalence does not preserve the structure of saturated proofs: an equivalence proof may go through intermediate steps that are neither saturated nor focused or in β-normal form.

We will thus go through an intermediate relation, which we will write $(\approx_{\mathtt{sat}})$, defined as follows on arbitrary well-typed lambda-terms:

$$\frac{\begin{array}{c}\emptyset; \Sigma \vdash_{\mathsf{inv}} t : A \qquad \emptyset; \Sigma \vdash_{\mathsf{inv}} u : A \qquad \emptyset \vdash_{\mathsf{inv}} \Sigma : \mathsf{NF}_\beta(t) \rightsquigarrow t' A \\ \emptyset \vdash_{\mathsf{inv}} \Sigma : \mathsf{NF}_\beta(u) \rightsquigarrow u' A \qquad \emptyset; \Sigma \vdash_{\mathit{sinv}} t' \rightsquigarrow t'' : A \mid \qquad \emptyset; \Sigma \vdash_{\mathit{sinv}} u' \rightsquigarrow u'' : A \mid \\ t'' \approx_{\mathtt{icc}} u''\end{array}}{\Sigma \vdash t \approx_{\mathtt{sat}} u : A}$$

It follows from the previous results that if $t \approx_{\mathtt{sat}} u$, then $t \approx_{\beta\eta} u$. We will now prove the converse inclusion: if $t \approx_{\beta\eta} u$ (and they have the same type), then $t \approx_{\mathtt{sat}} u$ holds. In the particular case of terms that happen to be (let-expansions of) valid saturated focused derivations, this will tell us in particular that $t \approx_{\mathtt{icc}} u$ holds – the desired result.

The computational content of this canonicity proof is an equivalence algorithm: $(\approx_{\mathtt{sat}})$ is a decidable way to check for $\beta\eta$-equality, by normalizing terms to their saturated (or maximally multi-focused) structure.

β-reductions It is immediate that $(\approx_\beta)$ is included in $(\approx_{\mathtt{sat}})$. Indeed, if $t \approx_\beta u$ then $\mathsf{NF}_\beta(t) = \mathsf{NF}_\beta(u)$ and $t \approx_{\mathtt{sat}} u$ is trivially satisfied.

Negative η-expansions We can prove that if $t \approx_\eta u$ through one of the equations

$$(t : A \to B) \approx_\eta \lambda x.\, t\ x \qquad\qquad (t : A \times B) \approx_\eta (\pi_1\ t, \pi_2\ t)$$

then both t and u are rewritten in the same focused proof r. We have both $\emptyset; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow r : N \mid$ and $\emptyset; \Sigma \vdash_{\mathsf{inv}} u \rightsquigarrow r : N \mid$, and thus $t \approx_{\mathtt{sat}} u$. Indeed we have:

$$\frac{\emptyset; \Sigma, x : P \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t\ x) \rightsquigarrow r : N \mid}{\emptyset; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow \lambda x.\, r : P \to N \mid}$$

$$\frac{\mathsf{NF}_\beta((\lambda x.\, t\ x)\ x) = \mathsf{NF}_\beta(t\ x) \qquad \emptyset; \Sigma, x : P \vdash_{\mathsf{inv}} \mathsf{NF}_\beta((\lambda x.\, t\ x)\ x) \rightsquigarrow r : N \mid}{\emptyset; \Sigma \vdash_{\mathsf{inv}} \lambda x.\, t\ x \rightsquigarrow \lambda x.\, r : P \to N \mid}$$

and

$$\frac{\forall i \in \{1,2\}, \quad \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(\pi_i\ t) \rightsquigarrow r_i : N_i \mid}{\emptyset; \Sigma \vdash_{\mathsf{inv}} t \rightsquigarrow (r_1, r_2) : (N_1, N_2) \mid}$$

$$\frac{\pi_i\ (\pi_1\ t, \pi_2\ t) = t \qquad \forall i \in \{1,2\}, \quad \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(\pi_i\ (\pi_1\ t, \pi_2\ t)) \rightsquigarrow r_i : N_i \mid}{\emptyset; \Sigma \vdash_{\mathsf{inv}} (\pi_1\ t, \pi_2\ t) \rightsquigarrow (r_1, r_2) : N_1 \times N_2 \mid}$$

Positive η-expansion: sum type The interesting case is the positive η-expansion

$$\forall C[\Box : \lfloor P_1 \rfloor_\pm + \lfloor P_2 \rfloor_\pm], \quad C[t] \approx_\eta \mathtt{match}\ t\ \mathtt{with} \left| \begin{array}{l} \sigma_1\ x \to C[\sigma_1\ x] \\ \sigma_2\ x \to C[\sigma_2\ x] \end{array} \right.$$

We do a case analysis on the (weak head) β-normal form of t. If it is an injection of the form $\sigma_i\ t'$, then the equation becomes true by a simple β-reduction:

$$\mathtt{match}\ \sigma_i\ t'\ \mathtt{with} \left| \begin{array}{l} \sigma_1\ x \to C[\sigma_1\ x] \\ \sigma_2\ x \to C[\sigma_2\ x] \end{array} \right. \rightsquigarrow_\beta C[\sigma_i\ t']$$

Otherwise the β-normal form of t is a term of sum type that does not start with an injection. In particular, $\mathsf{NF}_\beta(t)$ is not reduced when reducing the whole term $C[t]$ (only

possibly duplicated): for some multi-hole context $C'[x]$ we have $\mathsf{NF}_\beta(C[t]) = C'[\mathsf{NF}_\beta(t)]$ and

$$\mathsf{NF}_\beta(\texttt{match}\ t\ \texttt{with} \mid \sigma_1\ x \to C[\sigma_1\ x] \mid \sigma_2\ x \to C[\sigma_2\ x]) \quad = \quad \texttt{match}\ \mathsf{NF}_\beta(t)\ \texttt{with} \mid \sigma_1\ x \to C'[\sigma_1\ x] \mid \sigma_2\ x \to C'[\sigma_2\ x]$$

Without loss of generality, we can assume that $\mathsf{NF}_\beta(t)$ is a neutral term. Indeed, if it is not, it starts with a (possibly empty) series of non-invertible elimination forms, applied to a positive elimination – which is itself either a neutral or of this form. It eventually contains a neutral strict subterm of strictly positive type valid in the current scope. The focused translation can then cut on this strictly positive neutral. It this neutral is of empty type 0, both terms get translated to an $\texttt{absurd}(\_)$ construction so they are $(\approx_{\mathtt{icc}})$-related. If it is a sum type, the translation splits on it, and replace occurrences of this neutral with either $\sigma_1\ z$ or $\sigma_2\ z$ for some fresh z. This can be done on both terms equated by the η-equivalence for sums, and returns (two pairs of) η-equivalent terms with one less strictly possible neutral strict subterm.

Let $n \stackrel{\text{def}}{=} \mathsf{NF}_\beta(t)$. It remains to show that the translations of $C'[n]$ is equal modulo $(\approx_{\mathtt{icc}})$ to the translation of $\texttt{match}\ n\ \texttt{with} \mid \sigma_1\ x \to C'[\sigma_1\ x] \mid \sigma_2\ x \to C'[\sigma_2\ x]$. In fact, we show that they translate to the same focused proof:

$$\dfrac{\dfrac{\Gamma^{\mathsf{at}} \vdash n : P_1 + P_2 \qquad \Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow P_1 + P_2 \qquad \Gamma^{\mathsf{at}}; x : P_1 \vdash_{\mathsf{inv}} C'[\sigma_1\ x] \rightsquigarrow r_1 : \emptyset \mid Q^{\mathsf{at}} \qquad \Gamma^{\mathsf{at}}; x : P_2 \vdash_{\mathsf{inv}} C'[\sigma_2\ x] \rightsquigarrow r_2 : \emptyset \mid Q^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; x : P_1 + P_2 \vdash_{\mathsf{inv}} C'[x] \rightsquigarrow \texttt{match}\ x\ \texttt{with} \mid \sigma_1\ x \to r_1 \mid \sigma_2\ x \to r_2 : \emptyset \mid Q^{\mathsf{at}}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} C'[n] \rightsquigarrow \texttt{let}\ x = n\ \texttt{in match}\ x\ \texttt{with} \mid \sigma_1\ x \to r_1 \mid \sigma_2\ x \to r_2 : Q^{\mathsf{at}}}$$

$$\dfrac{\dfrac{\begin{array}{c}\Gamma^{\mathsf{at}} \vdash n : P_1 + P_2 \\ \Gamma^{\mathsf{at}} \vdash n \rightsquigarrow n' \Downarrow A + B \qquad \mathsf{NF}_\beta(\texttt{match}\ \sigma_i\ x\ \texttt{with} \mid \sigma_1\ x \to C'[\sigma_1\ x] \mid \sigma_2\ x \to C'[\sigma_2\ x]) = C'[\sigma_i\ x] \\ \Gamma^{\mathsf{at}}; x : P_1 \vdash_{\mathsf{inv}} C'[\sigma_1\ x] \rightsquigarrow r_1 : \emptyset \mid Q^{\mathsf{at}} \qquad \Gamma^{\mathsf{at}}; x : P_2 \vdash_{\mathsf{inv}} C'[\sigma_2\ x] \rightsquigarrow r_2 : \emptyset \mid Q^{\mathsf{at}}\end{array}}{\Gamma^{\mathsf{at}}; x : P_1 + P_2 \vdash_{\mathsf{inv}} \texttt{match}\ x\ \texttt{with} \mid \sigma_1\ x \to C'[\sigma_1\ x] \mid \sigma_2\ x \to C'[\sigma_2\ x] \rightsquigarrow \texttt{match}\ x\ \texttt{with} \mid \sigma_1\ x \to r_1 \mid \sigma_2\ x \to r_2 : \emptyset \mid Q^{\mathsf{at}}}}{\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} \texttt{match}\ n\ \texttt{with} \mid \sigma_1\ x \to C'[\sigma_1\ x] \mid \sigma_2\ x \to C'[\sigma_2\ x] \rightsquigarrow \texttt{let}\ x = n\ \texttt{in match}\ x\ \texttt{with} \mid \sigma_1\ x \to r_1 \mid \sigma_2\ x \to r_2 : Q^{\mathsf{at}}}$$

Positive η-expansion: empty type This case cannot happen by Theorem 11.3.4 (Inconsistent canonicity); in an inconsistent context, saturation always finds a proof of 0, and thus all saturated proof terms are of the form $\texttt{absurd}(\_)$ and are thus $(\approx_{\mathtt{icc}})$-equivalent.

Transitivity Given $t \approx_{\mathtt{sat}} u$ and $u \approx_{\mathtt{sat}} r$, do we have $t \approx_{\mathtt{sat}} r$? In the general case we have

$$\dfrac{\begin{array}{c}\emptyset; \Sigma \vdash_{\mathsf{inv}} t : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{inv}} u : A \mid \emptyset \\ \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(t) \rightsquigarrow t' : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(u) \rightsquigarrow {u'}_1 : A \mid \emptyset \\ \emptyset; \Sigma \vdash_{\mathsf{sinv}} t' \rightsquigarrow t'' : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{sinv}} {u'}_1 \rightsquigarrow {u''}_1 : A \mid \emptyset \\ t'' \approx_{\mathtt{icc}} {u''}_1\end{array}}{\Sigma \vdash t \approx_{\mathtt{sat}} u : A}$$

$$\dfrac{\begin{array}{c}\emptyset; \Sigma \vdash_{\mathsf{inv}} u : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{inv}} r : A \mid \emptyset \\ \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(u) \rightsquigarrow {u'}_2 : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{inv}} \mathsf{NF}_\beta(r) \rightsquigarrow r' : A \mid \emptyset \\ \emptyset; \Sigma \vdash_{\mathsf{sinv}} {u'}_2 \rightsquigarrow {u''}_2 : A \mid \emptyset \qquad \emptyset; \Sigma \vdash_{\mathsf{sinv}} r' \rightsquigarrow r'' : A \mid \emptyset \\ {u''}_2 \approx_{\mathtt{icc}} r''\end{array}}{\Sigma \vdash u \approx_{\mathtt{sat}} r : A}$$

By Lemma 11.4.5 (Determinacy of saturated translation) we have that $u''_1 \approx_{\mathtt{icc}} u''_2$. Then, by transitivity of $(\approx_{\mathtt{icc}})$:

$$t'' \approx_{\mathtt{icc}} u''_1 \approx_{\mathtt{icc}} u''_2 \approx_{\mathtt{icc}} r''$$

Congruence If $\Sigma \vdash t_1 \approx_{\mathtt{sat}} t_2 : A$, do we have that $C[t_1] \approx_{\mathtt{sat}} C[t_2]$ for any term context C?

This is an immediate application of Lemma 11.4.6 (Saturation congruence): it tells us that $\mathsf{NF}_{\mathsf{sat}}(C[t_1]) \approx_{\mathtt{icc}} \mathsf{NF}_{\mathsf{sat}}(C[\mathsf{NF}_{\mathsf{sat}}(t_1)])$ and $\mathsf{NF}_{\mathsf{sat}}(C[t_1]) \approx_{\mathtt{icc}} \mathsf{NF}_{\mathsf{sat}}(C[\mathsf{NF}_{\mathsf{sat}}(t_2)])$. So, by transitivity of $(\approx_{\mathtt{icc}})$ we only have to prove $\mathsf{NF}_{\mathsf{sat}}(C[\mathsf{NF}_{\mathsf{sat}}(t_1)]) \approx_{\mathtt{icc}} \mathsf{NF}_{\mathsf{sat}}(C[\mathsf{NF}_{\mathsf{sat}}(t_1)])$, which is a consequence of our assumption $\mathsf{NF}_{\mathsf{sat}}(t_1) \approx_{\mathtt{icc}} \mathsf{NF}_{\mathsf{sat}}(t_2)$ and congruence of $(\approx_{\mathtt{icc}})$. □

12. From the logic to the algorithm: deciding unicity

The saturating focused logic corresponds to a computationally complete presentation of the structure of canonical proofs we are interested in. From this presentation it is extremely easy to derive a terminating search algorithm complete for unicity – we moved from a whiteboard description of the saturating rules to a working implementation of the algorithm usable on actual examples in exactly one day of work. The implementation [Scherer, 2015b] is around 700 lines of readable OCaml code.

In Section 6.2 (Rudiments of proof search), we proved the decidability of inhabitation for propositional logic. Decidability results for quantifier-free logics are easily obtained by constructing a search space, for the proofs of a given judgment, that is both complete for provability (it contains a proof it he judgment is at all provable) and finite. Three key observations were used to exhibit this finite search space:

1. Cut-free proofs in propositional logic have the subformula property, which bounds the formula appearing in the proof the finite set of sub-formulas of the root judgment.

2. The contexts of the logic are *sets* of formulas, and in particular the set of contexts over the finite set of formulas is finite. Thus, the set of possible judgments is finite.

3. We can restrict ourselves to the subset of proof where, along any path of the proof tree, all judgments occurs at most once – and all provable formulas remain provable under that restriction. This sub-system of *recurrence-free* proofs is thus complete for provability, and is finite – as the set of possible judgments is finite.

In the present chapter, we would like to justify our implementation by proposing a similarly finite subsystem of our saturation logic, which enjoys canonical proofs. The goal is to be able to decide whether a type is uniquely inhabited by exploring this subsystem, so it should be unicity complete.

The subformula property is preserved in saturated proof terms, which are cut-free proofs with additional structure. But the two other restrictions above are too brutal for our needs. They preserve completeness for provability, but they lose many computational behaviors, they break computational completeness and even unicity completeness. We refine them into two restrictions that give us finiteness (and, in particular, break computational completeness for types with infinitely many distinct inhabitants) but preserve unicity completeness, and in fact let us enumerate at least n different inhabitants if they exist.

1. To detect non-unicity, it suffices to keep at most *two* variables of each type in the context. This suggest a definition of contexts as 2-bounded multisets of formulas, which give a finite context space over a finite space of formulas. The fact that this restriction is unicity complete was proved in Chapter 9 (Counting terms and proofs).

2. Similarly, we restrict ourselves to the subset of proofs where, along any path of the proof tree, all judgments occur at most two times. This relaxation of the *recurrence-free* criterion suffices to recover completeness for unicity, as we shall prove in this chapter.

12.1. Implementing search

12.1.1. Implementation overview

The central idea to cut the search space while remaining complete for unicity is the *two-or-more* approximation. We use a *plurality* monad `Plur`, defined in set-theoretic terms as $\texttt{Plur}(S) \stackrel{\text{def}}{=} 1 + S + S \times S$, representing zero, one or "at least two" distinct elements of the set S. Each typing judgment is reformulated into a search function which takes as input the context(s) of the judgment and its goal, and returns a plurality of proof terms – we search not for *one* proof term, but for (a bounded set of) *all* proof terms. Reversing the usual mapping from variables to types, the contexts map types to pluralities of formal variables – just as we did in Chapter 9 (Counting terms and proofs).

In the search algorithm, the SINV-END rule does not merely pass its new context Γ' to the saturation rules, but it also *trims* it by applying the two-or-more rule: if the old context Γ already has two variables of a given formula N, drop all variables for N from Γ'; if it already has one variable, retain at most one variable in Γ'. This amounts to defining a selection function $\texttt{Select}_{\Gamma,\Gamma'}(\_)$ for use in the SAT rule. This trimming respects the selection requirement SELECT-SPECIF, as it always keep at least one proof of each formula provable in either Γ or Γ'. Proving that it is complete for unicity was the topic of Chapter 9 (Counting terms and proofs).

To effectively implement the saturation rules, a useful tool is an *obligation search* function (called `select_oblis` in our prototype) which takes a selection predicate on positive or atomic formulas P^{at}, and searches for (a plurality of) each negative formula N from the context that might be the starting point of an elimination judgment of the form $\Gamma \vdash n \Downarrow P^{\mathsf{at}}$, for a P^{at} accepted by the selection predicate. For example, if we want to prove X and there is a formula $Y \to Z \times X$, this formula will be part of the search results – although we do not know yet if we will be able to prove Y. For each such P^{at}, it returns a *proof obligation*, that is either a valid derivation of $\Gamma \vdash n \Downarrow P^{\mathsf{at}}$, or a *request*, giving some formula Q and expecting a derivation of $\Gamma \vdash ? \Uparrow Q^{\mathsf{at}}$ before returning another proof obligation for P^{at}.

The rule SAT-ATOM $(\Gamma; \emptyset \vdash_{\mathsf{sat}} ? : X^-)$ uses this obligation search function to search for all negatives that could potentially be eliminated into a X^-, and feeding (pluralities of) answers to the returned proof obligations (by recursively searching for introduction judgments) to obtain (pluralities of) elimination proofs of X^-.

The rule SAT uses the selection function to find the negatives that could be eliminated in any strictly positive formula and tries to fullfill (pluralities of) proof obligations. This returns a binding context (with a plurality of neutrals for each positive formula), which is filtered a posteriori to keep only the "new" bindings – that use the new context. The new binding are all added to the search environment, and saturating search is called recursively. It returns a plurality of proof terms; each of them results in a proof derivation (where the saturating set is trimmed to retain only the bindings useful to that particular proof term).

Finally, to ensure termination while remaining complete for unicity, we do not search for proofs where a given subgoal occurs strictly more than twice along a given search path. This is easily implemented by threading an extra "memory" argument through each recursive call, which counts the number of identical subgoals below a recursive call and kills the search (by returning the "zero" element of the plurality monad) at two. Note that this does not correspond to memoization in the usual sense, as information is only propagated along a recursive search branch, and never shared between several branches.

This fully describes the algorithm, which is easily derived from the logic. It is effective, and our implementation answers instantly on all the (small) types of polymorphic functions we tried. But it is not designed for efficiency, and in particular saturation duplicates a lot of work (re-computing old values before throwing them away).

We can give a presentation of the algorithm as a system of inference rules that is terminating and deterministic. Using the two-or-more counting approximation result of

Chapter 9 (Counting terms and proofs), we can prove the correctness of this presentation.

12.1.2. A formal description of the algorithm

In Figure 12.1 (Saturation algorithm) we present a complete set of inference rules that captures the behavior of our search algorithm.

Data structures The judgments uses several kinds data-structures.

- 2-*sets* $S, T \ldots$, are sets restricted to having at most two (distinct) elements; we use $\{\ldots\}_2$ to build a 2-set, and $(\cup_2)$ for union of two-sets (keeping at most two elements in the resulting union). We use the usual notation $x \in S$ for 2-set membership. To emphasize the distinction, we will sometimes write $\{\ldots\}_\infty$ for the usual, unbounded sets. Remark that 2-sets correspond to the "plurality monad" of Section 12.1.1 (Implementation overview): a monad is more convenient to use in an implementation, but for inference rules we use the set-comprehension notation.

- 2-*mappings* are mappings from a set of keys to 2-sets. In particular, Γ^{at} denotes a 2-mapping from negative or atomic types to 2-sets of formal variables. We use the application syntax $\Gamma^{\mathsf{at}}(N^{\mathsf{at}})$ for accessing the 2-set bound to a specific key, $N^{\mathsf{at}} \mapsto S$ for the singleton mapping from one variable to one 2-set, and $(\oplus)$ for the union of 2-mappings, which applies $(\cup_2)$ pointwise:

  $$(\Gamma^{\mathsf{at}} \oplus \Gamma^{\mathsf{at}\prime})(N^{\mathsf{at}}) \stackrel{\mathsf{def}}{=} \Gamma^{\mathsf{at}}(N^{\mathsf{at}}) \cup_2 \Gamma^{\mathsf{at}\prime}(N^{\mathsf{at}})$$

  Finally, we write $\emptyset$ for the mapping that maps any key to the empty 2-set.

- *multisets* M are mappings from elements to a natural number count. The "memories" of subgoal ancestors are such mappings (where the keys are "judgments" of the form $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$), and our rules will guarantee that the value of any key is at most 2. We use the application syntax $M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$ to access the count of any element, and $(+)$ for pointwise addition of multisets:

  $$(M + M')(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}) \stackrel{\mathsf{def}}{=} M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}) + M'(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$$

- (ordered) *lists* Σ of strictly positive formulas.

Finally, we use a *substraction operation* $(-_2)$ between 2-mappings, that can be defined from the 2-*set restriction* operation $S \setminus_2 n$ (where n is a natural number in $\{0, 1, 2\}$). Recall that $\mathsf{cardinal}(S)$ is the cardinal of the set (or 2-set) S.

$$(\Gamma^{\mathsf{at}\prime} -_2 \Gamma^{\mathsf{at}})(N^{\mathsf{at}}) \stackrel{\mathsf{def}}{=} \Gamma^{\mathsf{at}\prime}(N^{\mathsf{at}}) \setminus_2 \mathsf{cardinal}(\Gamma^{\mathsf{at}}(N^{\mathsf{at}}))$$

$$S \setminus_2 0 \stackrel{\mathsf{def}}{=} S \qquad \emptyset \setminus_2 1 \stackrel{\mathsf{def}}{=} \emptyset \qquad \{a, \ldots\}_2 \setminus_2 1 \stackrel{\mathsf{def}}{=} \{a\}_2 \qquad S \setminus_2 2 \stackrel{\mathsf{def}}{=} \emptyset$$

Note that $\{a, b\} \setminus_2 1$ is not uniquely defined: it could be either a or b, the choice does not matter. The defining property of $S \setminus_2 n$ is that it is a minimal 2-set S' such as $S' \cup_2 T = S$ for some set T.

Judgments The algorithm is presented as a system of judgment-directed (that is, directed by the types in the goal and the context(s)) inference rules. It uses the following five judgment forms:

- invertible judgments $M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid Q^{\mathsf{at}}$
- saturation judgments $M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}$

- post-saturation judgments $M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}}_{\mathsf{post}} S : Q^{\mathsf{at}}$
- introduction judgments $M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow P$
- elimination judgments $M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Downarrow N$

All algorithmic jugments respect the same conventions:

- M is a *memory* (remembering ancestors judgments for termination), a multiset of judgments of the form $\Gamma \vdash A$
- $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime}$ are 2-mappings from negative or atomic types to 2-sets of formal variables (we will call those "contexts")
- Σ is an ordered list of pairs $x : P$ of formal variables and positive types
- S is a 2-set of proof terms of the saturating focused logic

The S position is the output position of each judgment (the algorithm returns a 2-set of distinct proof terms); all other positions are input positions; any judgment has exactly one applicable rule, determined by the value of its input positions.

Sets of terms We extend the term construction operations to 2-sets of terms:

$$
\begin{array}{lcl}
\lambda x.\, S & \stackrel{\mathsf{def}}{=} & \{\lambda x.\, t \mid t \in S\}_2 \\
S\ T & \stackrel{\mathsf{def}}{=} & \{t\ u \mid t \in S, u \in T\}_2 \\
(S, T) & \stackrel{\mathsf{def}}{=} & \{(t, u) \mid t \in S, u\ in T\}_2 \\
\pi_i\ S & \stackrel{\mathsf{def}}{=} & \{\pi_i\ t \mid t \in S\}_2 \\
\sigma_i\ S & \stackrel{\mathsf{def}}{=} & \{\sigma_i\ t \mid t \in S\}_2 \\
\texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ x \to S_1 \\ \sigma_2\ x \to S_2 \end{array} \right. & \stackrel{\mathsf{def}}{=} & \{\texttt{match}\ x\ \texttt{with} \left| \begin{array}{l} \sigma_1\ x \to t_1 \\ \sigma_2\ x \to t_2 \end{array} \right. \mid t_i \in S_i\}_2
\end{array}
$$

Invertible rules The invertible focused rules $\Gamma^{\mathsf{at}}; \Sigma \vdash_{\mathsf{inv}} ? : N \mid Q^{\mathsf{at}}$ exhibit "don't care" non-determinism in the sense that their order of application is irrelevant and captured by invertible commuting conversions (see Section 7.2.6). In the algorithmic judgment, we enforce a specific order through the two following restrictions.

First, we use the incremental move rules instead of a batch release rule (see Section 7.2.4 (Batch or incremental validation of non-polarized contexts) for a discussion of the design space). The negative or atomic formulas that are shifted in the positive context Σ are moved incrementally to a temporary context $\Gamma^{\mathsf{at}\prime}$. By using an ordered list for the positive context, we fix the order in which positives are deconstructed. When the head of the ordered list has been fully deconstructed (it is negative or atomic), the new rule ALG-SINV-RELEASE moves it into $\Gamma^{\mathsf{at}\prime}$.

Second, the invertible right-introduction rules are restricted to judgments whose ordered context Σ is empty. This enforces that left-introductions are always applied fully before any right-introduction. Note that we could arbitrarily decide to enforce the opposite order by un-restricting right-introduction rules, and requiring that left-introduction (and releases) only happen when the succedent is positive or atomic.

After the decomposition of Σ is finished, the final invertible rule ALG-SINV-END uses 2-mapping substractions $\Gamma^{\mathsf{at}} -_2 \Gamma^{\mathsf{at}\prime}$ to trim the new context $\Gamma^{\mathsf{at}\prime}$ before handing it to the saturation rules: for any given formula N^{at}, all bindings for N^{at} are removed from $\Gamma^{\mathsf{at}\prime}$ if there are already two in Γ^{at}, and at most one binding is kept if there is already one in Γ^{at}. Morally, the reason why it is *correct* to trim (that is, it does not endanger unicity completeness is that the next rules in bottom-up search will only use the merged context $\Gamma^{\mathsf{at}} \cup_2 \Gamma^{\mathsf{at}\prime}$ (which is preserved by trimming by construction of $(-_2)$), or saturate with

Figure 12.1.: Saturation algorithm

ALG-SINV-SUM

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : P_1, \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S_1 : N \mid Q^{\mathsf{at}} \qquad M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : P_2, \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S_2 : N \mid Q^{\mathsf{at}}}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : P_1 + P_2, \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} \mathtt{match}\ x\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x \to S_1 \\ \sigma_2\ x \to S_2 \end{array} : N \mid Q^{\mathsf{at}}}$$

ALG-SINV-PROD

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} S_1 : N_1 \mid \emptyset \qquad M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} S_2 : N_2 \mid \emptyset}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} (S_1, S_2) : N_1 \times N_2 \mid \emptyset}$$

ALG-SINV-ARR

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : P \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid \emptyset}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} \lambda x.\, S : P \to N \mid \emptyset}$$

ALG-SINV-UNIT

$$\frac{}{\mathcal{M} \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} \{()\} : 1 \mid \emptyset}$$

ALG-SINV-EMPTY

$$\frac{}{\mathcal{M} \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : 0, \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} \{\mathtt{absurd}(x)\} : 1 \mid \emptyset}$$

ALG-SINV-RELEASE

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \oplus (N^{\mathsf{at}\prime} \mapsto \{x\}_2); \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid Q^{\mathsf{at}}}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; x : \left\langle N^{\mathsf{at}\prime} \right\rangle^{+\mathsf{at}}, \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid Q^{\mathsf{at}}}$$

ALG-SINV-END

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; (\Gamma^{\mathsf{at}\prime} -_2 \Gamma^{\mathsf{at}}) \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : \emptyset \mid Q^{\mathsf{at}}}$$

ALG-SAT-KILL

$$\frac{M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}) = 2}{M \mathbin{@} \Gamma^{\mathsf{at}}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{sat}} \emptyset : Q^{\mathsf{at}}}$$

ALG-SAT-POST

$$\frac{M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}) < 2 \qquad M \oplus_2 (\Gamma \vdash P) \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}}_{\mathsf{post}} S : Q^{\mathsf{at}}}{M \mathbin{@} \Gamma^{\mathsf{at}}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}}$$

ALG-POST-INTRO

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow P}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}}_{\mathsf{post}} S : P}$$

ALG-POST-ATOM

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Downarrow X^-}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}}_{\mathsf{post}} S : X^-}$$

ALG-SAT

$$\frac{\begin{array}{c} \Gamma' \neq \emptyset \\ \forall (P \mid P \text{ subformula } (\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}\prime})), \quad S_P \stackrel{\mathsf{def}}{=} \bigcup_2 \{S_{\mathsf{ne}} \mid M \mathbin{@} \Gamma, \Gamma' \vdash^{\mathsf{alg}} S_{\mathsf{ne}} \Downarrow P\} \\ B \stackrel{\mathsf{def}}{=} \bigoplus_P \{P \mapsto \{x_n\}_2 \mid n \in S_P\} \\ M \mathbin{@} \Gamma, \Gamma'; \emptyset; B \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : \emptyset \mid Q^{\mathsf{at}} \\ S' \stackrel{\mathsf{def}}{=} \left\{ \mathtt{let}\ \bar{x} = \bar{n}\ \mathtt{in}\ t \;\middle|\; \begin{array}{l} t \in S, \\ (\bar{x}, \bar{n}) \stackrel{\mathsf{def}}{=} \{(x_n, n) \mid \exists P,\ x_n \in B(P)\}_\infty \end{array} \right\}_2 \end{array}}{M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash^{\mathsf{alg}}_{\mathsf{sat}} S' : Q^{\mathsf{at}}}$$

ALG-SINTRO-SUM

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S_1 \Uparrow P_1 \qquad M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S_2 \Uparrow P_2}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} (\sigma_1\ S_1) \cup_2 (\sigma_2\ S_2) \Uparrow P_1 + P_2}$$

ALG-SINTRO-VAR

$$\frac{S \stackrel{\mathsf{def}}{=} \{x \mid (x : X^+) \in \Gamma^{\mathsf{at}}\}_2}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow X^+}$$

ALG-SINTRO-END

$$\frac{M \mathbin{@} \Gamma^{\mathsf{at}}; \emptyset; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid \emptyset}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow \langle N \rangle^-}$$

ALG-SELIM

$$\frac{\begin{array}{c} N \text{ subformula } \Gamma^{\mathsf{at}} \\ S_{\mathsf{var}} \stackrel{\mathsf{def}}{=} \Gamma^{\mathsf{at}}(N) \\ S_{\mathsf{proj}} \stackrel{\mathsf{def}}{=} \bigcup_2 \{\pi_i\ S \mid M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Downarrow M_1 \times M_2,\ M_i = N\} \\ S_{\mathsf{app}} \stackrel{\mathsf{def}}{=} \bigcup_2 \{S\ T \mid M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Downarrow P \to N,\ M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} T \Uparrow P\} \end{array}}{M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S_{\mathsf{var}} \cup_2 S_{\mathsf{proj}} \cup_2 S_{\mathsf{app}} \Downarrow N}$$

bindings from $\Gamma^{\mathsf{at}'}$. Any strictly positive that can be deduced by using one of the variables present in $\Gamma^{\mathsf{at}'}$ but removed from $\Gamma^{\mathsf{at}} \cup_2 \Gamma^{\mathsf{at}'}$ has already been deduced from Γ^{at}. It is *useful* to trim in this rule (we could trim much more often) because subsequent saturated rules will test the new context $\Gamma^{\mathsf{at}'} -_2 \Gamma^{\mathsf{at}}$ for emptyness, so it is interesting to minimize it. In any case, we need to trim in at least one place in order for typing judgments not to grow unboundedly.

Saturation rules If the (trimmed) new context is empty, we test whether the judgment of the current subgoal has already occurred twice among its ancestors; in this case, the rule ALG-SAT-KILL terminates the search process by returning the empty 2-set of proof terms. In the other case, the number of occurrences of this judgment is incremented in the rule ALG-SAT-POST, and one of the (transparent) "post-saturation" rules ALG-POST-INTRO or ALG-POST-ATOM are applied.

This is the only place where the memory M is accessed and updated. The reason why this suffices is any given phase (invertible phase, or phase of non-invertible eliminations and introductions) is only of finite length, and either terminates or is followed by a saturation phase; because contexts grow monotonously in a finite space (of 2-mappings rather than arbitrary contexts), the trimming of rule ALG-SINV-END returns the empty context after a finite number of steps: an infinite search path would need to go through ALG-SAT-POST infinitely many times, and this suffices to prove termination.

The most important and complex rule is ALG-SAT, which proceeds in four steps. First, we compute the 2-set S_P of all ways to deduce any strict positive P from the context – for any P we need not remember more than two ways. We know that we need only look for P that are deducible by elimination from the context $\Gamma^{\mathsf{at}}, \Gamma^{\mathsf{at}'}$ – the finite set of subformulas is a good enough approximation. Because we retain at least one neutral of each newly provable positive P, this algorithm corresponds to a selection function that satisfies SELECT-SPECIF.

Second, we build a context B binding a new formal variable x_n for each elimination neutral n – it is crucial for canonicity that all n are new and semantically distinct from each other at this point, otherwise duplicate bindings would be introduced. Third, we compute the 2-set S of all possible (invertible) proofs of the goal under this saturation context B, and add the `let`-bindings to those proof terms in the final returned 2-set.

Non-invertible introduction and elimination rules The introduction rule ALG-SINTRO-SUM collects solutions using either left or right introductions, and unites them in the result 2-set. Similarly, all elimination rules are merged in one single rule ALG-SELIM, which corresponds to all ways to deduce a given formula N: directly from the context, by projection of a pair, or application of a function. The search space for this sequent is finite, as goal types grow strictly at each type, and we can kill search for any type that does not appear as a subformula of the context.

(The inference-rule presentation differs from our OCaml implementation at this point. The implementation is more effective, it uses continuation-passing style to attempt to provide function arguments only for the applications we know are found in context and may lead to the desired result. Such higher-order structure is hard to render in an inference rule, so we approximated it with a more declarative presentation here. This is the only such simplification.)

12.2. Correctness

Lemma 12.2.1 (Termination)**.**
The algorithmic inference system only admits finite derivations.

Proof. We show that each inference rule is of finite degree (it has a finite number of premises), and that there exists no infinite path of inference rules – concluding with

König's Lemma.

Degree finiteness The rules that could be of infinite degree are ALG-SAT (which quantifies over all positives P) and ALG-SELIM (which quantifies over arbitrarily many elimination derivations). But both rules have been restricted through the subformula property to only quantify on finitely many formulas (ALG-SAT) or possible elimination schemes (ALG-SELIM).

Infinite paths lead to absurdity We first assert that any given phase (invertible, saturation, introductions/eliminations) may only be of finite length. Indeed, invertible rules have either the context or the goal decreasing structurally. Saturation rules are either ALG-SAT if $\Gamma^{\mathsf{at}\prime} \neq \emptyset$, which is immediately followed by elimination and invertible rules, or ALG-SAT-KILL or ALG-SAT-POST if $\Gamma^{\mathsf{at}\prime} = \emptyset$, in which case the derivation either terminates or continues with a non-invertible introduction or elimination. Introductions have the goal decreasing structurally, and eliminations have the goal *increasing* structurally, and can only form valid derivations if it remains a subformula of the context Γ^{at}.

Given that any phase is finite, any infinite path will necessarily have an infinite number of phase alternation. By looking at the graph of phase transitions (invertible goes to saturating which goes to introductions or eliminations, which go to invertible), we see that each phase will occur infinitely many times along an infinite path. In particular, an infinite path would have infinitely many invertible and saturation phases; the only transition between them is the rule ALG-SINV-END which must occur infinitely many times in the path.

Now, because the rules grow the context monotonically, an infinite path must eventually reach a maximal stable context Γ^{at}, that never grows again along the path. In particular, for infinitely many ALG-SINV-END we have Γ^{at} maximal and thus $\Gamma^{\mathsf{at}\prime} -_2 \Gamma^{\mathsf{at}} = \emptyset$ – if the trimming was not empty, $\Gamma^{\mathsf{at}\prime}$ would grow strictly after the next saturation phase, while we assumed it was maximal.

This means that either ALG-SAT-KILL or ALG-SAT-POST incurs infinitely many times along the infinite path. Those rules check the memory count of the current (context, goal) pair $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$. Because of the subformula property (formulas occurring in subderivations are subformulas of the root judgment concluding the complete proof), there can be only finitely many different $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$ pair (Γ^{at} is a 2-mapping which grows monotonically).

An infinite path would thus necessarily have infinitely many steps ALG-SAT-KILL or ALG-SAT-POST with the same (context, goal) pair. This is impossible, as a given pair can only go at most twice through ALG-SAT-POST, and going through ALG-SAT-KILL terminates the path. There is no infinite path. □

Lemma 12.2.2 (Totality and Determinism)**.**
For any algorithmic judgment there is exactly one applicable rule.

Proof. Immediate by construction of the rules. Invertible rules $M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid Q^{\mathsf{at}}$ are directed by the shape of the context Σ and the goal N. Saturation rules $M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}$ are directed by the new context $\Gamma^{\mathsf{at}\prime}$. If $\Gamma^{\mathsf{at}\prime} = \emptyset$, the memory $M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$ decides whether to kill or post-saturate, in which case the shape of the goal (either strict positive or atomic) directs the post-saturation rule. Finally, non-invertible introductions $M \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow P$ are directed by the goal P, and there is exactly one non-invertible elimination rule. □

Remark 12.2.1. The choice we made to restrict the ordering of invertible rules is not necessary – we merely wanted to demonstrate an example of such restrictions, and reflect the OCaml implementation. We could keep the same indeterminacy as in previous systems; totality would be preserved (all judgments have one applicable rule), but determinism dropped. There could be several S such that $M \mathbin{@} \Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime}; \Sigma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : A \mid$, which would correspond to (2-set restrictions of) sets of terms equal upto invertible commuting conversion. ∗

Lemma 12.2.3 (Soundness)**.**

For any algorithmic judgment returning a 2-set S, any element $t \in S$ is a valid proof term of the corresponding saturating judgment.

Proof sketch. By induction, this is immediate for all rules except ALG-SAT. This rule is designed to fit the requirements of the saturated logic SAT rule. ⊔

Definition 12.2.1 Recurrent ancestors**.**
This definition is a reminder and specialization of Definition 6.2.5 (Recurrent ancestor).

Consider a complete algorithmic derivation of a judgment with empty initial memory $\emptyset$. Given any subderivation P_{leafward}, we call *recurrent ancestor* any other subderivation Π_{rootward} that is on the path between Π_{leafward} and the root (it has Π_{leafward} as a strict subderivation) and whose derived judgment is identical to the one of Π_{leafward} except for the memory M and the output set S.

Lemma 12.2.4 (Correct Memory)**.**
In a complete algorithmic derivation whose conclusion's memory is M, each subderivation of the form $M' \mathbin{@} \Gamma^{\mathsf{at}}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}$ has a number of recurrent ancestors equal to

$$M'(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}) - M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$$

Proof. This is immediately proved by reasoning on the path from the start of the complete derivation to the subderivation. By construction of the algorithmic judgment, each judgment of the form $M' \mathbin{@} \Gamma^{\mathsf{at}\prime}; \emptyset \vdash^{\mathsf{alg}}_{\mathsf{sat}} S' : Q^{\mathsf{at}}$ is proved by either the rule ALG-SAT-KILL, which terminates the path with the invariant maintained, or the rule ALG-SAT-POST, which continues the path with the invariant preserved by incrementing the count in memory. □

Lemma 12.2.5 (Recurrence Decrementation)**.**
If a saturated logic derivation contains $n+2$ occurrences of the same judgment along a given path, then there is a valid saturated logic derivation with $n+1$ occurrences of this judgment.

Proof. We have actually already proved this in Section 6.2.2 (Recurrent ancestors in derivations).

If t is the proof term with $n+2$ occurrences of the same judgment along a given path, let u_1 be the subterm corresponding to the very last occurrence of the judgment, and u_2 the last-but-one. The term $t[u_1/u_2]$ is a valid proof term (of the same result as t), with only $n+1$ occurrences of this same judgment. □

Note that this transformation changes the computational meaning of the term – it must be used with care, as it could break unicity completeness.

Theorem 12.2.6 (Provability completeness)**.**
If a memory M contains multiplicities of either 0 or 1 (never 2 or more), then any algorithmic judgment with memory M is complete for unicity: if the corresponding saturating judgment is inhabited, then the algorithmic judgment returns an inhabited 2-set.

Proof. If the saturating judgment $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} t : Q^{\mathsf{at}}$ holds for a given t, we can assume without loss of generality that t contains no two recurring occurrences of the same judgment along any path – indeed, it suffices to repeatedly apply Lemma 12.2.5 (Recurrence Decrementation) to obtain such a t with no recurring judgment.

The proof of our result goes by induction on (the saturated derivation of) this no-recurrence t, mirroring each inference step into an algorithmic inference rule returning an inhabited set. Consider the following saturated rule for example:

$$\frac{\Gamma^{\mathsf{at}} \vdash u \Uparrow P_1}{\Gamma^{\mathsf{at}} \vdash \sigma_1\ u \Uparrow P_1 + P_2}$$

We can build the corresponding algorithmic rule

$$\frac{M' \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S_1 \Uparrow P_1 \qquad M' \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S_2 \Uparrow P_2}{M' \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} \sigma_1\ S_1 \cup_2 \sigma_2\ S_2 \Uparrow P_1 + P_2}$$

By induction hypothesis we have that S_1 is inhabited; from it we deduce that $\sigma_1\ S_1$ is inhabited, and thus $\sigma_1\ S_1 \cup_2 \sigma_2\ S_2$ is inhabited.

It would be tempting to claim that the resulting set is inhabited *by* t. That, in our example above, u inhabits S_1 and thus $t = \sigma_1\ u$ inhabits $\sigma_1\ S_1 \cup_2 \sigma_2\ S_2$. This stronger statement is incorrect, however, as the union of 2-sets may drop some inhabitants if it already has found two distinct terms.

The first difficulty in the induction are with judgments of the form $\Gamma^{\mathsf{at}}; \emptyset \vdash_{\mathsf{sat}} u : Q^{\mathsf{at}}$: to build an inhabited result set, we need to use the rule ALG-SAT-POST and thus check that $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$ does not occur twice in the current memory M'. By Lemma 12.2.4 (Correct Memory), we know that $M'(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$ is the sum of the number of recurrent ancestors and of $M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$. By definition of t (as a term with no repeated judgment), we know that $\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}}$ did not already occur in t itself – the count of recurrent ancestors is 0. By hypothesis on M we know that $M(\Gamma^{\mathsf{at}} \vdash_{\mathsf{foc}} Q^{\mathsf{at}})$ is at most 1, so the sum cannot be 2 or more.

The second and last subtlety happens at the SINV-END rule for $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}'} \vdash_{\mathit{sinv}} f : \emptyset \mid Q^{\mathsf{at}}$. We read saturated derivation of the premise $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}'} \vdash_{\mathsf{sat}} f : Q^{\mathsf{at}}$, but build an algorithmic derivation in the trimmed context $M \mathbin{@} \Gamma^{\mathsf{at}}; (\Gamma^{\mathsf{at}'} -_2 \Gamma^{\mathsf{at}}) \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}$. It is not necessarily the case that f is well-defined in this restricted context. But that is not an issue for inhabitation: the only variables removed from $\Gamma^{\mathsf{at}'}$ are those for which at least one variable of the same type appears in Γ^{at}. We can thus replace each use of a trimmed variable by another variable of the same type in Γ^{at}, and get a valid derivation of the exact same size.[1] □

Theorem 12.2.7 (Unicity completeness)**.**
If a memory M contains multiplicities of 0 only, then any algorithmic judgment with memory M is complete for unicity: if the corresponding saturating judgment has two distinct inhabitants, then the algorithmic judgment returns a 2-set of two distinct elements.

Proof. Consider a pair of distinct inhabitants $t \neq u$ of a given judgment. Without loss of generality, we can assume that t has no judgment ocurring twice or more. (We cannot also assume that u has no judgment occurring twice, as the recurrence reduction of a general u may be equal to t.)

Without loss of generality, we will also assume that t and u use a consistent ordering for invertible rules (for example the one presented in the algorithmic judgment); this assumption can be made because reordering inference steps gives a term in the $(\approx_{\mathtt{icc}})$ equivalence class, that is thus $\beta\eta$-equivalent to the starting term.

Finally, to justify the SINV-END rule we need to invoke the "two or more" result of Chapter 9 (Counting terms and proofs), as we detail here. Without loss of generality we assume that t and u never use more than two variables of any given type (additional variables are weakened as soon as they are introduced). If t and u have distinct shapes (they are in disjoint equivalent classes of terms that erase to the same logic derivation), we immediately know that the disequality $t \neq u$ is preserved. If they have the same shape, we need to invoke Corollary 9.3.6 (Two-or-more approximation) to know that we can pick two distinct terms in this restricted space.

We then prove our result by parallel induction on t and u: the saturated judgment is inhabited by at least two distinct inhabitants. As long as their subterms start with the same syntactic construction, we keep inducing in parallel. Their head constructor may only differ in a non-invertible introduction or elimination rule (we assumed that invertible

[1] This argument was already used in Scherer and Rémy [2015], but there it was invalid. Indeed, there was a strict condition (Section 11.2.4 (Comparison with the previous approach of Scherer and Rémy [2015])) on the fact that each saturated variable had to be used in the right-hand side, which was not robust to the replacement of one variable for another. Our use of a saturation function in the current presentation avoids this issue.

steps were performed in the same order), for example we may have

$$\frac{\Gamma^{\mathsf{at}} \vdash p \Uparrow P_1}{\Gamma^{\mathsf{at}} \vdash \sigma_1\ p \Uparrow P_1 + P_2} \qquad \frac{\Gamma^{\mathsf{at}} \vdash q \Uparrow P_2}{\Gamma^{\mathsf{at}} \vdash \sigma_2\ q \Uparrow P_1 + P_2}$$

We then invoke Theorem 12.2.6 (Provability completeness) on p and q: we can build corresponding derivations $M' \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} S \Uparrow A$ and $M' \mathbin{@} \Gamma^{\mathsf{at}} \vdash^{\mathsf{alg}} T \Uparrow B$ where S and T are inhabited, and thus $\sigma_1\ S \cup_2 \sigma_2\ T$ is inhabited by at least two distinct terms. The memory hypothesis of the provability theorem is fulfilled: because we know that there are no repetitions in t, and that we iterated in parallel on the structures of t and u, we know that each judgment was seen at most once during the parallel induction. As we assumed our starting memory was all 0, the memory M' at the point where t and u differ is thus, by Lemma 12.2.4 (Correct Memory), of at most 1 for any judgment.

There is one difficulty during the parallel induction, which is the SINV-END case. We read a saturated derivations of premise $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} t : Q^{\mathsf{at}}$ and $\Gamma^{\mathsf{at}}; \Gamma^{\mathsf{at}\prime} \vdash_{\mathsf{sat}} u : Q^{\mathsf{at}}$, but build an algorithmic derivation in the trimmed context $\mathcal{M} \mathbin{@} \Gamma^{\mathsf{at}}; (\Gamma^{\mathsf{at}\prime} -_2 \Gamma^{\mathsf{at}}) \vdash^{\mathsf{alg}}_{\mathsf{sat}} S : Q^{\mathsf{at}}$. This is why we restricted t and u to not use more than two different variables of each type, so that they remain well-typed under this restriction. □

Theorem 12.2.8.
Our unicity-deciding algorithm is terminating and unicity complete.

Proof. Our unicity-deciding algorithm takes a judgment $\lfloor \Sigma \rfloor_{\pm} \vdash \lfloor (N \mid X^+) \rfloor_{\pm}$ and returns the 2-set S uniquely determined by a complete algorithmic derivation of the judgment $\emptyset \mathbin{@} \emptyset; \emptyset; \Gamma \vdash^{\mathsf{alg}}_{\mathsf{inv}} S : N \mid X^+$ – whose memory is empty. There always exists exactly one derivation by Lemma 12.2.2 (Totality and Determinism), and it is finite by Lemma 12.2.1 (Termination). Our algorithm can compute the next rule to apply in finite time, and all derivations are finite, so the algorithm is terminating. This root judgment has an empty memory, hence it is complete for unicity by Theorem 12.2.7 (Unicity completeness). □

12.3. Going further

12.3.1. Optimizations

The search space restrictions described above are those necessary for *termination*. Many extra optimizations are possible, that can be adapted from the proof search literature – with some care to avoid losing completness for unicity. For example, there is no need to cut on a positive if its atoms do not appear in negative positions (nested to the left of an odd number of times) in the rest of the goal. We did not develop such optimizations, except for two low-hanging fruits we describe below.

Eager redundancy elimination Whenever we consider selecting a proof obligation to prove a strict positive during the saturation phase, we can look at the negatives that will be obtained by cutting it. If all those atoms are already present at least twice in the context, this positive is *redundant* and there is no need to cut on it. Dually, before starting a saturation phase, we can look at whether it is already possible to get two distinct neutral proofs of the goal from the current context. In this case it is not necessary to saturate at all.

This optimization is interesting because it significantly reduces the redundancy implied by only filtering of old terms after computing all of them. Indeed, we intuitively expect that most types present in the context are in fact present twice (being unique tends to be the exception rather than the rule in programming situations), and thus would not need to be saturated again. Redundancy of saturation still happens, but only on the "frontier formulas" that are present exactly once.

Subsumption by memoization One of the techniques necessary to make the inverse method competitive is *subsumption* [McLaughlin and Pfenning, 2008]: when a new judgment is derived by forward search, it is only added to the set of known results if it is not subsumed by a more general judgment (same goal, smaller context) already known.

In our setting, being careful not to break computational completeness, this rule becomes the following. We use (monotonic) mutable state to grow a memoization table of each proved subgoal, indexed by the right-hand side formula. Before proving a new subgoal, we look for all already-computed subgoals of the same right-hand side formula. If one exists with exactly the same context, we return its result. But we also return eagerly if there exists a *larger* context (for inclusion) that returned zero result, or a *smaller* context that returned two-or-more results.

Interestingly, we found out that this optimization becomes unsound in presence of the empty type 0. Its equational theory tells us that in an inconsistent context (0 is provable), all proofs are equal. Thus a type may have two inhabitants in a given context, but a larger context that is inconsistent (let us prove 0) will have a unique inhabitant, breaking monotonicity. The optimization could still be applied for all judgments that do not have 0 as a subformula of the context.

12.4. Evaluation

In this section, we give some practical examples of code inference scenarios that our current algorithm can solve, and some that it cannot – because the simply-typed theory is too restrictive.

The key to our application is to translate a type using prenex-polymorphism into a simple type using atoms in stead of type variables – this is semantically correct given that bound type variables in System F are handled exactly as simply-typed atoms. The approach, of course, is only a very first step and quickly shows it limits. For example, we cannot work with polymorphic types in the environment (ML programs typically do this, for example when typing a parametrized module, or type-checking under a type-class constraint with polymorphic methods), or first-class polymorphism in function arguments. We also do not handle higher-kinded types – even pure constructors.

All the examples mentioned in this section are available as tests in our prototype implementation [Scherer, 2015b].

12.4.1. Inferring polymorphic library functions

The Haskell standard library contains a fair number of polymorphic functions with unique types. The following examples have been checked to be uniquely defined by their types:

$$\texttt{fst} : \forall\alpha\beta.\ \alpha \times \beta \to \alpha \qquad \texttt{curry} : \forall\alpha\beta\gamma.\ (\alpha \times \beta \to \gamma) \to \alpha \to \beta \to \gamma$$
$$\texttt{uncurry} : \forall\alpha\beta\gamma.\ (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$
$$\texttt{either} : \forall\alpha\beta\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \alpha + \beta \to \gamma$$

When the API gets more complicated, both types and terms become harder to read and uniqueness of inhabitation gets much less obvious. Consider the following operators chosen arbitrarily in the lens [Kmett, 2012] library.

```
(<.) :: Indexable i p => (Indexed i s t -> r)
       -> ((a -> b) -> s -> t) -> p a b -> r
(<.>) :: Indexable (i, j) p => (Indexed i s t -> r)
        -> (Indexed j a b -> s -> t) -> p a b -> r
(%@~) :: AnIndexedSetter i s t a b
        -> (i -> a -> b) -> s -> t
non :: Eq a => a -> Iso' (Maybe a) a
```

The type and type-class definitions involved in this library usually contain first-class polymorphism, but the documentation [Kmett, 2013] provides equivalent "simple types" to help user understanding. We translated the definitions of `Indexed`, `Indexable` and `Iso` using those simple types. We can then check that the first three operators are unique inhabitants; `non` is not.

12.4.2. Inferring module implementations or type-class instances

The `Arrow` type-class is defined as follows:

```
class Arrow (a : * -> * -> * ) where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

It is self-evident that the arrow type ($\to$) is an instance of this class, and *no code should have to be written* to justify this: our prototype is able to infer that all those required methods are uniquely determined when the type constructor `a` is instantiated with an arrow type. This also extends to subsequent type-classes, such as `ArrowChoice`.

As most of the difficulty in inferring unique inhabitants lies in sums, we study the "exception monad", that is, for a fixed type X, the functor $\alpha \mapsto X + \alpha$. Our implementation determines that its `Functor` and `Monad` instances are uniquely determined, but that its `Applicative` instance is not.

Indeed, the type of the `Applicative` method `ap` specializes to the following: $\forall \alpha\beta.\ X + (\alpha \to \beta) \to X + \alpha \to X + \beta$. If both the first and the second arguments are in the error case X, there is a non-unique choice of which error to return in the result.

This is in fact a general result on applicative functors for types that are also monads: there are two distinct ways to prove that a monad is also an applicative functor.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do                ap mf ma = do
  f <- mf                      a <- ma
  a <- ma                      f <- mf
  return (f a)                 return (f a)
```

Note that the type of `bind` for the exception monad, namely $\forall \alpha\beta.\ X + \alpha \to (\alpha \to X + \beta) \to X + \beta$, has a sum type thunked under a negative type. It is one typical example of a type which cannot be proved unique by the focusing discipline alone, and which is correctly recognized unique by our algorithm.

12.4.3. Artificial examples

Our prototype will correctly detect that

$$\forall \alpha\beta.\ \alpha \to (\alpha \to \beta + \beta) \to \beta$$

is uniquely inhabited. This type is an example of uniquely inhabited type that is not "negatively non-duplicated", as the type β has several occurrences in negative position (Section 6.2.4 (Positive and negative positions in a formula)); negative non-duplication is a sufficient criterion used in previous work on unique inhabitation [Aoto and Ono, 1994] that does not scale to sums.

A more interesting example is the continuation monad. If we define with a monomorphic return type

$$\texttt{Cont}\ \gamma\ \alpha \qquad \stackrel{\text{def}}{=} \qquad (\alpha \to \gamma) \to \gamma$$

then the `bind` operation on an arbitrary monad `Cont` A is not uniquely inhabited. In fact, the identity at this type, `Cont` $A \to$ `Cont` A, is already not uniquely inhabited.

If, however, we choose 0 as the return type, then both `Cont` $0 \to$ `Cont` 0 and the `bind` operation on `Cont` 0 are uniquely inhabited.

This example highlights the theoretical importance of properly handling the empty type. The equational theory is very different from a fixed atom X^+ with variable of this type in the environment. We conjecture that a similar result would be obtained with a definition of continuations using a polymorphic return type, but handling polymorphism comes at a higher cost in complexity.

12.4.4. Non-applications

Here are two related ideas we wanted to try, but that do not fit in the simply-typed lambda-calculus; the uniqueness algorithm must be extended to richer type systems to handle such applications.

We can check that specific instances of a given type-class are canonically defined, but it would be nice to show as well that some of the operators defined on *any* instance are uniquely defined from the type-class methods – although one would expect this to often fail in practice if the uniqueness checker doesn't understand the equational laws required of valid instances. Unfortunately, this would require uniqueness check with polymorphic types in context (for the polymorphic methods).

Another idea is to verify the coherence property of a set of declared instances by translating instance declarations into terms, and checking uniqueness of the required instance types. In particular, one can model the inheritance of one class upon another using a pair type (`Comp` α as a pair of a value of type `Eq` α and `Comp`-specific methods); and the system can then check that when an instance of `Eq` X and `Comp` X are declared, building `Eq` X directly or projecting it from `Comp` X correspond to $\beta\eta$-equivalent elaboration witnesses. Unfortunately, all but the most simplistic examples require parametrized types and polymorphic values in the environment to be faithfully modelled.

12.4.5. On impure host programs

The type system in which program search is performed does not need to exactly coincide with the ambiant type system of the host programming language, for which the code-inference feature is proposed – forcing the same type-system would kill any use from a language with non-termination as an effect. Besides doing term search in a pure, terminating fragment of the host language, one could also refine search with type annotations in a richer type system, for example using dependent types or substructural logic – as long as the found inhabitants can be erased back to host types.

However, this raises the delicate question of, among the unique $\beta\eta$-equivalence class of programs, which candidate to select to be actually injected into the host language. For example, the ordering or repetition of function calls can be observed in a host language passing impure function as arguments, and η-expansion of functions can delay effects. Even in a pure language, η-expanding sums and products may make the code less efficient by re-allocating data. There is a design space here that we have not explored.

Conclusion

13. Related Work

13.1. Previous work on unique inhabitation

The problem of unique inhabitation for the simply-typed lambda-calculus (without sums) has been formulated by Mints [1981], with early results by Babaev and Soloviev [1982], and later results by Aoto and Ono [1994], Aoto [1999] and Broda and Damas [2005].

These works have obtained several different *sufficient conditions* for a given type to be uniquely inhabited. While these cannot be used as an algorithm to decide unique inhabitation for any type, it reveals fascinating connections between unique inhabitation and proof or term structures. Some sufficient criteria are formulated on the types/formulas themselves, other on terms (a type is uniquely inhabited if it is inhabited by a term of a given structure).

A simple criterion on types given in Aoto and Ono [1994] is that "negatively non-duplicated formulas", that is formulas where each atom occurs at most once in negative position (nested to the left of an odd number of arrows), have at most one inhabitant. This was extended by Broda and Damas [2005] to a notion of "deterministic" formulas, defined using a specialized representation for simply-typed proofs named "proof trees".

Aoto [1999] proposed a criterion based on terms: a type is uniquely inhabited if it "provable without non-prime contraction", that is if it has at least *one* inhabitant (not necessarily cut-free) whose only variables with multiple uses are of atomic type. Recently, Bourreau and Salvati [2011] used game semantics to give an alternative presentation of Aoto's results, and a syntactic characterization of *all* inhabitants of negatively non-duplicated formulas.

Those sufficient conditions suggest deep relations between the static and dynamics semantics of restricted fragments of the lambda-calculus – it is not a coincidence that contraction at non-atomic type is also problematic in definitions of proof equivalence coming from categorial logic [Dosen, 2003]. However, they give little in the way of a decision procedure for all types – conversely, our decision procedure does not by itself reveal the structure of the types for which it finds unicity.

An indirectly related work is the work on retractions in simple types (A is a retract of B if B can be surjectively mapped into A by a λ-term). Indeed, in a type system with a unit type 1, a given type A is uniquely inhabited if and only if it is a retract of 1. Stirling [2013] proposes an algorithm, inspired by dialogue games, for deciding retraction in the lambda-calculus with arrows and products; but we do not know if this algorithm could be generalized to handle sums. If we remove sums, focusing already provides an algorithm for unique inhabitation.

13.2. Counting inhabitants

Broda and Damas [2005] remark that normal inhabitants of simple types can be described by a context-free structure. This suggests, as done in Zaoinc [1995], counting terms by solving a set of polynomial equations. Further references to such "grammatical" approaches to lambda-term enumeration and counting can be found in Dowek and Jiang [2011].

Of particular interest to us was the recent work of Wells and Yakobowski [2004]. It is similar to our work both in terms of expected application (program fragment synthesis) and methods, as it uses (a variant of) the focused calculus LJT [Herbelin, 1994] to perform proof search. It has sums (disjunctions), but because it only relies on focusing for canonicity it only implements the *weak* notion of η-equivalence for sums – it is not canonical,

as discussed in Section 10.6.4 (Non-canonicity of the full focused system), it counts an infinite number of inhabitants in presence of a sum thunked under a negative. Their technique to ensure termination of enumeration is very elegant. Over the graph of all possible proof steps in the type system (using multisets as contexts: an infinite search space), they superimpose the graph of all possible non-cyclic proof steps in the logic (using sets as contexts: a finite search space). Termination is obtained, in some sense, by traversing the two in lockstep. We took inspiration from this idea to obtain our termination technique: our bounded multisets can be seen as a generalization of their use of set-contexts.

13.3. Non-classical theorem proving and more canonical systems

Automated theorem proving has motivated fundamental research on more canonical representations of proofs: by reducing the number of redundant representations that are equivalent as programs, one can reduce the search space – although that does not necessarily improve speed, if the finer representation requires more book-keeping. Most of this work was done first for (first-order) classical logic; efforts porting them to other logics (linear, intuitionistic, modal) were of particular interest, as it often reveals the general idea behind particular techniques, and is sometimes an occasion to reformulate them in terms closer to type theory.

An important brand of work studies connection-based, or matrix-based, proof methods. They have been adapted to non-classical logic as soon as Wallen [1987]. It is possible to present connection-based search "uniformly" for many distinct logics [Otten and Kreitz, 1996], changing only one logic-specific check to be performed a posteriori on connections (axiom rules) of proof candidates. In intuitionistic setting, that would be a comparison on indices of Kripke Worlds; it is strongly related to *labeled logics* [Galmiche and Méry, 2013]. On the other hand, matrix-based methods rely on guessing the number of duplications of a formula (contractions) that will be used in a particular proof, and we do not know whether that can be eventually extended to second-order polymorphism – by picking a presentation closer to the original logic, namely focused proofs, we hope for an easier extension.

Some contraction-free calculi have been developed with automated theorem proving for intuitionistic logic in mind. A presentation is given in Dyckhoff [1992] – the idea itself appeared as early as Vorob'ev [1958]. The idea is that sums and (positive) products do not need to be deconstructed twice, and thus need not be contracted on the left. For functions, it is actually sufficient for provability to implicitly duplicate the arrow in the argument case of its elimination form ($A \to B$ may have to be used again to build the argument A), and to forget it after the result of application (B) is obtained. More advanced systems typically do case-distinctions on the argument type A to refine this idea, see Dyckhoff [2013] for a recent survey. Unfortunately, such techniques to reduce the search space break computational completeness: they completely remove some programmatic behaviors. Consider the type $\texttt{Stream}(A, B) \stackrel{\text{def}}{=} A \times (A \to A \times B)$ of infinite streams of state A and elements B: with this restriction, the next-element function can be applied at most once, hence $\texttt{Stream}(X, Y) \to Y$ is uniquely inhabited in those contraction-free calculi. (With focusing, only negatives are contracted, and only when picking a focus.)

Focusing was introduced for linear logic [Andreoli, 1992a], but is adaptable to many other logics. For a reference on focusing for intuitionistic logic, see Liang and Miller [2007]. To easily elaborate programs as lambda-terms, we use a natural deduction presentation (instead of the more common sequent-calculus presentation) of focused logic, closely inspired by the work of Brock-Nannestad and Schürmann [2010] on intuitionistic linear logic.

Some of the most promising work on automated theorem proving for intuitionistic logic comes from applying the so-called "Inverse Method" (see Degtyarev and Voronkov [2001] for a classical presentation) to focused logics. The inverse method was ported to linear logic in Chaudhuri and Pfenning [2005], and turned into an efficient implementation of proof

search for intuitionistic logic in McLaughlin and Pfenning [2008]. It is a "forward" method: to prove a given judgment, start with the instances of axiom rules for all atoms in the judgment, then build all possible valid proofs until the desired judgment is reached – the subformula property, bounding the search space, ensures completeness for propositional logic. Focusing allows important optimization of the method, notably through the idea of "synthetic connectives": invertible or non-invertible phases have to be applied all in one go, and thus form macro-steps that speed up saturation.

In comparison, our own search process alternates forward and backward-search. At a large scale we do a backward-directed proof search, but each non-invertible phase performs saturation, that is a complete forward-search for positives. Note that the search space of those saturation phases is not the subformula space of the main judgment to prove, but the (smaller) subformula space of the current subgoal's context. When saturation is complete, backward goal-directed search restarts, and the invertible phase may grow the context, incrementally widening the search space. (The forward-directed aspects of our system could be made richer by adding positive products and positively-biased atoms; this is not our main point of interest here. Our coarse choice has the good property that, in absence of sum types in the main judgment, our algorithm immediately degrades to simple, standard focused backward search.)

13.3.1. Maximal multi-focusing

An important result for canonical proof structures is *maximal multi-focusing* [Miller and Saurin, 2007, Chaudhuri, Miller, and Saurin, 2008a]. Multi-focusing refines focusing by introducing the ability to focus on several formulas at once, in parallel, and suggests that, among formulas equivalent modulo valid permutations of inference rules, the "more parallel" ones are more canonical. Indeed, *maximal* multi-focused proofs turn out to be equivalent to existing more-canonical proof structures such as linear proof nets [Chaudhuri, Miller, and Saurin, 2008a] and classical expansion proofs [Chaudhuri, Hetzl, and Miller, 2012].

In Scherer [2015a] we proposed a multi-focused natural deduction and a λ-calculus interpretation for it, whose maximal multi-focused terms are canonical for $\mathsf{\Lambda C}{\to}, \times, +$. Saturating focused proofs are almost maximal muli-focused proofs in this sense. The difference is that multi-focusing allow to focus on both variables in the context and the goal in the same time, while our right-focusing rule SAT-INTRO can only be applied sequentially after SAT (which does multi-left-focusing). To recover the exact structure of maximal multi-focusing, one would need to allow SAT to also focus on the right, and use it only when the right choices do not depend on the outcome on saturation of the left (the foci of the same set must be independent), that is when none of the bound variables are used (typically to saturate further) before the start of the next invertible phase. This is a rather artificial restriction from a backward-search perspective. Maximal multi-focusing is more elegant, declarative in this respect, but is less suited to proof search.

Unfortunately, it is unclear how to extend the definition of maximal multi-focusing in presence of units, in particular of the empty type 0. Two distinct left-focusing phases may both release the empty type 0 in the following invertible context, and this means that they be equated in the multi-focusing phase. We have worked on such formulations, but found them unsatisfying; the saturating logic, adapted to use selection functions, seems to lends itself to the empty type more gracefully.

13.3.2. Lollimon: backward and forward search together

We described in Section 11.2.5 (The roles of forward and backward search in a saturated logic) the way our saturated proof search mixes backward and forward search. It is interesting to compare it to Lollimon [López, Pfenning, Polakow, and Watkins, 2005], a system which similarly mixes backward and forward search.

Lollimon is part of the research on logic programming that understands the execution of logic program as given by the operational behavior of proof search in a well-chosen logic – typically with uniform proofs or focusing. Cut-elimination is not the only way to give an operational semantics to proof systems that is suitable for programming, proof search also has a rich "programmable" operational behavior.

More specifically, the research arc on Concurrent LF and related systems tries to studies a wider range of logic to capture the operational behavior of interesting systems, typically concurrent systems with several interacting actors or processes. Lollimon uses a mix of intuitionistic logic and linear logic – linear logic is suitable to represent consumable resources and, thus, essential to the modeling of systems with modifiable state.

In Lollimon, as in our case, forward search comes from the behavior of the left-focusing rule with positive conclusion, that is the forward-chaining rule of the logic. This forward search ingredient provides an elegant way to describe behaviors that are asynchronous (they do not necessarily rely on a communication between independent parts of a formula) but non-invertible – one example is the computation of a future alongside the rest of the program. Furthermore, when the forward search strategy performs forward search until saturation is reached, Lollimon can easily describe algorithms that rely on saturation, such as computing the transitive closure of a graph.

Because of this focus on representing the operation behavior of a variety of system, the Lollimon logic is not prescriptive: it does not actually enforce saturating or any other forward-search strategy, it is their implementation of the proof search algorithm that made specific implementation choices. In contrast, saturated logic is formulated is a strongly prescriptive way: while the choice of the saturation function gives some leeway, the logic enforces saturation phase as long as new hypotheses are present, and a form of completeness for provability through the SELECT-SPECIFIC restriction.

Saturated logic is prescriptive because we can afford it: in the more limited applications that we are interested in, either the search of a unique inhabitant or equivalence checking, there is a natural choice of selection function that allows some form of "full saturation" and yet remains terminating, so enforcing (restricted) saturation is practical.

I believe that the consideration of program terms – the type-theoretic rather than proof-theoretic setting – also gives some intuitions that would be harder to acquire in the Lollimon setting. Our distinction between "old" and "new" formulas would be possible in a purely logical setting, but the idea of only saturating on the neutrals that use the "new" formulas relies on the intuition of considering proof terms as programs – those new neutral may have new values that we did not know about yet. The saturation selection strategy used in our unicity-checking algorithm, the "two or more" criterion (we can keep at most two variables of each type to find out if two distinct programs are possible), would not at all be natural in a purely proof-theoretic setting.

13.4. Equivalence of terms in presence of sums

Ghani [1995b] first proved the decidability of equivalence of lambda-terms with sums, using sophisticated rewriting techniques. The two works that followed [Altenkirch, Dybjer, Hofmann, and Scott, 2001, Balat, Di Cosmo, and Fiore, 2004] used normalization-by-evaluation instead. Finally, Lindley [2007] was inspired by Balat, Di Cosmo, and Fiore [2004] to re-explain equivalence through rewriting. Our idea of "cutting sums as early as possible" was inspired from Lindley [2007], but in retrospect it could be seen in the "restriction (A)" in the normal forms of Balat, Di Cosmo, and Fiore [2004], or directly in the "maximal conversions" of Ghani [1995b].

Note that the existence of unknown atoms is an important aspect of our calculus. Without them (starting only from base types 0 and 1), all types would be finitely inhabited. This observation is the basis of the promising unpublished work of Ahmad, Licata, and Harper [2010], also strongly relying on (higher-order) focusing. Finiteness hypotheses also

play an important role in Ilik [2014], where they are used to reason on type *isomorphisms* in presence of sums.

In Munch-Maccagnoni and Scherer [2015], I collaborated with Guillaume Munch-Maccagnoni to rephrase the problem of sum equivalence in a notational framework of *abstract machine calculi* called System L. Historically this work comes from both the search for a term notation that would give a clear computational meaning to classical logic, and the fine-grained study of weak reduction strategies, notably the duality between call-by-name and call-by-value reduction. It subsumes both by using a "polarized" reduction strategy. In a typed setting – System L can also be studied as an untyped calculus – this "polarization" can be seen as going beyond focusing. In particular, the relation between System L's reduction and cut-elimination in strongly focused systems is similar to the relation between reduction in a direct-style effectful λ-calculus and an indirect-style monadic calculus.

13.5. Elaboration of implicits

Probably the most visible and the most elegant uses of typed-directed code inference for functional languages are *type-classes* [Wadler and Blott, 1989] and *implicits* [Oliveira, Moors, and Odersky, 2010]. Type classes elaboration is traditionally presented as a satisfiability problem (or constraint solving problem [Stuckey and Sulzmann, 2002]) that happens to have operational consequences. Implicits recast the feature as elaboration of a programming *term*, which is closer to our methodology. Type-classes traditionally try (to various degrees of success) to ensure *coherence*, namely that a given elaboration goal always give the same dynamic semantics wherever it happens in the program – often by making instance declarations a toplevel-only construct. Implicits allow a more modular construction of the elaboration environment, but have to resort to priorities to preserve determinism [Oliveira, Schrijvers, Choi, Lee, Yi, and Wadler, 2014].

We propose to reformulate the question of determinism or ambiguity by presenting elaboration as a *typing* problem, and proving that the elaborated problems intrinsically have unique inhabitants. This point of view does not by itself solve the difficult questions of which are the good policies to avoid ambiguity, but it provides a more declarative setting to expose a given strategy; for example, priority to the more recently introduced implicit would translate to an explicit weakening construct, removing older candidates at introduction time, or a restricted variable lookup semantics.

(The global coherence issue is elegantly solved, independently of our work, by using a dependent type system where the values that semantically depend on specific elaboration choices (for example a balanced tree ordered with respect to some specific order) have a type that syntactically depends on the elaboration witness. This approach meshes very well with our view, especially in systems with explicit equality proofs between terms, where features that grow the implicit environment could require proofs from the user that unicity is preserved.)

13.6. Smart completion and program synthesis

Type-directed program synthesis has seen sophisticated work in the recent years, notably Perelman, Gulwani, Ball, and Grossman [2012], Gvero, Kuncak, Kuraj, and Piskac [2013]. Type information is used to fill missing holes in partial expressions given by the users, typically among the many choices proposed by a large software library. Many potential completions are proposed interactively to the user and ordered by various ranking heuristics.

Our uniqueness criterion is much more rigid: restrictive (it has far less potential applications) and principled (there are no heuristics or subjective preferences at play). Complementary, it aims for application in richer type systems, and in *programming constructs* (implicits, etc.) rather than *tooling* with interactive feedback.

An aspect of interaction which could be interesting in our system is the *failure* case were at least two distinct inhabitants are found. A first question is, among all the possible counter-examples our algorithm could provide, which will be the more beneficial to the user? We suspect that having a computationally-observable difference as *early* in the terms as possible is preferable. A second is whether the user could interact with the system to refine the search space, possibly navigating between alternatives proposed by the system – for now the only refinement tools are type annotations.

Synthesis of glue code interfacing whole modules has been presented as a type-directed search, using type isomorphisms [Aponte and Di Cosmo, 1996] or inhabitation search in combinatory logics with intersection types [Düdder et al., 2014].

13.6.1. Focusing and program synthesis

We were very interested in the recent Osera and Zdancewic [2015], which generates code from both expected type and input/output examples. It is based on bidirectional type-checking, but we believe that it is in fact using focusing. The works are complementary: they have interesting proposals for data-structures and algorithm to make term search efficient, while we bring a deeper connection to proof-theoretic methods. They independently discovered the idea that saturation must use the "new" context, in their work it plays the role of an algorithmic improvement they call "relevant term generation".

This work has been expanded upon in Frankle, Osera, Walker, and Zdancewic [2016], and at the time of writing there is work underway to strengthen the connection to focusing. It is fully in line with the approach we proposed in Section (Motivation: Unicity as the ideal code inference criterion), and we hope to be able to study the connections more in detail. This work, notably, seems more advanced in terms of study of applicability to real scenarios, so a cooperation could be very fruitful.

14. Future work

14.1. A semantic proof of canonicity for saturating logic

I am uncomfortable with the proof technique used in the presented canonicity proof Theorem 11.4.7 (Canonicity of saturating focused logic) in Section 11.4 (Canonicity of saturated proofs). In my experience, proving canonicity by induction on a $\beta\eta$-equivalence derivation is fragile; for example, in very the first iteration of the proof, I completely forgot to prove the congruence case, considering only β and η-reductions at the toplevel of the case. This proof is crucial to the development, and in particular it is the one that justifies the correctness of our saturation approach as an *equivalence checking* algorithm, a question which deserves a better, robust, conclusive proof.

I would like to provide an alternative proof of canonicity using a more semantic proof technique using the results of Chapter 8 (Semantics). In the preparation of this document I attempted to prove that if two saturated derivations are not $(\approx_{\mathtt{icc}})$-equivalent, then they are semantically distinct (Section 8.3 (Semantic equivalence for $\mathsf{PIL}(\to, \times, 1, +, 0)$)), by building a model $\mathcal{M}$ in which their interpretation differ. This proof technique is simple on paper but the details are subtle; for example, the presence of the empty type implies that we may not be able to build a semantic valuation for all environments, and the results on saturated consistency in Section 11.3.2 (Saturated consistency) are crucial. I have lacked the time to finish this proof effort, but that is a goal in the short to medium term.

Manipulating semantic equivalence directly involves a fair amount of boilerplate, moving from terms to semantic values and conversely. An option to clarify such a proof would be to first propose a more syntactic logical relation that corresponds to semantic equivalence, and perform a proof against this logical relation.

Such a more semantic approach is that it would prove that saturation decides not only $\beta\eta$-equivalence, but more generally the contextual/semantic equivalence, the correct golden standard for equivalence. As a side-effect, this implies in particular – combined to the soundness result of Theorem 8.4.3 (Semantic soundness of $\beta\eta$-equivalence) – that contextual equivalence implies $\beta\eta$-equivalence, which is not a trivial result, even though it could be established more directly.

14.2. Pushing the application front

Despite some interesting experiments with our software prototype, we have not yet pushed efforts in the direction of practical application of this work to real-world programming language. I think that supporting richer type systems would help to make it more widely applicable, but it may already be possible to provide the current capabilities as a code inference tool for typed functional languages, and thus gather some usage experience.

14.3. Substructural logics

Instead of moving to more polymorphic type systems, one could move to substructural logics. We could expect to refine a type annotation using, for example, linear arrows, to get a unique inhabitant. We observed, however, that linearity is often disappointing in getting "unique enough" types. Take the polymorphic type of mapping on lists, for example: $\forall\alpha\beta.\,(\alpha \to \beta) \to (\mathtt{List}\ \alpha \to \mathtt{List}\ \beta)$. Its inhabitants are the expected map composed with any function that can reorder, duplicate or drop elements from a list.

Changing the two inner arrows to be linear gives us the set of functions that may only reorder the mapped elements: still not unique. An idea to get a unique type is to request a mapping from $(\alpha \leq \beta)$ to $(\texttt{List}\ \alpha \leq \texttt{List}\ \beta)$, where the subtyping relation $(\leq)$ is seen as a substructural arrow type.

(Dependent types also allow to capture `List.map`, as the unique inhabitant of the dependent induction principle on lists is unique.)

14.4. Equational reasoning

We have only considered pure, strongly terminating programs so far. One could hope to find monadic types that uniquely defined transformations of impure programs (e.g. $(\alpha \to \beta) \to \texttt{M}\ \alpha \to \texttt{M}\ \beta$). Unfortunately, this approach would not work by simply adding the unit and bind of the monad as formal parameters to the context, because many programs that are only equal up to the monadic laws would be returned by the system. It could be interesting to enrich the search process to also normalize by the monadic laws.[1] In the more general case, can the search process be extended to additional rewrite systems?

14.5. Unique inhabitation with polymorphism or dependent types

We have started experimenting with an extension of saturated proof search to System F, with no strong results so far.

The general problem with polymorphism is the loss of the subformula property, and thus the loss of termination in our algorithm – or any algorithm, as the problem becomes undecidable as shown by reducing unicity to inhabitation.[2] In the details, this appears when trying to build a negative neutral out of $\forall$-quantified formula during a left-focusing phase: there is an infinite space of possible instantiations choices.

First, remark that the algorithm of Chapter 11 (Saturation logic for canonicity) directly extends to the sub-system where $\forall$-quantifiers are only present in positive subformulas occurrences – this is the easy subset where no instantiation choices have to be made. Gilles Dowek and Ying Jiang studied this almost-non-polymorphic fragment in Dowek and Jiang [2009]; it gives a precise formal status to our handling of prenex polymorphism in our experiments. Note that formulas with positive $\forall$ occurrences are a more general fragment than just prenex polymorphism, although type systems such as Mitchell's $\mathsf{F}\eta$ [Mitchell, 1988] bridge the gap by allowing to lift positive quantifiers into prenex position by subtyping/containment.

Second, our suggestion for future work would be to replace the problem of "at a use site, how to instantiate this polymorphic neutral to make further progress", which leads to a natural explosion of the saturation dynamics – there will often be infinitely many strict positives to deduce – by the different question of "at the abstraction site, is there a set of instantiations that summarizes the polymorphic value in its full generality?".

For example, if the polymorphic type $\forall\alpha, (X^+ \to \alpha) \to (Y^+ \to \alpha) \to \alpha$ is in an invertible context, we could in a sense "invertibly decompose" it by instantiating it either with X^+ or with Y^+, as we can easily prove that no other instantiation leads to an inhabited type. Note that we are taking a "closed world" view here: we are assuming that the context has no other way to build a value of this type that we have ourselves, and thus that we can

[1] While reviewing this manuscript, Sam Lindley remarked that the specific case of monad laws should be relatively easy, as monad laws can be seen as a weaker form of sum laws. If we consider an abstract monad $\texttt{M}\ A$ as a sum $0 + A$, with the expected implementations of `bind` and `return`, the reduction and *weak* η-expansion on sums suffice to recover the usual monad laws – the equational theory of Eugenio Moggi's computational λ-calculus.

[2] Undecidability of inhabitation in System F is an old result recalled in Wells [1994] – an article that is itself related to the different issue of decidability of typability of a term.

reason on the possible values that were passed to us by enumerating the terms we could build ourselves at this type.

In a more general setting, this suggests a generalization of Noam Zeilberger's higher-order focusing rule [Zeilberger, 2009] that "decomposes" polymorphic hypotheses that could look somewhat like

$$\textsc{draft-polymorphic-higher-order-rule}\quad \frac{\forall \Sigma',\qquad \Sigma', \alpha \Vdash A(\alpha) \quad \Longrightarrow \quad \Gamma^{\mathsf{at}}; \Sigma, \Sigma' \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}{\Gamma^{\mathsf{at}}; \Sigma, \forall \alpha, A(\alpha) \vdash_{\mathsf{inv}} N \mid P^{\mathsf{at}}}$$

Where the $\Sigma \Vdash A$ relation ranges over the minimal set of contexts that must be inhabited for A to be inhabitable.

We have been trying to find a way to enumerate those "most general contexts" by reusing our (unicity-aware) proof search procedure on $A(\alpha)$, in a mode that would collect inhabitation constraints (the minimal context is an output, rather than an input, of the enumeration procedure). If this succeeded, it would give a new understanding of parametricity results in terms of syntactic proof search.

Note that the interaction between this idea of closed-world proof search and focusing is unknown and quite likely to be a delicate issue. The fact that $\forall$-quantifiers in positive position are invertibly introduced would suggest to consider polymorphic types as negatives, but our higher-order focusing approach instead consider them (in negative position) as positives.

Finally, on a more technical level, we think that extending our proof search procedure to System F (and beyond) would benefit from an explicit handling of metavariables as done in Lengrand, Dyckhoff, and McKinna [2011]. Explicit meta-variables let us explicitly represent the state of proof search as a derivation, and this let us explore a richer setting of proof search strategies – choices metavariable instantiation order – notably breadth-first search strategies. Without this explicit representation of search state, the natural approach is to have a recursive proof search procedure that provides complete proof of each judgment when called, so it imposes a depth-first approach. This inflexibility is acceptable in a simply-typed setting where each search branch terminates, but in a undecidable setting it makes the system halt as soon as some subspace becomes infinite – we would hope for a better behavior in this case.

Bibliography

Bibliography

Arbob Ahmad, Daniel R. Licata, and Robert Harper. Deciding coproduct equality with focusing. Online draft, 2010. 183, 274

Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for lambda^{-2}. In *FLOPS*, 2004. URL www.cs.nott.ac.uk/~psztxa/publ/flops04.pdf. 183

Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, 2001. 183, 231, 274

Jean-Marc Andreoli. Logic Programming with Focusing Proof in Linear Logic. *Journal of Logic and Computation*, 1992a. 272

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992b. doi: 10.1093/logcom/2.3.297. URL http://logcom.oxfordjournals.org/content/2/3/297.abstract. 20, 154

Takahito Aoto. Uniqueness of normal proofs in implicational intuitionistic logic. *Journal of Logic, Language and Information*, 1999. 271

Takahito Aoto and Hiroakira Ono. Non-Uniqueness of Normal Proofs for Minimal Formulas in Implication-Conjunction Fragment of BCK. *Bulletin of the Section of Logic*, 1994. 266, 271

Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *PLILP*, 1996. 276

Ali Babaev and Sergei Soloviev. A coherence theorem for canonical morphisms in cartesian closed categories. *Journal of Soviet Mathematics*, 1982. 271

Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 2004. 183, 231, 234, 274

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inf. Comput.*, 119(2), 1995. 87

Alessandro Berarducci and Corrado Böhm. Automatic synthesis of typed [Lambda]-programs on term algebras. *Theoretical Computer Science*, 1985. 65

Pierre Bourreau and Sylvain Salvati. Game semantics and uniqueness of type inhabitance in the simply-typed λ-calculus. In *TLCA*, 2011. 271

Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In *LPAR-17*, 2010. 202, 272

Sabine Broda and Luís Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 2005. 271

Aloïs Brunel. *The monitoring power of forcing program transformations*. PhD thesis, Université Paris 13, June 2014. URL https://hal.archives-Touvertes.fr/tel-T01162997. 172

Kaustuv Chaudhuri. Focusing strategies in the sequent calculus of synthetic connectives. In *LPAR*, 2008. 164

Kaustuv Chaudhuri. Magically constraining the inverse method using dynamic polarity assignment. In *LPAR*, October 2010. URL https://hal.inria.fr/inria-T00535948. 243

Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In *CSL*, 2005. 272

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *IFIP TCS*, 2008a. 273

Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. volume 40, 2008b. 156, 243

Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A Systematic Approach to Canonicity in the Classical Sequent Calculus. In *CSL*, 2012. 273

Julien Crétin. *Erasable coercions: a unified approach to type systems*. PhD thesis, Université Paris-Diderot - Paris VII, January 2014. 75

Julien Cretin and Didier Rémy. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM, July 2014. 87

Valéria de Paiva and Luiz Pereira. A short note on intuitionistic propositional logic with multiple conclusions. *Manuscrito*, 2005. 125

Anatoli Degtyarev and Andrei Voronkov. Introduction to the inverse method. In *Handbook of Automated Reasoning*. 2001. 272

Kosta Dosen. Identity of proofs based on normalization and generality. *Bulletin of Symbolic Logic*, 2003. 271

Gilles Dowek and Ying Jiang. Enumerating proofs of positive formulae. *Comput. J.*, 52 (7), 2009. 278

Gilles Dowek and Ying Jiang. On the expressive power of schemes. *Inf. Comput.*, 2011. 271

Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, 2014. 276

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 1992. 109, 272

Roy Dyckhoff. Intuitionistic decision procedures since gentzen, 2013. Talk notes. 109, 272

Mahfuza Farooque, Stéphane Graham-Lengrand, and Assia Mahboubi. A bisimulation between dpll(T) and a proof-search strategy for the focused sequent calculus. In *LFTMP*, 2013. 243

Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Ann. Pure Appl. Logic*, 2006. 33

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016. 276

Didier Galmiche and Daniel Méry. A connection-based characterization of bi-intuitionistic validity. *J. Autom. Reasoning*, 2013. 272

Herman Geuvers. Inductive and Coinductive Data Types in Typed Lambda Calculus Revisited, July 2015. URL http://www.cs.ru.nl/~herman/talk_TLCA2015.pdf. Slides from an excellent invited talk at TLCA'15, Warsaw. 65

Neil Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, November 1995a. 234

Neil Ghani. Beta-Eta Equality for Coproducts. In *TLCA*, 1995b. 231, 274

Timothy G Griffin. A formulae-as-type notion of control. In *POPL*, 1989. 117

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013. 275

Fritz Henglein and Ralf Hinze. Sorting and searching by distribution: From generic discrimination to generic tries. In *APLAS*, 2013. 92

Hugo Herbelin. A Lambda-calculus Structure Isomorphic to Gentzen-style Sequent Calculus Structure. In *CSL*, 1994. URL https://hal.inria.fr/inria-T00381525. 157, 271

Hugo Herbelin. C'est maintenant qu'on calcule, au cœur de la dualité, 2005. URL http://pauillac.inria.fr/~herbelin/habilitation/memoire+errata.pdf. habiliation thesis. 164

Danko Ilik. Axioms and decidability for type isomorphism in the presence of sums. *CoRR*, abs/1401.2567, 2014. URL http://arxiv.org/abs/1401.2567. 33, 275

Danko Ilik. The exp-log normal form of types and canonical terms for lambda calculus with sums. *CoRR*, abs/1502.04634, 2015. URL http://arxiv.org/abs/1502.04634. 183

Delia Kesner. The theory of calculi with explicit substitutions revisited. In *Computer Science Logic*, 2007. URL https://hal.archives-Touvertes.fr/hal-T00111285. 110

Edward Kmett. Lens, 2012. URL https://github.com/ekmett/lens. 265

Edward Kmett. Lens wiki – types, 2013. URL https://github.com/ekmett/lens/wiki/Types. 266

Joachim Lambek and Philip Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986. 90, 91

Olivier Laurent. A proof of the focalization property of linear logic. 2004. 213

Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A focused sequent calculus framework for proof search in Pure Type Systems. *Logical Methods in Computer Science*, 7(1), 2011. 164, 279

Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. *CoRR*, 2007. URL http://arxiv.org/abs/0708.2252. 155, 213, 272

Sam Lindley. Extensional rewriting with sums. In *TLCA*, 2007. 231, 274

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP*, 2005. 244, 273

Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In *LPAR*, 2008. 265, 273

Dale Miller and Alexis Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In *CSL*, 2007. 273

Grigori Mints. Closed categories and the theory of proofs. *Journal of Soviet Mathematics*, 1981. 271

John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76), 1988. 278

Guillaume Munch-Maccagnoni. Focalisation and Classical Realisability. In *CSL*, 2009. 172

Guillaume Munch-Maccagnoni. Calcul l pour les séquents. Exposé au Groupe de Travail de Logique, 2012. 177

Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised intermediate representation of lambda calculus with sums. In *LICS*, 2015. URL https://hal.inria.fr/hal-T01160579. 132, 275

Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010. 275

Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi, and Philip Wadler. The implicit calculus: A new foundation for generic programming. 2014. 275

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015. 276

Jens Otten and Christoph Kreitz. A uniform proof procedure for classical and non-classical logics. In *KI Advances in Artificial Intelligence*, 1996. 272

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012. 275

Gabriel Scherer. Multi-focusing on extensional rewriting with sums. In *TLCA*, 2015a. URL http://gallium.inria.fr/~scherer/drafts/multifoc_sums.pdf. 132, 231, 246, 273

Gabriel Scherer, 2015b. URL http://gallium.inria.fr/~scherer/research/unique_inhabitants/. 255, 265

Gabriel Scherer and Didier Rémy. Full Reduction in the Face of Absurdity. In *ESOP'15*, 2015. URL http://gallium.inria.fr/~remy/coercions/Remy-TScherer!fich@esop2015.pdf. 50, 75

Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *ICFP*, 2015. URL http://gallium.inria.fr/~scherer/research/unique_inhabitants/unique_stlc_sums-Tlong.pdf. 212, 215, 227, 234, 241, 242, 246, 263

Aleksy Schubert and Ken-etsu Fujita. A note on subject reduction in $(\rightarrow, \exists)$-curry with respect to complete developments. *Inf. Process. Lett.*, 114(1-2), 2014. 87

Robert J. Simmons. Structural focalization. *CoRR*, abs/1109.6273, 2011. 164, 213

Richard Statman. The typed lambda-calculus is not elementary recursive. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 1977. 52

Colin Stirling. Proof systems for retracts in simply typed lambda calculus. In *Automata, Languages, and Programming - ICALP*, 2013. 271

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP*, 2002. 275

David Turner. Elementary strong functional programming. In *FPLE'95*, 1995. URL http://hssc.sla.mdx.ac.uk/staffpages/dat/fple.pdf. 65

Nikolay Vorob'ev. A new algorithm of derivability in a constructive calculus of statements. In *Problems of the constructive direction in mathematics*, 1958. 109, 272

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989. 275

Lincoln A. Wallen. Automated proof search in non-classical logics: Efficient matrix proof methods for modal and intuitionistic logic, 1987. 272

Joe B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *LICS*, July 1994. 278

Joe B. Wells and Boris Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, 2004. 271

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, November 1994. URL http://www.cs.princeton.edu/~appel/proofsem/papers/wright94.ps. 68, 78

Marek Zaoinc. Fixpoint technique for counting terms in typed lambda-calculus. Technical report, State University of New York, 1995. 271

Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009. 221, 279

Noam Zeilberger. Polarity in proof theory and programming, August 2013. URL http://noamz.org/talks/logpolpro.pdf. Lecture Notes for the Summer School on Linear Logic and Geometry of Interaction in Torino, Italy. 50, 167

Remerciements

Une thèse, c'est beaucoup de travail sur une période assez longue pendant laquelle on n'est pas seul, heureusement.

Faire une thèse, c'est montrer que l'on a appris à faire de la recherche. Didier Rémy est mon directeur de thèse, c'est lui qui me l'a appris. Didier est mon directeur de thèse préféré; il est disponible et agréable, rigoureux et intéressant. Je retiendrai en particulier son sens du détail, de la bonne présentation, de la bonne notation, qui est précieux quand il faut dégrossir une idée nouvelle ou mettre le doigt sur un problème. J'ai aussi eu souvent l'occasion d'admirer le lien intime entre son intuition concrète de l'activité de programmation et le choix des outils formels utilisés dans sa recherche; en cela il incarne une approche de la recherche en langage de programmation qui m'a toujours convaincue.

J'ai eu le plaisir de côtoyer Roberto Di Cosmo à l'IRILL, en tant que collègue si sympathique et directeur dévoué, mais aussi en admirant sa pédagogie comme élève de son cours de logique linéaire, puis chargé de TP dans son cours de programmation fonctionnelle avancée, exigeant et gratifiant pour ses enseignants comme pour ses élèves. Nous partageons aussi une appartenance aux communautés OCaml et du logiciel libre. Je ne m'attendais pas, en commençant ma thèse, à m'appuyer aussi sur ses travaux de recherche, et c'est sous ces aspects si divers que je suis fier qu'il ait accepté de présider mon jury.

J'ai rencontré Dale Miller en suivant sa partie du cours de logique linéaire au MPRI, qui m'a ouvert les yeux sur la programmation logique comme une autre interprétation calculatoire de la logique. Mon travail s'inscrit dans la tradition opposée de la programmation fonctionnelle, mais utilise les outils logiques du focusing que Dale et son équipe, Parsifal, a su étendre et faire fructifier. J'ai eu grand plaisir à interagir directement avec ses membres (en particulier Stéphane, Kaustuv, Noam, Taus, Danko, Zak, Sonia, Tomer, Ulysse et Nicolas) et étudier leurs travaux.

Sam Lindley est l'auteur de travaux sur lequel je me suis directement appuyé pendant ma thèse, et aussi une personne qu'il est toujours agréable de rencontrer, frappant par sa vision accueillante et modeste de la recherche. Il m'a rendu un grand service, ainsi que Dale Miller, en acceptant d'être rapporteur de ma thèse; je crains que mes choix d'écriture n'aient fait de cette activité une corvée plus pénible qu'ils ne le laissent paraître.

Gilles Dowek et Olivier Laurent ont accepté d'être membres de mon jury, et je les en remercie grandement. J'ai peu interagi directement avec Olivier, mais j'ai pu admirer son intransigeance et son dévouement à l'organisation de la recherche française. J'ai pris beaucoup de plaisir à interagir avec Gilles et les membres de son équipe (en particulier Simon, Arnaud, Ali, Ronan, Raphaël et Bruno) qui animaient le cinquième étage de la Place d'Italie quand j'y travaillais.

Merci aussi à celles et ceux qui m'ont fourni une aide précieuse en acceptant de relire des chapitres de ce manuscrit. En plus de mon directeur et des membres de mon jury, Anne Lacerna a relu les chapitres 1 et 2, Luc Maranget et Adrien Guatto le chapitre 3, Pierre Courtieu le chapitre 4, Thomas Williams le chapitre 5 et Max New le chapitre 7. Ce document a grandement profité de leurs points de vue divers et remarques bienveillantes.

J'ai eu grand plaisir à travailler à Gallium pendant mes années de thèse. J'ai découvert cette équipe pendant mon stage de M1, moment d'adaptation et de découverte de la pause café Gallium. L'ambiance de Gallium est la meilleure que j'ai rencontrée dans un laboratoire de recherche. Pendant ma thèse, au bout de deux semaines d'absence, Gallium commençait à me manquer. J'y ai rencontré des collègues qui sont devenus des amis, membres permanents (Xavier, Didier, Luc, François, Damien), thésards vieux (Tahina, Nicolas, Arthur, Benoît, Alexandre) ou moins vieux (Jonathan, Jacques-Henri), post-docs

(Thomas et Thibaut, Pierre-Évariste et Maxime, Mike et Filip) et stagiaires ou visiteurs (Valentin, Joseph, Raphaël, Armaël, Cyprien, Robert, Sigurd). J'espère que les nouveaux jeunes (Thomas, Vitalii, Armaël, Ambroise et Jacques-Pascal) sauront prendre la relève et inciter les autres membres à écrire des billets de blog de temps en temps.

Andreas Abel et Jérôme Vouillon m'ont encadré en stage. J'ai pris plaisir à travailler avec eux et j'espère avoir l'occasion de recommencer. Je remercie aussi mes autres collaborateurs : Guillaume, Pierre-Évariste et Lionel, Thomas et Jonathan, Jan, et Silvain au tout début. Je n'ai pas toujours été un bon collaborateur; être mal organisé ou prendre trop de responsabilités est une habitude personnelle, mais j'en ai parfois fait souffrir les autres aussi.

Le centre INRIA de Rocquencourt était un endroit aussi pénible d'accès qu'agréable à vivre, et je remercie les gens que j'ai eu le plaisir d'y rencontrer (en particulier Thierry, Pauline, Thomas, Sarah, Victorien, Renaud et Maël) – merci aussi aux efforts constants de Jonathan de rencontrer des gens en dehors de son équipe, qui ont fait vivre notre coin café et permis certaines de ces rencontres. Merci au personnel qui a su faciliter notre travail, nos assistantes Stéphanie, Virginie et Cindy, le personnel de la cantine de Rocquencourt (ma cantine préférée) et du centre, qui ont su faire vivre et maintenir un endroit agréable et propice au travail.

Plusieurs laboratoires de recherche m'ont accueilli comme invité à plusieurs reprises et j'en garde un très bon souvenir. En plus de Parsifal et Deducteam, j'ai passé une grande partie de ma thèse à PPS; un jour, Thomas est venu me demander, avec l'air embêté de celui qui a été distrait, de lui rappeler lequel des membres du labo était mon directeur de thèse. Merci à ses nombreux membres avec qui j'ai interagi. J'ai aussi eu le plaisir de passer du temps à Parkas, en particulier dans de longs cafés avec Adrien, Guillaume et Nhat – et parfois aussi Tim, Marc ou Albert. Merci à Tie à Abstraction, puis avec François à Antique. Enfin, j'ai passé peu de temps à Plume, à Lyon, mais je m'y suis très bien senti – et je regrette d'avoir découvert le Chocola si tard dans ma thèse car c'est une denrée rare. Les rencontres francophones annuelles, que ce soient les JFLA ou bien les groupes de travail de GdR (LTP, LAC et GeoCal) sont des lieux qui m'ont intéressé et auxquels j'ai pris plaisir à contribuer. Une pensée aussi pour le groupe de travail de logique de Marc Bagnol, le joyeux mélange des études parisiennes.

J'ai beaucoup apprécié mes activités d'enseignement; apprendre la programmation est un sujet qui reste passionnant. J'ai enseigné le Caml en classe préparatoire et en M1, le Java en L1 et le C en L3; j'ai apprécié mes élèves, très divers selon les groupes, et les multiples facettes de cette activité. Merci à mes élèves, que j'ai plaisir à retrouver au hasard des rencontres, et à mes collègues enseignants, en particulier Yann et ses efforts d'encadrement, et Juliusz et ses techniques de fourbe.

Ma vie non-professionnelle pendant ces années de thèse a été organisée en collaboration avec Irène; nous avons eu grand plaisir à vivre à Paris, et en particulier d'y voir nos ami-e-s. Merci à eux et elles. Je me permets de ne pas citer chacun et chacune nommément; d'une part, cela ferait beaucoup de gens et j'ai très peur d'en oublier, d'autre part je préfère concentrer ces remerciements sur la thèse comme activité professionnelle.

Merci aussi à la famille d'Irène, qui a été accueillante dès le premier jour (rates comprises) et que j'ai grand plaisir à retrouver – ce manuscrit en particulier doit beaucoup à un séjour dans le pays basque avec rédaction de neuf à cinq et baignade ensuite.

Merci à ma famille, restreinte comme étendue, dont j'aime la compagnie. Je crois que j'ai souvent été un peu grincheux pendant les week-end où je retrouvais mes parents, et j'espère que c'est lié à un excès de travail (aussi difficile à croire que cela puisse sembler) plutôt qu'un trait de caractère permanent. Nous verrons.

Merci à Irène.