Extending Implicit Computational Complexity and Abstract Machines to Languages with Control

Giulio Pellitta

Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXVI

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF01

Extending Implicit Computational Complexity and Abstract Machines to Languages with Control

Presentata da: Giulio Pellitta

Coordinatore Dottorato:
Maurizio Gabbrielli

Relatore:
Simone Martini

Esame finale anno 2014

Abstract

The Curry-Howard isomorphism is the idea that proofs in natural deduction can be put in correspondence with lambda terms in such a way that this correspondence is preserved by normalization. The concept can be extended from Intuitionistic Logic to other systems, such as Linear Logic. One of the nice conseguences of this isomorphism is that we can reason about functional programs with formal tools which are typical of proof systems: such analysis can also include quantitative qualities of programs, such as the number of steps it takes to terminate. Another is the possiblity to describe the execution of these programs in terms of abstract machines.

In 1990 Griffin proved that the correspondence can be extended to Classical Logic and *control operators*. That is, Classical Logic adds the possiblity to manipulate *continuations*. In this thesis we see how the things we described above work in this larger context.

Acknowledgements

Writing acknowledgements is no easy task, since one risks forgetting about many people whose contribution has been precious, if not crucial. Hence, I would first like to thank all the professors, technicians, colleagues and friends with whom I had a chance to interact.

I would like to thank Ugo Dal Lago for the time spent teaching me and guiding me during my studies, I have learned a lot from him. I would also like to thank Simone Martini for accepting to be my supervisor and for all the advice and encouragement he has given me. I am grateful to them both, without them this thesis may not have been possible. I am grateful to Stefano Guerrini for the opportunity of working six months under his supervision in Paris XIII.

I am thankful to Davide Sangiorgi for having me in the Focus team, and for leading it with enthusiasm. I would also like to thank Maurizio Gabbrielli, for fulfilling competently his role as Coordinatore of the PhD program these last three years.

I would also like to thank Marco Gaboardi and Virgile Mogbil for reviewing my thesis, which I have improved also thanks to their suggestions.

I would like to thank the various members of LIPN I have had the chance of interact with while I was in Paris, especially those of the LCR team.

I also want to thank all the friends I have had these last three years in Bologna. Among them my office colleagues, here is a non-exhaustive list (in no particular

order): Paolo Parisen Toldin, Jacopo Mauro, Marco Solieri, Andrea Giovanni Nuzzolese, Ornela Dardha, Tudor Alexandru Lascu, Sara Zuppiroli, Jean-Marie Madiot, Alessandro Rioli, Giuseppe Profiti, Francesco Poggi, Ivan Lanese, Gustavo Marfia, Silvio Peroni, Michael Lienhardt, Wilmer Ricciotti, Dominic Mulligan, Jaap Boender, Claudio Sacerdoti Coen, Marco Di Felice, Saverio Giallorenzo and Beniamino Accattoli.

Last but not least, I am especially grateful to my parents for their unfailing support and encouragement.

Contents

Contents

Thesis Outline

In Part I we analyze some variants of $\lambda\mu$-calculus (an extension of λ-calculus with two control operators for manipulating continuations), with special attention to the calculus of de Groote; and the main ideas of implicit computational complexity (a series of tools and techniques to characterize complexity classes), especially restrictions of Linear Logic with limited complexity and in particular Bounded Linear Logic.

In Part II we investigate how to adapt techniques of implicit computational complexity to characterize a fragment of $\lambda\mu$-calculus which is expressive enough to represent all functions that can be computed in time polynomial w.r.t. the input size.

In Part III we study some abstract machines for evaluating λ- and $\lambda\mu$-terms; we ultimately review an abstract machine for lambda terms which has not received a lot of attention in the last few years and adapt it to $\lambda\mu$-calculus.

What is not here During my PhD I have devoted myself to other works, some of which are still in progress, while others are not yet mature for publication. Although they are all on more or less related topics, I have decided not to included them in my thesis.

- I have written a paper on abstract machines based on Geometry of Interaction, together with Marco Pedicini and Mario Piazza, aimed at formalizing a parallel implementation named PELCR [PQ07] as a stream-processing machine. I have also envisioned connection with control operators (inspired by [LM00]), but the paper is currently being rewritten and extended.

- Paolo Parisen Toldin and I have worked on establishing a connection between the existence of complete problems and the existence of an ICC system for a complexity class. The most interesting case studies of the subject of the paper are **BPP** (see [DLPT12b, §1.3, 5] and other semantical classes. The results obtained so far are encouraging but still preliminary.

Original Contributions The original material can be found mainly in Chapters 4 and 6. Chapter 4 is based on [DLP13a] (a joint work with PhD Ugo Dal Lago), while Chapter 6 contains (yet) unpublished material which is the result of six months spent studying under the supervision of Professor Stefano Guerrini.

Part I

Introduction

Chapter 1

Continuations

1.1 Introduction

The continuation (a concept that dates back at least to the 1970s, cf., e.g., [Rey72]) is the rest of the computation, i.e., the context of the expression currently being evaluated. A good intuition (somewhat informal, but operationally accurate) is that a continuation is an expression with a "hole": once the hole is filled the result of the whole expression is returned. For example, in $((\lambda f.\lambda x.(f)(f)x)\lambda y.y)(\underline{2})\underline{2}$ (where as usual $\underline{n}$ denotes the Church integer for n) the continuation of $(\lambda f.\lambda x.(f)(f)x)\lambda y.y$ is $(\_)(\underline{2})\underline{2}$, where $\_$ is a hole that can be filled once we have evaluated the sub-term $(\lambda f.\lambda x.(f)(f)x)\lambda y.y$. The expression `(3*2)+1` may be decomposed into `(3*2)` and `_+1`. An easy way to get an intuitive idea of continuations is to think of them as a kind of `goto` [Dij68], but more structured, less arbitrary[Cun12]. Basically the idea of control is that once the continuation is *named*, then the parameter corresponding to the continuation can be bound and it becomes possible to explicitly manipulate the execution flow.

Enriching a language with continuations improves its expressiveness and helps the programmer write more readable code, since he is not forced to code how to manipulate the control flow, but can use high-level primitives to do the job. Furthermore the enrichment provided by continuations to functional languages permits to extend the analysis techniques typical of the functional world to languages that

have some features with imperative-flavor. Quasi-functional languages are sort of best of both worlds.

Several concrete functional-languages provide some facilities for handling continuations, i.e., treat continuations as first-class objects. The most known example is Scheme, which has the primitive `callcc` (i.e., *call-with-current-continuation*), which we discuss later on in more details. Other examples include Scala (2.8 and later versions), OCaml (e.g., through the library `delimcc`).

Continuation-Passing-Style Continuations can be used in two fashions. Either the program is written in *continuation-passing-style* (CPS), in which continuations have to be managed explicitly, i.e., functions have an extra argument which corresponds to the continuation. Or the control flow of a program is manipulated by suitable operators, an approach called *direct-style* (DS) as opposite of CPS. In cases as simple as those above there is no need to talk about continuations. Conversely, continuations are useful when one needs to have a non-sequential flow of control: they can be used to implement coroutines [HFW86], exceptions, back-tracking, threads or multiprocessing [Wan80], etc...

On one hand, even though it is possible to do so in CPS, the resulting program can be extremely complex, making considerably hard for the programmer to reason about it (study its semantics, running time, etc...). On the other hand the CPS representation can be useful in those situation when one needs precise control of the execution flow. For example, some compilers for higher-order languages can use a CPS translation as an intermediate representation for the code [FSDF93] [App07]. Furthermore, translating a richer language (e.g., including extra feautres or new evaluation strategies) into a simpler, more primitive language allows to define the extended semantics of the first language in terms of the latter [Rey72] (in particular, there is no need to build new interpreters/compilers).

It should be noted that a straightforward/naïve CPS translation adds a number of so-called *administrative redexes*, so the program obtained can be significantly

bigger than the original, not to mention less efficient.[1]

A CPS translation is also a way to enforce a particular evaluation order. Different evaluation strategies correspond to different CPS translations.

(Non-)Linear Continuations Once the programmer has access to the current continuation, he can simply save the current context and resume it later. This is but the most basic usage of continuations. Once a continuations has been captured it can be used any number of times.

A program can, for an instance, enter a procedure twice. This can be a crucial issue from an efficiency point of view, since a trivial implementation may simply duplicate the corresponding stack/continuation.

It is also possible to use a continuation zero times, i.e., *abort* or *escape* a computation. This corresponds to do a `return` within the code, i.e., continuations allows (possibly multiple) returns to escape from nested procedures.

(Un)delimited Continuations Whenever a continuation is captured, the whole context of the current expression is saved on a stack. The current stack may consist of an arbitrary number of arbitrarily large arguments. But sometimes only part of the continuation is actually needed. In the latter case we talk of *delimited* continuation (e.g., the operators shift and reset [DF90] [DF92]).

Undelimited continuations operators capture the whole continuation and are thus less efficient, although all known delimited continuations operators can be implemented in terms of undelimited continuations operators (and mutable state) [Sha04]. For example the operator `callcc` can be used to implement a number of facilities, but for some reasons it is sometimes preferred not to use it [PCM+05] [Sai12].

[1] Some more sophisticated CPS translation may reduce the administrative redexes. It is also possible to use some optimizing compilations specifically tailored to deal with continuation parameters (in certain cases the CPS representation allows to perform more optimization than would be possible on the source language [App07]).

1.2 Applications

1.2.1 Imperative Languages

Krivine has studied the use of control operators from *Classical Logic* to model some features of imperative programming in functional languages using an extended notion of head-reduction [Kri96].

As argued by Krivine, a call-by-name evaluation has a number of advantages (head reduction strategy always yields a normal form, if it exists; and using programs with side-effects does not produce unintended results), but it forces to evaluate the argument of a function as many time as it is used. To avoid this drawback, *storage operators*[2] [Kri94] can be used when we need to simulate call-by-value, thus enriching lambda calculus with imperative-like assignments.

Furthermore, control operators can be used in various way to escape from a procedure, similarly to what happens in the C language with a `break` or `exit` instruction (which can be simulated by an *abort* operator) or more sophisticated commands. See also Section 1.3.1.

1.2.2 Programs for the Web

In [KHM+07] Felleisen et al. describe a Web server written in PLT Scheme that uses first-class continuations to enable direct-style programming for Web applications.

> According to Queinnec, Web browsing in the presence of dynamic Web content generation is the process of capturing and resuming continuations.

A proper handling of continuations at the level of programming language is essential for having a code which is both natural to write and efficient to execute.

Among other things they present an interesting application for conference paper management called CONTINUE which has been used in several workshop and conferences.

[2] Called output operators by Parigot, see Section 1.3.2.

1.2.3 Process calculi

Abramsky shows in [Abr94] that there is an analogy between cut-elimination in Linear Logic and process I/O interaction (e.g., in CCS [Mil82]). In the one-sided presentation of Linear Logic the cut rule corresponds to an operator of *parallel composition*.

Other works exhibiting a connection between calculi with continuation and concurrency can be found in literature. For example, an encoding of $\bar{\lambda}\mu\tilde{\mu}$ calculus [3] into π-calculus, whose typed image identifies a non trivial-subset of terminating processes can be found in [CCS09]. More recently, proofs in multiplicative linear logic have been interpreted as scheduling for processes [BM12].

On a closely related topic, it has been known for some time that continuations can be used to model concurrent operating systems [Wan80] [Shi97] since they provide a clean and efficient way to do save and restore operations.

1.2.4 Natural language

Let us conclude with something quite different. De Groote has studied the use of continuation to cope with semantic ambiguities of quantifiers (whose semantical scope may be larger of the syntactical scope) [DG01b]. The idea of using logic for a number of linguistic phenomena has been analyzed also in the context of Linear Logic [MP01].

1.3 Computational Interpretation of Classical Logic

For a long time Classical Logic was believed to have no computational content, i.e., that the Curry-Howard isomorphism does not extend to Classical Logic. The correspondence was restricted to constructive logics (such as Intuitionistic and Linear Logic) since classical cut-elimination behaves badly in presence of structural rules

[3] A non-confluent classical extension of λ-calculus [CH00].

and thus there seemed to be no chance of a meaningful computational interpretation. However, the situation changes if one allows computation to explicitly control the current continuation.

1.3.1 Control Operators

Griffin [Gri90] has shown a correspondence between classical propositional logic and a typed Idealized Scheme containing a control operator (Felleisen's $\mathcal{C}$) which (roughly) resembles Scheme's `callcc`. More precisely, he has shown that the operator $\mathcal{C}$ of Felleisen can be given a type which corresponds to the law of *double negation elimination*.

$$\neg\neg A \Rightarrow A \tag{1.1}$$

where $\neg(\cdot)$ should be interpreted constructively/intuitionistically as $(\cdot) \Rightarrow \bot$. Finally, in [Gri90, §6] it is shown that CPS translations correspond to the encoding of classical logic into constructive logic.[4] The operator `callcc` [Fel90], instead, can be given the type corresponding to *Peirce's Law* (PL).

$$((A \to B) \to A)) \to A. \tag{1.2}$$

Logical axioms have different relative strength and correspondingly control operators have different expressive power. A thorough comparison of the relative expressive power of control operators can be found in [AH03]. See also Section 1.3.5.

Felleisen's $\lambda_{\mathcal{C}}$ The calculus studied by Felleisen [FFKD87] [FH92], usually denoted $\lambda_{\mathcal{C}}$ is a call-by-value lambda calculus enriched with an operator $\mathcal{C}$. Its syntax is the following

$$t, u = x \quad | \quad \lambda x.t \quad | \quad (t)u \quad | \quad (\mathcal{C})t. \tag{1.3}$$

[4] The first calculus of control, namely $\lambda\mu$-calculus [Par92] (see next section), derives from classical natural deduction. Which is not too surprising considering that Griffin used Classical Logic to study control operators.

Let $\mathcal{A}$ be the *abort operator* defined by $(\mathcal{A})t \stackrel{\text{def}}{=} (\mathcal{C})\lambda x.t$, where $x \notin t$. The operational semantics is obtained adding two rewriting rules

$$C[(\lambda x.t)v] \to C[t[v/x]] \tag{C1}$$

$$C[(\mathcal{C})t] \to (t)\lambda x.(\mathcal{A})C[x] \tag{C2}$$

where v is a value and v/x is the substitution of v in place of x (with usual α-conversion to avoid capture of variables) and $C[\,]$ is an *applicative context* defined by:

$$C[\,] = \quad [\,] \quad | \quad (C[\,])t \quad | \quad (v)C[\,]. \tag{1.4}$$

Felleisen's $PS(\texttt{callcc})$ Here we briefly sketch a calculus with control by Felleisen, obtained adding `callcc` to *Pure Scheme* (PS), an untyped call-by-value lambda calculus enriched with numerals, numeric constants, ... We do not give the syntax or the semantics of PS, and we only give a partial description of the syntax or the semantics of the calculus extended with control, the reader is referred to [Fel90].

The syntax is enriched as follows:

$$u := \ldots \quad | \quad (\texttt{callcc})u. \tag{1.5}$$

Extra rules are added to the semantics (only a few here, see [Fel90, §3] for all details)

$$(\texttt{callcc})\lambda k.C[(k)e] \to (\texttt{callcc})\lambda k.e \tag{1.6}$$

$$(\texttt{callcc})\lambda k.e \to e, \quad k \notin e \tag{1.7}$$

$$\ldots$$

where $C[\,]$ denotes a call-by-value evaluation context as in Equation (1.4).

1.3.2 Parigot's $\lambda\mu$-Calculus

In [Par92] Parigot defines a classical extension of lambda calculus. The calculus, which enjoys confluence and subject reduction, is obtained adding two new operators μ and $[\cdot]$.

Term grammar

The syntax term is given by the following definition

$$u, v := x \quad | \quad \lambda x.u \quad | \quad \mu\alpha[\beta]u \quad | \quad (u)v \tag{1.8}$$

where α and β belong to a denumerable set of variables (disjoint from usual λ-variables). μ-variables, also known as μ-*names* or simply names [MPW92], correspond to communication channels or to stack names. The μ operator is a binder for μ-variables, so β is free in $\mu\alpha[\beta]u$ iff $\alpha \neq \beta$. The μ and $[\cdot]$ operators are called μ-*abstraction* and *naming* for the reasons we have just explained.

Typed $\lambda\mu$-calculus

Let X denote an atomic type and let A denote a type obtained by $\perp$ and atomic formulas by $\rightarrow$ and (for simplicity, here we omit first and second order universal quantifier $\forall$). Let Γ, Π (resp. Δ, Σ) be sets of formulas indexed by λ-variables (resp. μ-variables). Sequents are expressions of the form $\Gamma \vdash t : A \mid \Delta$, where t is any term and A is called the *active* formula. Γ, Γ' denotes the union of Γ and Γ', with the convention that they do not contain two contradictory type declarations $x : A$ and $x : B$ with $A \neq B$ (basically an implicit form of contraction). We write A^x when we want to explicit that A is indexed by the variable x. All formulas to the left of $\vdash$ are indexed by λ-variables, while all formulas to the right of $\vdash$, except at most one, are indexed by μ-variables.

In [Par92] Parigot uses the notation $t : \Gamma \vdash A, \Delta$, which we change slightly for the sake of uniformity and readability. Analogously, he allows using non-active formulas which do not actually appear in the premises.[5]

[5]Using a generalized rule for introducing a variable has the unpleasant consequence that a term can be assigned different types, although they only differ for unessential type declarations of variables which do not actually appear.

$$\frac{}{A^x, \Gamma \vdash x : A \mid \Delta}$$

$$\frac{\Pi, A^x \vdash u : B \mid \Sigma}{\Pi \vdash \lambda x.u : A \to B \mid \Sigma} \qquad \frac{\Gamma \vdash t : A \to B, \Delta \qquad \Gamma' \vdash u : A, \Delta'}{\Gamma, \Gamma' \vdash (t)u : B \mid \Delta, \Delta'}$$

(a) Logical rules

$$\frac{\Pi \vdash t : A \mid \Sigma}{\Pi \vdash [\alpha]t : A^\alpha \mid \Sigma} \qquad \frac{\Gamma \vdash e : A^\alpha \mid \Delta}{\Gamma \vdash \mu\alpha.e : A \mid \Delta}$$

(b) Structural rules

Figure 1.1: Logical and structural rules of $\lambda\mu$.

Operational semantics

The expression $\mu\alpha[\beta]u$ should be interpreted as "take the sub-term u, which is in context α, and evaluate it in context β instead". This is made precise by the reduction rules of the calculus:

$$(\lambda x.u)v \to u[v/x] \tag{β}$$

$$(\mu\beta.u)v \to \mu\beta.u[[\beta](w)v/[\beta]w] \tag{μ}$$

$$[\alpha]\mu\beta.u \to u[\alpha/\beta] \tag{ρ}$$

There is no need to comment on the first rule (the usual logical reduction). The second rule, called *structural reduction*, says that the argument v is sent to any receiving ends of channel β, i.e., wherever we find a sub-term $[\beta]w$ we replace it with $[\beta](w)v$. The third rule, i.e., renaming, means that whenever we send to channel β whatever we receive from channel α we can instead simply rename the endpoints of channel β to endpoints of channel α and delete the intermediate channel β (which does nothing more than "forwarding").

Definition 1.1 *The structural substitution, denoted $u[t/^*\delta]$, is defined inductively*[6] *as follows:*

[6] Cf. also Remark 4.

(i) $x[t/^*\delta] = x$;

(ii) $(\lambda x.u)[t/^*\delta] = \lambda x.u[t/^*\delta]$;

(iii) $((u)v)[t/^*\delta] = (u[t/^*\delta])v[t/^*\delta]$;

(iv) $(\mu\alpha.u)[t/^*\delta] = \mu\alpha.u[t/^*\delta]$;

(v) $([\delta]u)[t/^*\delta] = [\delta](u[t/^*\delta])t$;

(vi) $(\mu\alpha.[\beta]u)[t/^*\delta] = \mu\alpha.[\beta]u[t/^*\delta]$, *if* $\beta \neq \delta$.

Using the previous definition, structural reduction is formally defined as:

$$(\mu\alpha.u)v \to_\mu \mu\alpha.u[v/^*\alpha].$$

Remark 1 *Note that when reducing μ-redexes the binder does* not *disappear, contrarily to β-redexes. This means that:*

- *in an* untyped setting*, the μ binder acts like a λ with* unbounded *arity;*
- *in a* typed setting*, the type of the μ-variable* changes *after reduction.*[7]

Remark 2 *Parigot also mentions (cf. [Par92] and [Par93]) an additional reduction rule (later named θ by De Groote [dG98]) which is similar to η reduction in lambda calculus:*

$$\mu\alpha[\alpha]u \to u, \alpha \notin u \qquad (\theta)$$

Parigot considers the possibility of further enriching the set of rules, but warns it can result in the loss of confluence or even subject reduction.

[7] E.g., $(\mu\beta^{A\Rightarrow B}.[\beta]\lambda x^A.u^B)v^A \to_\mu \mu\beta^B.[\beta](\lambda x^A.u^B)v^A$.

Further topics

Failure of Uniqueness Property One possible "defect" of the $\lambda\mu$-calculus of Parigot is the non-uniqueness of results. If a term represents some data type, this is clearly undesirable (e.g., we would expect that the only terms representing natural numbers are Church integers). Nonetheless, this is not a serious issue since there is an *output operator* (a typed $\lambda\mu$-term) which allows to extract the intuitionistic result among classical ones. Output operators are connected to Krivine's *memorization operators* [Kri90] (also known as *storage operators* [Kri94]). Parigot studies this question more deeply in [Par93].

Failure of Böhm Separation Theorem Böhm separation Theorem [Böh68] essentially states that two different $\beta\eta$-normal forms r, s cannot be observationally equivalent, i.e., there is at least one context $C[\,]$ s.t. $C[r]$ and $C[s]$ do not have the same normal form. This result does not extend to the $\lambda\mu$ calculus of Parigot [DP01], but there are some variants of $\lambda\mu$ for which it holds (cf. [Sau08]).

1.3.3 De Groote's Calculus

An alternative classical extension of λ-calculus has been provided by de Groote [dG94a] (cf. also [dG94b] and [dG98]).

Term grammar

The calculus of de Groote conservatively extends the calculus of Parigot since the μ and $[\cdot]$ operators no longer need to be paired:

$$u, v := x \quad | \quad \lambda x.u \quad | \quad \mu\alpha.u \quad | \quad [\alpha]u \quad | \quad (u)v. \tag{1.9}$$

Taken separately, the operators can be interpreted this way: $\mu\alpha$ is used to *save* the current context to the stack[8] α (i.e., *send* data along channel α), while $[\alpha]$ is used to *restore* the context α (i.e., *receive* the data from channel α).

[8]Here we informally talk about stacks because of how the calculus is implemented on abstract machines. Cf. Chapter 5.

Remark 3 *We can also see this as a mechanism for* exception handling. $(\mu\alpha.t)u_1 \dots u_n$ *can be interpreted as "if trying to evaluate* t *you run into a subterm* $[\alpha]w$*, continue with* $[\alpha](w)u_1 \dots u_n$ *instead". Thus, the role of* $[\alpha]$ *is to "raise the exception", which is then "caught" by the continuation bound by* $\mu\alpha$ *(see Section 1.3.3 and [dG95]).*

As first observed by Saurin [Sau05], in the untyped case the calculus of de Groote is strictly more expressive than the calculus of Parigot; in the typed case (where continuations are usually given the type $\bot$), on the other hand, it is equivalent (modulo a simple encoding) to Parigot's calculus enriched with a single continuation constant [HG08]. Thus de Groote's calculus is often denoted $\Lambda\mu$ (cf. also [HS09] and Remark 11), a notation that we adopt henceforth.

De Groote's Typed $\Lambda\mu$

The set of types is defined as follows:

$$\sigma, \tau := \bot \mid X \mid \sigma \to \tau \tag{1.10}$$

where X is an atomic type and $\bot$ stands for absurdity. The type system is defined by an intuitionistic sequent calculus: sequents are of the form $\Gamma \vdash t : \tau$ where Γ is a set containing variable declarations of the form $(x_i : \tau_i)$ or $(\alpha_j : \tau_j^\bot)$, where $\tau^\bot$ is a *cotype* whenever τ is a type. The typing rules can be found in Figure 1.2. The type system uses implicit weakening (resp. contraction) in rule (ID) (resp. (APP)). Also note that (first and second order) quantifiers rules are missing from the figure. De Groote does not explicitly consider them since they are clearly, *mutatis mutandis*, the same of the calculus of Parigot.[9] Last but not least, observe that $\mu\alpha.M$ is well-typed only if M has type $\bot$, for example (but not necessarily) if M has the form $[\beta]N$ for some β, N.

[9] They are not "essential" (they simply allow more terms to be typed) and they pose no particular technical difficulty.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,(\text{ID})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\,(\text{ABS}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M)N : \tau}\,(\text{APP})$$

(a) Logical rules

$$\frac{\Gamma, \alpha : \tau^{\perp} \vdash M : \tau}{\Gamma, \alpha : \tau^{\perp} \vdash [\alpha]M : \perp}\,(\text{NAME}) \qquad \frac{\Gamma, \alpha : \tau^{\perp} \vdash M : \perp}{\Gamma \vdash \mu\alpha.M : \tau}\,(\text{MUABS})$$

(b) Naming rules

Figure 1.2: Typing rules of $\Lambda\mu$.

Extended grammar

To give a formal and uniform treatment of the various notion of reductions, de Groote [dG98] temporanely extends the grammar of the terms using the notion of *name*. Let $\lambda^\circ f$ be a linear binder, i.e., if u contains exactly one occurrence of f then $\lambda^\circ f.u$ is a well-formed term. Then the resulting grammar is the following

$$u, v := x \quad | \quad \lambda x.u \quad | \quad \mu\alpha.u \quad | \quad [\alpha]u \quad | \quad (u)v \quad | \quad [N]u \tag{1.11a}$$

$$N := \lambda^\circ f.u. \tag{1.11b}$$

where essentially $[N]u$ is interpreted as $(N)u$. Let β° be the following reduction rule:

$$[\lambda f^\circ.M]N \to M[f := N]. \tag{β°}$$

Terms of the extended calculus are (strongly) normalizable and β° does not break confluence. Since β°-normal terms are just usual $\Lambda\mu$-terms, we can simply consider terms up to α and β° conversion. In other words, the extended grammar does not really define a new calculus.

Remark 4 *The above extension of the grammar had already been envisaged by Parigot in [Par93, §2.1].*

Operational semantics

Thanks to the extended grammar, de Groote can easily express μ reduction as

$$(\mu\alpha.M)N \to \mu\beta.M[\alpha := \lambda^\circ f.[\beta](f)N]. \tag{1.12}$$

Remark 5 *Of course expressing μ reduction in this fashion introduces β° conversions, although they are computationally trivial they still needed to be taken into account. This formulation of μ reduction is simpler to describe, but it has the downside of requiring extra linear β reductions (which can be seen as a sort of* reification*).*

Besides rules β, μ, ρ (and θ), in [dG98] de Groote also proposes an additional rule (ϵ) whose logical counterpart is the elimination of absurd weakening. In literature the calculus obtained with the addition of ϵ is sometimes called $\lambda\mu\epsilon$. He justifies auxiliary reductions first in the typed calculus (with proof-theoretical motivations) and then in the untyped calculus.

Remark 6 *In particular, since some rules (namely ϵ) use types in their definition, he defines the untyped variants of the various rules by imposing that the evaluation context has a certain form. Thus in a typed setting the untyped formulation of the rules is more restrictive, since types allow to disregard the form of the context.*

De Groote defines reduction rules for untyped terms because he does not want types to play an explicit *role in the dynamic part of the calculus. This is perfectly understandable for various reasons, both theoretical (types are not needed for β reduction) and practial (an implementation which uses type would be significantly more complex).*

Auxiliary reduction rules are described in Figure 1.3, where M_α denotes that each subterm $[\alpha]N$ of M has been replaced by N (i.e., $M_\alpha = M[\alpha := \lambda^\circ f.f]$). Some explanations are in order. First, note that the rules for the $\lambda\mu\epsilon$ calculus are expressed in term of the extended grammar, which is more operationally oriented.[10] Second,

[10] Also note that η-like rules like θ are not suitable to be expressed in that way.

$$[\beta](\mu\alpha.M) \;\to_\rho\; M[\beta/\alpha] \qquad \mu\alpha.[\alpha]M \;\to_\theta\; M, \alpha \notin M$$

(no ϵ rule)

(a) $\Lambda\mu$

$$[\alpha](\mu\beta.\mu\gamma.M) \;\to_\rho\; \mu\gamma.M[\beta := \lambda^\circ f.[\alpha]f]$$
$$\mu\alpha.\mu\beta.M \;\to_\epsilon\; \mu\alpha.M[\beta := \lambda^\circ f.f]$$
$$\mu\alpha.[\alpha]M \;\to_\theta\; M, \alpha \notin M$$

(b) $\lambda\mu\epsilon$

Figure 1.3: Auxiliary reductions ρ, ϵ, θ.

the elimination of absurd weakening rule of $\lambda\mu\epsilon$ is only defined for term of the form $\mu\alpha.\mu\beta.M$ because the previous term can be typed only if $\mu\beta.M$ has type $\bot$, i.e., only if β has cotype $\bot^\bot$. Third, the renaming rule needs to be phrased as shown rather than, say, $[\alpha](\mu\beta.M) \to M[\beta := \lambda^\circ f.[\alpha]f]$ (which *looks* perfectly reasonable[11]), since otherwise the Church-Rosser property is lost, as shown by de Groote [dG98, §2.4]:

$$[\alpha](\mu\beta.\mu\gamma.M) \to_\rho \mu\gamma.M[\beta := \lambda^\circ f.[\alpha]f]$$
$$[\alpha](\mu\beta.\mu\gamma.M) \to_\epsilon [\alpha](\mu\beta.M[\gamma := \lambda^\circ f.f]).$$

Figure 1.4: Loss of confluence caused by unrestricted ρ.

Remark 7 *The ϵ rule had not been previously considered by Parigot, not even informally. In his calculus it would not make sense at all.*

In the typed case the rule entails replacing named terms $[\alpha]N$, which are not proper terms but only appear in terms of the form $\mu\beta.[\alpha]N$, with N.

- *Thus $\mu\alpha.[\alpha]\mu\beta.[\alpha]N$ would be rewritten into $\mu\beta.N$, which is not a legal term (since clearly N may not have the form $[\gamma]O$). This settle the case in which the name occurs multiple times.*
- *On the other hand the case in which the name has zero occurrence turns a term of the form $\mu\alpha.M$ into M, which is not a proper term.*

[11] It is simply the rule ρ of $\Lambda\mu$ expressed in the extended syntax.

- *The only exception, i.e., the only meaningful case, would be* $\mu\alpha^{\perp}.[\alpha]N^{\perp} \to N^{\perp}$, *where* $\alpha \notin N$, *which is but a particular* θ *reduction.*

In the untyped case, analogously, it never applies since $\mu\alpha.\mu\beta.M$ *is not a well-formed term.*

Separation The ϵ-free version of de Groote's calculus, i.e., the version presented in [dG94b], does satisfy Böhm's separation theorem. However, it is unknown whether $\lambda\mu\epsilon$ [dG98] satisfies separation or not. Cf. [Sau08].

1.3.4 Other Calculi

$\lambda\mu^{\to\wedge\vee\perp}$

A further extension of Parigot's $\lambda\mu$, called $\lambda\mu^{\to\wedge\vee\perp}$ is described by de Groote in [dG01a], for which he proves confluence, subject reduction and strong normalization.

$$\begin{aligned} t, u, v := {} & x \mid (u)v \mid \lambda x.t \mid [\alpha]t \mid \mu\alpha.t \mid \imath_1 u \mid \imath_2 v \mid \delta(t, x.u, y.v) \\ & \mid \langle u, v\rangle \mid \pi_1 t \mid \pi_2 t. \end{aligned} \tag{1.13}$$

This calculus also provides pairs and injections constructors and destructors, which correspond to introduction and elimination rules for conjunction and disjunction. Correspondingly, the set of types is also enlarged to include the *unit* type $\mathbf{1}$, product and coproduct of types:

$$\sigma, \tau := \mathbf{1} \mid \perp \mid X \mid \sigma \to \tau \mid \sigma \vee \tau \mid \sigma \wedge \tau. \tag{1.14}$$

Remark 8 *While the grammar and the types are a strict superset of the corresponding ones for the "traditional"* $\Lambda\mu$ *of de Groote, things change when it comes to operational semantics: the auxiliary reduction rules* ρ, θ *and* ϵ *are not discussed.*

The reduction rules are divided into three groups: *detour*-reduction rules (for the intuitionistic fragment of natural deduction), δ reductions (for the *choice* operator δ) and μ-reduction rules (where the notion of structural substitution is modified to fit the new grammar).

Definition 1.2 *The structural substitution, denoted $u[w/^*\alpha]$ is defined as in Definition 1.1 and, in the remaining cases, as follows:*

(vii) $\langle u, v\rangle[w/^*\alpha] = \langle u[w/^*\alpha], v[w/^*\alpha]\rangle$;

(viii) $(\pi_i t)[w/^*\alpha] = \pi_i t[w/^*\alpha]$;

(ix) $(\imath_i t)[w/^*\alpha] = \imath_i t[w/^*\alpha]$;

(x) $\delta(t, x.u, y.v)[w/^*\alpha] = \delta(t[w/^*\alpha], x.u[w/^*\alpha], y.v[w/^*\alpha])$.

Since in the rest of this thesis we do not discuss disjunction operators, for the sake of brevity we omit the operational semantics of $\lambda\mu^{\rightarrow\wedge\vee\bot}$.

Ong-Stewart's $\lambda\mu_{\mathsf{v}}$ Calculus

Another variant of Parigot's $\lambda\mu$ has been described in [OS97]. This is a (typed-only) calculus using call-by-value, just like the calculus of Felleisen $\lambda_{\mathcal{C}}$. It is based on a previous call-by-name calculus $\lambda\mu_{\mathsf{n}}$ [Ong96]. It enjoys Church Rosser and subject reduction and strong normalization.

Term grammar The term syntax of $\lambda\mu_{\mathsf{v}}$ is the same of the calculus of de Groote, with the proviso that $\bot$-typed names are forbidden.

$$u, t := x \mid \lambda x^A.u \mid (u)t \mid [\alpha^A]u \mid \mu\alpha^A.u. \tag{1.15}$$

Conseguently, there is no need of a ϵ rule.

Operational semantics Contrarily to the approach of de Groote, rewriting rules make use of types. Besides standard substitution and renaming for μ variable there is no structural substitution. In its place the *mixed substitution* is employed, which is a sort of *recursive* structural substitution:[12]

$$t[\alpha^A, C^A/\beta^B] := t[[\alpha^A]C[u]/[\beta^B]u]. \qquad (\mathsf{mix_v})$$

Let K range over (call-by-value) *singular contexts*

$$K[\,] := [\,] \mid (K[\,])t \mid (v)K[\,] \qquad (1.16)$$

where v is a value. The rewriting rules are defined as follows:

β_v-reduction:

$$(\lambda x^A.t)u \to t[u/x] \qquad (\beta_v)$$

μ-reduction:

$$[\alpha^A](\mu\gamma^A.t) \to t[\alpha/\gamma] \qquad (\mu\text{-}\beta)$$

$$\mu\alpha^A.[\alpha^A]t \to t, \quad \alpha \notin t \qquad (\mu\text{-}\eta)$$

ζ-reduction:

$$K^B[\mu\alpha^A.t] \to \begin{cases} \mu\beta^B.t[\beta, K/\alpha] & \text{if } B \neq \bot \\ t[\bot, K/\alpha] & \text{otherwise} \end{cases} \qquad (\zeta)$$

$\bot$-reduction:

$$v^{\bot \Rightarrow B} t^\bot \to \begin{cases} \mu\beta^B.t^\bot & \text{if } B \neq \bot \\ t^\bot & \text{otherwise} \end{cases}. \qquad (\bot)$$

Remark 9 *The lack of $\bot$-typed names forces to have rules (namely, ζ- and $\bot$-reduction rules) that behaves differently depeding on whether the type involved is $\bot$ or not, something which de Groote strived to avoid. This depends on mixed reduction and in fact the same thing happens in $\lambda\mu_{\mathsf{n}}$.*

[12] Using mixed substitution we could rephrase de Groote's μ-reduction in the following way: $C[\mu\alpha.t[\alpha^A, C^A/\beta^B]] = \mu\beta.t[\alpha := \lambda^\circ f.C[f]]$.

Ong and Stewart show that we can distinguish two cases for the ζ rules[13], depending on whether the μ-abstraction is in function or argument position.

$$(\mu\alpha^{A\Rightarrow B}.u)t \to \begin{cases} \mu\beta^{B}.u[\beta,([\,])t/\alpha] & \text{if } B \neq \bot \\ e[\bot,([\,])t/\alpha^{A\Rightarrow\bot}] & \text{otherwise} \end{cases} \qquad (\zeta_{\mathsf{fun}})$$

$$(v^{A\Rightarrow B})\mu\alpha^{A}.t \to \begin{cases} \mu\beta^{B}.t[\beta,(v)[\,]/\alpha] & \text{if } B \neq \bot \\ t[\bot,(v)[\,]/\alpha^{A}] & \text{otherwise.} \end{cases} \qquad (\zeta_{\mathsf{arg}})$$

Remark 10 *ζ_{fun} is essentially usual μ-reduction. On the other hand, ζ_{arg} is what Parigot [Par92, §3.2] figures should be its symmetrical counterpart.*

In a call-by-name setting this rule can be used to obtain a stronger notion of normal form which, in particular, avoids the problem of "false" data representation of Parigot's calculus. As already briefly mentioned (cf. pag. 16), Parigot envisages the possiblity of such a rule, but does not include it for the sake of confluence.

However, Stewart and Ong [OS97, §2] claim that uniqueness of data representation and confluence can be reconciled provided the rule is only used when the functional part is a variable or an abstraction and it is employed together with an appropriate evaluation strategy.

A quick look at $\lambda\mu_{\mathsf{n}}$ Mixed substitution generalizes a notion already appeared in [Ong96].

$$t[\beta^{B},x^{A}/\alpha^{A\Rightarrow B}] := t[[\alpha^{A\Rightarrow B}]u/[\beta^{B}](u)x], \quad B \neq \bot. \qquad (\mathsf{mix}_{\mathsf{n}})$$

Here the context $C^{A}[\,]$ has the particular shape $([\,])x^{A}$. This substitution is used to "unroll" a λ from a μ (a possibility *de facto* already suggested by Parigot), as illustrated by the following rule.

$$\mu\alpha^{A\Rightarrow B}.t = \lambda x^{A}.\mu\beta^{B}.t[\beta^{B},x^{A}/\alpha^{A\Rightarrow B}]. \qquad (\zeta)$$

[13] Aside for the usual $\bot$ issue, so there are actually four cases to consider.

The side condition in Equation ($\mathsf{mix_n}$) is due to the absence of $\bot$-typed names in the syntax. Thus an additional rule is necessary for the $\bot$ case

$$\mu\alpha^{A\Rightarrow\bot}.t = \lambda x^A.t[\bot, x^A/\alpha^{A\Rightarrow\bot}] \qquad (\zeta^\bot)$$

where $t[\bot, x^A/\alpha^{A\Rightarrow\bot}] = t[(u)x/[\alpha^{A\Rightarrow\bot}]u]$.[14] However, the $\zeta^\bot$ rule can also be applied to terms of the form $\mu\alpha^{A\Rightarrow\bot}[\alpha]t$, where $\alpha \notin t$. Since these terms can be reduced by μ-η-reduction (i.e., θ-reduction) to t, and by $\zeta^\bot$ reduction to $\lambda x^\bot.(t)x$, it evidently becomes necessary to include also η-reduction in order to handle this kind of critical pairs.

$$\lambda x^A.(t)x = t, \quad x \notin t. \qquad (\eta)$$

In [Ong96] Ong also briefly mentions that the calculus can be extended with product types and a ζ rule he calls $\zeta^\times$.

It is not evident [HS09, §2.3] whether the formulation proposed by Ong has definitive advantages with respect to Parigot's original work. It also looks somewhat contrived, in its effort to avoid $\bot$-typed names.

$\mu\mathsf{PCF_v}$ From a computational perspective, the main contribution of [OS97] is the definition of an extension of PCF [Plo77] with control features. Put otherwise, $\lambda\mu_\mathsf{v}$ can be enriched with basic arithmetic, conditionals and fix-points.

Saurin's $\Lambda\mu$-Calculus

Saurin's calculus [Sau05] derives from a previous calculus of David and Py [DP01], which is designed as a conservative extension of Parigot to include η-reduction.

A quick look at David and Py's $\lambda\mu\eta$-Calculus η reduction is essential in order to formulate Böhm's Theorem. However, the resulting calculus $\lambda\mu\theta\rho + \eta$ is

[14] This rule is obviously a variant of the former which actually hides a ϵ-like rule.

not confluent:[15]

$$\mu\alpha.t_\eta \leftarrow \lambda x.\mu\alpha.(t)x \rightarrow_\mu \lambda x.\mu\alpha.t[x/^*\alpha]. \tag{1.17}$$

Since it is unclear what would be an equivalent of Böhm's theorem in a non-confluent calculus, if any, confluence is restored adding one extra rule called ν-reduction, which is an η-expansion followed by a μ-reduction.

$$\mu\alpha.t \rightarrow \lambda x.\mu\alpha.t[x/^*\alpha]. \tag{ν}$$

Ultimately, separation still fails and confluence only holds for μ-closed terms [Sau08, §3.2].

Term grammar Back to $\Lambda\mu$, the syntax is basically the more liberal one of de Groote, but naming is denoted $(t)\alpha$ instead (of course, this causes no ambiguity with usual application).

$$t, u := x \mid \lambda x.t \mid \mu\alpha.t \mid (t)\alpha \mid (t)u. \tag{1.18}$$

Saurin's Typed $\Lambda\mu$ Saurin typing rules make a clean separation between free λ and μ variables. Names are put in a separate context.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \mid \Delta}\,(\text{Var}) \qquad \frac{\Gamma, x : \sigma \vdash t : \tau \mid \Delta}{\Gamma \vdash \lambda x.t : \sigma \to \tau \mid \Delta}\,(\lambda\text{Abs})$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \mid \Delta \qquad \Gamma \vdash u : \sigma \mid \Delta}{\Gamma \vdash (t)u : \tau \mid \Delta}\,(\lambda\text{App})$$

$$\frac{\Gamma \vdash t : \bot \mid \Delta, \alpha : \tau}{\Gamma \vdash \mu\alpha.t : \tau \mid \Delta}\,(\mu\text{Abs}) \qquad \frac{\Gamma \vdash t : \tau \mid \Delta, \alpha : \tau}{\Gamma \vdash (t)\alpha : \bot \mid \Delta, \alpha : \tau}\,(\mu\text{App})$$

Figure 1.5: Typing rules of $\Lambda\mu$.

[15] In [Sau08, §2.2] Saurin argues that it may be the very reason Parigot did not include η-reduction in the first place.

Operational semantics Saurin takes the $\beta\mu\theta\rho$ rules, adds fst (which is basically ν) and removes μ.

$$(\lambda x.t)u \to_\beta t[u/x] \tag{1.19}$$

$$\lambda x.(t)x \to_\eta t, \quad x \notin t \tag{1.20}$$

$$(\mu\alpha.u)\beta \to_{\beta_s} u[\beta/\alpha] \tag{1.21}$$

$$\mu\alpha.(t)\alpha \to_{\eta_s} t, \quad \alpha \notin t \tag{1.22}$$

$$\mu\alpha.t \to_{fst} \lambda x.\mu\alpha.t[(u)x\alpha/(u)\alpha], \quad x \notin t. \tag{1.23}$$

W.r.t. other $\lambda\mu$-calculi, except Ong's $\lambda\mu_{\mathsf{n}}$, the distinctive characteristic of the calculus of Saurin is precisely that he does *not* consider μ-reduction, which becomes unnecessary since it can be simulated by β and fst. Expressing μ reduction by means of fst allows to see $\Lambda\mu$ as a "calculus of streams", building on the observation of Parigot that μ looks like an unbounded version of λ (which we already mentioned, cf. Remark 1) and because of the fact that fst is applied only when there are no $\beta\eta\beta_s\eta_s$-redexes and fst creates $\beta\eta\beta_s\eta_s$-redexes.

Separation $\Lambda\mu$ satisfies Böhm theorem, a fact that Saurin relates to two features of the calculus: the rich syntax and the possibility to freely η/η_s expand. Note, however, that confluence only holds for μ-closed terms.

Remark 11 *Saurin's $\Lambda\mu$ coincides, up to extensional rules, with de Groote's $\Lambda\mu$-calculus: in [HG08, §1] Herbelin and Ghilezan actually suggest to call it de Groote-Saurin calculus. Its more expressive syntax makes it significantly richer that Parigot's calculus, rich enough to satisfy Böhm's theorem, which justifies putting a capital lambda in its name (as anticipated in Section 1.3.3). Cf. also [HS09, §2.6].*

1.3.5 Conclusion

Which Classical Logic?

In the calculus of Parigot it is not possible to have terms of the form $[\alpha]t$, hence there is no need of ϵ-reduction for the elimination of absurdity. However, this has

important consequences: the $\lambda\mu$ term of type $\neg\neg A \to A$ that Parigot considers to have a behavior close to Felleisen's $\mathcal{C}$ operator has one free μ-variable, as pointed out by several people (cf., e.g., [dG98, §2.6], [OS97, §2], [HS09, §2.1]). This is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \neg\neg A, x : A \vdash x : A}}{y : \neg\neg A, \alpha : A^{\perp}, x : A \vdash [\alpha]x :}}{y : \neg\neg A, \alpha : A^{\perp}, x : A \vdash \mu\beta.[\alpha]x : \perp}}{y : \neg\neg A, \alpha : A^{\perp} \vdash \lambda x.\mu\beta.[\alpha]x : \neg A} \qquad y : \neg\neg A, \alpha : A^{\perp} \vdash y : \neg\neg A}{y : \neg\neg A, \alpha : A^{\perp} \vdash (y)\lambda x.\mu\beta.[\alpha]x : \perp}}{y : \neg\neg A, \alpha : A^{\perp} \vdash [\gamma](y)\lambda x.\mu\beta.[\alpha]x :}}{y : \neg\neg A, \alpha : A^{\perp} \vdash \mu\alpha.[\gamma](y)\lambda x.\mu\beta.[\alpha]x : A}}{\alpha : A^{\perp} \vdash \lambda y.\mu\alpha.[\gamma](y)\lambda x.\mu\beta.[\alpha]x : y : \neg\neg A \to A}$$

Figure 1.6: Felleisen's $\mathcal{C}$ in the calculus of Parigot.

unsatisfactory from a proof-theoretical point of view and is a serious impediment for an implementation which uses de Brujin indexes for variable names. On the

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \neg\neg A, x : A \vdash x : A}}{y : \neg\neg A, \alpha : A^{\perp}, x : A \vdash [\alpha]x : \perp}}{y : \neg\neg A, \alpha : A^{\perp} \vdash \lambda x.[\alpha]x : \neg A} \qquad y : \neg\neg A, \alpha : A^{\perp} \vdash y : \neg\neg A}{y : \neg\neg A, \alpha : A^{\perp} \vdash (y)\lambda x.[\alpha]x : \perp}}{y : \neg\neg A, \alpha : A^{\perp} \vdash \mu\alpha.(y)\lambda x.[\alpha]x : A}}{\alpha : A^{\perp} \vdash \lambda y.\mu\alpha.(y)\lambda x.[\alpha]x : y : \neg\neg A \to A}$$

Figure 1.7: Felleisen's $\mathcal{C}$ in the calculus of de Groote.

contrary, in the calculus of de Groote the operator $\mathcal{C}$ can be represented with a properly closed term.

Since the continuation of $\perp$ type is used by Parigot only to adjust the type, it makes sense to consider it a sort of constant of the calculus. This is the idea of $\lambda\mu\hat{\mathsf{tp}}$ [AH03] [HG08], which extends the calculus of Parigot with a constant $\hat{\mathsf{tp}}$, a single dinamically bound continuation variable. The calculus can be used either with a call-by-value or with a call-by-name formulation. The reduction system of $\Lambda\mu$ and

$$
\begin{array}{rll}
M, N := V \mid (M)M \mid \mu q.c & & \text{(terms)} \\
V := x \mid \lambda x.M & & \text{(values)} \\
c := [q]M & & \text{(commands)} \\
q := \alpha \mid \hat{\mathsf{tp}} & & \text{(ev. context variables)}
\end{array}
$$

Figure 1.8: Syntax of $\lambda\mu\hat{\mathsf{tp}}$.

$$
\begin{array}{ll}
\beta_v : & (\lambda x.M)V \to M[V/x] \\
\mu_{app} : & (\mu\alpha.c)M \to \mu\beta.c[[\beta](N)M/[\alpha]N], \quad \beta \text{ fresh} \\
\mu'_{app} : & (V)\mu\alpha.c \to \mu\beta.c[[\beta](V)N/[\alpha]N], \quad \beta \text{ fresh} \\
\mu_{var} : & [q]\mu\alpha.c \to c[q/\alpha] \\
\eta_{\hat{\mathsf{tp}}} : & \mu\hat{\mathsf{tp}}.[\hat{\mathsf{tp}}]V \to V, \quad \text{even if } \hat{\mathsf{tp}} \text{ occurs in } V
\end{array}
$$

Figure 1.9: Rewriting rules of call-by-value $\lambda\mu\hat{\mathsf{tp}}$.

$$
\begin{array}{ll}
\beta : & (\lambda x.M)N \to M[N/x] \\
\mu_{app} : & (\mu\alpha.c)M \to \mu\beta.c[[\beta](N)M/[\alpha]N], \quad \beta \text{ fresh} \\
\mu^n_{var} : & [\beta]\mu\alpha.c \to c[\beta/\alpha] \\
\mu^n_{\hat{\mathsf{tp}}} : & \mu\hat{\mathsf{tp}}.[\hat{\mathsf{tp}}]M \to M, \quad \text{even if } \hat{\mathsf{tp}} \text{ occurs in } M
\end{array}
$$

Figure 1.10: Rewriting rules of call-by-name $\lambda\mu\hat{\mathsf{tp}}$.

$$\neg\neg A \to A \quad \begin{array}{c}\Rightarrow\\ \not\Leftarrow\end{array} \quad ((A \to B) \to A) \to A \quad \begin{array}{c}\Rightarrow\\ \not\Leftarrow\end{array} \quad A \vee \neg A.$$

Figure 1.11: Relative expressive power of classical principles within minimal classical logic.

call-by-name $\lambda\mu\hat{\mathsf{tp}}$ are bisimilar (cf. [HG08, §4.3]) and, although its syntax does not look as expressive, Böhm's separation theorem holds also for the latter.

$\lambda\mu\hat{\mathsf{tp}}$ is a calculus born from λ-$\mathcal{C}^-$-top, a theory of control which extends Felleisen's λ-$\mathcal{C}$ with a top level continuation (and uses $\mathcal{C}$ notation in place of μ notation). In [App07] Ariola and Herbelin show that a formula A is provable in classical logic iff there is closed $\lambda\mu\hat{\mathsf{tp}}$ term of type A, the analogous of Griffin's result for λ-$\mathcal{C}$. Furthermore they prove that there is a subset of classical logic they call *minimal classical logic* (which has no rules for $\bot$) s.t. a formula A is provable in minimal classical logic iff there is a closed term in Parigot's calculus of type A. Since minimal logic is strictly less expressive than classical logic (within it the ex falso quodlibet principle cannot be proved) then Parigot's $\lambda\mu$ is also less expressive. Another way to see it is that $\mathcal{C}$ (of type $\neg\neg A \to A$) can express `callcc` (of type $((A \to B) \to A) \to A$), but not viceversa. However, it is possible to express $\mathcal{C}$ from `callcc` and the abort operator $\mathcal{A}$.

Which Evaluation Order?

Many (if not most) of the calculi for control we have presented are call-by-name, or at least have also a call-by-name variant. On the contrary, Felleisen's studies have started with an untyped call-by-value calculus, since its aim was to study programs written in Scheme, which uses CBV.

Some languages also permit to have mixed evaluations orders (and are thus non-confluent) like the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [CH00] (which can be embedded into π-calculus [CCS09]), but non-confluent languages are not considered here. What is important is that there is no theoretical reason to favour CBN or CBV.

Indeed, Selinger [Sel01] considers call-by-name and call-by-value variants of Parigot's $\lambda\mu$ enriched with product and disjunction types and shows they are equivalent (in the sense that there are syntactical translations between them which are sound w.r.t. the operational semantics). For example, Ong and Stewart mention that $\lambda\mu$ works equally well with both CBN and CBV: from $\lambda\mu_{\mathsf{v}}$ they define a $\mu\mathsf{PCF}_{\mathsf{v}}$ calculus, which uses call-by-value, but they remark that a $\mu\mathsf{PCF}_{\mathsf{n}}$ from Ong's $\lambda\mu_{\mathsf{n}}$ would have also been possible).

Final Remarks

Although the calculi derived from the syntax of de Groote generally have nicer properties than those following the original syntax for $\lambda\mu$, I must concede that many of the ideas emerged later were already present, albeit in a rough, prototipal and un-refined form, in the original paper of Parigot [Par92]. Furthermore, in the early papers of de Groote [dG94a] [dG94b] he keeps referring to the calculus as Parigot's $\lambda\mu$, although the syntax he uses is more general. Some time later, Ong and Stewart [OS97] pointed out that it was de Groote who first studied $\lambda\mu$ from a computational viewpoint, while Parigot simply shows two terms whose behaviour is "close" to Felleisen's $\mathcal{C}$ operator[16] and Scheme `callcc`. Eventually [dG98, §2.6] de Groote fully realizes the two calculi are indeed different.

The work of Ariola, Herbelin and Saurin has been of particular relevance in the proper understanding of the properties of the various calculi and the relation between them. For further readings the author would recommend [AH03] [HG08] [Sau08] [HS09].

[16] Again, they also point out that Parigot uses a term which is not closed.

Chapter 2

Implicit Computational Complexity

2.1 Introduction

Traditionally, complexity of programs is studied considering a specific computational model (e.g., Turing machines) and a notion of cost (e.g., number of steps/transitions). However, studying properties like termination, time or space complexity, is in general a very hard task. If we are (also) interested in more sophisticated aspects like amount of bandwidth consumed, maximum bandwidth required, energy consumed, maximum power required, ... the situation becomes even more complex.

A different approach consists in analysing the *abstract* complexity of programs. As an example, one can take the number of instructions executed by the program as a measure of its execution time. This is of course a less informative metric, which however becomes more accurate if the actual time taken *by each instruction* is kept low. One advantage of this analysis is the independence from the specific hardware platform executing the program at hand: the latter only needs to be analysed once. A variety of *complexity analysis* techniques have been employed in this context, from abstract interpretation [Gul09] to type systems [JHLH10] to program logics [dBdBZ80] to interactive theorem proving. Properties of programs written in higher-order functional languages are for various reasons well-suited to be verified by way of type systems. This includes not only safety properties (e.g. well-typed programs do not go wrong), but more complex ones, including resource

bounds [JHLH10, BT09, GRDR07, DLG12].

Purpose of ICC Implicit Computational Complexity (ICC) [Cob65] [BC92] [GSS92] [Jon97] [Gir98] (see also [Hof00] for a relatively recent survey) aims at providing formal methods for asserting computational properties of programs. More precisely, it aims at defining machine-free characterization of complexity classes based on Mathematical Logic. It has provided:

1. some machine-independent characterizations of complexity classes of functions,
2. some criteria for verifying statically that a program admits a certain complexity bound.

What do we mean for *machine-independent*? Let us consider the notion of *reasonable* machine models [vEB90]:

> "Reasonable machines can simulate each other within a polynomially-bounded overhead in time and a constant-factor overhead in space."

Since (up to a polynomial time factor and a constant space factor) reasonable machines are computationally equivalent, it stands to reason that one can define a class of functions of a bounded complexity without specifying a computational model.

Formal Systems developed in ICC are mainly for polynomial time, but there are also some for other complexity classes, both above and below **P** (**L**[Nee04], **PSPACE**[LM95], ...).

Characterizing a complexity class is not enough. We want to obtain *practical* programming languages, i.e., languages natural and rich enough to be used.

The ICC problem Determining whether a given program P written in any Turing-complete language (C, Caml, ...) runs in time polynomial in the size of its input is an undecidable problem (Σ_2-complete, even harder than the halting problem) [MM$^+$06]. This is true not just for **PTIME** but also **L**, **PSPACE**, ... any classical complexity class.

The workaround used by ICC systems is that they can only recognize *some* programs, i.e., they give only sufficient conditions. The more **PTIME** programs an ICC system recognizes, the more complex it is.

Definition 2.1 (Soundness) *A logical system $\mathcal{G}$ is sound for $\mathcal{C}$ if for every program $p \in \mathcal{I}$, p reduces or calculates in resources bounded by a function in $\mathcal{C}$.*

Definition 2.2 (Extensional Completeness) *A logical system $\mathcal{G}$ is extensionally complete for $\mathcal{C}$ if for every function that can be computed in resources bounded by $\mathcal{C}$ there is (at least) an element of $\mathcal{I}$ that computes that function.*

A logical system $\mathcal{I}$ is an ICC system characterizing a complexity class $\mathcal{C}$ if it is both sound and extensionally complete for $\mathcal{C}$. E.g. suppose the class $\mathcal{C}$ is the complexity class $\mathbf{P}$. If $\mathcal{I}$ is sound for $\mathbf{P}$, then all the programs written in $\mathcal{I}$ reduce/compute in polynomial time.

All the ICC systems have the properties of soundness and extensionally completeness. Some systems are more intensionally expressive than other, though. Two different ICC systems for the same class $\mathcal{C}$ may have different degrees of expressiveness, i.e., one may be able to capture more programs within $\mathcal{C}$ w.r.t. the other (i.e., it is more *intensionally* expressive than the other). We introduce the notion of intensional completeness:

Definition 2.3 (Intensional Completeness) *$\mathcal{I}$ is $\mathcal{C}$-intensionally complete if $\mathcal{I}$ contains all programs that runs in resources bounded by a function in $\mathcal{C}$.*

Soundness and Completeness are the fundamental properties of each ICC system. They express that all the programs generated by the systems need to have a complexity bound that are characteristic of the analyzed class; moreover, the system should be expressive to generate enough programs and being able to capture all the functions in the class.

Implicit Computational Complexity uses different approaches and is strictly connected to different fields of computational logic. We give a brief introduction in the following section.

2.2 Main branches of ICC

ICC is a research area in which a variety of tools and techniques are employed. Let us briefly review the various branches of ICC (w.r.t. to the tools used).[1]

Restricted recursion

This research field of ICC started with Cobham's pioneering work [Cob65] in 1965. Define $f(x, \vec{y})$ from $g(\vec{y})$, $h(x, \vec{y}, z)$ and $k(x, \vec{y})$ as

$$f(x, \vec{y}) = \begin{cases} g(\vec{y}) & x = 0 \\ h(x, \vec{y}, f(\lfloor x/2 \rfloor, \vec{y}) & x > 0 \end{cases}, \qquad f(x, \vec{y}) \leq k(x, \vec{y}). \tag{2.1}$$

The above recursion scheme is called *bounded recursion on notation* and characterizes programs running in a polynomial number of steps.

Cobham's characterization of polynomial time requires that a bound is checked to determine whether a program is polynomial. There is a similar limitation in the approach based on finite model theory (interpret everything in a finite domain $\{0, 1, ..., N\}$ for some fixed N), cf. [Fag73]. On the contrary, Bellantoni-Cook [BC92] and Leivant-Marion [LM93] [LM95] gave more intrinsic characterization of polynomial time. The common idea is to prevent complexity explosion by preventing "bad" compositions of recursion.

Quasi-interpretations

We consider first-order functional programs written as systems of terms with rewriting rules $l \to r$. Quasi-interpretation [JYM00] is about bounding the size of values and restricting recursion, just as in Cobham's system, to obtain time bounds. Better intensionality than primitive recursion may be achieved.

To each term t is associated the size of its normal form $[\![t]\!]$. A program admits a quasi-interpretation if each rule $l \to r$ verifies $[\![l]\!] \geq [\![r]\!]$. Quasi-interpretation is not

[1] Here we do not discuss the approach which uses finite model theory, which consists in interpreting everything in a finite model (so, for example, the successor function has a fixed upper bound). The interested reader can check, e.g., [Goe92].

sufficient for termination, however interpretation plus termination give a complexity bound. Quasi-interpretation generalizes interpretation, a similar technique in which each rule strictly decreases the size of the term.

Although typically first-order, an extension of quasi-interpretation to higher-order has been designed by Dal Lago and Baillot [BDL12] using a simple termination criterion based on linear types and path-like orders.

Linear Logic and Typed λ-Calculus

Types ensure some qualitative properties such as termination, but they can also be used to express *quantitative* properties. The basic idea is to use variants of Linear Logic with a restricted exponential modality ! (and dually also ?), which are called *light logics*, as type systems for λ-calculus (exponentials mark those formulas that may be duplicated during normalization, thus are responsible for complexity explosions). Different restrictions in general correspond to different complexity classes; or at least different formulations, some more intensionally expressive than others. An introduction to Linear Logic can be found here .

2.3 Linear Logic

2.3.1 Introduction

Intuitionistic Logic [Pra65] is a restriction of Classical Logic obtained by limiting the use of structural rules. It is obtained imposing that in the sequent calculus there can be (at most) one formula on the right side of each sequent, thus contraction is forbidden on the right.

Linear Logic (LL) [Gir87a] [GLT89, §A], on the contrary, does not impose restriction on the shape of the sequents but rather adds new modalities ! (*of course* or *bang*) and ? (*why not*) to control structural rules. Intuitionistic implication $A \Rightarrow B$ is thus expressed as $(!A) \multimap B$ or $?(A^{\perp}) \parr B$ (where $\parr$ is the multiplicative disjunc-

tion, see Figure 2.2, and $\multimap$ is the *linear implication*[2] and $(\cdot)^\perp$ denotes the involutive negation).

More precisely, sequents are usually considered one-sided because splitting the formulas on the left and the right of the turnstile symbol $\vdash$ only increases the number of rules to consider. This simplification can be adopted w.l.o.g.:

Theorem 2.1 *A two-sided sequent $\Gamma \vdash \Delta$ is provable if and only if the sequent $\vdash \Gamma^\perp, \Delta$ is provable in the one-sided system.*

On the other hand, when Linear Logic is used as a type system for lambda calculus, sequents are split with exactly one rule on the right. The idea is that the formula on the right corresponds to the type of the term obtained with the derivation in the sequent calculus, provided that its free variables have the types determined by the formulas on the left. This presentation of Linear Logic is called *Intuitionistic Linear Logic* (ILL).

2.3.2 Sequent calculus

As we can see from Figure 2.4, structural rules can only be applied to modal formulas. In particular non-modal formulas can only be used *linearly*, i.e., exactly once, since

$$\frac{}{\vdash A, A^\perp}\,\mathsf{Ax} \qquad\qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,\mathsf{Cut}$$

Figure 2.1: Axiom and Cut rules of LL.

weakening and contraction are not allowed.

$$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\,\parr \qquad\qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\,\bot$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\Gamma, A \otimes B, \Delta}\,\otimes \qquad\qquad \frac{}{\vdash 1}\,1$$

Figure 2.2: Multiplicative rules of LL.

[2]For simplicity notation $\multimap$ is often associated to the right.

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B} \oplus_2 \qquad \text{(no rule for } \mathsf{0}\text{)}$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\Gamma, A \& B} \& \qquad \frac{}{\vdash \Gamma, \top} \top$$

Figure 2.3: Additive rules of LL.

One of the consequences of the introduction of exponential rules is that additive and multiplicative rules are not equivalent (i.e., contrarily to Classical Logic, it is not possible to derive the first ones from the second ones or vice versa). In particular, there are two formulas for "true" and "false".

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?A} ?w \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} ?c \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} ?d$$

(a) Weakening and contraction rules. (b) Promotion and dereliction rules.

Figure 2.4: Exponential rules of LL.

Remark 12 *The intuitive meaning of* $!A$ *is that* A *can be used any number of times, i.e.,* $!A \equiv A \otimes !A$ *(or less formally,* $!A \equiv \otimes_{k=0}^{\infty} A$*).*

Exponential rules are called this way because of the isomorphism below:

$$!(A \& B) \equiv !A \otimes !B. \tag{2.2}$$

The intuitive meaning of Equation (2.2) is that having any number of either A or B is equivalent to having any number of A and any number of B. The equation is very similar to the following

$$e^{a+b} = e^a \cdot e^b$$

in which the exponential function transforms a sum into a product and vice versa. Hence "exponential" rules.

Fragments of Linear Logic

Linear Logic has several interesting subsystems, here is a short list.

MLL Figure 2.1 and 2.2.

MALL Figure 2.1, 2.2 and 2.3.

MELL Figure 2.1, 2.2 and 2.4.

MELL$^-$ Figure 2.1, 2.2 (except the rules for the constants $\mathbf{1}$ and $\bot$) and 2.4.

A subsystem of LL can be either *affine* or linear depending on whether the weakening rule ($?w$) is included or not. Sometimes the term linear is used to mean *affine* linear (i.e., formulas may be erased but not duplicated).

2.3.3 Full Linear Logic

Finally we show the rules for quantifiers, since second-order quantifiers are needed for polymorphism, which in turn is needed for obtaining the extensional completeness of various fragments of LL. Note that the variable x (resp. X) in the first (resp. second) order $\forall$ rule is intended not to appear free in the context Γ of the rule. To

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, \forall x.A}\,\forall \qquad\qquad \frac{\vdash \Gamma, A[a]}{\vdash \Gamma, \exists x.A}\,\exists$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, \forall X.A}\,\forall \qquad\qquad \frac{\vdash \Gamma, A[\alpha]}{\vdash \Gamma, \exists X.A}\,\exists$$

Figure 2.5: First and Second order quantifiers rules.

explicit the presence of quantifiers in a sub-system of LL one may say, e.g., *second order* MELL.

2.3.4 Proof-boxes

A significant novelty of Linear Logic is the introduction of *proof-nets* [Gir87b] [Gir96] [Gir96, §3] to represent proofs. A box, which corresponds to a promotion rule, has one conclusion $!A$, which is called the *main door* of the box, and a certain number of conclusions $?\Gamma$, which are called *auxiliary doors*.

Definition 2.4 (Box) *A* box *B is a proof structure whose conclusions are ?-formulas but one, its* principal door*, which is the conclusion of an !-link. The ?-conclusions of B are its* auxiliary doors*.*

A proof in sequent calculus is a sequential object, but proof-nets are not. In particular, contrarily to Natural Deduction [GLT89, §2] (in which proofs are represented by trees) there is no last rule. In other words, proof-nets offer a way to look at proofs disregarding the unessential order of rules, whenever they can commute. An important feature of proof-nets is the use of *proof-boxes* (or simply boxes) to represent those part of the proofs that can be duplicated or erased during cut-elimination. When a promotion rule is used the whole subproof is put into a box (in particular, two boxes are either *nested* or disjoint).

Computational meaning

Computationally boxes represent thunks to be evaluated a certain number of times.

- When reducing a cut between a promotion rule and a contraction the box is *duplicated*;
- when the cut is between a promotion rule and a weakening rule, the box is *erased*;
- finally when the cut is between a promotion and a dereliction rule the box is *opened*, i.e., it can be evaluated.

Until then the inside and the outside of the box cannot interact.

One case we have not mentioned is when a promotion rule interacts with a why not formula introduced by an axiom. This is not a particularly crucial nor meaningful case, in fact we can assume w.l.o.g. that we work with proofs whose axioms introduce only formulas without modalities. In particular, a cut between a promotion and an axiom is basically the same thing of a cut between a promotion and a dereliction.

$$\frac{}{\vdash ?^n A^\perp, !^n A}\,\mathsf{Ax} \qquad\qquad \frac{\dfrac{\dfrac{}{\vdash A^\perp, A}\,\mathsf{Ax}}{\vdash ?^n A^\perp, A}\,?d}{\vdash ?^n A^\perp, !^n A}\,!$$

Figure 2.6: Two sequent calculus proofs of $\vdash ?^n A^\perp, !^n A$, where $n \in \mathbb{N}$.

Handling boxes

If we do not want normalization to take too much time we have to be careful at how we handle boxes. Boxes may contain any number of nodes but, until they are opened, they are treated as a single object. The boxes are handled modularly, the only characterizing elements are their terminal ports.

Parallelism This *black-box* principle is quite reasonable, but it has its flaws. Boxes impose synchronization, and cut-reductions involving boxes are not local, thus boxes limit parallelism. An implementation of proof-nets which overcomes these limitations, using sharing graphs for optimal reduction [AG98], can be found in [GAL92].

Strong normalization Boxes are an important aspect to consider when reasoning on complexity, but they are problematic already when studying *strong* normalization. The use of proof-nets does not completely eliminate the need of commutations of rules. Some administrative steps are still needed, which complicates the proof of strong normalization (and might also be bad for the complexity of reduction).

A simplified proof by Accattoli has recently appeared [Acc13b]. The paper builds on ideas from explicit substitutions [DCK97] and basically exploits the clever observation that once the commutative steps are performed, the box still has to interact through its main door with the node which introduces the matching why not formula. So if we "peek" into the box, i.e., if we violate the black-box principle and look deep enough into boxes to find the matching node, we can define reduction and commutation in one step. Thus one can consider instead a reduction rule which either moves one box inside the other (axiom and dereliction case) and reduces the cut; or erases the box (weakening case) and adds weakening nodes to replace the

auxiliary doors of the deleted box; or duplicates the box (contraction case) and puts contractions on auxiliary doors of the duplicated boxes.

2.3.5 Light logics

The whole simply typed λ-calculus, in which normalization is known to be not elementary [Sta79], can be encoded in Linear Logic, thus cut-elimination of full linear logic is also not elementary. To obtain proofs on which cut-elimination can be performed in reasonable time, we must consider a strict subset of Linear Logic. Light logics are obtained by restricting or modifying exponential rules. Multiplicative rules are needed to express proofs corresponding to the simplest programs (roughly speaking $\parr$ and $\otimes$ correspond respectively to abstraction and application), so they are not touched.[3] Additive rules, on the other hand, are not an essential part of cut-elimination (but they are needed for *non-deterministic* computations [MT03]).

By tuning the rules governing the exponential modality, then, one can define logical systems for which cut-elimination can be performed within appropriate resource bounds. Usually, this is coupled with an encoding of all functions in a complexity class $\mathcal{C}$ into the system at hand, which makes the system a *characterization* of $\mathcal{C}$.

Two possible ways to impose complexity bounds are:

- forbidding some rules governing the exponential modality "?" (this is the case of LLL, Light Linear Logic [Gir95a]);
- replacing existing exponential modalities with restricted variants (as in BLL, Bounded Linear Logic [GSS92]).

We define two properties called *stratification* and *boundedness* which characterize light logics and are closely related to the kind of restriction imposed.

Remark 13 *Weakening is not essential for extensional completeness (it corresponds to the possibility to write programs which can discard one or more arguments), but it*

[3] The rules for multiplicative *constants*, however, may or may not be included. The constants-free fragment of multiplicative linear logic is often denoted MLL^- to distinguish it from MLL.

does help to have systems which are simpler (i.e., more natural) to use. Essentially, adding weakening to a subset of logic is computationally harmless so, unless one wants a system with a minimal set of rules, there is no reason not to include it.

Stratification Removing the the linear logical principles called *dereliction* ($?d$) and *digging* ($??$), the depth of a node (i.e., the number of nested boxes in which it is contained) does not change throughout the cut-elimination procedure. In this way only node at the same depth can interact, thus limiting the complexity of cut-elimination.

More in general, if a proof can be divided in *strata* or *levels* s.t. two nodes belonging to different strata never interact along cut-elimination, then we say the proof is *stratified*. A proper account of stratification is outside the scope of this thesis, the interested reader is referred to [BM10] and [BMdF12]. When imposing

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

Figure 2.7: Digging.

such restriction, the promotion rule is modified in such a way to add a modality ? to each formula in the context. The modified promotion is called t-promotion or

$$\frac{\Gamma, A}{?\Gamma, !A}\ \mathsf{f!}$$

Figure 2.8: Functorial promotion.

functorial promotion. Correspondingly, the boxes are called functorial boxes. LLL and ELL, plus their affine version LAL and EAL, are two fragments of linear logic with these characteristics. ELL (and EAL) characterizes the class of elementary functions, those that can be computed by a Turing machine in a number of steps bounded by a tower of exponentials of fixed height. On the other hand, LLL (and LAL) proof-nets can be normalized in polynomial time, but for technical reasons they require the introduction of a further exponential modality called *paragraph* (§).[4]

[4]The context of functorial promotion consists of at most one formula and thus without some

Boundedness The idea of boundedness is to impose some kind of bound on the number of copies of a box that can be created during cut-elimination, thus denying the principle $!A \otimes A \equiv !A$ of Linear Logic. Girard's Bounded Linear Logic is the most significant light logic which has this property.

2.4 Bounded Linear Logic

Bounded Linear Logic (BLL) [GSS92] [HS04] [DLH09] is a subset of Linear Logic obtained restricting the exponential modalities: $!_x A$ denotes, roughly speaking, that A may be used up to x times (we may think of $!_x A$ as $1 \otimes \underbrace{A \otimes \ldots \otimes A}_{x \text{ times}}$). To distinguish the various copies of A we use the notation $!_{y<x} A$ for $1 \otimes A\{y/0\} \otimes \ldots \otimes A\{y/x-1\}$.

2.4.1 Introduction

Definition 2.5 (Resource polynomial) *A* resource monomial *is any (finite) product of binomial coefficients,* $\prod_{i=1}^{p} \binom{x_i}{n_i}$*, where* x_i *are distinct variables and* n_i *are non-negative integers. A* resource polynomial *is any finite sum of resource monomials.*

Given resource polynomials p, q *write* $p \sqsubseteq q$ *to denote that* $q - p$ *is a resource polynomial. If* $p \sqsubseteq p'$ *and* $q \sqsubseteq q'$ *then also* $q \circ p \sqsubseteq q' \circ p'$*.*

Remark 14 *For simplicity we can think of resource labels as* constants *rather than* polynomials*. Proofs which only use constant polynomials indeed correspond, as already noted by Asperti, to Lafont's Soft Linear Logic (*SLL*) [Laf04], in which contraction is replaced by the following rule, where* n *is any natural number and* $A^{(n)}$ *denotes the sequence of* n *copies of* A*. Using the more general polynomials makes* BLL *much more intensionally expressive. See also Section 2.5.3.*

adjustment duplication would be completely forbidden: hence § has to be added to compensate for this restriction and obtain a logical system strong enough to be complete for **PTIME**.

$$\frac{\Gamma, A^{(n)} \vdash C}{\Gamma, !A \vdash C}$$

Figure 2.9: Multiplexing.

$!_{x<p}A$ can be seen as a form of *bounded storage*, i.e., it represents a bounded amount of function calls to memory. Indeed, in BLL the promotion rule of sequent calculus is named *storage*.

Remark 15 *Nevertheless, as pointed out by Girard himself in [Gir95b, §1.4.2], the price paid (keeping track of explicit polynomial bounds) was too high.*

Definition 2.6 (Formula) *Formulae (= types): atomic formulae have the form $\alpha(\vec{p})$, where α is a second-order variable of given finite positive arity and $\vec{p}$ here denotes an appropriate non-empty list of resource polynomials.*

Formulae are closed under the following operations:

1. *$A \otimes B$, $A \multimap B$;*
2. *$(\forall\alpha)A$ (second order universal quantification);*
3. *$!_{x<p}A$ (bounded exclamation mark with p a resource polynomial not containing x).*

Occurrences of resource terms in formulae are either *positive* or *negative*. The resource polynomials $\vec{p}$ in $\alpha(p)$ all occur positively; p occurs negatively in $!_{x<p}A$; the connectives $\otimes$ and $\forall$ are monotone, while $\multimap$ is monotone in the first argument and antitone in the second one.

Definition 2.7 (Subtyping) *Let A, A' be types of* BLL*. $A \sqsubseteq A'$ iff A and A' only differ in their resource polynomials and if p is a positive (resp. negative) occurrence of a resource polynomial in A then the omologous p' is s.t. $p \sqsubseteq p'$ (resp. $p' \sqsubseteq p$).*

Sequents have the form $\Gamma \vdash B$, where Γ is a finite (possibly empty) multiset of formulae. The formulae in Γ are considered indexed but not ordered. $A\{\alpha := B\}$ is the result of substituting a second order abstraction term B for all free occurrences of the propositional variable α in A.

The rules of BLL are the following:

Axiom

$$\frac{A \sqsubseteq B}{B \vdash A}\ \mathsf{Ax}$$

Cut

$$\frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\ \mathsf{Cut}$$

Logical

$$\otimes L \quad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad \otimes R \quad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

$$\multimap L \quad \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \qquad \multimap R \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

$$\forall L \quad \frac{\Gamma, A\{\alpha := C\} \vdash B}{\Gamma, (\forall\alpha)A \vdash B} \qquad \forall R \quad \frac{\Gamma \vdash A}{\Gamma \vdash (\forall\alpha)A}$$

(provided α is not free in Γ)

Exponential

$$!W \quad \frac{\Gamma \vdash B}{\Gamma, !_{x<r}A \vdash B}$$

$$!D \quad \frac{\Gamma, A\{x/0\} \vdash B}{\Gamma, !_{x<1+r}A \vdash B}$$

$$!C \quad \frac{\Gamma, !_{x<p}A, !_{y<q}A\{x/p+y\} \vdash B}{\Gamma, !_{x<p+q+r}A \vdash B}$$

where $p + y$ is free for x in A.

$$!S \quad \frac{!_{z<q_1(x)}A_1\{y/v_1(x)+z\}, \ldots, !_{z<q_n(x)}A_n\{y/v_n(x)+z\} \vdash B}{!_{y<v_1(p)+w_1}A_1, \ldots, !_{y<v_n(p)+w_n}A_n \vdash !_{x<p}B}$$

where $v_i(x) + z$ is free for y in A_i and $v_i(x) = \sum_{z<x} q_i(z)$.

2.4.2 Normalization

Some results of BLL are better phrased in the context of proof-nets (which, are anyway a more elegant formalism). Soundness, for an instance, also holds for sequent calculus[GSS92, Appendix A] but the bound is worse because of the presence of commutative steps.

Data types Usual data types like (tally) integers and (dyadic) lists (as well as trees, etc...) can be represented by proofs of BLL.

$$\mathbf{N}_x \equiv \forall\alpha !_{y<x}(\alpha(y) \multimap \alpha(y+1)) \multimap (\alpha(0) \multimap \alpha(x))$$
$$\mathbf{N}^2_x \equiv \forall\alpha !_{y<x}(\alpha(y) \multimap \alpha(y+1)) \multimap !_{y<x}(\alpha(y) \multimap \alpha(y+1)) \multimap \alpha(0) \multimap \alpha(x).$$

In the above definitions, x represents the maximum size of elements.

Definition 2.8 *A function ϕ from dyadic lists to dyadic lists is* represented *in bounded linear logic by a proof F of $\mathbf{N}^2_x \multimap !_{y<1}\mathbf{N}^2_{p(x)}$ if for every dyadic list b of length $\leq n$ and the corresponding cut-free proof $\mathbf{b}$ of $\mathbf{N}^2_{p(n)}$, the irreducible proof net with conclusion $!_{y<1}\mathbf{N}^2_{p(n)}$ that corresponds to the dyadic list $\phi(b)$ is the irreducible form of the proof net representation of the BLL proof displayed in Fig. 2.10.*

A function ϕ from dyadic lists to dyadic lists is representable in bounded linear logic if there exists a resource polynomial $p(x)$ and a BLL proof of $\vdash \mathbf{N}^2_x \multimap \mathbf{N}^2_{p(x)}$ that represents ϕ.

Similarly for integers, trees, ... For example, there is a proof representing zero and a function representing the successor function.

$$\dfrac{\dfrac{\dfrac{\mathbf{b}}{\vdots}}{\vdash \mathbf{N}^2_n} \qquad \dfrac{\dfrac{F(x := n)}{\vdots}}{\vdash \mathbf{N}^2_n \multimap \mathbf{N}^2_{p(n)}} \qquad \dfrac{\mathbf{N}^2_n \vdash \mathbf{N}^2_n \qquad \mathbf{N}^2_{p(n)} \vdash \mathbf{N}^2_{p(n)}}{\mathbf{N}^2_n \multimap \mathbf{N}^2_{p(n)}, \mathbf{N}^2_n \vdash \mathbf{N}^2_{p(n)}} \multimap\mathrm{L}}{\mathbf{N}^2_n \vdash \mathbf{N}^2_{p(n)}}\,\mathsf{Cut}}{\mathbf{N}^2_{p(n)}}\,\mathsf{Cut} \qquad \dfrac{\dfrac{\text{Reuse, with } x := p(n)}{\vdots}}{\vdash \mathbf{N}^2_{p(n)} \multimap !_{y<1}\mathbf{N}^2_{p(n)}} \qquad \dfrac{\mathbf{N}^2_{p(n)} \vdash \mathbf{N}^2_{p(n)} \qquad !_{y<1}\mathbf{N}^2_{p(n)} \vdash !_{y<1}\mathbf{N}^2_{p(n)}}{\mathbf{N}^2_{p(n)} \multimap !_{y<1}\mathbf{N}^2_{p(n)}, \mathbf{N}^2_{p(n)} \vdash !_{y<1}\mathbf{N}^2_{p(n)}} \multimap\mathrm{L}}{\mathbf{N}^2_{p(n)} \vdash !_{y<1}\mathbf{N}^2_{p(n)}}\,\mathsf{Cut}$$

$$!_{y<1}\mathbf{N}^2_{p(n)}\,\mathsf{Cut}$$

Figure 2.10: F represents the function ϕ.[5]

[5] "Reuse" denotes a proof of conclusion $\vdash \mathbf{N}^2_{p(n)} \multimap !_{y<1}\mathbf{N}^2_{p(n)}$ (see [GSS92, Example 5.5] for details).

Soundness and completeness It is possible to associate a polynomial called *weight* $\|\pi\|$to a proof-net π in such a way that the weight strictly decreases at each reduction step.

The weight of a contraction link Contraction is 2. The weight of every other link, including the Axiom link, is 1. Finally, the weight of a box whose immediate sub-proof σ is $\sum_{x<p}(\|\sigma\|(x)+1)+2np+n+1$, where n is the number of auxiliary doors and p is the resource polynomial at the main door.

Definition 2.9 *In* BLL *proof nets, an instance of the cut link is irreducible if it is boxed or if one of its premises is a box with at least one auxiliary door, where the cut formula is at the main door, and the other premise is a conclusion of a Weakening, Dereliction, or Contraction link, or a box.*

Definition 2.10 *A* BLL *proof net is irreducible if it contains only irreducible cuts (if any).*

It is understood that the reduction steps do not apply to irreducible cuts.

Theorem 2.2 *Any function from dyadic lists to dyadic lists represented by a proof of* $\mathbf{N}_x^2 \multimap \mathbf{N}_{p(x)}^2$ *in* BLL *is computable in polynomial time. Furthermore, the required polynomial can be obtained explicitly from the weight of the representing* BLL *proof of* $\mathbf{N}_x^2 \multimap \mathbf{N}_{p(x)}^2$.

Theorem 2.3 *Every polynomial time computable function can be represented in* BLL *by a proof of* $\mathbf{N}_x^2 \multimap \mathbf{N}_{p(x)}^2$, *for some resource polynomial* p.

Expressive power BLL is very versatile. It can represents natural algorithms which would otherwise be rejected in Light Linear Logic. It is also possible to embed into it Lafont's SLL [Laf04] which in turn has inspired several type system for λ-calculus which characterize **PTIME** [GDR07], **PSPACE** [GMRDR12], ...

2.4.3 Related systems

Quantified Bounded Affine Logic A generalized version, named Quantified Bounded Affine Logic (QBAL) [DLH09] (obtained by adding bounded first order quantifiers), is at least as expressive as two other polynomial systems: Leivant's RRW [Lei93] and Hofmann's LFPL [Hof03] can both be embedded into it, although they are very heterogeneous (see also Section 2.5.3 for a brief description of dlPCF).

Stratified Bounded Affine Logic Restricting second order quantifiers yields a variant of BLL called Stratified Bounded Affine Logic (SBAL) [Sch07]. SBAL-proofs can be compiled into **L** functions, borrowing ideas from Game Semantics [AJ92].

Since SBAL functions are (already) very efficient in terms of space, optimization such as tail-call recursion cannot be implemented (since in general storing intermediate results requires an amount of space which is polynomial in the input size). Indeed, only a limited form of recursion called *skewed iteration* is available.

IntML Although SBAL cannot be properly considered a language for programs, it has led to the design of IntML [DLS10] [DLS], a functional languages which also characterizes **L**. IntML also borrows a few ideas from DLAL, since (bounded) quantification is only available to the left of the arrow. The key idea of IntmL is that a program can be represented by a fixed graph and its execution is the traversing of the graph with messages corresponding to inputs: this is achieved using two kind of types, for terms of the *upper class* and the *working class* (although the latter class would be enough, since term of the upper class are compiled to terms of the working class). Thanks to the *interactive* nature of computation, in this model composition of functions is still in **L**.

2.5 Further topics

2.5.1 Evaluation strategy

Within the setting of purely-functional languages, call-by-value is sometimes preferred to call-by-name since it is generally more efficient [DLP12c] (at least in terms of time) and it is closely related to the implicit complexity cost [DLM08b] [DLM09] (although under certain conditions, i.e., weak reduction and shared representation of terms, this is also true for call-by-name [DLM08a]).

Remark 16 *The* standard *normalization strategy (which evaluates the leftmost outermost redex) is always able to reach the normal form of a normalizing term. It may not be the shortest reduction strategy, since it may duplicate inner redexes, but how much longer it is with respect to the shortest reduction strategy? Recent work of Asperti and Lévy on standardization [AL13] shows that a previous bound obtained by Xi on the length of the standard reduction with respect to an arbitrary reduction can be improved from a double exponential to a factorial.*

2.5.2 Related fields

ICC has contribued to the understanding of traditional existing complexity class in the classical model of sequential computation. Nevertheless, it has helped to pave the way towards the definition of non-traditional complexity classes (e.g., probabilistic classes). We give a brief description of a few relevant cases of these alternative models of computations outlining a few significant contributions of ICC.

Quantum Computing One of the main interests in computational models based on quantum mechanics is the possibility to exploit parallelism to reduce the complexity of problems that are considered unfeasible in traditional models (such as the problem of integers factorization). Even though — from a practical/algorithmic point of view — the research is still in the early stages, a few interesting works have appeared in the last few years.

In [DLMZ10] a quantum calculus based on Lafont's Soft Linear Logic [Laf04] is presented. Roughly speaking, the intuition is to exploit the fact that in Linear Logic an arbitrary formula may not be duplicated, just like it is not possible (in general) to duplicate quantum bits.

Probabilistic Complexity Classes A closely related topic is that of probabilistic classes. Two interesting examples are the class of Probabilistic Polynomial Time (**PP**), the class of problems which can be solved by a probabilistic Turing machine in polynomial time; and the class of Bounded-Error Probabilistic Polynomial Time (**BPP**) in polynomial time with probability of error bounded by a constant *strictly* smaller than 0.5, (that is, it all problems whose solution can be found for "most" instances with a "small" error probability). This class is called **BPP** [Gil77] and it is nowadays considered a good model of "feasible computations" [DLPT12a] [MPT13].

These complexity classes are useful tools (in cryptography and other settings) for studying the security of protocols, particularly those based on problems computationally hard to solve[6] [BGZB09].

> In a typical provable security setting, one reasons about effective adversaries, modeled as arbitrary probabilistic polynomial-time Turing machines, and about their probability of thwarting a security objective, e.g. secrecy. In a similar fashion, security assumptions about cryptographic primitives bound the probability of polynomial algorithms to solve hard problems, e.g. computing discrete logarithms.

Process Calculi In a concurrent setting the overall system may run forever, in general (e.g. an operating system, the Internet, ...). Nevertheless it is reasonable to expect that single components terminate their execution. Furthermore, it is desirable to know that the interaction terminates within a certain time bound (and thus does not consume too much memory or other resource).

[6] Rather than those with perfect secrecy, e.g., one-time pad (OTP).

Types system can be used to ensure these kind of properties in Higher-Order π-calculus as in [DLMS10] (which uses types inspired by Light Linear Logic to characterize polytime interactions). This line of work has found application even for session types [DLDG11], which are types used to model those interactions which are generated by two initial parties, which can be identified with client and server.

Geometry of Interaction The program of Geometry of Interaction (GoI) was started by Girard in [Gir89]. GoI is an operational semantics of Linear Logic which was designed to capture the *dynamic* aspects of proof-theory, i.e., cut-elimination, and to get rid of the concept of *ad hoc* global time (towards a parallel implementation).

In [BP99] we can find a model of Geometry of Interaction to interpret proofs in Elementary Linear Logic [Gir95a], a variant of which characterizes elementary complexity.

2.5.3 The problem of intensional expressiveness

Although an ICC system characterizes a certain class, it might not do so in a satisfactory way. One of the typical concerns is whether the functions that can be computed within a certain resource bound can be represented by natural (and efficient) algorithms. Suppose we have a list of natural numbers and we want to sort it (say, in ascending order) and suppose an ICC system is at least **PTIME**-complete: what sorting algorithms are available? Can we find a program which implements `quicksort` or `mergesort`?

Conversely, if an ICC system is "too rich", it may become complicate to use. For example, the problem of deciding whether a term in polymorphic lambda calculus can be typed in a certain system is in general undecidable. Typability aside, the system might use an intricate type system (possibly it necessarily makes use of dependent types), so intensional expressivity might be sacrificed for the sake of practicality. This is the case of DLAL, a subsystem of LAL (see §2.2).

Another aspect to consider is that to characterize a certain complexity class sometimes it is necessary to adopt a particular syntax and/or a tricky notion of execution.

- Within LLL/LAL, the number of steps is not polynomial if one choses to work with sequent calculus, but only with *proof-nets*. In BLL, on the contrary, both ways to execute programs ensure a polynomial bound (although the bound is better working with proof-nets).
- Obviously standard computational models like lambda calculus or Turing Machines cannot be used to characterize peculiar complexity classes like **L** [DLS10] (resp. **NC** [BKMO08]).

Remark 17 *This is not overly surprising, since a similar phenomenon also occurs, e.g., for optimal reduction [AG98]. Optimal reduction can only be achieved with* parallel reductions*, but the usual syntax for lambda terms does not allow to reduce a family of redexes at once, so it is necessary to use a representation with explicit sharing [Lam89].*

dlPCF Linear dependent types can be used to overcome the limitedness of intensional expressiveness of usual ICC systems obtained by syntactical restriction. The idea is to work with semantics, formulating constraints in the form of inequalities, in a way which can be tailored w.r.t. a certain complexity class. dlPCF [DLG12] is a type system obtained decorating PCF types with *index terms* (denoted I, J, K) generated from variables, function symbols, ... in such a way two properties are ensured:

soundness if t is a program and $\vdash_K t : \mathtt{Nat}[I, J]$, then t evaluates to a natural number between I and J in a number of steps which is at most linear in K;

completeness if t is typable in PCF and evaluates to a natural number n in m steps, then $\vdash_I t : \mathtt{Nat}[n, n]$ where $I \leq m$.

$\texttt{Nat}[I, J]$ denotes the type of natural numbers between I and J. The type derivation of a term uses some assumption (i.e., the type derivation holds only if they are satisfied) which take the form of (in)equalities between index terms. Once we have the various constraints, we can solve them automatically with an SMT solver (not in every case, of course, since the underlying problem is *undecidable*). See also [DLP12a] and [DLP12c].

Part II

Polytime Quasi-Functional Languages

Chapter 3

Polarized Linear Logic

3.1 Introduction

In this chapter we explain the notion of *polarization* in logic starting from the earlier concept of *focalization*. Then we sketch a few connections with Game Semantics and π-calculus. Finally we describe Laurent's polarized version of Linear Logic and show how to translate $\lambda\mu$-calculus into it.

Focalization Logic programming [Mil04] is the idea that there is a correspondence between computation and the bottom-up search of a (cut-free) sequent proof. Such search is complicated by the the fact that for a given sequence there could be several proofs and thus several ways to *build* a proof. Which means that proof-searching has a computational over-head. This is a significant issue to consider also in the setting of (semi-)automated proof-search.

The search space of proofs can be restricting by considering only *focused* proofs. The subject has been first studied by Andreoli [And92] in the context of Linear Logic; it has been done for Intuitionistic Logic [LM07] and for Classical Logic [CMM10]; and more recently also in the context of Ludics [BST10]. Focalization is a concept which has been introduced essentially for cut-free proofs.

In this system there is a clear distinction between *asynchronous* connectives $\bot, ⅋, ?, \top, \&, \forall$ (whose right introduction rules are reversible) and *synchronous* con-

nective $\mathsf{1}, \otimes, !, \mathsf{0}, \oplus, \exists$ (whose right introduction rule is generally not reversible). Essentially, if a sequent has a synchronous principal formula, then that formula must be decomposed until either it is reduced to asynchronous formulas or to atomic formulas. We say that the *focus* is kept on that formula as long as possible. This causes a clean alternation between positive and negative connectives, so one may consider generalized positive (resp. negative) connectives as well. Thus a generalized positive connective can be seen as a *critical focusing section* of the proof.

Proposition 3.1 (Andreoli) *If a sequent is provable in Linear Logic, then it is provable in Linear Logic with a focusing proof.*

Formulas without connectives (i.e., atomic formulas) do not have a polarity (or are of *neutral* polarity). The choice of polarity is not relevant w.r.t. completeness, but it can be significant for computational complexity [LM09, §1.1].

Polarized Linear Logic What is *Polarized* Linear Logic (LLP)? It is a refinement of usual Linear Logic in which contraction can be applied to any negative formula, not just formulas of the shape $?A$. In this logic proofs are said to be *focalized*, in the sense that each sequent may contain at most one positive formula. Thus in particular the $\top$ rule (see Figure 2.3) may introduce at most one positive formula.

The intersting thing about LLP [Lau02] [Lau03b] is that it can be used to encode $\lambda\mu$-calculus: two terms which are *operationally equivalent* are translated into the same polarized proof-net, i.e., they are σ-equivalent (thus extending the usual σ-equivalence on λ-calculus [Reg92] [Reg94]). For simply-typed $\lambda\mu$-calculus it is enough to consider the polarized fragment of MELL, although some extensions are also possible.

Game Semantics PCF terms can be interpreted as innocent strategies in games between a player and an opponent. The opponent starts the game and alternates with the player (two consecutive moves have different polarities) until the sequence of moves played cannot be further extended, meaning computation is over. Here we omit technical details about games since in this thesis we do not reason in terms

of games. The important message is that the Hylang-Ong (HO) Games can be considered as an implementation of head linear reduction (each move in a game — except the initial move — points to a previous move), thus leading to the definition of the PAM (see Chapter6).

In the usual formulations of Game Semantics composition of strategies is not associative [A$^+$03], thus a Game Semantics and Linear Logic need to be reconciled somehow. The problem is related to the involutive nature of negation in Linear Logic, in which A is equivalent to $\neg\neg A$. Weakening this principle in $A \to \neg\neg A$ yields Tensorial Logic [MT10], which is a sort of synthesis between Game Semantics and Linear Logic.

3.2 Polarized Linear Logic

Extending structural rules to all negative formulas yields a proof system which is still confluent (as well as strongly-normalizing, in the typed case). LLP is categorically equivalent to Girard's Classical Logic (LC) [Gir91], so Laurent's work establishes a Curry-Howard correpondence between $\lambda\mu$-calculus and LC.

Proof-nets in this system are slightly modified. The resulting notion allows to establishes a correspondence with an asynchronous typed π-calculus [HL10] (a work which builds on [HYB04] to study control in the setting of π-calculus), which also shows that σ-equivalence on $\lambda\mu$-calculus corresponds to structural congruence.

Proposition 3.2 *Let $\vdash P : A$ and $\vdash Q : A$ be two typed π-terms,*

$$P \equiv Q \Longleftrightarrow \vdash P \simeq_\sigma Q : A.$$

3.2.1 Polarized Proof-Nets

Definition 3.1 (Polarized formula) *Let X range over atomic formulas,* output *(resp.* anti-output*) formulas are defined as follows.*

$$N := X \mid \bot \mid ?P \parr N, \tag{3.1}$$

$$P := X^\perp \mid 1 \mid !N \otimes P. \tag{3.2}$$

Formulas of the shape $?P$ *(resp.* $!N$*) are called* input *(resp.* anti-input*) formulas.* Negative formulas *(resp.* positive formulas*) are input and output (resp. anti-input and anti-output) formulas.*

Remark 18 *The terminology adopted by Laurent depends on the role of these types to encode terms. The type* $?P \parr N$ *can be rewritten as the (linear) arrow type* $(!P^{\perp}) \multimap N$*, where thus* $?P$ *is the input part and* N *is the output. The prefix* anti *is for the involutive negation.*

The polarized system can be extended to the full Linear Logic, but here we consider only MELL. One one hand we want to follow [Lau03b], where Laurent considers only the fragment needed to encode $\lambda\mu$, on the other in [Lau02, §4.3] we see that Polarized Linear Logic is syntactically isomorphic to some of its fragments. The main difference with the presentation followed in [Lau03b] is that we include multiplicative constants, since we want to talk about the richer calculus of de Groote (something which Laurent only does implicitly,[1] sticking essentially to Parigot's $\lambda\mu$).

Definition 3.2 (Proof-structure) *A* proof-structure *is a finite acyclic graph built from the nodes in Figure 3.1 (where* A *stands for a negative formula). Each node labelled by one of the symbols* $\mathsf{Ax}, \mathsf{Cut}, \parr, \otimes, \perp, \mathsf{1}, \forall, \exists, !, ?d, ?c, ?w$*, which determines the number of ingoing (resp. outgoing) edges, which are called the* premises *(resp.* conclusions*) of the node. Each edge is* conclusion *of exactly one node and* premise *of at most one node. Edges which are not premise of any node are the conclusions of the proof-structure. Additionally to each* $!$*-node of conclusions* $\{N, \mathcal{N}\}$ *is associated a* box *of conclusions* $\{N, \mathcal{N}\}$*.*

We say that an edge is positive (resp. negative) if the associated formula is positive (resp. negative).

Definition 3.3 (Correction graph) *Given a proof-structure, its* correction graph *is obtained by orienting upwardly (resp. downwardly) the positive (resp. negative) edges, i.e., by reversing the orientation of positive edges, and by erasing boxes (just keeping the* $!$*-node).*

[1] He mentions a "slight generalization" of $\lambda\mu$-calculus which allows to encode Felleisen's $\mathcal{C}$.

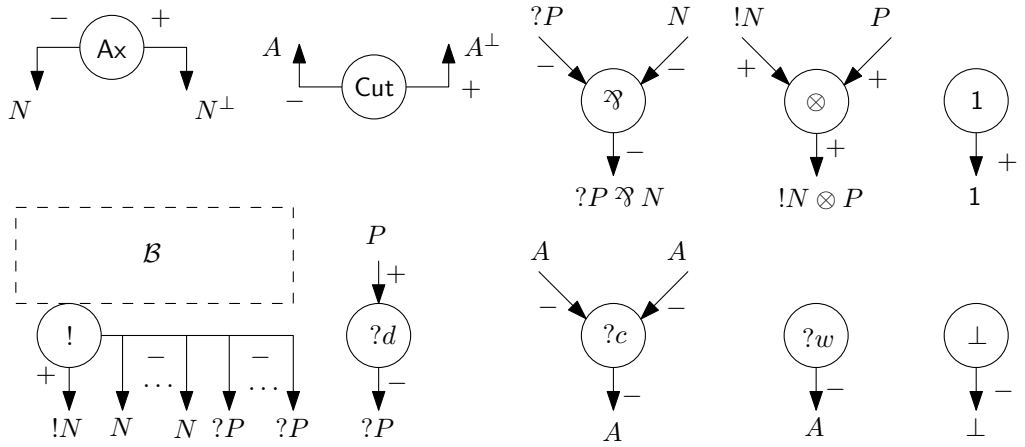

Figure 3.1: Nodes of polarized proof-structures.

Definition 3.4 (Proof-net) *A proof-structure is* correct *or is a* proof-net *if:*

- *its correction graph is an acyclic oriented graph;*
- *the number of positive conclusions plus ?d nodes is one;*
- *and recursively the boxes are also correct proof-structures.*

Remark 19 *Note that, contrarily to* MELL*, it is not necessary to consider the switching graphs of a proof-structure. Correctness can be verified in time linear w.r.t. the size of the proof-structure.*[2] *The correctness criterion of full* LLP *(i.e., in presence of quantifiers and additives) is a bit more complex, see [Lau02, §5].*

Definition 3.5 (⊗-tree) *The set of nodes above a node labeled with a positive type is called a ⊗-tree.*

A ⊗-tree is either an axiom, a box, a 1-node or a tensor formed by a box and a ⊗-tree. Cut-elimination of usual Linear Logic can duplicate (or erase) only *boxes*, while in LLP any ⊗*-tree* can be duplicated (or erased).

[2] At least for MELL$^-$.

The correctness of a proof-structure is preserved by cut-elimination, i.e., reducing a proof-net yields a proof-net. Polarized proof-nets can be translated to usual proof-nets by the following encoding:

$$\overline{X} = ?X$$
$$\overline{\bot} = ?\bot$$
$$\overline{?P \parr N} = ?(?\overline{P} \parr \overline{N})$$
$$\overline{?P} = ?\overline{P}.$$

The encoding is extended to positive formulas by duality.

Remark 20 *Laurent informally observes that this translation corresponds to CPS-translations: roughly speaking, adding $\neg\neg$ in suitable places is like adding ? in linear logic. De Groote shows in [dG94a, §5] that the translation of $\lambda\mu$ calculus (see next section) at the type level indeed resembles Kolmogorov's negative translation of classical logic into intuitionistic logic.*

Confluence and strong normalization of polarized proof-nets can be reduced to the same properties for standard proof-nets.

3.2.2 The $\lambda\mu$-calculus

As anticipated, here we consider de Groote's $\lambda\mu$ instead of Parigot's calculus (see Sections 1.3.2 and 1.3.3). Consider the type derivations in Figure 3.2. A proof of the sequent $\Gamma \vdash u : N \mid \Delta$ can be interpreted as the proof-net[3] with conclusions $?\Gamma^\diamond, N^\diamond, \Delta^\diamond$, where $(\cdot)^\diamond$ is the mapping below:

$$X^\diamond = X$$
$$\bot^\diamond = \bot$$
$$(N \to M)^\diamond = (!N^\diamond) \multimap M^\diamond = ?N^{\diamond\bot} \parr M^\diamond.$$

[3] See Chapter 4.

$$\frac{}{x : N \vdash x : N \mid} \,\mathsf{var}$$

$$\frac{\Gamma, x : N \vdash u : M \mid \Delta}{\Gamma \vdash \lambda x.u : N \to M \mid \Delta} \,\mathsf{abs}$$

$$\frac{\Gamma \vdash u : N \to M \mid \Delta \qquad \Gamma' \mid v : N \mid \Delta'}{\Gamma, \Gamma' \vdash (u)v : M \mid \Delta, \Delta'} \,\mathsf{app}$$

$$\frac{\Gamma \vdash u : N \mid \Delta}{\Gamma \vdash [\alpha]u : \bot \mid \alpha : N, \Delta} \,\mu\text{-}\mathsf{name}$$

$$\frac{\Gamma \vdash u : \bot \mid \alpha : N, \Delta}{\Gamma \vdash \mu\alpha.u : N \mid \Delta} \,\mu\text{-}\mathsf{abs}$$

Figure 3.2: Type system for $\lambda\mu$.

This translation is sound, i.e., we can simulate term reduction as proof-net normalization. This allows to extend sigma-equivalence to $\lambda\mu$-calculus. We can translate into proof-nets also untyped terms, we just need to apply to types the usual recursive equation $N =!N \multimap N$.

3.2.3 σ-equivalence

Translating λ-calculus into proof-nets identifies terms which are σ-equivalent [Reg92] [Reg94]. The same happens for $\Lambda\mu$-calculus, extending appropriately the concept of σ-equivalence [Lau02] [Lau03b]. While σ-equivalent λ-terms can be translated as the same proof-net, two σ-equivalent $\Lambda\mu$-terms are translated as the same *polarized* proof-net.

Definition 3.6 *The σ-equivalence is the smallest equivalence relations on $\Lambda\mu$-terms which is compatible (i.e., preserved by application, abstraction, μ-naming and μ-*

abstraction) and contains:

$$((\lambda x.u)v)w \equiv_\sigma (\lambda x.(u)w)v \qquad x \notin w, \tag{3.3}$$

$$(\lambda x.\lambda y.u)v \equiv_\sigma \lambda y.(\lambda x.u)v \qquad x \neq y, y \notin v, \tag{3.4}$$

$$(\lambda x.\mu\alpha.u)v \equiv_\sigma \mu\alpha.(\lambda x.u)v \qquad \alpha \notin v, \tag{3.5}$$

and

$$[\alpha'](\mu\alpha.[\beta'](\mu\beta.u)v)w \equiv_\sigma [\beta'](\mu\beta.[\alpha'](\mu\alpha.u)w)v \qquad \alpha \notin v, \beta \notin w, \tag{3.6}$$

$$[\alpha']\lambda x.\mu\alpha.[\beta']\lambda y.\mu\beta.u \equiv_\sigma [\beta']\lambda y.\mu\beta.[\alpha']\lambda x.\mu\alpha.u \qquad x \neq y, \tag{3.7}$$

$$[\alpha'](\mu\alpha.[\beta']\lambda x.\mu\beta.u)v \equiv_\sigma [\beta']\lambda x.\mu\beta[\alpha'](\mu\alpha.u)v \qquad x \notin v, \beta \notin v, \tag{3.8}$$

where in the last group of equations $\alpha \neq \beta, \alpha \neq \beta'$ *and* $\alpha' \neq \beta$.

3.3 Final remarks

LLP has all the desirable properties of Linear Logic (confluence, strong normalization, ...) and the polarization allows a simple correctness criterion and a generalization of the concept of boxes to $\otimes$-trees. Overall the system is elegant and could be adapted to encode richer calculi. A term typable in LLP is *strongly normalizable*, but the types do not give any information about how many steps long normalization could be.

Chapter 4

Bounded Polarized Linear Logic

An early version of this work has appeared in [DLP13c]. A large part of the material of this chapter is from [DLP13a], and a few more proofs and other details can be found in [DLP13b] (the proofs in Appendix A are unpublished).

4.1 Introduction

In this chapter, we delineate a methodology for complexity analysis of higher-order programs *with control operators*. The latter are constructs which are available in most concrete functional programming languages (including Scheme and OCaml), and allow control to flow in non-standard ways. The technique we introduce takes the form of a type system for de Groote's $\lambda\mu$-calculus [dG94b] derived from Girard, Scedrov and Scott's Bounded Linear Logic [GSS92] (BLL in the following). We prove it to be sound: typable programs can indeed be reduced in a number of steps lesser or equal to a (polynomial) bound which can be read from the underlying type derivation. A similar result can be given when the cost model is the one induced by an abstract machine. To the knowledge of Dal Lago and I, this is the first example of a complexity analysis methodology coping well not only with higher-order functions, but also with control operators.[1]

[1] This point has been confirmed also in all the reviews of [DLP13a] (as well as in other occasions).

In the rest of this section, we sketch the crucial role Linear Logic has in this work. See Section 2.2 and Section 2.3 for more details on LL.

4.1.1 Linear Logic and Complexity Analysis

Linear Logic [Gir87a] is one of the most successful tools for characterizing complexity classes in a higher-order setting, through the Curry-Howard correspondence. Subsystems of it can indeed be shown to correspond to the polynomial time computable functions [GSS92, Gir98, Laf04] or the logarithmic space computable functions [Sch07]. Many of the introduced fragments can then be turned into type systems for the λ-calculus [BT09, GRDR07], some of them being relatively complete in an intensional sense [DLG12].

Not much is known about whether this approach scales to languages in which not only functions but also first-class continuations and control operators are present. Understanding the impact of these features to the complexity of programs is an interesting research topic, which however has received little attention in the past.

4.1.2 Linear Logic and Control Operators

On the other hand, more than twenty years have passed since Classical Logic has been shown to be amenable to the Curry-Howard paradigm [Gri90]. And, interestingly enough, classical axioms (e.g. Peirce's law or the law of the Excluded Middle) can be seen as the type of control operators like Scheme's `callcc`. In the meantime, the various facets of this new form of proofs-as-programs correspondence have been investigated in detail, and many extensions of the λ-calculus for which Classical Logic naturally provides a type discipline have been introduced (e.g. [Par92, CH00]).

Moreover, the decomposition provided by Linear Logic is known to scale up to Classical Logic [Gir91]. Actually, Linear Logic was known to admit an involutive notion of negation from its very inception [Gir87a]. A satisfying embedding of Classical Logic into Linear Logic, however, requires restricting the latter by way

of polarities [Lau02]: this way one is left with a logical system with most of the desirable dynamical properties.

In this chapter, we define BLLP, a polarized version of Bounded Linear Logic. The kind of enrichment resource polynomials provide in BLL is shown to cope well with polarization. Following the close relationship between Polarized Linear Logic and the $\lambda\mu$-calculus [Lau03b], BLLP gives rise to a type system for the $\lambda\mu$-calculus. Proofs and typable $\lambda\mu$-terms are both shown to be reducible to their cut-free or normal forms in a number of steps bounded by a polynomial weight. Such a result for the former translates to a similar result for the latter, since any reduction step in $\lambda\mu$-terms corresponds to one or more reduction steps in proofs. The analysis is then extended to the reduction of $\lambda\mu$-terms by a Krivine-style abstract machine [dG98].

4.2 Bounded Polarized Linear Logic as A Sequent Calculus

In this section, we define BLLP as a sequent calculus. Although this section is self-contained, some familiarity with both Bounded [GSS92] and Polarized [Lau03b] Linear Logic would certainly help (see Sections 2.4 and 3.2).

4.2.1 Polynomials and Formulas

A *resource monomial* is any (finite) product of binomial coefficients in the form $\prod_{i=1}^{p} \binom{x_i}{n_i}$, where the x_i are distinct variables and the n_i are non-negative integers. A *resource polynomial* is any finite sum of resource monomials. Given resource polynomials p, q we write $p \sqsubseteq q$ to denote that $q - p$ is a resource polynomial. If $p \sqsubseteq r$ and $q \sqsubseteq s$ then also $q \circ p \sqsubseteq s \circ r$. Resource polynomials are closed by addition, multiplication, bounded sums and composition [GSS92].

A *polarized formula* is a formula (either positive or negative) generated by the

following grammar

$$P ::= \alpha(\vec{p}) \;\Big|\; P \otimes P \;\Big|\; 1 \;\Big|\; \exists \alpha P \;\Big|\; !_{x<p} N;$$

$$N ::= \alpha^{\perp}(\vec{p}) \;\Big|\; N \parr N \;\Big|\; \bot \;\Big|\; \forall \alpha N \;\Big|\; ?_{x<p} P.$$

where α ranges over a countable sets of atoms. Throughout this chapter, formulas (but also terms, contexts, etc.) are considered modulo α-equivalence. Formulas (either positive or negative) are ranged over by metavariables like A, B. Formulas like $\alpha^{\perp}$ are sometimes denoted as X, Y.

In a polarized setting, contraction can be performed on any negative formula. As a consequence, we need the notion of a *labelled formula* $[A]^p_x$, namely the *labelling* of the formula A with respect to x and p. The labelled formula $[N]^p_x$ (resp. $[P]^p_x$) can be thought of roughly as $?_{x<p} N^{\perp}$ (resp. $!_{x<p} P^{\perp}$), i.e., in a sense we can think of labelled formulas as formulas hiding an implicit exponential modality. All occurrences of x in A are bound in $[A]^p_x$. Metavariables for labellings of positive (respectively, negative) formulas are $\mathbf{P}, \mathbf{Q}, \mathbf{R}$ (respectively, $\mathbf{N}, \mathbf{M}, \mathbf{L}$). Labelled formulas are sometimes denoted with metavariables $\mathbf{A}, \mathbf{B}$ when their polarity is not essential. Negation, as usual in classical linear systems, can be applied to any (possibly labelled) formula, *à la* De Morgan. When the resource variable x does not appear in A, then we do not need to mention it when writing $[A]^p_x$, which becomes $[A]^p$. Similarly for $!_{x<p} N$ and $?_{x<p} P$.

Both the space of formulas and the space of labelled formulas can be seen as partial orders by stipulating that two (labelled) formulas can be compared iff they have *exactly* the same skeleton and the polynomials occurring in them can be compared.

Formally,

$$\begin{aligned}
\alpha(p_1,\ldots,p_n) &\sqsubseteq \alpha(q_1,\ldots,q_n) \text{ iff } \forall i.p_i \sqsubseteq q_i;\\
\alpha^{\perp}(p_1,\ldots,p_n) &\sqsubseteq \alpha^{\perp}(q_1,\ldots,q_n) \text{ iff } \forall i.q_i \sqsubseteq p_i;\\
1 &\sqsubseteq 1;\\
\perp &\sqsubseteq \perp;\\
P \otimes Q &\sqsubseteq R \otimes S \text{ iff } P \sqsubseteq R \wedge Q \sqsubseteq S;\\
N \parr M &\sqsubseteq O \parr K \text{ iff } N \sqsubseteq O \wedge M \sqsubseteq K;\\
!_{x<p}N &\sqsubseteq !_{x<q}M \text{ iff } q \sqsubseteq p \wedge N \sqsubseteq M;\\
?_{x<p}P &\sqsubseteq ?_{x<q}Q \text{ iff } p \sqsubseteq q \wedge P \sqsubseteq Q;\\
\forall\alpha.N &\sqsubseteq \forall\alpha.M \text{ iff } N \sqsubseteq M;\\
\exists\alpha.P &\sqsubseteq \exists\alpha.Q \text{ iff } P \sqsubseteq Q.
\end{aligned}$$

In a sense, then, polynomials occurring next to atoms or to the *whynot* operator are in positive position, while those occurring next to the *bang* operator are in negative position. In all the other cases, $\sqsubseteq$ is defined component-wise, in the natural way, e.g. $P \otimes Q \sqsubseteq R \otimes S$ iff both $P \sqsubseteq R$ and $Q \sqsubseteq S$. Finally $[N]^p_x \sqsubseteq [M]^q_x$ iff $N \sqsubseteq M \wedge p \sqsupseteq q$. And dually, $[P]^p_x \sqsubseteq [Q]^q_x$ iff $N \sqsubseteq M \wedge p \sqsubseteq q$.

Lemma 4.1 *$A \sqsubseteq B$ iff $B^{\perp} \sqsubseteq A^{\perp}$. Moreover, $\mathbf{A} \sqsubseteq \mathbf{B}$ iff $\mathbf{B}^{\perp} \sqsubseteq \mathbf{A}^{\perp}$.*

Proof: $A \sqsubseteq B$ iff $B^{\perp} \sqsubseteq A^{\perp}$ can be proved by induction on the structure of A. Consider the second part of the statement, now. Suppose that A, B are positive, and call them P, Q respectively. Then

$$\begin{aligned}
[P]^p_x \sqsubseteq [Q]^q_x &\Leftrightarrow P \sqsubseteq Q \wedge p \sqsubseteq q\\
&\Leftrightarrow Q^{\perp} \sqsubseteq P^{\perp} \wedge p \sqsubseteq q\\
&\Leftrightarrow [Q^{\perp}]^q_x \sqsubseteq [P^{\perp}]^p_x.
\end{aligned}$$

The case when A, B are negative is similar. □

Certain operators on resource polynomials can be lifted to formulas. As an example, we want to be able to *sum* labelled formulas provided they have a proper

form:

$$[N]_x^p \uplus [N\{x/y+p\}]_y^q = [N]_x^{p+q}.$$

We are assuming, of course, that x, y are not free in either p or q. This construction can be generalized to *bounded* sums: suppose that a labelled formula is in the form

$$[M]_y^r = [N\{x/y + \sum_{u<z} r\{z/u\}\}]_y^r,$$

where y and u are not free in N nor in r and z is not free in N. Then the labelled formula $\sum_{z<q}[M]_y^r$ is defined as $[N]_x^{\sum_{z<q} r}$. See [GSS92, §3.3] for more details about the above constructions.

An *abstraction formula* of arity n is simply a formula A, where the n resource variables $x_1, \ldots, x_n$ are meant to be bound. $A\{\alpha := B\}$ is the result of substituting a second order abstraction term B (of arity n) for all free occurrences of the propositional variable α (of the same arity) in A. This can be defined formally by induction on the structure of A, but the only interesting clauses are the following two:

$$\alpha(p_1, \ldots, p_n)\{\alpha := B\} = B\{x_1, \ldots, x_n/p_1, \ldots, p_n\}$$
$$\alpha^\perp(p_1, \ldots, p_n)\{\alpha := B\} = B^\perp\{x_1, \ldots, x_n/p_1, \ldots, p_n\}$$

4.2.2 Sequents and Rules

The easiest way to present BLLP is to give a sequent calculus for it. Actually, proofs will be structurally identical to proofs of Laurent's LLP. Of course, only *some* of LLP proofs are legal BLLP proofs — those giving rise to an exponential blow-up cannot be decorated according to the principles of Bounded Linear Logic.

A *sequent* is an expression in the form $\vdash \Gamma$, where $\Gamma = \mathbf{A}_1, \ldots \mathbf{A}_n$ is a multiset of labelled formulas such that at most one among $\mathbf{A}_1, \ldots, \mathbf{A}_n$ is positive. If Γ only contains (labellings of) negative formulas, we indicate it with metavariables like

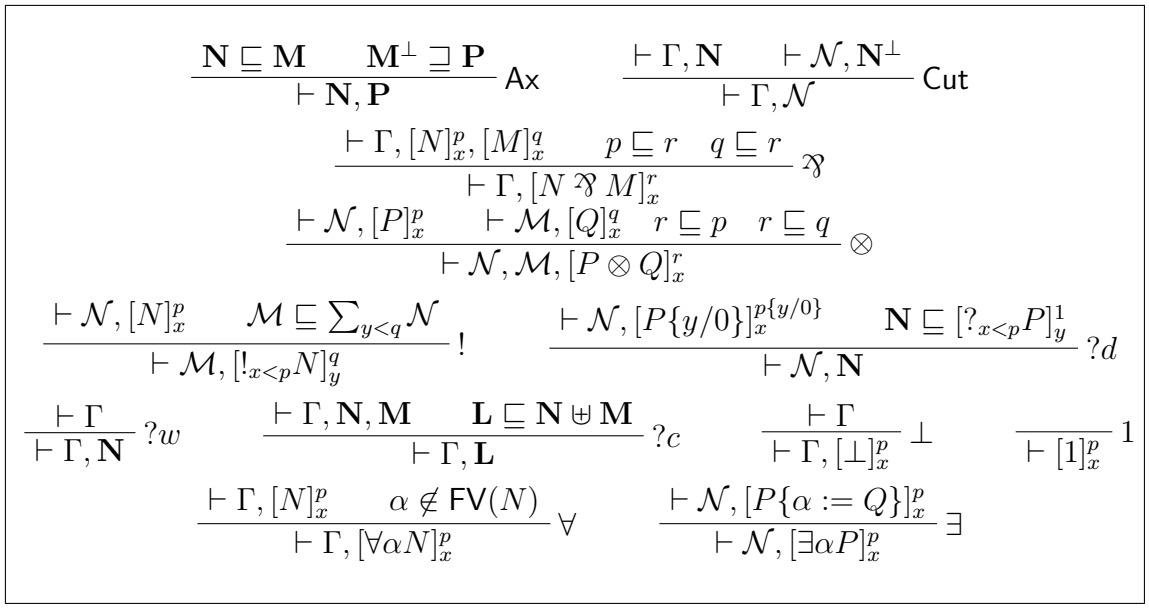

Figure 4.1: BLLP, Sequent Calculus Rules

$\mathcal{N}, \mathcal{M}$. The operator $\uplus$ can be extended to one on multisets of formulas componentwise, so we can write expressions like $\mathcal{N} \uplus \mathcal{M}$: this amounts to sum the polynomials occurring in $\mathcal{N}$ and those occurring in $\mathcal{M}$. Similarly for bounded sums.

The rules of the sequent calculus for BLLP are in Figure 4.1. Please observe that:

- The relation $\sqsubseteq$ is implicitly applied to both formulas and polynomials whenever possible in such a way that "smaller" formulas can always be derived (see Section 4.2.3).
- As in LLP, structural rules can act on any negative formula, and not only on exponential ones. Since all formulas occurring in sequents are labelled, however, we can still keep track of how many times formulas are "used", in the spirit of BLL.
- A byproduct of taking sequents as multisets of *labeled* formulas is that multiplicative rules themselves need to deal with labels. As an example, consider rule $\otimes$: the resource polynomial labelling the conclusion $P \otimes Q$ is anything smaller or equal to the polynomials labeling the two premises.

The sequent calculus we have just introduced could be extended with additive logical connectives. For the sake of simplicity, however, we have kept the language of formulas very simple here.

As already mentioned, BLLP proofs can be seen as obtained by decorating proofs from Laurent's LLP [Lau03b] with resource polynomials. Given a proof π, $\langle\pi\rangle$ is the LLP proof obtained by erasing all resource polynomials occurring in π. If π and ρ are two BLLP proofs, we write $\pi \sim \rho$ iff $\langle\pi\rangle = \langle\rho\rangle$, i.e., iff π and ρ are two decorations of the same LLP proof.

Even if structural rules can be applied to all negative formulas, only certain proofs will be copied or erased along the cut-elimination process, as we will soon realize. A *box* is any proof which ends with an occurrence of the ! rule. In non-polarized systems, only boxes can be copied or erased, while here the process can be applied to $\otimes$-*trees*, which are proofs inductively defined as follows:

- Either the last rule in the proof is Ax or ! or 1;
- or the proof is obtained from two $\otimes$-trees by applying the rule $\otimes$.

A $\otimes$-tree is said to be *closed* if it does not contain any axiom nor any box having auxiliary doors (i.e., no formula in the context of the ! rules).

4.2.3 Malleability

The main reason for the strong (intensional) expressive power of BLL [DLH09] is its *malleability*: the conclusion of any proof π can be modified in many different ways without altering its structure. Malleability is not only crucial to make the system expressive, but also to prove that BLLP enjoys cut-elimination. In this section, we give four different ways of modifying a sequent in such a way as to preserve its derivability. Two of them are anyway expected and also hold in BLL, while the other two only make sense in a polarized setting.

First of all, taking smaller formulas (i.e., more general — cf. [GSS92, §3.3, p. 21]) preserves derivability:

Lemma 4.2 (Subtyping) *If* $\pi \triangleright \ \vdash \Gamma, \mathbf{A}$ *and* $\mathbf{A} \sqsupseteq \mathbf{B}$*, then there is* $\rho \triangleright \ \vdash \Gamma, \mathbf{B}$ *such that* $\pi \sim \rho$*.*

Proof: By a simple induction on π. The crucial cases:

- If the last rule used is an axiom:
$$\frac{\mathbf{N} \sqsubseteq \mathbf{M} \qquad \mathbf{M}^{\perp} \sqsupseteq \mathbf{P}}{\vdash \mathbf{N}, \mathbf{P}}\,\mathsf{Ax}$$
If $\mathbf{B} \sqsubseteq \mathbf{N}$, then we know that $\mathbf{B} \sqsubseteq \mathbf{N} \sqsubseteq \mathbf{M}$, from which it follows that $\mathbf{B} \sqsubseteq \mathbf{M}$. We can thus take ρ as
$$\frac{\mathbf{B} \sqsubseteq \mathbf{M} \qquad \mathbf{M}^{\perp} \sqsupseteq \mathbf{P}}{\vdash \mathbf{B}, \mathbf{P}}\,\mathsf{Ax}$$
If $\mathbf{B} \sqsubseteq \mathbf{P}$, then we know that $\mathbf{B} \sqsubseteq \mathbf{P} \sqsubseteq \mathbf{M}^{\perp}$, from which it follows that $\mathbf{B} \sqsubseteq \mathbf{M}^{\perp}$. We can thus take ρ as
$$\frac{\mathbf{B} \sqsubseteq \mathbf{M} \qquad \mathbf{M}^{\perp} \sqsupseteq \mathbf{B}}{\vdash \mathbf{B}, \mathbf{M}}\,\mathsf{Ax}$$
- Suppose the last rule used is !:
$$\frac{\sigma \triangleright \ \vdash \mathcal{N}, [N]^{r}_{y} \qquad \mathcal{M} \sqsubseteq \sum_{x<q} \mathcal{N}}{\vdash \mathcal{M}, [!_{y<r} N]^{q}_{x}}\,!$$
If $\mathbf{B} \sqsubseteq [!_{y<r} N]^{q}_{x}$, then necessarily $\mathbf{B} = [!_{y<s} M]^{p}_{x}$, where $N \sqsupseteq M$, $q \sqsupseteq p$ and $s \sqsupseteq r$. Hence $[N]^{r}_{y} \sqsupseteq [M]^{s}_{y}$ and, by induction hypothesis, there is λ such that $\lambda \sim \sigma$ and $\lambda \triangleright \ \vdash \mathcal{N}, [M]^{s}_{y}$. As a consequence ρ can be simply defined as
$$\frac{\lambda \triangleright \ \vdash \mathcal{N}, [M]^{s}_{y} \qquad \mathcal{M} \sqsubseteq \sum_{x<p} \mathcal{N}}{\vdash \mathcal{M}, [!_{y<s} M]^{q}_{x}}\,!$$
since $\sum_{x<q} \mathcal{N} \sqsubseteq \sum_{x<p} \mathcal{N}$. If $\mathbf{B} \sqsubseteq \mathbf{N} \in \mathcal{M}$, then we can just derive the thesis from transitivity of $\sqsubseteq$.
- If the last rule used is $?d$:
$$\frac{\sigma \triangleright \ \vdash \mathcal{N}, [P\{x/0\}]^{r\{x/0\}}_{y} \qquad \mathbf{N} \sqsubseteq [?_{y<r} P]^{1}_{x}}{\vdash \mathcal{N}, \mathbf{N}}\,?d$$
Then the induction hypothesis immediately yields the thesis.

This concludes the proof. □

Substituting resource variables with polynomials itself preserves typability:

Lemma 4.3 (Substitution) *Let* $\pi \triangleright \ \vdash \Gamma$*. Then there is a proof* $\pi\{x/p\}$ *of* $\vdash \Gamma\{x/p\}$*. Moreover,* $\pi\{x/p\} \sim \pi$*.*

Proof: By an easy induction on the structure of π. □

Lemma 4.4 $[A]^p_x \sqsupseteq [B]^p_x \Rightarrow [A\{x/y+q\}]^p_y \sqsupseteq [B\{x/y+q\}]^p_y$.

Proof: $[A]^p_x \sqsupseteq [B]^p_x \Rightarrow A \sqsupseteq B \Rightarrow A\{x/y+q\} \sqsupseteq B\{x/y+q\} \Rightarrow [A\{x/y+q\}]^p_y \sqsupseteq [B\{x/y+q\}]^p_y$ □

As we have already mentioned, one of the key differences between Linear Logic and its polarized version is that in the latter, arbitrary proofs can potentially be duplicated (and erased) along the cut-elimination process, while in the former only special ones, namely boxes, can. This is, again, a consequence of the fundamentally different nature of structural rules in the two systems. Since BLLP is a refinement of LLP, this means that the same phenomenon is expected. But beware: in a bounded setting, contraction is not symmetric, i.e., the two copies of the proof π we are duplicating are not identical to π.

What we need to prove, then, is that proofs can indeed be *split* in BLLP. But preliminary to that is the following technical lemma:

Lemma 4.5 (Shifting Sums) *If* $\sum_{z<q}[M]^r_y = [N]^{\sum_{z<q} r}_y$, *then the formula* $\mathbf{N} = [M]^r_y\{z/z+q\}$ *is such that*

$$\sum_{z<p} \mathbf{N} = [N\{x/x + \sum_{z<q} r\}]^{\sum_{z<p} r\{z/z+q\}}_y$$

Proof: The fact $\sum_{z<q}[M]^r_y$ exists implies that there exist N, x, u such that

$$M = N\{x/y + \sum_{u<z} r\{z/u\}\}$$

and $y, z \notin \mathsf{FV}(N)$ and $y \notin \mathsf{FV}(r)$. As a consequence:

$$\begin{aligned}
[M]^r_y\{z/z+q\} &= [N\{x/y + \sum_{u<z} r\{z/u\}\}\{z/z+q\}]^{r\{z/z+q\}}_y \\
&= [N\{x/y + \sum_{u<z+q} r\{z/u\}\}]^{r\{z/z+q\}}_y \\
&= [N\{x/y + \sum_{u<q} r\{z/u\} + \sum_{u<z} r\{z/u+q\}\}]^{r\{z/z+q\}}_y \\
&= [N\{x/x + \sum_{u<q} r\{z/u\}\}\{x/y + \sum_{u<z} r\{z/u+q\}\}]^{r\{z/z+q\}}_y \\
&= [N\{x/x + \sum_{u<q} r\{z/u\}\}\{x/y + \sum_{u<z} r\{z/z+q\}\{z/u\}\}]^{r\{z/z+q\}}_y
\end{aligned}$$

Call the last formula $\mathbf{N}$. As a consequence, $\sum_{z<p} \mathbf{N}$ exists and is equal to

$$[N\{x/x + \sum_{u<q} r\{z/u\}\}]^{\sum_{z<p} r\{z/z+q\}}_y.$$

This concludes the proof. □

Lemma 4.6 (Splitting) *If $\pi \triangleright \vdash \mathcal{N}, [P]^p_x$ is a $\otimes$-tree and $p \sqsupseteq r+s$ then there exist $\mathcal{M}, \mathcal{O}$ such that $\rho \triangleright \vdash \mathcal{M}, [P]^r_x$, $\sigma \triangleright \vdash \mathcal{O}, [P\{x/y+r\}]^s_y$. Moreover, $\mathcal{N} \sqsubseteq \mathcal{M} \uplus \mathcal{O}$ and $\rho \sim \pi \sim \sigma$.*

Proof: By induction on π:

- If the last rule used is an axiom then it is in the form
$$\frac{[N]^q_x \sqsubseteq [M]^t_x \qquad [M^\perp]^t_x \sqsupseteq [P]^p_x}{\vdash [N]^q_x, [P]^p_x}\,\mathsf{Ax}$$
for some M, t. We know that
$$r + s \sqsubseteq p \sqsubseteq t \sqsubseteq q.$$
Observe that, we can form the following derivations
$$\frac{[N]^r_x \sqsubseteq [M]^r_x \qquad [M^\perp]^r_x \sqsupseteq [P]^r_x}{\vdash [N]^r_x, [P]^r_x}\,\mathsf{Ax}$$
$$\frac{[N\{x/y+r\}]^s_y \sqsubseteq [M\{x/y+r\}]^s_y \qquad [M^\perp\{x/y+r\}]^s_y \sqsupseteq [P\{x/y+r\}]^s_y}{\vdash [N\{x/y+r\}]^s_y, [P\{x/y+r\}]^s_y}\,\mathsf{Ax}$$

where in building the second one we made use, in particular, of Lemma 4.4.

- If the last rule used is $\otimes$ then we can write π as
$$\frac{\lambda_1 \triangleright \vdash \mathcal{N}_1, [P_1]_x^{p_1} \qquad \lambda_2 \triangleright \vdash \mathcal{N}_2, [P_2]_x^{p_2}}{\vdash \mathcal{N}_1, \mathcal{N}_2, [P_1 \otimes P_2]_x^{p}} \otimes$$
where $p \sqsubseteq p_1$ and $p \sqsubseteq p_2$. As a consequence, $p_1 \sqsupseteq q + r$ and $p_2 \sqsupseteq q + r$, and we can thus apply the induction hypothesis to λ_1, λ_2 easily reaching the thesis.
- If the last rule used is promotion ! then π has the following shape:
$$\frac{\lambda \triangleright \vdash \mathcal{N}, [N]_z^{q} \qquad \mathcal{M} \sqsubseteq \sum_{x<r} \mathcal{N}}{\vdash \mathcal{M}, [!_{z<q} N]_x^{p}} !$$
Then ρ is simply
$$\frac{\lambda \triangleright \vdash \mathcal{N}, [N]_z^{q}}{\vdash \sum_{x<r} \mathcal{N}, [!_{z<q} N]_x^{r}} !$$
About σ, observe that $\lambda\{x/y + r\}$ has conclusion
$$\vdash \mathcal{N}\{x/y + r\}, [N\{x/y + r\}]_z^{q\{x/y+r\}}$$
By Lemma 4.5, it is allowed to form $\sum_{y<s} \mathcal{N}\{x/y + r\}$. As a consequence, σ is
$$\frac{\vdash \mathcal{N}\{x/y + r\}, [N\{x/y + r\}]_z^{q\{x/y+r\}}}{\vdash \sum_{y<s} \mathcal{N}\{x/y + r\}, [(!_{z<q} N)\{x/y + r\}]_y^{s}} !$$
Observe that the conclusions of ρ and σ are in the appropriate relation, again because of Lemma 4.5.

This concludes the proof. □

Observe that not every proof can be split, but only $\otimes$-trees can. A parametric version of splitting is also necessary here:

Lemma 4.7 (Parametric Splitting) *If $\pi \triangleright \vdash \mathcal{N}, [P]_x^{p}$, where π is a $\otimes$-tree and $p \sqsupseteq \sum_{x<r} s$, then there exists $\rho \triangleright \vdash \mathcal{M}, [P]_x^{s}$ where $\sum_{x<r} \mathcal{M} \sqsupseteq \mathcal{N}$ and $\rho \sim \pi$.*

While splitting allows to cope with duplication, parametric splitting implies that an arbitrary $\otimes$-tree can be modified so as to be lifted into a box through one of its auxiliary doors.

The following is useful when dealing with cuts involving the rule $?d$:

Lemma 4.8 *If $q \sqsupseteq 1$, then $\sum_{z<q} [M]_y^{r} \sqsubseteq [M]_y^{r}\{z/0\}$.*

Proof: By hypothesis, we have that $\sum_{z<q}[M]^r_y = [N]^p_y$ for some N, y, p. As a consequence

$$M = N\{x/y + \sum_{u<z} p\{z/u\}\}.$$

Now:

$$\begin{aligned}[M]^r_y\{z/0\} &= [N\{x/y + \sum_{u<0} p\{z/u\}\}]^{r\{z/0\}}_y \\ &= [N\{x/y\}]^{r\{z/0\}}_y = [N]^{r\{z/0\}}_x \sqsupseteq [N]^{\sum_{z<q} r}_x\end{aligned}$$

This concludes the proof. □

4.3 Cut Elimination

In this section, we show how a cut-elimination procedure for BLLP can be defined. We start by showing how *logical* cuts can be reduced, where a cut is logical when the two immediate subproofs end with a rule introducing the formula involved in the cut. We describe how logical cuts can be reduced in the critical cases in Figure 4.2, which needs to be further explained:

- When reducing multiplicative logical cuts, we extensively use the Subtyping Lemma.
- In the dereliction reduction step, $\pi\{x/0\}$ (obtained through Lemma 4.3) has conclusion $\vdash \mathcal{N}\{x/0\}, [N\{x/0\}]^{p\{x/0\}}_y$. By Lemma 4.8, $\mathcal{M} \sqsubseteq \sum_{x<q} \mathcal{N} \sqsubseteq \mathcal{N}\{x/0\}$, and as a consequence, there is $\sigma \triangleright \ \vdash \mathcal{M}, [N\{x/0\}]^{p\{x/0\}}_y$. From $[?_{y<p} N^\perp]^q_x \sqsupseteq [?_{y<r} M^\perp]^1_x$, it follows that $[M^\perp\{x/0\}]^{r\{x/0\}}_y \sqsupseteq [N\{x/0\}]^{p\{x/0\}}_y$, and there is a proof $\lambda \triangleright \ \vdash \mathcal{O}, [N\{x/0\}]^{p\{x/0\}}_y$.
- In the contraction reduction step, we suppose that π is a $\otimes$-tree. Then we can apply Lemma 4.6 and Lemma 4.2, and obtain $\sigma \triangleright \ \vdash \mathcal{M}, [O^\perp]^p_x$ and $\lambda \triangleright \ \vdash \mathcal{O}, [O^\perp\{x/y + p\}]^q_y$ such that $\mathcal{M} \uplus \mathcal{O} \sqsupseteq \mathcal{N}$.
- In digging, by Lemma 4.7 from π we can find $\sigma \triangleright \ \vdash \mathcal{K}, [O^\perp]^s_y$, where $\mathcal{N} \sqsubseteq \sum_{x<q} \mathcal{K}$.

All instances of the Cut rule which are not logical are said to be *commutative*, and induce an equivalent relation on proofs. As an example, the proof

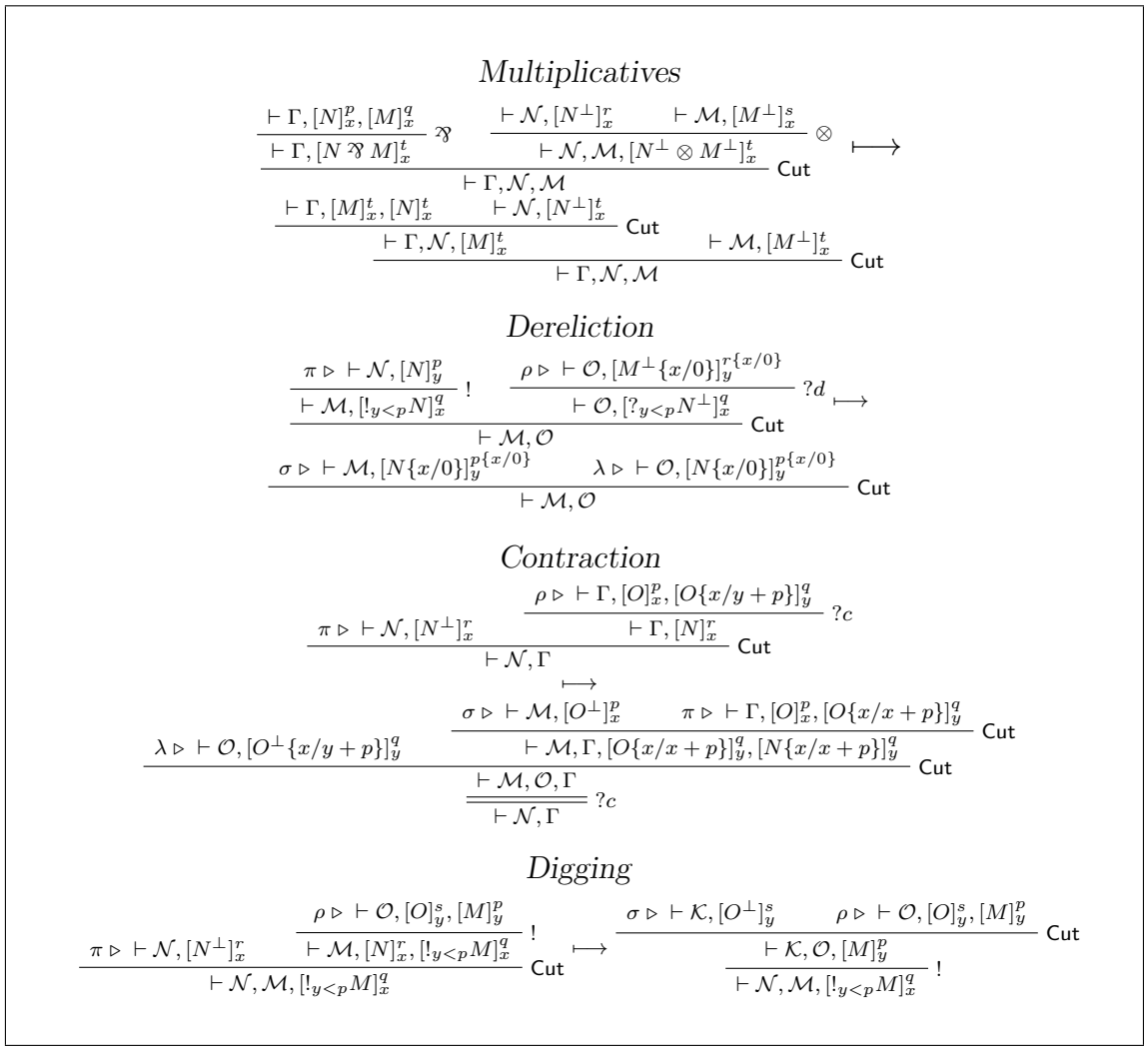

Figure 4.2: Some Logical Cut-Elimination Steps

$$\dfrac{\dfrac{\pi \triangleright \vdash \Gamma, \mathbf{N}, [N]^p_x, [M]^q_x}{\vdash \Gamma, \mathbf{N}, [N \parr M]^r_x}\parr \qquad \rho \triangleright \vdash \mathcal{N}, \mathbf{N}^\perp}{\vdash \Gamma, \mathcal{N}, [N \parr M]^r_x}\mathsf{Cut}$$

is equivalent to

$$\dfrac{\pi \triangleright \vdash \Gamma, \mathbf{N}, [N]^p_x, [M]^q_x \qquad \rho \triangleright \vdash \mathcal{N}, \mathbf{N}^\perp}{\dfrac{\vdash \Gamma, \mathcal{N}, [N]^p_x, [M]^q_x}{\vdash \Gamma, \mathcal{N}, [N \parr M]^r_x}\parr}\mathsf{Cut}$$

This way we can define an equivalence relation $\cong$ on the space of proofs. In general, not all cuts in a proof are logical, but any cut can be turned into a logical one:

Lemma 4.9 *Let π be any proof containing an occurrence of the rule* Cut*. Then, there are two proofs ρ and σ such that $\pi \cong \rho \longmapsto \sigma$, where ρ can be effectively obtained from π.*

The proof of Lemma 4.9 goes as follows: given any instance of the Cut rule

$$\dfrac{\pi \triangleright \vdash \Gamma, [N]^p_x \qquad \rho \triangleright \vdash \mathcal{N}, [P]^p_x}{\vdash \Gamma, \mathcal{N}}\mathsf{Cut}$$

consider the path (i.e., the sequence of formula occurrences) starting from $[N]^p_x$ and going upward inside π, and the path starting from $[P]^p_x$ and going upward inside ρ. Both paths end either at an Ax rule or at an instance of a rule introducing the main connective in N or P. The game to play is then to show that these two paths can always be *shortened* by way of commutations, thus exposing the underlying logical cut.

Lemma 4.9 is implicitly defining a cut-elimination procedure: given any instance of the Cut rule, turn it into a logical cut by the procedure from Lemma 4.9, then fire it. This way we are implicitly defining another reduction relation $\longrightarrow$. The next question is the following: is this procedure going to terminate for every proof π (i.e., is $\longrightarrow$ strongly, or weakly, normalizing)? How many steps does it take to turn π to its cut-free form?

Actually, $\longrightarrow$ produces reduction sequences of very long length, but is anyway strongly normalizing. A relatively easy way to prove it goes as follows: any BLLP

proof π corresponds to a LLP sequent calculus proof $\langle\pi\rangle$, and the latter itself corresponds to a polarized proof net $\langle\!\langle\pi\rangle\!\rangle$ [Lau03b]. Moreover, $\pi \longrightarrow \rho$ implies that $\langle\!\langle\pi\rangle\!\rangle \mapsto \langle\!\langle\rho\rangle\!\rangle$, where $\mapsto$ is the canonical cut-elimination relation on polarized proof-nets. Finally, $\langle\!\langle\pi\rangle\!\rangle$ is identical to $\langle\!\langle\rho\rangle\!\rangle$ whenever $\pi \cong \rho$. Since $\mapsto$ is known to be strongly normalizing, $\longrightarrow$ does not admit infinite reduction sequences:

Proposition 4.1 (Cut-Elimination) *The relation $\longrightarrow$ is strongly normalizing.*

This does not mean that cut-elimination can be performed in (reasonably) bounded time. Already in BLL this can take exponential time: the whole of Elementary Linear Logic [Gir98] can be embedded into it.

4.3.1 Polystep Soundness

To get a soundness result, then, we somehow need to restrict the underlying reduction relation $\longrightarrow$. Following [GSS92], one could indeed define a subset of $\longrightarrow$ just by imposing that in dereliction, contraction, or box cut-elimination steps, the involved $\otimes$-trees are closed. Moreover, we could stipulate that reduction is external, i.e., it cannot take place inside boxes. Closed and external reduction, however, is not enough to simulate head-reduction in the $\lambda\mu$-calculus, and not being able to reduce under the scope of μ-abstractions does not make much sense anyway. We are forced, then, to consider an extension of closed reduction. The fact that this new notion of reduction still guarantees polynomial bounds is technically a remarkable strengthening with respect to BLL's Soundness Theorem [GSS92].

There is a quite natural notion of *downward* path in proofs: from any occurrence of a negative formula $\mathbf{N}$, just proceed downward until you either find (the main premise of) a Cut rule, or a conclusion. In the first case, the occurrence of $\mathbf{N}$ is said to be *active*, in the second it is said to be *passive*. Proofs can then be endowed with a new notion of reduction: all dereliction, contraction or box digging cuts can be fired only if the negative formula occurrences in its rightmost argument are all passive. In the literature, this is sometimes called a *special cut* (e.g. [BCDL11]).

Moreover, reduction needs to be external, as usual. This notion of reduction, as we will see, is enough to mimic head reduction, and is denoted with $\Longrightarrow$.

The next step consists in associating a weight, in the form of a resource polynomial, to every proof, similarly to what happens in BLL. The *pre-weight* $\pi^\diamond$ of a proof π with conclusion $\vdash \mathbf{A}_1, \ldots, \mathbf{A}_n$ consists in:

- a resource polynomial p^π.
- n disjoints sets of resource variables $S_1^\pi, \ldots, S_n^\pi$, each corresponding to a formula in $\mathbf{A}_1, \ldots, \mathbf{A}_n$; if this does not cause ambiguity, the set of resource variables corresponding to a formula $\mathbf{A}$ will be denoted by $S^\pi(\mathbf{A})$. Similarly for $S^\pi(\Gamma)$, where Γ is a multiset of formulas.

If π has pre-weight $p^\pi, S_1^\pi, \ldots, S_n^\pi$, then the *weight* q^π of π is simply p^π where, however, all the variables in $S_1^\pi, \ldots, S_n^\pi$ are substituted with 0: $p^\pi\{\cup_{i=1}^n S_i^\pi/0\}$. The pre-weight of a proof π is defined by induction on the structure of π, following the rules in Figure 4.3. Please notice how any negative formula $\mathbf{N}$ in the conclusion of π is associated with some fresh variables, each accounting for the application of a rule to it. When $\mathbf{N}$ is then applied to a cut, all these variables are set to 1. This allows to discriminate between the case in which rules can "produce" time complexity along the cut-elimination, and the case in which they do not. Ultimately, this leads to:

Lemma 4.10 *If $\pi \cong \rho$, then $q^\pi = q^\rho$. If $\pi \Longrightarrow \rho$, then $q^\pi \sqsupset q^\rho$.*

The main idea behind Lemma 4.10 is that even if the logical cut we perform when going from π to ρ is "dangerous" (e.g. a contraction) *and* the involved $\otimes$-tree is not closed, the residual negative rules have null weight, because they are passive.

We can conclude that:

Theorem 4.1 (Polystep Soundness) *For every proof π, if $\pi \Longrightarrow^n \rho$, then $n \leq q^\pi$.*

Proof:[Sketch] The idea is that reducing one cut strictly reduces the weight. Analogously to [GSS92, Theorem 4.4], the proof proceeds by case analysis on the various cut-elimination steps (cf. Figure 4.2). The weight of a proof is defined in such a way

π	$\pi^\diamond$
$\dfrac{[M]^q_x \sqsubseteq [N]^p_x \qquad [N^\perp]^p_x \sqsupseteq [P]^r_x}{\vdash [M]^q_x, [P]^r_x}\ \mathsf{Ax}$	$\{y\}, \emptyset, y$
$\dfrac{\rho \triangleright \vdash \Gamma, [N]^p_x \qquad \sigma \triangleright \vdash \mathcal{N}, [N^\perp]^p_x}{\vdash \Gamma, \mathcal{N}}\ \mathsf{Cut}$	$S^\rho(\Gamma), S^\sigma(\mathcal{N}), p^\rho\{S^\rho([N]^p_x)/1\} + p^\sigma\{S^\sigma([N^\perp]^p_x)/1\}$
$\dfrac{\rho \triangleright \vdash \Gamma, [N]^p_x, [M]^q_x \qquad p \sqsubseteq r \quad q \sqsubseteq r}{\vdash \Gamma, [N \parr M]^r_x}\ \parr$	$S^\rho(\Gamma), S^\rho([N]^p_x) \cup S^\rho([M]^p_x) \cup \{y\}, p^\rho + y$
$\dfrac{\rho \triangleright \vdash \mathcal{N}, [P]^p_x \qquad \sigma \triangleright \vdash \mathcal{M}, [Q]^p_x \quad r \sqsubseteq p \quad r \sqsubseteq q}{\vdash \mathcal{N}, \mathcal{M}, [P \otimes Q]^r_z}\ \otimes$	$S^\rho(\mathcal{N}), S^\sigma(\mathcal{M}), S^\rho([P]^p_x) \cup S^\sigma([Q]^p_x), p^\rho + p^\sigma$
$\dfrac{\rho \triangleright \vdash \mathbf{N}_1, \ldots, \mathbf{N}_n, [M]^p_x \qquad \mathbf{M}_i \sqsubseteq \sum_{y<q} \mathbf{N}_i}{\vdash \mathbf{M}_1, \ldots, \mathbf{M}_n, [!_{x<p} M]^q_y}\ !$	$S^\rho(\mathbf{N}_1) \cup \{y_1\}, \ldots, S^\rho(\mathbf{N}_n) \cup \{y_n\}, p \cdot p^\rho + y_1 + \ldots + y_n$
$\dfrac{\rho \triangleright \vdash \mathcal{N}, [P\{y/0\}]^{p\{y/0\}}_x \qquad \mathbf{N} \sqsubseteq [?_{x<p} P]^1_y}{\vdash \mathcal{N}, \mathbf{N}}\ ?d$	$S^\rho(\mathcal{N}), S^\rho([P\{y/0\}]^{p\{y/0\}}_x) \cup \{y\}, p^\rho + y$
$\dfrac{\rho \triangleright \vdash \Gamma}{\vdash \Gamma, \mathbf{N}}\ ?w$	$S^\rho(\Gamma), \{y\}$
$\dfrac{\rho \triangleright \vdash \Gamma, \mathbf{N}, \mathbf{M} \qquad \mathbf{L} \sqsubseteq \mathbf{N} \uplus \mathbf{M}}{\vdash \Gamma, \mathbf{L}}\ ?c$	$S^\rho(\Gamma), S^\rho(\mathbf{N}) \cup S^\rho(\mathbf{M}) \cup \{y\}$
$\dfrac{\rho \triangleright \vdash \Gamma}{\vdash \Gamma, [\perp]^p_x}\ \perp$	$S^\rho(\Gamma), \{y\}, p^\rho + y$
$\dfrac{}{\vdash [1]^p_x}\ 1$	$\emptyset, 0$

Figure 4.3: Pre-weights for Proofs.

that eliminating special-cuts (cf. the beginning of this section), rather than simply closed cuts, it strictly decreases. □

In a sense, then, the weight of any proof π is a resource polynomial which can be easily computed from π (rules in Figure 4.3 are anyway inductively defined), but which is also an upper bound on the number of logical cut-elimination steps separating π from its normal form. Please observe that q^π continues to be such an upper bound even if any natural number is substituted for any of its free variables, an easy consequence of Lemma 4.3.

Why then, are we talking about *polynomial* bounds? In BLL, and as a consequence also in BLLP, one can write programs in such a way that the size of the input is reflected by a resource variable occurring in its type. Please refer to [GSS92].

4.4 A Type System for the $\lambda\mu$-Calculus

We describe here a version of the $\lambda\mu$-calculus as introduced by de Groote [dG94a]. Terms are as follows

$$t, u \ ::= \ x \ \mid \ \lambda x.t \ \mid \ \mu\alpha.t \ \mid \ [\alpha]t \ \mid \ (t)t,$$

where x and α range over two infinite disjoint sets of variables (called λ-variables and μ-variables, respectively). In contrast with the $\lambda\mu$-calculus as originally formulated by Parigot [Par92], μ-abstraction is not restricted to terms of the form $[\alpha]t$ here.

4.4.1 Notions of Reduction

The reduction rules we consider are the following ones:

$$(\lambda x.t)u \to_\beta t[{}^u/_x]; \qquad (\mu\alpha.t)u \to_\mu \mu\alpha.t[{}^{[\alpha](v)u}/_{[\alpha]v}]; \qquad \mu\alpha.[\alpha]t \to_\theta t;$$

where, as usual, $\to_\theta$ can be fired only if $\alpha \notin \mathsf{FV}(t)$. In the following, $\to$ is just $\to_{\beta\mu\theta}$. In so-called *weak reduction*, denoted $\to_{\mathsf{w}}$, reduction simply cannot take place in the scope of binders, while *head reduction*, denoted $\to_{\mathsf{h}}$, is a generalization of the same concept from pure λ-calculus [dG98]. Details are in Figure 4.4. Please notice

$$\frac{t \to u}{t \to_{\mathsf{w}} u} \qquad \frac{t \to_{\mathsf{w}} u}{tv \to_{\mathsf{w}} uv} \qquad \frac{t \to_{\mathsf{w}} u}{[\alpha]t \to_{\mathsf{w}} [\alpha]u}$$

$$\frac{t \to_{\mathsf{w}} u}{t \to_{\mathsf{h}} u} \qquad \frac{t \to_{\mathsf{h}} u}{\lambda x.t \to_{\mathsf{h}} \lambda x.u} \qquad \frac{t \to_{\mathsf{h}} u}{\mu\alpha.t \to_{\mathsf{h}} \mu\alpha.u}$$

Figure 4.4: Weak and Head Notions of Reduction

how in head reduction, redexes can indeed be fired even if they lie in the scope of λ-or-μ-abstractions, which, however, cannot themselves be involved in a redex. This harmless restriction, which corresponds to taking the *outermost* reduction order, is needed for technical reasons that will become apparent soon.

Remark 21 *Here we have not explicitly considered the auxiliary rules ρ and ϵ for renaming and the elimination of absurd weakening. Nevertheless, the application of these rules is not computationally problematic since they decrease the size of the term and De Groote proves [dG98, Appendix A] that all auxiliary rules can be postponed w.r.t. β and μ reduction.*

4.4.2 The Type System

Following Laurent [Lau03b], types are just negative formulas. Not all of them can be used as types, however: in particular, $N \parr M$ is a legal type only if N is in the form $?_{x<p}O^{\perp}$, and we use the following abbreviation in this case: $N \multimap_x^p M = (?_{x<p}N^{\perp}) \parr M$. In particular, if M is $\perp$ then $N \multimap_x^p M$ can be abbreviated as $\neg_x^p N$. *Typing formulas* are negative formulas which are either $\perp$, or X, or in the form $N \multimap_x^p M$ (where N and M are typing formulas themselves). A *modal formula* is one in the form $?_{x<p}N^{\perp}$ (where N is a typing formula).[2] Please observe that all the constructions from Section 4.2.1 (including labellings, sums, etc.) easily apply

[2]Basically typing formulas (resp. modal formulas) are what Laurent calls *output types* (resp. *input types*), see § 3.2.1.

$$\frac{1 \sqsubseteq p, r\{y/0\} \sqsubseteq q, M \sqsubseteq N\{y/0\}}{\Gamma, x : {}^{r}_{z}[N]^{p}_{y} \vdash x : [M]^{q}_{z} \mid \Delta}\ \mathsf{var}$$

$$\frac{\Gamma, x : {}^{s}_{z}[N]^{p}_{y} \vdash t : [M]^{q}_{y} \mid \Delta \qquad r \sqsupseteq q, r \sqsupseteq p}{\Gamma \vdash \lambda x.t : [N \multimap^{s}_{z} M]^{r}_{y} \mid \Delta}\ \mathsf{abs}$$

$$\frac{\Theta \vdash t : [N \multimap^{p}_{x} M]^{q}_{y} \mid \Psi \qquad \Xi \vdash u : [N]^{p}_{x} \mid \Phi \qquad \begin{array}{l} h \sqsupseteq q \quad k \sqsupseteq q \\ \Gamma \sqsubseteq \Theta \uplus \Upsilon \quad \Upsilon \sqsubseteq \sum_{b<h} \Xi \\ \Delta \sqsubseteq \Psi \uplus \Pi \quad \Pi \sqsubseteq \sum_{b<h} \Phi \end{array}}{\Gamma \vdash (t)u : [M]^{k}_{y} \mid \Delta}\ \mathsf{app}$$

$$\frac{\Gamma \vdash t : \mathbf{N} \mid \alpha : \mathbf{M}, \Delta \qquad \mathbf{L} \sqsubseteq \mathbf{N} \uplus \mathbf{M}}{\Gamma \vdash [\alpha]t : [\bot]^{q}_{z} \mid \alpha : \mathbf{L}, \Delta}\ \mu\text{-name} \qquad \frac{\Gamma \vdash t : [\bot]^{q}_{z} \mid \beta : \mathbf{N}, \Delta}{\Gamma \vdash \mu\beta t : \mathbf{N} \mid \Delta}\ \mu\text{-abs}$$

Figure 4.5: (Additive) Type Assignment Rules

to typing formulas. Finally, we use the following abbreviation for labeled modal formulas: ${}^{q}_{y}[N]^{p}_{x} = [?_{y<q}N^{\perp}]^{p}_{x}$.

A *typing judgment* is a statement in the form $\Gamma \vdash t : \mathbf{N} \mid \Delta$, where:

- Γ is a context assigning labelled modal formulas to λ-variables;
- t is a $\lambda\mu$-term;
- $\mathbf{N}$ is a typing formula;
- Δ is a context assigning labelled typing formulas to μ-variables.

The way typing judgments are defined allows to see them as BLLP sequents. This way, again, various concepts from Section 4.2.2 can be lifted up from sequents to judgments, and this remarkably includes the subtyping relation $\sqsubseteq$.

Typing rules are in Figure 4.5. The typing rule for applications, in particular, can be seen as overly complicated. In fact, all premises except the first two are there to allow the necessary degree of malleability for contexts, without which even subject reduction would be in danger. Alternatively, one could consider an explicit subtyping rule, the price being the loss of syntax directedness. Indeed, all malleability results from Section 4.2.3 can be transferred to the just defined type assignment system.

4.4.3 Subject Reduction and Polystep Soundness

The aim of this section is to show that *head* reduction preserves types, and as a corollary, that the number of reduction steps to normal form is bounded by a polynomial, along the same lines as in Theorem 4.1. Actually, the latter will easily follow from the former, because so-called Subject Reduction will be formulated (and in a sense proved) with a precise correspondence between type derivations and proofs in mind.

In order to facilitate this task, Subject Reduction is proved on a modified type-assignment system, called $\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}$ which can be proved equivalent to $\mathsf{BLLP}_{\lambda\mu}$. The only fundamental difference between the two systems lies in how structural rules, i.e., contraction and weakening, are reflected into the type system. As we have already noticed, $\mathsf{BLLP}_{\lambda\mu}$ has an *additive* flavour, since structural rules are implicitly applied in binary and 0-ary typing rules. [3] This, in particular, makes the system syntax directed and type derivations more compact. The only problem with this approach is that the correspondence between type derivations and proofs is too weak to be directly lifted to a dynamic level (e.g., one step in $\rightarrow_{\mathsf{h}}$ could correspond to possibly many steps in $\Longrightarrow$). In $\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}$, on the contrary, structural rules are explicit, and turns it into a useful technical tool to prove properties of $\mathsf{BLLP}_{\lambda\mu}$.

$\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}$'s typing judgments are precisely the ones of $\mathsf{BLLP}_{\lambda\mu}$. What changes are typing rules, which are in Figure 4.6. Whenever derivability in one of the systems needs to be distinguished from derivability on the other, we will put the system's name in subscript position (e.g. $\Gamma \vdash_{\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}} t : \mathbf{N} \mid \Delta$). Not so surprisingly, $\mathsf{BLLP}_{\lambda\mu}$ and $\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}$ type exactly the same class of terms:

Lemma 4.11 $\Gamma \vdash_{\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}} t : \mathbf{N} \mid \Delta$ *iff* $\Gamma \vdash_{\mathsf{BLLP}_{\lambda\mu}} t : \mathbf{N} \mid \Delta$

Proof: The left-to-right implication follows from weakening and contraction lemmas for $\mathsf{BLLP}_{\lambda\mu}$, which are easy to prove. The right-to-left implication is more direct, since additive var and app are multiplicatively derivable. □

[3] E.g., the var (resp. app) rule implicitly uses weakening (resp. contraction). In multiplicative sequent calculus the context of the conclusion is always a juxtaposition of the contexts of the premises, but that is not always the case with additive rules (cf. Section 2.3).

$$\frac{1 \sqsubseteq p, r\{y/0\} \sqsubseteq q, M \sqsubseteq N\{y/0\}}{x : {}^{r}_{z}[N]^{p}_{y} \vdash x : [M]^{q}_{z}|}\ \mathsf{var}$$

$$\frac{\Gamma, x : {}^{s}_{z}[N]^{p}_{y} \vdash t : [M]^{q}_{y}|\Delta \qquad r \sqsupseteq q, r \sqsupseteq p}{\Gamma \vdash \lambda x.t : [N \multimap^{s}_{z} M]^{r}_{y}|\Delta}\ \mathsf{abs}$$

$$\frac{\Gamma \vdash t : [N \multimap^{p}_{x} M]^{q}_{y}|\Delta \qquad \Theta \vdash u : [N]^{p}_{x}|\Xi \qquad \begin{array}{l} h \sqsupseteq q, k \sqsupseteq q, \\ \Psi \sqsubseteq \sum_{b<h} \Theta, \\ \Phi \sqsubseteq \sum_{b<h} \Xi \end{array}}{\Gamma, \Psi \vdash (t)u : [M]^{k}_{y}|\Delta, \Phi}\ \mathsf{app}$$

$$\frac{\Gamma \vdash t : \mathbf{N}|\Delta}{\Gamma \vdash [\alpha]t : [\bot]^{q}_{z}|\alpha : \mathbf{N}, \Delta}\ \mu\text{-name} \qquad \frac{\Gamma \vdash t : [\bot]^{q}_{z}|\beta : \mathbf{N}, \Delta}{\Gamma \vdash \mu\beta.t : \mathbf{N}|\Delta}\ \mu\text{-abs}$$

$$\frac{\Gamma \vdash t : \mathbf{N} \mid \Delta}{\Gamma, y : \mathbf{M} \vdash t : \mathbf{N} \mid \Delta}\ ?w^{\lambda} \qquad \frac{\Gamma, x : \mathbf{N}, y : \mathbf{M} \vdash t : \mathbf{O} \mid \Delta \qquad \mathbf{L} \sqsubseteq \mathbf{N} \uplus \mathbf{M}}{\Gamma, z : \mathbf{L} \vdash t\{x/z\}\{y/z\} : \mathbf{O} \mid \Delta}\ ?c^{\lambda}$$

$$\frac{\Gamma \vdash t : \mathbf{N} \mid \Delta}{\Gamma \vdash t : \mathbf{N} \mid \Delta, \alpha : \mathbf{M}}\ ?w^{\mu} \qquad \frac{\Gamma \vdash t : \mathbf{O} \mid \Delta, \alpha : \mathbf{N}, \beta : \mathbf{M} \qquad \mathbf{L} \sqsubseteq \mathbf{N} \uplus \mathbf{M}}{\Gamma \vdash t\{\alpha/\gamma\}\{\beta/\gamma\} : \mathbf{O} \mid \Delta, \gamma : \mathbf{L}}\ ?c^{\mu}$$

Figure 4.6: (Multiplicative) Type Assignment Rules

Let $\pi \triangleright \vdash \mathcal{N}, [N]^p_x$ be a proof of BLLP without a positive conclusion and let h a resource polynomial. The proof obtained by h-boxing π is the following:

$$\frac{\pi \triangleright \vdash \mathcal{N} \qquad \mathcal{M} \sqsubseteq \sum_{y<h} \mathcal{N}}{\mathcal{M}, [!_{x<p} N]^h_y}\,!$$

Given a $\mathsf{BLLP}^{\mathsf{mult}}_{\lambda\mu}$ type derivation π, one can define a BLLP proof $\pi^\diamond$ following the rules in Figure 4.7, which work by induction on the structure of π. This way one not only gets some guiding principles for subject-reduction, but can also prove that the underlying transformation process is nothing more than cut-elimination:

Lemma 4.12 (λ-Substitution) *If $\pi \triangleright \Gamma, x : {}^s_z[N]^p_y \vdash t : [M]^q_y \mid \Delta$ and $\rho \triangleright \Theta \vdash u : [N]^s_z \mid \Xi$, then for all $h \sqsubseteq q$ there is σ_h such that*

$$\sigma_h \triangleright \Gamma, \sum_{b<h} \Theta \vdash t\{x/u\} : [M]^q_y \mid \Delta, \sum_{b<h} \Xi.$$

Moreover, the proof obtained by h-boxing $\rho^\diamond$ and cutting it against $\pi^\diamond$ is guaranteed to $\Longrightarrow$-reduce to σ_h.

Proof: As usual, this is an induction on the structure of π. We only need to be careful and generalize the statement to the case in which a *simultaneous* substitution for many variables is needed. □

Lemma 4.13 (μ-Substitution) *If $\pi \triangleright \Gamma \vdash t : [\bot]^q_y \mid \Delta, \alpha : [N \multimap^s_z M]^p_y$ and $\rho \triangleright \Theta \vdash u : [N]^s_z \mid \Xi$, then for all $h \sqsupseteq q$ there is σ_h such that*

$$\Gamma, \sum_{b<h} \Theta \vdash t\{[\alpha]w/[\alpha](w)u\} : [\bot]^q_y \mid \Delta, \alpha : [M]^p_y, \sum_{b<h} \Xi$$

Moreover, the proof obtained by h-boxing $\rho^\diamond$, tensoring it with an axiom and cutting the result against $\pi^\diamond$ is guaranteed to $\Longrightarrow$-reduce to σ_h.

The proofs of the two lemmas above can be found in the Appendix (they are not part of this chapter for typographical reasons).

Theorem 4.2 (Subject Reduction) *Let $\pi \triangleright \Gamma \vdash t : \mathbf{N} \mid \Delta$ and suppose $t \to_{\mathsf{h}} u$. Then there is $\rho \triangleright \Gamma \vdash u : \mathbf{N} \mid \Delta$. Moreover $\pi^\diamond \Longrightarrow^+ \rho^\diamond$.*

π	$\pi^\diamond$
$\dfrac{1 \sqsubseteq p, r\{y/0\} \sqsubseteq q, M \sqsubseteq N\{y/0\}}{x : {}^{r}_{z}[N]^{p}_{y} \vdash x : [M]^{q}_{z}\lvert}\ \mathsf{var}$	$\dfrac{}{\dfrac{\vdash [N^\perp\{y/0\}]^{r\{y/0\}}_{z}, [M]^{q}_{z}}{\vdash [?_{z<r}N^\perp]^{p}_{y}, [M]^{q}_{z}}}$
$\dfrac{\rho \triangleright \Gamma, x : {}^{s}_{z}[N]^{p}_{y} \vdash t : [M]^{q}_{y}\lvert\Delta}{\Gamma \vdash \lambda x.t : [N \multimap^{s}_{z} M]^{r}_{y}\lvert\Delta}\ \mathsf{abs}$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, [?_{z<s}N]^{p}_{y}, [M]^{q}_{y}, \Delta}{\vdash \Gamma, [?_{z<s}N \parr M]^{r}_{y}, \Delta}$
$\dfrac{\begin{array}{c}\rho \triangleright \Gamma \vdash t : [N \multimap^{p}_{x} M]^{q}_{y}\lvert\Delta \\ \sigma \triangleright \Theta \vdash u : [N]^{p}_{x}\lvert\Xi\end{array}}{\Gamma, \Psi \vdash (t)u : [M]^{k}_{y}\lvert\Delta, \Phi}\ \mathsf{app}$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, [?_{x<p}N^\perp \parr M]^{q}_{y}, \Delta \qquad \dfrac{\dfrac{\sigma^\diamond \triangleright \vdash \Theta, [N]^{p}_{x}, \Xi}{\vdash \Psi, [!_{x<p}N]^{h}_{y}, \Phi} \qquad \vdash [M^\perp]^{k}_{y}, [M]^{k}_{y}}{\vdash \Psi, [!_{x<p}N \otimes M^\perp]^{q}_{y}, \Phi, [M]^{k}_{y}}}{\vdash \Gamma, \Psi, [M]^{k}_{y}, \Delta, \Phi}$
$\dfrac{\rho \triangleright \Gamma \vdash t : \mathbf{N}\lvert\Delta}{\Gamma \vdash [\alpha]t : [\perp]^{q}_{z}\lvert\alpha : \mathbf{N}, \Delta}\ \mu\text{-name}$	$\dfrac{\rho^\diamond \triangleright \Gamma, \mathbf{N}, \Delta}{\rho^\diamond \triangleright \Gamma, [\perp]^{q}_{z}, \mathbf{N}, \Delta}$
$\dfrac{\rho \triangleright \Gamma \vdash t : [\perp]^{q}_{z}\lvert\beta : \mathbf{N}, \Delta}{\Gamma \vdash \mu\beta t : \mathbf{N}\lvert\Delta}\ \mu\text{-abs}$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, [\perp]^{q}_{z}, \mathbf{N}, \Delta \qquad \vdash [1]^{q}_{z}}{\vdash \Gamma, \mathbf{N}, \Delta}$
$\dfrac{\rho \triangleright \Gamma \vdash t : \mathbf{N} \mid \Delta}{\Gamma, y : \mathbf{M} \vdash t : \mathbf{N} \mid \Delta}\ ?w^\lambda$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, \mathbf{N}, \Delta}{\vdash \Gamma, \mathbf{M}, \mathbf{N}, \Delta}$
$\dfrac{\rho \triangleright \Gamma \vdash t : \mathbf{N} \mid \Delta}{\Gamma \vdash t : \mathbf{N} \mid \Delta, \alpha : \mathbf{M}}\ ?w^\mu$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, \mathbf{N}, \Delta}{\vdash \Gamma, \mathbf{N}, \mathbf{M}, \Delta}$
$\dfrac{\rho \triangleright \Gamma, x : \mathbf{N}, y : \mathbf{M} \vdash t : \mathbf{O} \mid \Delta}{\Gamma, z : \mathbf{L} \vdash t\{x/z\}\{y/z\} : \mathbf{O} \mid \Delta}\ ?c^\lambda$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, \mathbf{N}, \mathbf{M}, \mathbf{O}, \Delta}{\vdash \Gamma, \mathbf{L}, \mathbf{O}, \Delta}$
$\dfrac{\rho \triangleright \Gamma \vdash t : \mathbf{O} \mid \Delta, \alpha : \mathbf{N}, \beta : \mathbf{M}}{\Gamma \vdash t\{\alpha/\gamma\}\{\beta/\gamma\} : \mathbf{O} \mid \Delta, \gamma : \mathbf{L}}\ ?c^\mu$	$\dfrac{\rho^\diamond \triangleright \vdash \Gamma, \mathbf{O}, \mathbf{N}, \mathbf{M}, \Delta}{\vdash \Gamma, \mathbf{O}, \mathbf{L}, \Delta}$

Figure 4.7: Mapping of (multiplicative) derivations into BLLP proofs

Proof: By induction on the structure of π. Here are some interesting cases:

- If t is an application, reduction takes place inside t, and π is as follows

$$\frac{\Gamma \vdash t : [N \multimap^p_x M]^q_y | \Delta \qquad \Theta \vdash v : [N]^p_y | \Xi \qquad h \sqsupseteq q, k \sqsupseteq q}{\Gamma \uplus \sum_{b<h} \Theta \vdash (t)v : [M]^k_z | \Delta \uplus \sum_{b<h} \Xi} \text{ app}$$

  then ρ is

$$\frac{\Gamma \vdash u : [N \multimap^p_x M]^q_y | \Delta \qquad \Theta \vdash v : [N]^p_y | \Xi \qquad h \sqsupseteq q, k \sqsupseteq q}{\Gamma \uplus \sum_{b<h} \Theta \vdash (u)v : [M]^k_z | \Delta \uplus \sum_{b<h} \Xi} \text{ app}$$

  which exists by induction hypothesis. We omit the other trivial cases.

- If t is a β-redex, then π looks as follows:

$$\frac{\dfrac{\Gamma, x : {}^s_z[N]^p_y \vdash t : [M]^q_y | \Delta \qquad r \sqsupseteq q, r \sqsupseteq p}{\Gamma \vdash \lambda x.t : [N \multimap^s_y M]^r_u | \Delta} \text{ abs} \qquad \Theta \vdash u : [N]^s_z | \Xi \qquad h \sqsupseteq q, k \sqsupseteq q}{\Gamma, \sum_{b<h} \Theta \vdash (\lambda x.t)u : [M]^k_y | \Delta, \sum_{b<h} \Xi} \text{ app}$$

  Lemma 4.12 ensures that the required type derivation actually exists:

$$\Gamma, \sum_{b<h} \Theta \vdash t\{x/u\} : [M]^k_y | \Delta, \sum_{b<h} \Xi$$

- If t is a μ-redex, then π looks as follows:

$$\frac{\dfrac{\Gamma \vdash t : [\bot]^q_z | \beta : [N \multimap^s_z M]^p_y, \Delta}{\Gamma \vdash \mu\beta t : [N \multimap^s_z M]^p_y | \Delta} \mu\text{-abs} \qquad \Theta \vdash u : [N]^s_y | \Xi \qquad h \sqsupseteq p, k \sqsupseteq p}{\Gamma, \sum_{b<h} \Theta \vdash (\mu\beta.t)u : [M]^k_z | \Delta, \sum_{b<h} \Xi} \text{ app}$$

  and Lemma 4.13 ensures us that ρ exists for

$$\Gamma \uplus \sum_{b<h} \Theta \vdash \mu\beta.t[{}^{[\beta](v)u}/_{[\beta]v}] : [M]^k_z | \Delta \uplus \sum_{b<h} \Xi$$

- If t is a θ-redex, then π looks as follows:

$$\frac{\dfrac{\dfrac{\pi \ \triangleright \ \Gamma \vdash t : [N]^p_x \mid \Delta}{\Gamma \vdash t : [N]^p_x \mid \Delta, \alpha : [N]^q_y} ?w^\mu \qquad r \sqsupseteq p + q}{\Gamma \vdash [\alpha]t : [\bot]^s_x \mid \Delta, \alpha : [N]^r_y} \mu\text{-abs}}{\Gamma \vdash t : [N]^r_x \mid \Delta} \mu\text{-name}$$

  Since $r = p + q \sqsupseteq p$ we know that

$$\pi^S \ \triangleright \ \Gamma \vdash t : [N]^r_x \mid \Delta$$

  where π^S is the derivation obtained from π, applying the Subtyping Lemma to the derivation π.

This concludes the proof. □

Observe how performing head reduction corresponds to $\Longrightarrow$, instead of the more permissive $\longrightarrow$. The following, then, is an easy corollary of Theorem 4.2 and Theorem 4.1:

Theorem 4.3 (Polystep Soundness for Terms) *Let* $\pi \triangleright \Gamma \vdash t : \mathbf{N} \mid \Delta$ *and let* $t \to_{\mathsf{h}}^{n} u$. *Then* $n \leq q^{\pi^{\diamond}}$.

4.5 Control Operators

In this section, we show that $\mathsf{BLLP}_{\lambda\mu}$ is powerful enough to type (the natural encoding of) two popular control operators, namely Scheme's `callcc` and Felleisen's $\mathcal{C}$ [AH03, Lau03b].

Control operators change the evaluation context of an expression. This is simulated by the operators μ and $[\cdot]$ which can, respectively, save and restore a stack of arguments to be passed to subterms. This idea, by the way, is the starting point of an extension of Krivine's machine for de Groote's $\lambda\mu$ [dG98] (see Section 4.6).

An extension of de Groote's calculus named $\Lambda\mu$-calculus [Sau05] satisfies a Böhm separation theorem that fails for Parigot's calculus [DP01]. Hence in an untyped setting the original $\lambda\mu$ of Parigot is strictly less expressive than de Groote's calculus.

4.5.1 `callcc`

An encoding of `callcc` into the $\lambda\mu$-calculus could be, e.g.,

$$\kappa = \lambda x.\mu\alpha.[\alpha](x)\lambda y.\mu\beta.[\alpha]y.$$

Does κ have the operational behavior we would expect from `callcc`? First of all, it should satisfy the following property (see [Fel90]): if $k \notin \mathsf{FV}(e)$, then $(\kappa)\lambda k.e \to^{*} e$. Indeed:

$$(\lambda x.\mu\alpha.[\alpha](x)\lambda y.\mu\beta.[\alpha]y)\lambda k.e \to_{\mathsf{h}} \mu\alpha.[\alpha](\lambda k.e)\lambda y.\mu\beta.[\alpha]y \to_{\mathsf{h}} \mu\alpha.[\alpha]e \to_{\mathsf{h}} e,$$

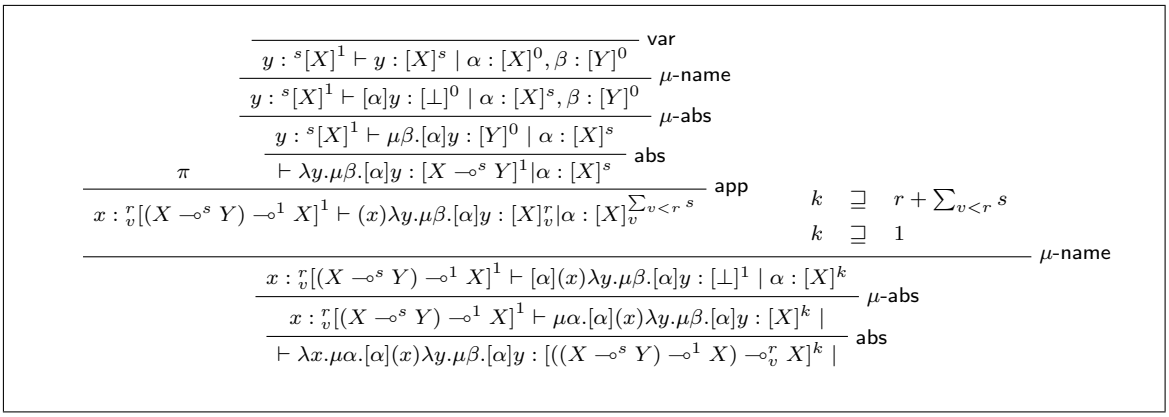

Figure 4.8: A Type Derivation for κ

where the second β-reduction step replaces $e\{k/\lambda y.\mu\beta.[\alpha]y\}$ with e since $k \notin \mathsf{FV}(e)$ by hypothesis. It is important to observe that the second step replaces a variable for a term with a free μ-variable, hence weak reduction gets stuck. Actually, our notion of weak reduction is even more restrictive than the one proposed by de Groote in [dG98]. Head reduction, on the contrary, is somehow more liberal. Moreover, it is also straightforward to check that the reduction of `callcc` in [Par92, §3.4] can be simulated by head reduction on κ.

But is κ typable in $\mathsf{BLLP}_{\lambda\mu}$? The answer is positive: a derivation typing it with (an instance of) Peirce's law is in Figure 4.8, where π is the obvious derivation of

$$x : {}^{r}_{v}[(X \multimap^{s} Y) \multimap^{1} X]^{1} \vdash x : [(X \multimap^{s} Y) \multimap^{1} X]^{r}_{v} \mid \alpha : [X]^{0}.$$

4.5.2 Felleisen's $\mathcal{C}$

The canonical way to encode Felleisen's $\mathcal{C}$ as a $\lambda\mu$-term is as the term $\aleph = \lambda f.\mu\alpha.(f)\lambda x.[\alpha]x$. Its behavior should be something like

$$(\aleph)wt_1 \dots t_k \to (w)\lambda x.(x)t_1 \dots t_k,$$

where $x \notin \mathsf{FV}(t_1, \dots, t_k)$, i.e., x is a fresh variable. Indeed:

$$(\aleph)wt_1 \dots t_k \to_{\mathsf{h}} (\mu\alpha.(w)\lambda x.[\alpha](x))t_1 \dots t_k \to^{k}_{\mathsf{h}} \mu\alpha.(w)\lambda x.[\alpha](x)t_1 \dots t_k.$$

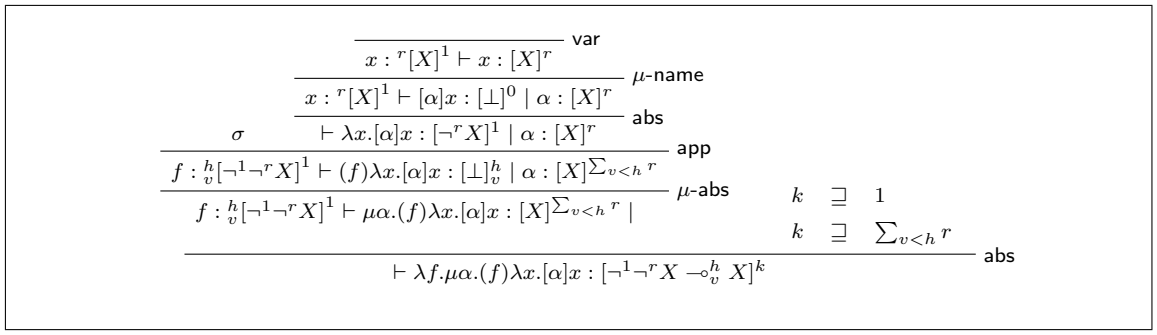

Figure 4.9: A Type Derivation for ℵ

A type derivation for ℵ is in Figure 4.9, where σ is a derivation for

$$f : {}^{h}_{v}[\neg^1\neg^r X]^1 \vdash f : [\neg^1\neg^r X]^h_v \mid \alpha : [X]^0.$$

It is worth noting that weak reduction is strong enough to properly simulating the operational behavior of $\mathcal{C}$. It is not possible to type $\mathcal{C}$ in Parigot's $\lambda\mu$, unless an open term is used. Alternatively, a free continuation constant must be used (obtaining yet another calculus [AH03]). This is one of the reasons why we picked the version of $\lambda\mu$-calculus proposed by de Groote over other calculi. See [dG94b] for a discussion about $\lambda\mu$-and-λ-calculi and Felleisen's $\mathcal{C}$.

4.6 Abstract Machines

Theorem 4.3, the main result of this paper so far, tells us that the number of *head-reduction steps* performed by terms typable in $\mathsf{BLLP}_{\lambda\mu}$ is bounded by the weight of the underlying type derivation. One may wonder, however, whether taking the number of reduction steps as a measure of term complexity is sensible or not — substitutions involve arguments which can possibly be much bigger than the original term. Recent work by Accattoli and Dal Lago [ADL12], however, shows that in the case of λ-calculus endowed with head reduction, the unitary cost model is polynomially invariant with respect to Turing machines. Dal Lago and I conjecture

that those invariance results can be extended to the $\lambda\mu$-calculus.[4]

In this section, we show that $\mathsf{BLLP}_{\lambda\mu}$ is polystep sound for another cost model, namely the one induced by de Groote's K, an abstract machine for the $\lambda\mu$-calculus. This will be done following a similar proof for PCF typed with linear dependent types [DLG12] and Krivine's Abstract Machine (of which K is a natural extension). The emphasis here is on terms which can be typed in $\mathsf{BLLP}_{\lambda\mu}$, so we anticipate some concepts which are introduced in Chapter 5.

Configurations of K are built around environments, closures and stacks, which are defined mutually recursively as follows:

- *Environments* are partial functions which makes λ-variables correspond to closures and μ-variables correspond to stacks; metavariables for environments are $\mathscr{E}, \mathscr{F}$, etc.;
- *Closures* are pairs whose first component is a $\lambda\mu$-term and whose second component is an environment; metavariables for closure are $\mathcal{C}, \mathcal{D}$, etc.
- *Stacks* are just finite sequences of closures; metavariables for stacks are $\mathscr{S}, \mathscr{T}$, etc.

Configurations are pairs whose first component is a closure and whose second component is a stack, and are indicated with C, D, etc. Reduction rules for configurations are in Figure 4.10. The K-machine is sound and complete with respect to head reduction [dG98], where however, reduction can take place in the scope of μ-abstractions, but not in the scope of λ-abstractions.[5]

Actually, $\mathsf{BLLP}_{\lambda\mu}$ can be turned into a type system for K's *configurations*. We closely follow Laurent [Lau03a] here. The next step is to assign a weight q^{π} to

[4]Subsequent study done by the author on the subject of abstract machines indeed leads in this direction. In particular the subterm property of head linear reduction has a similar variant for $\lambda\mu$.

[5]Dal Lago and I are aware of the work in [SR98], in which a Krivine machine for $\lambda\mu$ is derived semantically rather than syntactically (independently of [dG98]) and there is also a further extension of the machine which reduces under μ- and even λ-abstractions. The paper is not essential for the work described in this chapter since the machine of de Groote is enough to work with control operators. Still, in spite of some important differences w.r.t. our setting (the calculus considered is an untyped variant of Parigot's $\lambda\mu$), it might be worthwhile to investigate in the future.

$$
\begin{aligned}
((x, \mathscr{E}), \mathscr{S}) &\hookrightarrow (\mathscr{E}(x), \mathscr{S});\\
((\lambda x.t, \mathscr{E}), \mathcal{C} \cdot \mathscr{S}) &\hookrightarrow ((t, \mathscr{E}\{x/\mathcal{C}\}), \mathscr{S});\\
((tu, \mathscr{E}), \mathscr{S}) &\hookrightarrow ((t, \mathscr{E}), (u, \mathscr{E}) \cdot \mathscr{S});\\
((\mu\alpha.t, \mathscr{E}), \mathscr{S}) &\hookrightarrow ((t, \mathscr{E}\{\alpha/\mathscr{S}\}), \varepsilon);\\
(([\alpha]t, \mathscr{E}), \varepsilon) &\hookrightarrow ((t, \mathscr{E}), \mathscr{E}(\alpha)).
\end{aligned}
$$

Figure 4.10: K-machine Transitions.

every type derivation $\pi \triangleright C : \mathbf{N}$, similarly to what we have done in type derivations for *terms*. The idea then is to prove that the weight of (typable) configurations decreases at every transition step:

Lemma 4.14 *If $C \hookrightarrow D$, then $q^C \sqsupset q^D$.*

This allows to generalize polystep soundness to K:

Theorem 4.4 (Polystep Soundness for the K) *Let $\pi \triangleright \vdash C : \mathbf{N}$ and let $C \hookrightarrow^n D$. Then $n \leq q^\pi$.*

Please observe how Theorem 4.4 holds in particular when C is the initial configuration for a typable term t, i.e., $\langle\langle t, \emptyset\rangle, \varepsilon\rangle$.

4.7 Conclusions

In this chapter we have presented some evidence that the enrichment to Intuitionistic Linear Logic provided by Bounded Linear Logic is robust enough to be lifted to Polarized Linear Logic and the $\lambda\mu$-calculus. This paves the way towards a complexity-sensitive type system, which on the one hand guarantees that typable terms can be reduced to their normal forms in a number of reduction steps which can be read

from their type derivation, and on the other allows to naturally type useful control operators.

Many questions have been purposely left open here: in particular, the language of programs is the pure, constant-free, $\lambda\mu$-calculus, whereas the structure of types is minimal, not allowing any form of polymorphism. We expect that endowing BLLP with second order quantification[6] or $\mathsf{BLLP}_{\lambda\mu}$ with constants and recursion should not be particularly problematic, although laborious: the same extensions have already been considered in similar settings in the absence of control [GSS92, DLG12]. Actually, a particularly interesting direction would be to turn $\mathsf{BLLP}_{\lambda\mu}$ into a type system for Ong and Stewart's μPCF [OS97], this way extending the linear dependent paradigm to a language with control. This is of course outside the scope of this chapter, whose purpose was only to delineate the basic ingredients of the logic and the underlying type system.

As we stressed in the introduction, we are convinced this work is the first one giving a time complexity analysis methodology for a programming language with higher-order functions *and control*.[7] One could of course object that complexity analysis of $\lambda\mu$-terms could be performed by translating them into equivalent λ-terms, e.g. by way of a suitable CPS-transform [dG94a]. This, however, would force the programmer (or whomever doing complexity analysis) to deal with programs which are structurally different from the original one. And of course, translations could introduce inefficiencies, which are maybe harmless from a purely qualitative viewpoint, but which could make a difference for complexity analysis.

The type system for $\lambda\mu$ we obtained is somewhat sophisticated. In particular it necessarily needs to use a call-by-name evaluation strategy, since it is draws heavily on BLL (other commonly used light logics such as LLL or SLL do not appear to have a polarized variant[8]). Although call-by-name only evaluates needed redexes, its

[6] Building on syntactic isomorphisms discussed in [Lau02, §4.3].

[7] Tatsuta has investigated the maximum length of μ-reduction for a language *without* λ-*abstractions* (RTA 2007).

[8] As also remarked by D. Mazza (and confirmed by O. Laurent) during his lecture at the School of Linear Logic and GoI held in Turin last summer (`http://www.logoi.fr/events/school/`).

downside is that in general it evaluates redexes several times. Building on Krivine's work on storage operators [Kri90], which offer a way to force evaluation to happen exactly once, Girard defines a storage functional for BLL [GSS92, §5]. Even if the type of the corresponing proof is not polarized, it is conceivable that with a few modifications this approach could be adapted to BLLP.

Part III

Abstract Machines for λ and $\lambda\mu$.

Chapter 5

KAM

5.1 Introduction

Definition 5.1 *Let s, t, u be lambda terms and let $s\{x/t\}$ denote the usual capture-free substitution. Weak head reduction is the reflexive and transitive closure of the rewriting system shown in Figure 5.1.*

$$\frac{(\lambda x.s)t \to_\beta s\{x/t\}}{(\lambda x.s)t \to_{\mathsf{w}} s\{x/t\}} \qquad \frac{s \to_{\mathsf{w}} t}{(s)u \to_{\mathsf{w}} (t)u}$$

Figure 5.1: Weak head reduction rules.

Weak head reduction is a restricted form of β reduction which is *weak*, i.e., does not perform reduction within the scope of lambdas; and only reduces (if it exists) the *head* redex of the term.

Lemma 5.1 *Let t be a lambda term in weak head normal form. Then t is either a lambda abstraction or a variable applied to a certain number of terms (possibly zero). In particular, if t is closed then t is a lambda abstraction.*

Proof: Let t be a term which is not in the form $\lambda x.t_0$ nor $(x)t_1 \dots t_n$. Suppose t is normal and suppose that it is the smallest such term (i.e., there are no terms of smaller size which are also normal). Then, considering the rules in Figure 5.1, t must be an application $(s)u$ where s is normal. s cannot be an abstraction, otherwise t

would not be normal, thus it must be either a variable or an application. s cannot be a variable, otherwise t would be a term of the second form. Otherwise we have found a normal term s which is smaller than t (since it is a subterm), which is absurd. □

Interstingly, in spite of its simplicity, weak head reduction is powerful enough to evaluate terms representing data types [CHL96]:

> [...] weak reduction is powerful enough to evaluate (in normal order) normal forms of closed expressions of basic types in typed functional programming languages like ML or Miranda [...]

Remark 22 *Although here we are only interested in weak* head *reduction, in general weak reduction is not confluent. Special evaluation strategies such as call-by-name or call-by-value are needed to obtain a weak confluent calculus. Alternatively, it is necessary to work within a weak calculus with explicit substitutions (see also [ADL12, §3.1] and Section 6.2).*

Extending the reduction rules allowing the possiblity to reduce under lambda yields *strong* calculi.

Definition 5.2 *Let s, t be lambda terms and let x be a variable. Head reduction is the reflexive and transitive closure of the rewriting system in Figure 5.2.*

$$\frac{s \to_{\mathsf{w}} t}{s \to_{\mathsf{h}} t} \qquad\qquad \frac{s \to_{\mathsf{h}} t}{\lambda x.s \to_{\mathsf{h}} \lambda x.t}$$

Figure 5.2: Head Reduction rules.

Remark 23 *In the context of (strong) head reduction, using the usual syntax for variable names is not too satisfactory because of the need of α-conversion [CHL96]. For example (see also [Kes07]), in the following rule*

$$(\lambda y.t)[x/v] \to \lambda y.t[x/v] \tag{5.1}$$

it is assumed $x \neq y$ and $y \notin \mathsf{FV}(v)$. Nameless notation à la *De Bruijn, on the other hand, offers a clean way to deal with name clashes.*

Several mechanized systems for the evaluation of expressions have been considered during the last fifty years, the first example is Landin's SECD machine [Lan64] (so called because each state of the machine is identified by four components named stack, environment, control and dump). Here we describe the *Krivine Abstract Machine* [Kri07] (KAM) which computes the weak head normal form of a term. To be precise, the KAM does not perform weak head reduction but (weak) head *linear* reduction.

5.1.1 Head Linear Reduction

Head reduction (more generally, β reduction) involves the substitution of the argument in place of each occurrence of the corresponding variable. Of course, the variable can occur only once or many times (or not at all), so it is not reasonable (at least in principle) to consider head reduction as an operation that can be mechanized (since it does not appear to have a bounded cost). On the other hand, if we only substitute the argument in place of *one* occurrence, an operatorion called *head linear reduction* [DR04], then the cost of the reduction has a linear cost. Can we do any better?

Theorem 5.1 (Subterm property) *Linear head substitutions along a reduction $t \to_{\mathsf{hl}} u$ duplicates subterms of t only.*

Since duplicating a term amounts to creating a pointer to a subterm of an existsing term, which has a fixed size w.r.t. the size of the original term, then head linear reduction can be seen as an atomic operation of some reasonable machine.

Remark 24 *Note that the subterm property is not true for β-reduction, not even head reduction. A counter-example can be found in [DR04, p. 4]: let $t = (\lambda x.(x)(x)u)\lambda y.y \to_{\mathsf{h}} (\lambda y.y)(\lambda y'.y')u$, the subterm $(\lambda y'.y')u$ of the latter term is not a subterm of t.*

Recent work of Accattoli [Acc13a] (cf. also [ABM14]) shows that there is a similar property in a weak call-by-value setting.

5.2 A call-by-name lambda-calculus machine

The KAM is an abstract machine which performs transactions between states to compute the weak head normal form of a term.

De Bruijn notation Let us consider a tree representation for terms. For each (bound) variable occurrence we can reach its binder by just crossing as many λ nodes as necessary. This number is called the De Bruijn index. The definition can be extended also to free variables by stipulating that their De Bruijn number is equal to (or greater than) the number of lambda nodes between that occurrence and the root of the term (or even ∞). For the sake of briefness, we use the word "index" to mean "De Bruijn indexes".

Although using De Bruijn notation is a good way to implement the KAM it possible to do otherwise, as shown by Lippi [Lip02] [Lip07].

State representation The KAM works by doing a series of *transitions* between *states*, to be defined shortly.

Definition 5.3 *Let T be a term. A* closure *is a pair (T, ρ), where ρ is any sequence of closures.*

A finite sequence of closures is called an *environment* or a *stack*, depending on where it is used in the KAM. Each *state* of the machine is identified by a triple (T, ρ, σ), where ρ is an environment and σ is a stack.

Machine transitions The execution of the machine starts from the initial state $\langle T, (), \epsilon \rangle$. The possible transitions are as follows.

$$\langle x, [x_1 \mapsto (t_1, e_1), \ldots, x_n \mapsto (t_n, e_n)], \sigma \rangle \succ \langle t_i, e_i, \sigma \rangle, \qquad (x = x_i) \tag{5.2}$$

$$\langle \lambda x.t, e, c \cdot \sigma \rangle \succ \langle t, e[c/x], \sigma \rangle \tag{5.3}$$

$$\langle (u)v, e, \sigma \rangle \succ \langle u, e, (v, e) \cdot \sigma \rangle \tag{5.4}$$

Remark 25 *The transition rule for application duplicates the current environment. This is not truly needed, i.e., it is enough to use a pointer to the environment rather than actually duplicating it. Even without pointers, it would be smarter to duplicate only the part of the environment which corresponds to the variables that appear free in the subterm. That is, denoting $\rho|_t$ the restriction of ρ to the free variables of t:*

$$\langle (U)V, \rho, \sigma \rangle \succ \langle U, \rho|_U, (V, \rho|_V) \cdot \sigma \rangle \tag{5.5}$$

We use this simplification in examples without further comments.

Note that at any time it is not possible to apply two distinct transitions, since the left sides of the rules cannot match. The execution thus is deterministic. Also note the machine halts in two cases:

- when the current term is a variable z which does not appear in the environment (which corresponds to the case in which z is free);
- when the current term is an abstraction $\lambda x.U$ but there is no argument to pop from the stack.

This is not too surprising since the KAM implements weak head reduction and the only possible normal forms are variables applied to any number of arguments (possibly none) and abstractions.

Readback Once the execution has terminated as described above, how do we turn the final state of the machine into the normal form? This is formalized in [DR04] with the concept of *expansion*. Let $s = \langle T_s, \rho_s, \sigma_s \rangle$ be the final state, where $\sigma_s = ((T_1, \rho_1, \sigma_1), \ldots, (T_n, \rho_n, \sigma_n))$. The expansion of s is defined as

$$\mathsf{exp}(s) = [\![T_s]\!]_{\rho_s} [\![T_1]\!]_{\rho_1} \ldots [\![T_n]\!]_{\rho_n} \tag{5.6}$$

where the expansion of a term in an environment is defined as

$$[\![t]\!]_{[x_1\mapsto(t_1,e_1),\ldots,x_n\mapsto(t_n,e_n)]} = \begin{cases} [\![t_i]\!]_{\rho_i} & \text{if } x = x_i \\ x & \text{otherwise} \end{cases} \tag{5.7}$$

$$[\![(u)v]\!]_\rho = [\![u]\!]_\rho [\![v]\!]_\rho \tag{5.8}$$

$$[\![\lambda x.U]\!]_\rho = \lambda\bar{x}.[\![U]\!]_{(\bar{x},())::\rho} \tag{5.9}$$

where $\bar{x}$ is a fresh variable name.

Some explainations are in order. If s is a final state, what is the meaning of its expansion? The weak head normal form can only be an abstraction or a variable applied to some arguments.

- Suppose we are in the first case: this means that when the machine halts the stack σ_s is empty. Thus in Equation (5.6) $n = 0$.
- Suppose instead we are in the second case: so the machine halts because it finds a variable z which is free in the environment. Thus in Equation (5.6) $[\![T_s]\!]_{\rho_s} = [\![z]\!]_{\rho_s} = z$.

Correctness Let t and t' be respectively $\mathsf{exp}(s)$ and $\mathsf{exp}(s')$. Suppose that the KAM goes from s to s', then $t \to_{\mathsf{w}}^* t'$. The proof is rather straightforward (basically case analysis on s and calculations), but tedious.

Completeness If $t \to_{\mathsf{w}}^* t'$ then starting the KAM with the state $s = (t, (), ())$ then it arrives at a state s' s.t. $\mathsf{exp}(s') = t'$. **Proof:** We can assume w.l.o.g. that $t \to_{\mathsf{w}} t'$ (the general result is immediate). Let us proceed by induction on the structure of t. We have to consider two cases, corresponding to the rules in Figure 5.1.

- Suppose $t = (\lambda x.u)v$, then $s = \langle(\lambda x.u)v, (), \epsilon\rangle \succ \langle\lambda x.u, (), (v, ()) \cdot \epsilon\rangle \succ \langle u, (v, ((\))) \cdot ((\)), \epsilon\rangle$. Let s' be the latter state. From the rules for the expansion of a term in an environment, we easily see that the only thing that changes are the variables which are in the environment (in this case x would be replaced by v and nothing else). Hence the term we obtain is $u\{x/v\} = t'$.

- Suppose instead $t = (u)v$, where $u \to_{\mathsf{w}} u'$. Hence $s = \langle (u)v, (), \epsilon \rangle$. Let s_u be $\langle u, (), \epsilon \rangle$, by induction hypothesis we know that $s_u \succ^* s'_u$, where $\mathsf{exp}(s'_u) = u'$. Suppose now we take $\bar{s_u} = \langle u, (), (v, ()) \cdot \epsilon \rangle$, then the KAM goes into $\bar{s'_u}$ s.t. $\mathsf{exp}(\bar{s'_u}) = \mathsf{exp}(s'_u) = u'$ (since the element we add is deeper in the stack than the ones needed to go from u to u'). Let $s = ((u)v, (), ())$, trivially $s \succ \bar{s_u}$. Thus $s \succ s' = \bar{s'_u}$.

□

5.3 μKAM

De Groote [dG98] considers an extension of the KAM to $\lambda\mu$-calculus (see Section 1.3.3). More precisely, the machine is able to compute the $\beta\beta^\circ\mu\rho\epsilon$ normal form of a $\lambda\mu\epsilon$-term.

Weak head normal form We need to define what we mean for weak head normal form in the context $\lambda\mu$-calculus. If we reason by analogy with lambda calculus, we would guess we need to forbid reduction under μ. However, that would be wrong since when we contract a μ-redex the binder does not appear.

Definition 5.4 *Let s, t, u be lambda mu terms and let $t[\alpha := N]$ denote the usual structural substitution. Weak head reduction is the reflexive and transitive closure of the rewriting system obtained adding to the rules in Figure 5.1 those in Figure 5.3, which we still denote $\to_{\mathsf{w}}$.*

Such a notion of reduction involves an appropriate modification of the De Bruijn indexes. For simplicity here we consider only the version with variable names.

Transition rules Using the traditional syntax form terms, environment are not simply sequence of closures but rather finite maps from the set of variables to closures/stacks (depending on whether the variable is a λ variable or a μ variable). Aside from that, the basic definitions are essentially the same of the KAM. De

$$\frac{(\mu\alpha.s)t \to_\mu s[\alpha := \lambda f.[\alpha](f)t]}{(\mu\alpha.s)t \to_{\mathsf{w}} s[\alpha := \lambda f.[\alpha](f)t]} \qquad \frac{\mu\alpha.[\beta](\mu\gamma.s) \to_\rho \mu\alpha.s[\gamma := \lambda f.[\beta]f]}{\mu\alpha.[\beta](\mu\gamma.s) \to_{\mathsf{w}} \mu\alpha.s[\gamma := \lambda f.[\beta]f]}$$

$$\frac{\mu\alpha.\mu\beta.s \to_\epsilon \mu\alpha.\mu\beta.s[\beta := \lambda f.f]}{\mu\alpha.\mu\beta.s \to_{\mathsf{w}} \mu\alpha.\mu\beta.s[\beta := \lambda f.f]} \qquad \frac{\mu\alpha.[\alpha]t \to_\theta t, \quad \alpha \notin \mathsf{FV}(t)}{\mu\alpha.[\alpha]t \to_{\mathsf{w}} t}$$

$$\frac{s \to_{\mathsf{w}} t}{\mu\alpha.s \to_{\mathsf{w}} \mu\alpha.t} \qquad \frac{s \to_{\mathsf{w}} t}{[\alpha]s \to_{\mathsf{w}} [\alpha]t}$$

Figure 5.3: Weak head reduction rules for $\lambda\mu$.

Groote uses the word *dump* to mean "state", except that (in the preliminary definition) he includes a boolean to store some contextual information. For an instance, this extra information allows to distinguish the case of ρ reduction from the case of ϵ-reduction.

$$\langle x, [x_1 \mapsto (t_1, e_1), \ldots, x_n \mapsto (t_n, e_n)], \sigma\rangle \succ \langle t_i, e_i, \sigma\rangle, \qquad (x = x_i) \tag{5.10}$$

$$\langle \lambda x.t, e, c \cdot \sigma\rangle \succ \langle t, e[c/x], \sigma\rangle \tag{5.11}$$

$$\langle (u)v, e, \sigma\rangle \succ \langle u, e, (v, e) \cdot \sigma\rangle \tag{5.12}$$

$$\langle \mu\alpha.t, e, \sigma\rangle \succ \langle t, e[\alpha \mapsto \sigma], \epsilon\rangle \tag{5.13}$$

$$\langle [\alpha]t, e, \epsilon\rangle \succ \langle t, e, e(\alpha)\rangle \tag{5.14}$$

Readback Analogously to the case of lambda calculus, one can define a function to read the resulting term from the state of the machine. De Groote calls it the *unloading function*.

Example We describe here a simple run of the μKAM starting from the term $(\texttt{callcc})\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1} = (\lambda x.\mu\alpha.[\alpha](x)\lambda y.\mu\beta.[\alpha]y)\ldots$.

$$\langle(\texttt{callcc})\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, (), \epsilon\rangle \tag{5.15a}$$

$$\langle\texttt{callcc}, (), (\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, ()) \cdot \epsilon\rangle \tag{5.15b}$$

$$\langle\mu\alpha.[\alpha](x)\lambda y.\mu\beta.[\alpha]y, [x \mapsto (\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, ())], \epsilon\rangle \tag{5.15c}$$

$$\langle[\alpha](x)\lambda y.\mu\beta.[\alpha]y, [x \mapsto (\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, ()), \alpha \mapsto \epsilon], \epsilon\rangle \tag{5.15d}$$

$$\langle(x)\lambda y.\mu\beta.[\alpha]y, [x \mapsto (\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, ()), \alpha \mapsto \epsilon], \epsilon\rangle \tag{5.15e}$$

$$\langle x, [x \mapsto (\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, ())], (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon]) \cdot \epsilon\rangle \tag{5.15f}$$

$$\langle\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1}, (), (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon]) \cdot \epsilon\rangle \tag{5.15g}$$

$$\langle(\mathsf{or})\mathbf{0}(k)\mathbf{1}, [k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])], \epsilon\rangle \tag{5.15h}$$

$$\langle(\lambda a.\lambda b.(a)\mathbf{1}b)\mathbf{0}, (), ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15i}$$

$$\langle\lambda a.\lambda b.(a)\mathbf{1}b, (), (\mathbf{0}, ()) \cdot ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15j}$$

$$\langle\lambda b.(a)\mathbf{1}b, [a \mapsto (\mathbf{0}, ())], ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15k}$$

$$\langle(a)\mathbf{1}b, [a \mapsto (\mathbf{0}, ()), b \mapsto ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])], \epsilon\rangle \tag{5.15l}$$

$$\langle(a)\mathbf{1}, [a \mapsto (\mathbf{0}, ())], ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15m}$$

$$\langle a, [a \mapsto (\mathbf{0}, ())], (\mathbf{0}, ()) \cdot ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15n}$$

$$\langle\mathbf{0}, (), (\mathbf{0}, ()) \cdot ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15o}$$

$$\langle\lambda d.d, [c \mapsto (\mathbf{0}, ())], ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])]) \cdot \epsilon\rangle \tag{5.15p}$$

$$\langle d, [c \mapsto (\mathbf{0}, ()), d \mapsto ((k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])])], \epsilon\rangle \tag{5.15q}$$

$$\langle(k)\mathbf{1}, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])])], \epsilon\rangle \tag{5.15r}$$

$$\langle k, [\alpha \mapsto \epsilon, k \mapsto (\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon])])], (\mathbf{1}, ()) \cdot \epsilon\rangle \tag{5.15s}$$

$$\langle\lambda y.\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon], (\mathbf{1}, ()) \cdot \epsilon\rangle \tag{5.15t}$$

$$\langle\mu\beta.[\alpha]y, [\alpha \mapsto \epsilon, y \mapsto (\mathbf{1}, ())], \epsilon\rangle \tag{5.15u}$$

$$\langle[\alpha]y, [\alpha \mapsto \epsilon, y \mapsto (\mathbf{1}, ()), \beta \mapsto \epsilon], \epsilon\rangle \tag{5.15v}$$

$$\langle y, [\alpha \mapsto \epsilon, y \mapsto (\mathbf{1}, ()), \beta \mapsto \epsilon], \epsilon\rangle \tag{5.15w}$$

$$\langle\mathbf{1}, (), \epsilon\rangle \tag{5.15x}$$

Remark 26 *It is interesting to note that to extract the intuitionistic answer it is not enough to use β and μ reducion rules. It is necessary to use also auxiliary reductions.*

$$
\begin{aligned}
(\texttt{callcc})\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1} &\to_\beta \mu\alpha.[\alpha](\lambda k.(\mathsf{or})\mathbf{0}(k)\mathbf{1})\lambda y.\mu\beta.[\alpha]y && (5.16)\\
&\to_\beta \mu\alpha.[\alpha](\mathsf{or})\mathbf{0}(\lambda y.\mu\beta.[\alpha]y)\mathbf{1} && (5.17)\\
&\to_\beta \mu\alpha.[\alpha](\lambda b.b)(\lambda y.\mu\beta.[\alpha]y)\mathbf{1} && (5.18)\\
&\to_\beta \mu\alpha.[\alpha](\lambda y.\mu\beta.[\alpha]y)\mathbf{1} && (5.19)\\
&\to_\beta \mu\alpha.[\alpha]\mu\beta.[\alpha]\mathbf{1} && (5.20)\\
&\to_\rho \mu\alpha.[\alpha]\mathbf{1} && (5.21)\\
&\to_\theta \mathbf{1} && (5.22)
\end{aligned}
$$

5.4 Other variants of the KAM

Control operator Curien and Herbelin extends the Krivine machine with Felleisen's $\mathcal{C}$ operator. Then they show how, depending on whether call-by-name or call-by-value is used, the machine can be extended with new operators μ and $\tilde{\mu}$, which combined result in $\bar{\lambda}\mu\tilde{\mu}$ calculus.

Strong reduction Crégut [Cré07] and De Carvalho [DC09] have studied how to extend the Krivine machine to reduce under lambda and to obtain a full β-normal form.

Call-by-need A lazy call-by-need variant of the KAM known as Sestoft's machine [Ses97] is further improved in [FGSW07], both in terms of time and space. In the paper the authors compare four variants of the machine, implemented in Scheme, on a series of benchmark (just unprocessed closed lambda terms).

Chapter 6

PAM

6.1 Introduction

The *Pointer Abstract Machine* (a.k.a. PAM) is a variant of the KAM in which closures are never built, but rather the value of the closure is fetched when needed jumping back to the appropriate "move"[1] using *pointers*. This avoids creating closures which are never used and is thus efficient in many cases. The machine is inspired by Game Semantics [HO00], so it was originally conceived for (simply) typed lambda calculus [DHR96], but it has been studied also in the setting of untyped lambda calculus [DR04]. The main contribution of this chapter are: first and foremost a clearer explanation of the PAM, then some possible optimizations, and finally its extension to $\lambda\mu$-calculus (which first requires extending head linear reduction to $\lambda\mu$-calculus).

Unfortunately, although the PAM has been known since 1988, so far there have been few works in literature about the PAM (at least compared to the KAM); as for complexity in particular, besides *folklore* (e.g., the authors of [DR04] cite a private communication with Herbelin, who has worked with Curien on some variants of the PAM [CH96]), there is not that much. Thus the first contribution of this chapter is to give a clear explanation of the PAM itself.

[1] Simply put, a point of the "execution trace".

6.2 A more operationally-oriented approach

In spite of the strong connection with games, here we do not want to commit ourself to typed lambda calculus. Our treatment of the PAM is entirely general and the optimizations we propose can be used even working with terms which are not typable (thus we exploit no typing information, we cannot even assume, e.g., that all occurrence of the same variable have the same arity) although in the future it would be interesting to study their relevance on terms typable in BLLP. In the following we work using terms which respect the so-called Barendregt's convention, i.e., each binder has a unique name (to avoid variable captures). Another common assumption, especially if one want to work using De Bruijn nameless notation [DB72], is to work only with closed terms. Some of the terminology is inspired by Game Semantics and we adopt it here without considering explicitly the connection with Game Semantics.

Explicit Substitutions Some (functional) programming languages include a kind of `let` operator which allows to write expressions like `let` $x = u$ `in` t. The meaning of the previous expression is "what you would get replacing x with u in t", i.e., the denoted substitution is "frozen" (it differs from the meta-substitution $t\{u/x\}$). This kind of `let`-expression, a term with an *explicit substitution*, is denoted $t[u/x]$, to distinguish it from usual substitution. Actually, in the context of explicit substitutions, the order is reversed so typically we find respectively $t\{x/u\}$ and $t[x/u]$, instead. As pointed out in [CMM10], explicit substitutions are related to commutative cuts.

The simplest ES calculus is $\lambda\mathsf{x}$, which is the basis of most calculi in literature. Even though lambda calculus is confluent, explicit substitution calculi may contain critical pairs [Mel95], as shown by the following example (often called *Melliès critical pair*):

$$t[y/v][x/u[y/v]]^* \leftarrow ((\lambda x.t)u)[y/v] \rightarrow^* t[x/u][y/v]. \tag{6.1}$$

It is also possible that strong normalization (in the simply-typed case) is lost when moving to an ES calculus. Otherwise the calculus is said to have the *strong normalization preservation* property (PSN).

$$
\begin{array}{lll}
(\lambda x.t)u & \to t[x/u] & \\
x[x/u] & \to u & \\
y[x/u] & \to y & (x \neq y) \\
((t_1)t_2)[x/u] & \to (t_1[x/u])t_2[x/u] & \\
(\lambda y.v)[x/u] & \to \lambda y.v[x/u] &
\end{array}
$$

Figure 6.1: Rules of $\lambda\mathsf{x}$.

Both phenomena arise when $\lambda\mathsf{x}$ is enriched with more rewriting rules involving substitutions, but there are ways to avoid Melliès counter-example and recover the PSN property. See [Kes07] for a survey on this topic (cf. also [CHL96] [KL07] [AK12]) and [ABM14] for an analysis of the relation between some abstract machines[2] and the corresponding explicit substitution calculi.

6.2.1 Introduction

Head occurrence The hoc of a term is the leftmost variable occurrence. It can be defined in terms of head contexts.

Definition 6.1 (Head context) *An head-context (or simply H-context) is defined by the following grammar.*

$$H[\ ] = [\ ] \mid H[\ ]t \mid \lambda x.H[\ ]. \tag{6.2}$$

Lemma 6.1 *For each term t there exist (and are uniquely determined) a variable x and a H-context H s.t.*

$$t = H[x]. \tag{6.3}$$

[2]The only call-by-name abstract machine considered in a previous version of the paper was the KAM; latest version also considers the MAM (Milner's Abstract Machine, a variant of the KAM which uses only a global environment and has no concept of closure). The PAM is not discussed.

Proof: By induction on the structure of t.

- If $t = y$ then $x = y$ and $H[\ ] = [\ ]$.
- If $t = (u)v$ then by induction hypothesis there exist y and H_u s.t. $u = H_u[y]$. Then $x = y$ and $H = H_u[\ ]v$.
- If $t = \lambda z.u$ then by induction hypothesis there exist y and H_u s.t. $u = H_u[y]$. Then $x = y$ and $H = \lambda z.H_u[\ ]$.

□

Thus any term t can be written as $H[x]$. The occurrence of the variable x is called the *head occurrence* (hoc) and H is said to be the *maximal* head context of t.

Prime redexes Let t be a term, a node of its syntax tree is a *spine node* if it is the root node, or it is the child (resp. left child) of a λ (resp. @ node) which is also a spine node. A spine node which is also a variable (resp. application, abstraction) node is called a *spine variable node* (resp. *spine application node*, *spine abstraction node*).

A term u is said to be an argument subterm of t if the syntax tree of u is a subtree of the syntax tree of t which is on the right of an application node.

An argument subterm of a term t will be called a *spine argument* if it is the argument of some spine application node of t. Let z be an occurrence (of variable) in t, t_z denotes the maximal subterm of t whose hoc is z.

Definition 6.2 *The* head λ list $\lambda_h(t)$ *of* t *is a sequence* $(\lambda x_1, \ldots, \lambda x_n)$ *of abstraction subterms of* t*; the* prime redexes *are pairs* $(\lambda x, A)$ *where* λx *is a binder of* t *and* A *is an argument subterm of* t*, called the* argument of the prime redex*. Both are defined by induction on* t*:*

- *if* t *is a variable then* $\lambda_h(t)$ *is the empty sequence and* t *has no prime redexes;*

- *if* $t = (u)v$ *then either* $\lambda_h(u)$ *is empty in which case* $\lambda_h(t)$ *is defined to be empty and the prime redexes of* t *are those of* u*; or* $\lambda_h(u)$ *begins with* λx*, that is* $\lambda_h(u) = (\lambda x) \cdot l$*, and we set* $\lambda_h(t) = l$ *and define the prime redexes of* t *to be those of* u *with the addition of* $(\lambda x, v)$*;*

- *if* $t = \lambda x.u$ *then* $\lambda_h(t) = (\lambda x) \cdot \lambda_h(u)$ *and the prime redexes of* t *are those of* u*.*

Suppose the hoc of t *is an occurrence of the (bound) variable* x*; then the prime redex* $(\lambda x, v)$ *if it exists is called the* hoc redex *of* T*.*

Remark 27 *Head* λ *list is a somewhat confusing name and calls for an explanation. Let* t *be a term. If* t *is written using explicit substitutions*

$$t = \lambda y_1 \dots \lambda y_m.(x)u_1 \dots u_p[x_1/v_1] \dots [x_n/v_n],$$

or alternatively, if t *is in* σ*-equivalent to a* canonical form

$$t \equiv_\sigma \lambda y_1 \dots \lambda y_m.(\lambda x_1 \dots (\lambda x_n.(x)u_1 \dots u_p)v_n \dots v_1),$$

then the head λ *list is* exactly *the list of the "leading binders" of the term.*

$$\lambda_h(t) = (\lambda y_1) \cdot \dots (\lambda y_m)$$

For example, let t *be* $(\lambda x \lambda y \lambda z.z)uv$*: it has no leading* λ*s* per se*, but if we explicit the substitutions we get* $\lambda z.z[x/u][y/v]$ *where the leading binder* λz *is exactly the only element of* $\lambda_h(t)$*.*

Definition 6.3 *Let* t *be a term. If* t *is written using explicit substitutions*

$$t = \lambda y_1 \dots \lambda y_m.(x)u_1 \dots u_p[x_1/v_1] \dots [x_n/v_n],$$

or alternatively, if t *is in* σ*-equivalent to a canonical form* t_σ

$$t_\sigma = \lambda y_1 \dots \lambda y_m.(\lambda x_1 \dots (\lambda x_n.(x)u_1 \dots u_p)v_n \dots v_1),$$

then we say t *is in* quasi head normal form *iff* $x \notin \{x_1, \dots, x_n\}$*.*

State A *pointed sequence* of the term t is a finite sequence of *moves*: $((p_0, z_0, A_0), \ldots, (p_n, z_n, A_n))$ where $p_0 = 0$, $z_0 \notin t = A_0$ and for each $i > 0$ we have: $1 \leq p_i \leq i$, z_i is an occurrence of variable in t, A_i is an argument subterm of t. A PAM *state* for t is a pointed sequence $((p_1, z_1, A_1), \ldots, (p_n, z_n, A_n))$ satisfying:

- z_1 is the hoc of t and for each $i > 1$, z_i is the hoc of A_{i-1};
- A_i is *justified* by z_{p_i}, that is, A_i is a spine argument of t_z.

Again, we have to keep in mind the underlying idea of the PAM. Avoid building closures, use pointers instead.

Transitions Let t a term not in quasi head normal form and s a PAM state for t. The initial move is $(1, z_1, A_1)$, where z_1 is the hoc of t and A_1 is the argument of the hoc redex. In general, how do we build the new move $(p_{n+1}, z_{n+1}, A_{n+1})$? We start by recording the z_{n+1}, which is simply the hoc of A_n. Then we need to find its argument and note the pointer to the right move of the run. Let x the variable name of which z_{n+1} is an occurrence.

- If x is free no transition is defined.
- If x is bound then:
  - if λx has a corresponding argument u we set $A_n = u$;
  - otherwise execution halts.

Let us explain this process more in details.

6.2.2 How and why

Simply put, we want to compute the pointed sequence of a term t (starting from the initial move $\langle 0; z_0; t \rangle_{\mathsf{P}}$). Operationally, this amounts to building a table with numbered rows (where the initial move is number zero), and jumping using pointers means going back to the appropriate line.

Head occurrence Determining the hoc of a term is a pretty straightforward operation, as it is clear from Lemma 6.1. The statement can be generalized to work with explicit substitutions.

Corollary 6.1 *For each term t there exist (and are uniquely determined) a variable x, a H-context H and a sequence of pairs $(\lambda x_1, t_1), \dots (\lambda x_n, t_n)$ s.t.*

$$t = H[x][x_1/t_1] \dots [x_n/t_n][x_1/t_1] \dots [x_n/t_n]. \tag{6.4}$$

Proof: Straightforward from Lemma 6.1 and Remark 27. Note that, by the Barendregt's condition, there is no possible name clash between the usual binders and the λs in the explicit substitutions. □

Binder Given the hoc of A_n, we must determine if and where it is bound in the moves of the current state of the PAM run. To this end we follow Algorithm 1

Algorithm 1 Find binder (and relative rank) of z_{n+1}

Require: $n \geq 0$ {index of current move}
Require: z_{n+1} {hoc of A_n}
1: $k \Leftarrow n$
2: **if** (z_{n+1} is bound in A_k) is true **then**
3: $r \Leftarrow$ rank of λz in $\lambda_h(A_k)$
4: **return** (k, r) {binder has been found}
5: **else if** $k > 0$ **then**
6: $k \Leftarrow p_k - 1$ {jump back using the pointer[3]}
7: **go to** 2 {keep looking for the binder}
8: **else**
9: **return** $\perp$ {binder not found}
10: **end if**

[3]The index in the pointed sequence.

Remark 28 *If we find a binder, how can we be sure the binder we find is the right one? The PAM does not make use of α-conversion, so there can actually be several copies of the same name. The point is that the algorithm finds, if it exists, the closest copy of that name (basically the line number is implicitly used to disambiguate names). Indeed, that copy of the name has a smaller scope w.r.t. the original, which is essentially* shadowed *within that scope.*

Before proceeding further, we first note the *rank* r of λz, that is the position of λz in the head λ list of A_k.

Argument Assuming we have found a binder λz, we must now determine if there is a matching argument and which is it.

Lemma 6.2 *If λy is a spine λ of u then: either it is a head λ of u; or λy forms a primary redex $(\lambda y, v)$, where v is a spine argument of u.*

Proof: u can be written in a unique way as $H[x][x_1/t_1]\dots[x_n/t_n]$ (cf. Corollary 6.1). □

If λz is not a head λ of A_k then it forms a prime redex $(\lambda z, A)$, thus the matching argument A is in A_k itself.

If λz is a head λ of A_k, we proceed as described in Algorithm 2. Note that, by Remark 27, it is crucial that we only consider head λs to update the rank.

Pointer To understand the PAM is to understand pointers. Which means understanding first how they are *computed*, and then how they are *employed*.

1. If we have found a matching argument A for λz on the same line of its binder we have to record a pointer to the line $j+1$, i.e., where we find the hoc z_{j+1} of A_j s.t. A is an argument subterm of A_j.

2. Suppose we have not found the argument on the same line of the binder but on a previous line, instead. Then we have to record a pointer to the line j, i.e., where we find the hoc z_j of A_{j-1} s.t. A is an argument subterm of A_{j-1}.

Algorithm 2 Finding the argument of the binder.

Require: $n \geq 0$ {index of current move}

Require: λz {binder of current hoc}

Require: A_k {term where binder λz has been found}

Require: r {rank of the binder λz in $\lambda_h(A_k)$}

1: $j \Leftarrow k$

2: $a \Leftarrow$ number of arguments of z_j {compute arity of z_j}

3: **if** $r \leq a$ **then**

4: **return** r-th argument of z_j {argument has been found}

5: **else if** $j > 0$ **then**

6: $l \Leftarrow |\lambda_h(A_{j-1})|$

7: $r \Leftarrow r - a + l$

8: $j \Leftarrow j - 1$ {go back sequentially}

9: **go to** 2 {keep looking for the argument}

10: **else**

11: **return** $\perp$ {argument not found}

12: **end if**

Overall, the number we write down is simply the successor of the number of the line where the argument has been found. This concludes the calculation of p_{n+1}.

Remark 29 *The first case happens if A_j is not in head normal form. So either $j = 0$ (i.e., we are considering a variable bound in a primary redex of t); or if A_n is not in head normal form (i.e., t contains an inner redex).*

Remark 30 *Here it is important that the terms we work with are in canonical form w.r.t. σ-equivalence. Otherwise it is non-trivial to determine if a λ is paired with an argument subterm and which one. So it is likewise non-trivial to determine the rank of a binder.*

One way to greatly simplify this process is to work with explicit substitutions: if A_j has the form $H[x][x_1/t_1]\dots[x_n/t_n]$ of Corollary 6.1 we can rewrite it as $\lambda y_1 \dots \lambda y_m.(x)s_1 \dots s_l[x_1/t_1]\dots[x_n/t_n]$ (cf. Section 6.3), where we can easily read out if a binder has a matching argument (we just need to check if it is in a substitution) or otherwise (if it not *in a substitution), what is its rank (we simply look at its position in the list of leading binders).*

Was not the point of the PAM to avoid *building closures? Do we ultimately still need to build to build them? Not really. Suppose the term we want to evaluate is $(f)v_1 \dots v_r$, where f is "fixed". Even if we do not assume f is normal (not even head-normal) we can still explicit its primary redexes. This operation can be done* statically*, i.e., once for all, before the arguments $v_1, \dots, v_r$ are available (but it is not needed if f is normal). Cf. also Section 6.4.*

What is the use of pointers? A pointer essentially determines the line where a certain argument has been found. In general that argument contains some free variables, so we need to know where to find a certain binder. Thanks to the subterm property (cf. Section 5.1.1), the arguments is a subterm of some existing term in the state of the PAM, which is exactly where we need to look for the binder. If the term we jump to does not contain the binder, we need to look in the term where it has been taken from, and so on. Iterating these pointers-guided backwards jumps we

ultimately either find the binder or (if the variable is free in t) we jump all the way to the initial line.

We continue this section with a classical example from [DR04].

Example 6.1 Let $T = (\delta)\delta = (\lambda x.(x)x)\lambda y.y$. Let us study a partial run of this term. For a better understanding we do in parallel the calculation using explicit substitutions. Following [ABM14, §3] we denote $\multimap_m$ and $\multimap_e$ the two rewriting relations of the calculus (they correspond essentially to the first and second rules of Figure 6.1, respectively). Note that whenever a $\multimap_e$ rule involves a term with binders we need α-renaming, so we add an index to distinguish names.

ES	PAM	#
$(\lambda x.(x)x)\lambda y.(y)y$	$\langle 0; z_0; (\lambda x.(x)x)\lambda y.(y)y)\rangle_{\mathsf{P}}$	0
$\multimap_m (x)x[x/\lambda y.(y)y]$ $\multimap_e (\lambda y^1.(y^1)y^1)x[x/\lambda y.(y)y]$	$\langle 1; x_1; \lambda y.(y)y)\rangle_{\mathsf{P}}$	1
$\multimap_m (y^1)y^1[y^1/x][x/\lambda y.(y)y]$ $\multimap_e (x)y^1[y^1/x][x/\lambda y.(y)y]$	$\langle 1; y_1; x_2\rangle_{\mathsf{P}}$	2
$\multimap_e (\lambda y^2.(y^2)y^2)y^1[y^1/x][x/\lambda y.(y)y]$	$\langle 1; x_2; \lambda y.(y)y\rangle_{\mathsf{P}}$	3
$\multimap_m (y^2)y^2[y^2/y^1][y^1/x][x/\lambda y.(y)y]$ $\multimap_e (y^1)y^2[y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 2; y_1; y_2\rangle_{\mathsf{P}}$	4
$\multimap_e (x)y^2[y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; y_2; x_2\rangle_{\mathsf{P}}$	5
$\multimap_e (\lambda y^3.(y^3)y^3)y^2[y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; x_2; \lambda y.(y)y\rangle_{\mathsf{P}}$	6
$\multimap_m (y^3)y^3[y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$ $\multimap_e (y^2)y^3[y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 4; y_1; y_2\rangle_{\mathsf{P}}$	7
$\multimap_e (y^1)y^3[y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 2; y_2; y_2\rangle_{\mathsf{P}}$	8
$\multimap_e (x)y^3[y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; y_2; x_2\rangle_{\mathsf{P}}$	9
$\multimap_e (\lambda y^4.(y^4)y^4)y^3[y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; x_2; \lambda y.(y)y\rangle_{\mathsf{P}}$	10
$\multimap_m (y^4)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$ $\multimap_e (y^3)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 7; y_1; y_2\rangle_{\mathsf{P}}$	11
$\multimap_e (y^2)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 4; y_2; y_2\rangle_{\mathsf{P}}$	12

$\multimap_e (y^1)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 2; y_2; y_2\rangle_{\mathsf{P}}$	13
$\multimap_e (x)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; y_2; x_2\rangle_{\mathsf{P}}$	14
$\multimap_e (\lambda y^5.(y^5)y^5)y^4[y^4/y^3][y^3/y^2][y^2/y^1][y^1/x][x/\lambda y.(y)y]$	$\langle 1; x_2; \lambda y.(y)y\rangle_{\mathsf{P}}$	15

□

As we can see from the previous example, a step of the PAM correspond exactly to one $\multimap_e$ step in the calculus with explicit substitutions. A certain number of $\multimap_m$ steps, up to the maximum length of the head λ list of the term and its spine arguments, may also be implicitly involved.

To be or not to be Is the evaluation strategy corresponding to the PAM the (strong) head reduction or not? To put it more directly, does the PAM reduce inside abstractions? The short answer is "Yes". Another answer is "Yes, if we want to". Applying blindly the mechanism described produces the head normal form of a term (not its *weak* hnf). Nevertheless, if we only want the weak head normal form it is also possible. Simply put, before doing the substitution we must check if there is a non-matching lambda. This can done either on the fly or not. We go back to the last line in which there are head lambda and check if they have all been paired between that line and the current one.

6.3 Towards a better understanding of the PAM

A *head context*, or *H-context*, is a context in which the hole appears in the head position of t, namely

$$H = \Box \mid \lambda x.H \mid Ht$$

A head context of a term t is any H-context H s.t. $t = H[s]$, and we say that s is in head position in t. In particular, there is an unique head context of t, the *maximal head context* and an unique variable x, the *head variable* of t, s.t. $t = H[x]$. The *spine* of the term $t = H[x]$ and of its head context H is the sequence of applications and λ-abstractions in head position ordered from the root of t (the root of H) to

its head variable (hole). A λ-abstraction or an application in head position in t are said a spine λ-abstraction or a spine application of t, respectively. The variable x bound by a spine abstraction is a *spine variable*, while the right subterm of a spine application is a *spine argument* of t. By $SV(t)$ and $SV(H)$ we denote the set of the spine variables of a term t and of a H-context H.

The *spine string* of a term t, or of a context H, is the string $t^{\$}$, or $H^{\$}$, obtained by associating an indexed symbol λ_x to every spine λ-abstraction λx, and an indexed symbol $@_v$ to every spine application uv, namely

$$x^{\$} = \Box^{\$} = \epsilon \qquad (\lambda x.t)^{\$} = \lambda_x t^{\$} \qquad (st)^{\$} = @_t s^{\$} \tag{6.5}$$

From which, it is readily seen that $H_1[H_2]^{\$} = H_1^{\$} H_2^{\$}$.

Lemma 6.3 $(H_1[H_2])^{\$} = H_1^{\$} H_2^{\$}$.

Proof: By induction on H_1.

- If $H_1[\ ] = [\ ]$ then $(H_1[H_2])^{\$} = [H_2]^{\$} = H_2^{\$} = \epsilon H_2^{\$} = H_1^{\$} H_2^{\$}$.
- If $H_1[\ ] = (H[\ ])t$ then $(H_1[H_2])^{\$} = @_t(H[H_2])^{\$}$. By induction hypothesis the latter is $@_t H^{\$} H_2^{\$} = (@_t H^{\$}) H_2^{\$} = H_1^{\$} H_2^{\$}$.
- If $H_1[\ ] = \lambda x.H$ then $(H_1[H_2])^{\$} = \lambda_x(H[H_2])^{\$}$. By induction hypothesis the latter is $\lambda_x H^{\$} H_2^{\$} = (\lambda_x H)^{\$} H_2^{\$} = H_1^{\$} H_2^{\$}$.

□

An environment context, or *E-context*, is a particular H-context in which spine λ-abstractions and spine applications are balanced. More precisely, every spine λ-abstraction matches with a spine applications forming a λ-redex at distance. E-contexts are defined by the following grammar:

$$E_{1,2} := \Box \mid E_1[\lambda x.E_2]t \tag{6.6}$$

In terms of spine strings, the definition of E-context corresponds to a grammar of well-balanced bracket pairs

$$\Box^{\$} = \epsilon \qquad (E_1[\lambda x.E_2]t)^{\$} = @_t E_1^{\$} \lambda_x E_2^{\$}$$

where @ is the open bracket and λ the closed bracket. As a consequence, since this language is non-ambiguous, there is a bijection between the indexed symbols λ_x and $@_t$, which also induces a bijection between the spine variables and the spine arguments in E. We have then a natural correspondence between E-contexts and environments.

An *environment* $\eta = t_1/x_1, \ldots, t_k/x_k$ is an ordered sequence of variable substitutions t_i/x_i (where t_i is a term replacing the variable x_i). By $t[\eta]$ we denote the term obtained by applying the usual λ-calculus substitution corresponding to η, i.e., $t_0[t_1/x_1, \ldots, t_k/x_k] = t_0[t_1/x_1] \ldots [t_k/x_k]$ (in other words, for $i < j$, the occurrences of x_j in the terms t_i are replaced by the term t_j, while this is not the case for any occurrence of x_j in a term t_k with $k \geq j$). Given an E-context E, let us denote by $\eta(E)$ the environment corresponding to the sequence of mappings associating every spine variable of E to its matching spine argument, according to the positions in which the variables occur in the spine. More formally,

$$\eta(\Box) = \epsilon \tag{6.7}$$

$$\eta(E_1[\lambda x.E_2]t) = \eta(E_2), t/x, \eta(E_1) \tag{6.8}$$

Proposition 6.1 *For every E-context E and every term t, we have that*

$$E[t] =_\beta t[\eta(E)]$$

Proof: Let t be any term.

- If $E = \Box$ then $\eta(E) = \epsilon$ and $[t] = t = t[\epsilon]$.
- If $E = E_1[\lambda x.E_2]s$ then $\eta(E) = \eta(E_1), s/x, \eta(E_2)$ and $E_1[\lambda x.E_2[t]]s =_\beta E_1[\lambda x.t[\eta(E_2)]]s =_\beta (\lambda x.t[\eta(E_2)]]s)[\eta(E_1)] =_\beta t[\eta(E_2)][s/x][\eta(E_1)] =_\beta t[\eta(E_2), s/x, \eta(E_1)] =_\beta t[\eta(E_1[\lambda x.E_2[s]])]$.

□

A *canonical E-context* is an E-context in which every matching spine variable/argument is a β-redex. It is readily seen that the set of the canonical E-contexts

is defined by the grammar $E_c := \Box \mid (\lambda x.E_c)t$, and that any canonical E-redex has the shape $(\lambda x_n. \ldots (\lambda x_2.(\lambda x_1.\Box)t_1)t_2 \ldots)t_n$. We also remark that, for any pair of E-contexts E_1, E_2, the E-context $E_1[E_2]$ is canonical, iff E_1 and E_2 are canonical.

Lemma 6.4 *For any pair of E-contexts E_1, E_2, the E-context $E_1[E_2]$ is canonical, iff E_1 and E_2 are canonical.*

Every environment η can be seen as the explicit representation of a canonical E-context $\mathcal{E}(\eta)$ in which the order of the β-redex along the spine is the inverse of the substitution pairs in the environment

$$t_1/x_1, t_2/x_2, \ldots, t_n/x_n \quad \stackrel{\mathcal{E}}{\longmapsto} \quad (\lambda x_n. \ldots (\lambda x_2.(\lambda x_1.\Box)t_1)t_2 \ldots)t_n$$

which corresponds to the inductive definition

$$\mathcal{E}(\epsilon) = \Box \tag{6.9}$$

$$\mathcal{E}(t/x, \sigma) = \mathcal{E}(\sigma)[(\lambda x.\Box)t] \tag{6.10}$$

Now we consider an equivalence relation $\sim_\eta$, which is not to be confused with standard η-equivalence of λ-calculus (the greek letter η is chosen after "environment").

Proposition 6.2 *Let $\sim_\eta$ be the congruence of E-contexts defined by*

$$E_1[\lambda x.E_2]t \sim_\eta E_1[(\lambda x.E_2)t]$$

when $FV(t) \cap SV(E_1) = \emptyset$. We have that

1. *For every E-context E, there is a unique canonical E-context $\widetilde{E} \sim_\eta E$; moreover, $\widetilde{E} = \mathcal{E}(\eta(E))$.*
2. *For every pair of E-contexts E_1 and E_2, we have that $E_1 \sim_\eta E_2$ iff $\eta(E_1) = \eta(E_2)$.*

Proof:

1. By induction on E. If E is the trivial context $[\ ]$ then $E \sim_\eta \widetilde{E} = \mathcal{E}(\eta(E)) = [\ ]$. If E has the form $E_1[\lambda x.E_2]t$ then $\mathcal{E}(\eta(E)) = \mathcal{E}(\eta(E_2), t/x, \eta(E_1)) = \mathcal{E}(u_1/y_1, \ldots u_n/y_n, t/x, s_1/z_1, \ldots, s_m/z_m) = (\lambda z_m. \ldots (\lambda z1.(\lambda x.(\lambda y_n. \ldots (\lambda y_1.\Box) u_1) \ldots) u_n)t) s_1) \ldots) s_m = \widetilde{E_1}[(\lambda x.\widetilde{E_2})t]$. Clearly $\widetilde{E} \sim_\eta E$ since $\widetilde{E_1}[(\lambda x.\widetilde{E_2})t] \sim_\eta E_1[(\lambda x.\widetilde{E_2})t] \sim_\eta E_1[(\lambda x.E_2)t]$. The canonical context is unique since each E-context is written in a unique way and thus there is only one way to rewrite it using $\sim_\eta$.

2. First, the if part. By induction on $\eta(E_1)$. If $\eta(E_1) = \epsilon$ then necessarily $E_1 = E_2 = [\ ]$. If $\eta(E_1) \neq \epsilon$ necessarily $E_1 = E_1^a[\lambda x.E_1^b]t$ and $E_2 = E_2^a[\lambda y.E_2^b]t$. Since $\eta(E_1) = \eta(E_1^b), t/x, \eta(E_1^a)$ and $\eta(E_2) = \eta(E_2^b), t/y, \eta(E_2^a)$, then necessarily $y = x$, thus both E-context have the same canonical E-context and hence by transitivity of $\sim_\eta$ they must be equivalent.

   Now the only if part. Since $E_1 \sim_\eta E_2$ then they must have the same canonical E-context. Since $\widetilde{E_1} = \mathcal{E}(\eta(E_1)) = \mathcal{E}(\eta(E_2)) = \widetilde{E_2}$ and $\mathcal{E}$ is an injective function, then it must be $\eta(E_1) = \eta(E_2)$.

□

Example 6.2 *Let* $E_1 = (\lambda x.[\ ])$ and $E_2 = [\ ]$. The non-canonical E-context $E_1[\lambda y.E_2]t = ((\lambda x.[\lambda y.\Box]s)t$ has a canonical $\sim_\eta$-equivalent E-context $(\lambda x.[(\lambda y.\Box)t])s$.
□

Remark 31 *Note that* $\sim_\eta$ *equivalence preserves Barendregt's condition on contexts.*

Let us take a term $H = (E[\lambda x.\Box])$ for, some E-context. It is readily seen that $H[\ ]t \sim_\eta E[(\lambda x.\Box)t]$.

In an environment, the order of the variable substitutions is relevant. In fact, since $t[t_1/x_1, t_2/x_2] = t[t_1/x_1][t_2/x_2]$, in general $t[t_1/x_1, t_2/x_2] \neq t[t_2/x_2, t_1/x_1]$ (for instance, $x_1[x_2/x_1, z/x_2] = z$ while $x_1[z/x_2, x_2/x_1] = x_2$). However, when x_1 does not occur free in t_2 and x_2 does not occur free in t_1, it is readily seen that the above variable substitutions can be applied in any order.

We have then the following *permutation equivalence of environments*

$$\eta_1, t/x, s/y, \eta_2 \sim \eta_1, s/y, t/x, \eta_2 \qquad \text{if } x \notin FV(s) \text{ and } y \notin FV(t)$$

A *substitution* σ is the $\sim$-equivalence class of an environment η, i.e., $\sigma = \eta/\sim$. Since $t[\eta_1] = t[\eta_2]$ for every $\eta_1 \sim \eta_2$, we define $t[\sigma] = t[\eta]$ for any $\eta \in \sigma$ (and analogously for contexts). Next we define σ-equivalence between contexts, not to be confused with standard σ-equivalence of λ-calculus.

Definition 6.4 *The* σ-equivalence *is the least congruence on contexts and on terms s.t.*

$$\lambda x.E \sim_\sigma E[\lambda x.\Box] \qquad x \notin FV(E) \tag{6.11}$$

$$E\,t \sim_\sigma E[\Box\,t] \qquad FV(t) \cap SV(E) = \emptyset \tag{6.12}$$

Proposition 6.3 *For every pair of E-contexts* E_1, E_2*,*

1. $E_1 \sim_\eta E_2$ *implies* $E_1 \sim_\sigma E_2$
2. $E_1 \sim_\sigma E_2$ *iff* $\eta(E_1) \sim \eta(E_2)$*.*

Proposition 6.4 *For every H-context* H*, there is a unique sequence of variables* $x_1, \ldots, x_n$*, and a unique sequence of terms* $t_1, \ldots t_m$*, with* $n, m \geq 0$*, s.t.*

$$H \sim \lambda x_1. \ldots \lambda x_h. E[\Box\,t_1 \ldots t_k]$$

Moreover,

$$\lambda x_1. \ldots \lambda x_n. E_1[\Box\,t_1 \ldots t_m] \sim H \sim \lambda x_1. \ldots \lambda x_n. E_2[\Box\,t_1 \ldots t_m]$$

iff $E_1 \sim E_2$*.*

6.4 Optimizations

We discuss here a couple of optimizations, which we consider in the case of the term $(\delta)\delta = (\lambda x.(x)x)\lambda y.(y)y$. A few more ideas are sketched at the end of the section.

6.4.1 Compile prime redexes

How do we efficiently compile a program? We can use a more efficient representation for terms if we move to a richer calculus in which we use explicit substitutions to keep track of primary redexes.

Given the hoc of the n-th term of a run, we must determine if and where it is bound in the moves of the current state of the PAM run. Now binders can also be in substitutions, so if we find that line i contains the explicit substitution $[x/u]$ we set the $n+1$-th term (resp. pointer) as u (resp. $i+1$), just like we would if we were not using a syntax with explicit substitutions. If the binder is not involved in a substitution, we still follow Algorithm 1 and Algorithm 2.

Example 6.3 *T*he term $(\delta)\delta$ could then be represented as $(x)x[x/\lambda y.(y)y]$. □

This modified representation allows to immediately retrieve the argument of a variable which is involved in a primary redexes. This is a first optimization, not too original, but which we manage put to some use in what follows. In particular, the reduction of $(\delta)\delta$ involves a number of trivial substitutions which increases linearly (thus the number of states needed to fire some redexes is quadratic in their number) but we eventually succeed in keeping this number constant (see next section).

Partial evaluation We can use the PAM also if we want to *specialize* a function passing it some arguments. Suppose $\lambda x.\lambda y.f$ is a function of two arguments, we can process the term $(\lambda x.\lambda y.\lambda f)t_1$ or, with a bit of η-expansion, the term $\lambda z.(\lambda x.\lambda y.f)zt_2$ (assuming of course that $z \notin \mathsf{FV}(t)$), thus computation can be resumed from some $\lambda y.g[x/t_1]$ or $\lambda z.h[y/t_2]$. In particular, the partial evaluation of these terms halts when their head (linear) normal form is reached.

Note that in general this operation requires some η-expansion.

6.4.2 Collapse variable lines

If the n-th line of a partial PAM run has a term which is a variable then it can be safely overwritten by the next line (limitedly to the pointer and the term).

- Since the $n+1$-th term is a variable its head occurrence is *not* bound on line n, thus we have to jump to line p_n (which is why we do not overwrite the pointer) and keep looking for the binder.
- In principle we might still need to keep track of the line to look for the argument corresponding to some binder. But this is not actually the case since a variable is a term with *zero* arguments. Also notice that in this case the rank $r' = r + a - l$ is actually unchanged.

We denote $\mathsf{PAM}^{\mathsf{var}}$ the variant of the PAM which adopts this rule. Correspondingly, we are adopting an explicit substitution calculus with an extra rule $\multimap_a$, which we name $\mathrm{ES}^{\mathsf{var}}$.

Example 6.4 *F*or conciseness, here we abbreviate $\lambda y.(y)y$ as δ. Using (only) the above optimization the reduction of $(\lambda x.xx)\lambda y.yy$ becomes the following:

$\mathrm{ES}^{\mathsf{var}}$	$\mathsf{PAM}^{\mathsf{var}}$	#
$(\lambda x.(x)x)\lambda y.(y)y$	$\langle 0; z_0; (\lambda x.(x)x)\lambda y.(y)y)\rangle_{\mathsf{P}}$	0
$\multimap_m (x)x[x/\delta]$ $\multimap_e (\lambda y^1.(y^1)y^1)x[x/\delta]$	$\langle 1; x_1; \lambda y.(y)y)\rangle_{\mathsf{P}}$	1
$\multimap_m (y^1)y^1[y^1/x][x/\delta]$ $\multimap_a (y^1)y^1[y^1/\delta][x/\delta]$ $\multimap_e (\lambda y^2.(y^2)y^2)y^1[y^1/\delta][x/\delta]$	$\langle 1; y_1; \lambda y.(y)y\rangle_{\mathsf{P}}$	2
$\multimap_m (y^2)y^2[y^2/y^1][y^1/\delta][x/\delta]$ $\multimap_a (y^2)y^2[y^2/\delta][y^1/\delta][x/\delta]$ $\multimap_e (\lambda y^3.(y^3)y^3)y^2[y^2/\delta][y^1/\delta][x/\delta]$	$\langle 1; y_1; \lambda y.(y)y\rangle_{\mathsf{P}}$	3
$\multimap_m (y^3)y^3[y^3/y^2][y^2/\delta][y^1/\delta][x/\delta]$ $\multimap_a (y^3)y^3[y^3/\delta][y^2/\delta][y^1/\delta][x/\delta]$ $\multimap_e (\lambda y^4.(y^4)y^4)y^3[y^3/\delta][y^2/\delta][y^1/\delta][x/\delta]$	$\langle 1; y_1; \lambda y.(y)y\rangle_{\mathsf{P}}$	4
$\multimap_m (y^4)y^4[y^4/y^3][y^3/\delta][y^2/\delta][y^1/\delta][x/\delta]$ $\multimap_a (y^4)y^4[y^4/\delta][y^3/\delta][y^2/\delta][y^1/\delta][x/\delta]$ $\multimap_e (\lambda y^5.(y^5)y^5)y^4[y^4/\delta][y^3/\delta][y^2/\delta][y^1/\delta][x/\delta]$	$\langle 1; y_1; \lambda y.(y)y\rangle_{\mathsf{P}}$	5

Note that the final step of the PAM in the table above matches perfectly the last step of the table of Example 6.1.

What is actually going on? We still have to compute PAM moves as usual, but we can discard some useless information.

1. $(1, x_1, \lambda y.(y)y)$;
2. $(\cancel{1}, y_1, \cancel{x_2})$;

   $(1, \cancel{x_2}, \lambda y.(y)y)$;
3. $(1, y_1, \lambda y.(y)y)$;

   ...

Of course we could also just avoid memorizing intermediate trivial steps. □

The point of the previous example is that we evidently save space since we obviously need to store less moves. But even though these trivial steps still takes a bit of time, we also save time when go back looking for arguments (since the dynamic chain is shortened). We also note that there is a certain resemblance with tail-call optimization.

Remark 32 *As we have seen in the previous example, the reduction of $(\delta)\delta$ then becomes evidently non-terminating, since each state of the machine becomes exactly the same.*

Proposition 6.5 *Let $s = \langle 0; z_0; t_0\rangle_{\mathsf{P}}, \ldots, \langle p_{n-1}; z_{n-1}; t_{n-1}\rangle_{\mathsf{P}}, \langle p_n; z_n; z_{n+1}\rangle_{\mathsf{P}}, \langle p_{n+1}; z_{n+1}; t_{n+1}\rangle_{\mathsf{P}}$, $\ldots, \langle p_{n+m+1}; z_{n+m+1}; t_{n+m+1}\rangle_{\mathsf{P}}$ a PAM run. Then $s' = \langle 0; z_0; t_0\rangle_{\mathsf{P}}, \ldots, \langle p_{n-1}; z_{n-1}; t_{n-1}\rangle_{\mathsf{P}}$, $\langle p_{n+1}; z_n; t_{n+1}\rangle_{\mathsf{P}}, \langle p'_{n+2}; z_{n+2}; t_{n+2}\rangle_{\mathsf{P}} \ldots \langle p'_{n+m+1}; z_{n+m+1}; t_{n+m+1}\rangle_{\mathsf{P}}$, where p'_j is p_j if $p_j \leq n$ and $p_j - 1$ otherwise, is sound and operationally indistinguishable w.r.t. s. By sound we mean that the term which is the readback s and s' is the same. And by operationally indistinguishable we mean that if both runs can be extended, the extension is the same (modulo the shown index shift).*

It works well in combination with the optimization which compiles the prime redexes, since there is no need to keep track of the name of the placeholder for the spine argument.

Lemma 6.5 *Let* $s = \langle 0; z_0; t_0\rangle_{\mathsf{P}}, \ldots, \langle p_{n-1}; z_{n-1}; t_{n-1}\rangle_{\mathsf{P}}, \langle p_n; z_n; z_{n+1}\rangle_{\mathsf{P}}$ *a PAM run, where* $t_i = u_i[x_1^i/w_1^i]\ldots[x_{n_i}^i/w_{n_i}^i]$. *If* $z_{n+1} = x_j^i$, *then* s *can be extended with* $\langle i+1; z_{n+1}; w_j^i\rangle_{\mathsf{P}}$.

Lazy version At first sight, the statement of the Proposition 6.5 seems a bit too complex. Is there any reason to take $m \neq 0$? Yes, if we want to apply this optimization on the fly. Suppose we are either looking for a binder or an argument in a PAM run, once we go before a line whose term is an argument we can continue the execution of the machine *and* eliminate that line. Of course, it is simpler to eliminate the line right away, since otherwise we either have to adjust indexes, or we need to use an absolute address for each move (so we either waste time or space).

6.4.3 Argument search at most once

Let us show another way to look at the previous optimization. The optimization avoids repeating trivial substitutions, which would otherwise pile up. How is it done? In practice, we do the substitution once and for all at the level of closures. One can imagine doing a full-fledged call-by-need version of the PAM, by working with pointers to pointers or some other trick.

Here we sketch a simple way to modify the PAM in a similar direction. For all spine arguments of a term we associate a list of extra informations $\langle d_1, \ldots, d_k\rangle$ to its binders (whose number k is statically determined). A piece of data d_i has type $1 \oplus P$, where 1 is the unit type and P is for pointers (i.e., nat). Initially we set $d_i = \mathsf{inl}(*)$ for all i, when we find the argument for a binder we record the corresponding line (note that each move must store this information). A smarter thing would be to use two binders λ° and λ, where the first is used for those variable which occur exactly once: with this mixed strategy we can avoid storing the pointer if we are never going

to use it). This simple optimization does not shorten the length of a PAM run, it simply reduces the time spent searching for the argument of a term at the expenses of increased size of a single move (essentially, we build only *needed* closures).

6.5 μPAM

The notion of σ-equivalence has been extended to the case of $\lambda\mu$-calculus by Laurent [Lau03b]. Moreover, already in λ-calculus a step of head linear reduction does not remove the binder, so the fact that μ-abstractions are not removed by μ-reduction should not be a problem (see next section). So it stands to reason that head linear reduction could be also be adapted to $\lambda\mu$.

As we have seen for lambda calculus, w.r.t. the KAM the approach used by the PAM is lazier. Some works in literature [UMK$^+$03] have already established the importance of laziness for the efficient implementation of continuations, so studying how to adapt the PAM to $\lambda\mu$-calculus certainly appears to be a worthwhile task.

6.5.1 Head Linear Reduction

Extending head linear reduction to $\lambda\mu$ is not so straightforward. It requires a bit of tinkering, instead. First we have to extend the concept of head context:

$$H[\ ] = [\ ] \mid (H[\ ])t \mid \lambda x.H[\ ] \mid \mu\alpha.H[\ ] \mid [\alpha]H[\ ] \tag{6.13}$$

Thus, from the above definition of H-context, the head variable of $[\alpha]t$ (resp. $\mu\alpha.t$) is α (resp. the head variable of t). We also need to extend evaluation contexts

$$E[\ ] = [\ ] \mid E_1[\lambda x.E_2[\ ]]t \mid E_1[Q[(\mu\alpha.E_2)t_1 \dots t_m]] \tag{6.14}$$

where $Q[\ ]$ denotes a head context where the hole is not (directly) on the left of an application. In other words, non-applicative head contexts are

$$Q[\ ] = [\ ] \mid H_1[\lambda x.H_2[\ ]] \mid H_1[\mu\alpha.H_2[\ ]] \mid H_1[[\alpha]H_2[\ ]]. \tag{6.15}$$

The intuition is that they somehow identify the maximal list of argument of a term (thus we can manipulate the whole stack). Then we have to modify head linear reduction itself to include a rule for linear μ reduction (and possibly a corresponding rule for garbage collection). One first attempt could be to consider a rule like the following

$$H_1[Q[\mu\alpha.H_2[[\alpha]u]]t_1 \dots t_m]] \to H_1[Q[\mu\alpha.H_2[[\alpha](u)t_1 \dots t_m]]t_1 \dots t_m]]. \qquad (*)$$

The above rule is wrong because after its application the head variable of the term is still α and thus we would have to apply it again (and again). On the other hand we cannot simply take α out the hole of H_2 either

$$H_1[Q[\mu\alpha.H_2[[\alpha]u]]t_1 \dots t_m]] \to H_1[Q[\mu\alpha.[\alpha]H_2[(u)t_1 \dots t_m]]t_1 \dots t_m]]. \qquad (**)$$

The above rule is wrong because, e.g., H_2 could be of the form $\lambda x.H$, thus α would capture the argument of λx. There does not appear to be a straightforward solution, so we need to employ a small trick.

For each μ variable α we also consider another variable α^{c}, which we call *combusted* α. The definition of head context in Equation (6.13) thus suggests that the head variable of $[\alpha^{\mathsf{c}}]H[t]$ is the head variable of $H[t]$. Formally, we have simply extended the syntax of the calculus; computationally, the intuition is that we use a boolean variable to keep track of the copies of α where we have *already* restored the stack. Analogously, non applicative head context are modified as

$$Q[\ ] = [\ ] \mid H_1[\lambda x.H_2[\ ]] \mid H_1[\mu\alpha.H_2[\ ]] \mid H_1[[\alpha]H_2[\ ]] \mid H_1[[\alpha^{\mathsf{c}}]H_2[\ ]] \qquad (6.16)$$

Thus the rewriting rule we need to use is the following, which we name $\multimap'_\mu$.

$$H_1[Q[\mu\alpha.H_2[[\alpha]u]]t_1 \dots t_m]] \multimap'_\mu H_1[Q[\mu\alpha.H_2[[\alpha^{\mathsf{c}}](u)t_1 \dots t_m]]t_1 \dots t_m]]. \qquad (6.17)$$

We want to extend the PAM to $\lambda\mu$ following the approach of de Groote, thus μ reduction (see Equation (1.12)) is but a substitution followed by a β°-conversion, which in turn is but a linear β-conversion. Hence the above rule can be rephrased

as the rule $\multimap_\mu$

$$H_1[Q[\mu\alpha.H_2[[\alpha]u]]t_1 \dots t_m]] \multimap_\mu H_1[Q[\mu\alpha.H_2[[\alpha := \lambda f.[\alpha^{\mathsf{c}}](f)t_1 \dots t_m]u]]t_1 \dots t_m]]. \tag{6.18}$$

Remark 33 *One might object considering an operation which evidently has a cost which is linear in the number of arguments of the μ-abstraction. We might do otherwise, but the cost of the operation does not depend on the* size *of the arguments involved, so if we consider the functional part of the term as fixed this makes sense.*

In practice we work in the extended syntax of de Groote, which we further extend using two kinds of μ variables.

$$u, t := x \mid \lambda x.t \mid \mu\alpha.u \mid [\alpha]u \mid (u)t \mid [N]u \mid [\alpha^{\mathsf{c}}]u \tag{6.19}$$

$$N := \lambda^\circ f.u \tag{6.20}$$

As anticipated, we also need a corresponding notion of garbage collection.

$$Q[(\mu\alpha.u)t_1 \dots t_m] \multimap_{\mu_g} Q[\mu\alpha.u] \tag{6.21}$$

where it is assumed that u contains no occurrences of α, although it may contain *combusted* occurrences of α, i.e., occurrences of α^{c}. Note that contrarily to β-reduction, where the garbage rule is a trivial case of β, this garbage rule is not a particular case of μ-reduction.

What kind of normal form does this notion of head linear reduction yields? It produces a term where there is no β or μ redex which involves the head occurrence of the resulting term.

6.5.2 μPAM

Ultimately how do we define the PAM for $\lambda\mu$? To this aim we want to use one optimization, already proposed for λ calculus, applied to μ variables. Simply put we want to compute the possible terms of the form $\lambda f.[\alpha^{\mathsf{c}}](f)t_1 \dots t_m$ so that we can re-use the standard mechanism of the PAM with just a few modifications.

Is it even possible to precompute all stacks? How many stacks do we need to consider? How big they are? The following lemma shows that this approach is feasible.

Definition 6.5 *The* μ-compilation *of a* $\lambda\mu$ *term is defined as follows*

$$[\![x]\!] = x \tag{6.22}$$

$$[\![(\mu\alpha.u)t_1 \dots t_m]\!] = \mu\alpha.[\![u]\!][\alpha/\lambda f.[\alpha^{\mathsf{c}}](f)t_1 \dots t_m] \tag{6.23}$$

$$[\![\lambda x.t]\!] = \lambda x.[\![t]\!] \tag{6.24}$$

$$[\![[\alpha]t]\!] = [\alpha][\![t]\!] \tag{6.25}$$

$$[\![(u)t]\!] = [\![u]\!][\![t]\!] \tag{6.26}$$

where it is assumed that Equation (6.26) applies only if Equation (6.23) does not.

The above definition allows to reduce $\lambda\mu$ calculus to a calculus with explicit substitutions and apply the usual way of the PAM to implement head linear reduction, since μ-binders can now be ignored (there are no matching occurrences to substitute, thanks to the use of combusted variables).

Lemma 6.6 (Subterm property) *For any* $\lambda\mu$ *term* t*, the head linear reduction of* $[\![t]\!]$ *involves the substitution only of subterm of* $[\![t]\!]$*.*

Note that for λ-terms $[\![t]\!] = t$, so the above lemma is a conservative extension of the usual one [ADL12]. We can return to the usual syntax for $\lambda\mu$ after computing the head normal form. We call the resulting term *expansion*.

Lemma 6.7 *If* $[\![t]\!] = \tau$ *and* τ *reduces to* τ' *then the expansion of* τ' *can be obtained by linear head reduction from* t *(up to* σ*-equivalence).*

Let us show an interesting example (from [Lau02]) which is the equivalent for μ-reduction of $(\delta)\delta$, which is the example used in [DR04] to show a partial PAM run.

Example 6.5 $(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x$.

$$\begin{aligned}\underline{(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x} &\to_{\mu} \mu\alpha[\alpha](\lambda x.\mu\gamma[\alpha](x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x\\ &\to_{\beta} \mu\alpha[\alpha]\mu\gamma[\alpha]\underline{(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x}\\ &\to^{2}_{\mu\beta} \mu\alpha[\alpha]\mu\gamma[\alpha]\mu\alpha[\alpha]\mu\gamma[\alpha]\underline{(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x}\end{aligned}$$

In practice $\underline{(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x} \to^{2k}_{\mu\beta} \underbrace{\mu\alpha[\alpha]\mu\gamma[\alpha]}_{k \text{ times}} \underline{(\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x)\mu\alpha[\alpha]\lambda x.\mu\gamma[\alpha]x}$.

PAM	#
$\langle 0; z_0; \mu\beta^{\mathsf{c}}.[\beta]\lambda y.\mu\gamma.[\beta]y[\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]\rangle_{\mathsf{P}}$	0
$\langle 1; \beta_1; \lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x\rangle_{\mathsf{P}}$	1
$\langle 1; f_1; \lambda y.\mu\gamma.[\beta]y\rangle_{\mathsf{P}}$	2
$\langle 1; \beta_2; \lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]\rangle_{\mathsf{P}}$	3
$\langle 3; f_1; y\rangle_{\mathsf{P}}$	4
$\langle 2; y; \mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x\rangle_{\mathsf{P}}$	5
$\langle 4; \alpha_1; \beta_2\rangle_{\mathsf{P}}$	6

ES
$(\mu\beta.[\beta]\lambda y.\mu\gamma.[\beta]y)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x$
$\mu\beta^{\mathsf{c}}.[\beta]\lambda y.\mu\gamma.[\beta]y[\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\lambda^{\circ} f^1.[\beta^{\mathsf{c}}](f^1)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]\lambda y.\mu\gamma.[\beta]y[\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}](f^1)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x[f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}](\lambda y^1.\mu\gamma^1.[\beta^1]y^1)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x[f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\beta^1]y^1[y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\lambda^{\circ} f^2.[\beta^{\mathsf{c}}](f^2)\mu\alpha^1.[\alpha^1]\lambda x^1.\mu\delta^1.[\alpha^1]x^1]y^1[y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\beta^{\mathsf{c}}](f^2)\mu\alpha^1.[\alpha^1]\lambda x^1.\mu\delta^1.[\alpha^1]x^1[f^2/^{\circ}y^1][y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\beta^{\mathsf{c}}](y^1)\mu\alpha^1.[\alpha^1]\lambda x^1.\mu\delta^1.[\alpha^1]x^1[f^2/^{\circ}y^1][y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha^{\mathsf{c}}.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\beta^{\mathsf{c}}](\mu\alpha^2.[\alpha^2]\lambda x^2.\mu\delta^2.[\alpha^2]x^2)\mu\alpha^1.[\alpha^1]\lambda x^1.\mu\delta^1.[\alpha^1]x^1[f^2/^{\circ}y^1][y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$
$\mu\beta^{\mathsf{c}}.[\beta^{\mathsf{c}}]\mu\gamma^1.[\beta^{\mathsf{c}}](\mu\alpha^2.[\alpha^2]\lambda x^2.\mu\delta^2.[\alpha^2]x^2)\mu\alpha^1.[\alpha^1]\lambda x^1.\mu\delta^1.[\alpha^1]x^1[\alpha^2/\beta^2][f^2/^{\circ}y^1][y^1/\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x][f^1/^{\circ}\lambda y.\mu\gamma.[\beta]y][\beta/\lambda^{\circ} f.[\beta^{\mathsf{c}}](f)\mu\alpha.[\alpha]\lambda x.\mu\delta.[\alpha]x]$

□

Renaming If we employ for $\lambda\mu$-calculus the optimization we proposed in the previous section for collapsing lines where the replacing term is a variable we are able to handle properly the renaming rule. Thus if we adapt the PAM to Saurin's calculus we can collapse arbitrarily-long sequences of β_s moves into one move.

6.6 Related Works

6.6.1 Some recent developments

In 2008 Terui has given a talk intitled "On space efficiency of Krivine's abstract machine and pointer abstract machine" [Ter08]. In the talk Terui shows that a space-efficient implementation of functional composition must be interactive: as an example he shows the composition of Offline Turring Machines [Hen65] (see also pag. 51 for a brief describtion of IntML and a couple more references). In particular, although there is no evaluator which is uniformly more space-efficient of the KAM, the KAM seems to be good at "tall" terms such as $t_n = (\lambda f.\lambda x.(f)^n x)(\lambda y.y)*$ (since t_n can be evaluated by the KAM using only $\mathcal{O}(1)$ space) but not at "wide" terms such as $u_n = (\lambda x_1 \dots \lambda x_n.x_n) *_1 \cdots *_n$ (since u_n can be evaluated using $\mathcal{O}(n)$ pointers[4]). For simply typed terms with unbounded width and fixed *rank*, it seems that Hylang-Ong games are more efficient (HO games corresponds to the PAM runs [DHR96]). More precisely, Terui defines the rank as $\max(rank(A)+1, rank(B))$ for the arrow type $A \to B$ and 0 per the atomic type ι and states that terms of rank up to 3 can be evaluated in **PSPACE**.

[4] As we already mentioned, the flaw of the KAM is that it may build many closures it never actually uses.

6.6.2 Semantic-inspired approach to head linear reduction

After the initial studies by Danos, Regnier, Curien and Herbelin, some important advances on this topic (with even some attention to complexity issues) have been done, e.g., by Clairambault [Cla11] [Cla13]. He gives bounds on the length of interactions between strategies in HO game semantics.

The advantage of his approach is twofold

- the connection to game semantics allows (in principle) to extend the results to calculi with side-effects [AHM98];
- the result is obtained with purely semantics means, so it is compatible (in principle) with the kind of syntactical restrictions which are typically employed in ICC.

Unfortunately, so far the impact of this line of work has been rather limited [Cla13]; only very few people have been active on it in the last few years, although we can expect this to change in the future. Closely related appears other works on Game Semantics of Dal Lago et al. [DLL08] [DLG12] and Ghica [Ghi05] [GS13].

6.6.3 Relation with Geometry of Interaction

A machine based on Geometry of Interaction called *Interactive Abstract Machine* (IAM) is considered in [DR99], where it is shown that it can be turned into the more efficient *Jumping Abstract Machine* (JAM), which shortens execution paths exploiting the call/return symmetry [AL95]. Specializing the JAM results in either the KAM or the PAM depending on what embedding is chosen to represent λ-calculus into nets, respectively using the equation $D = !D \multimap !D$ or $D = !D \multimap D$. Unfortunately, the case of the PAM is not detailed as much as the case of the KAM.

Analogously Mackie [Mac95] shows how a low-level space-efficient evaluation mechanism for PCF programs can be derived from GoI, and how to improve it with a time-space trade off (similarly to environment machines).

6.7 Conclusions

In this chapter we have reviewed the key ideas of the PAM and also a somewhat clearer explanation of how it works, something which will hopefully give new life to the research on this topic. Most likely some of the abstract machines which have been proposed after the PAM are typically more efficient (e.g., Sestoft's lazy variant of the KAM [Ses97]), but to the author's knowledge the literature offers no comparison of the PAM w.r.t. other abstract machines, with the exception of the aforementioned work of Terui [Ter08] (which proves instead that the answer is by no means obvious, nor absolute).

To start closing the gap with similar studies on variants of the Krivine machine [FGSW07] [Cré07], a couple of simple optimizations have been briefly discussed. This is not only interesting *per se*, since it helps to see the PAM as a convincing implementation (it proves that in some cases there is no need to increase the size of the state of the PAM), but it potentially relevant (as pointed out, e.g., by Crégut [Cré07]) for all those settings in which (an efficient and certified implementation of) partial evaluation has an important role, such as reflective decision procedures of theorem provers [Coq09] [ARCT11] [DMVEP02].

Finally, we have shown how to extend the PAM to $\Lambda\mu$-calculus. There are some Krivine(-like) machines which process terms of $\lambda\mu$-calculus [SR98] or λ-calculus extended with control operators [CH00] [CMM10], but so far the PAM has only been studied to evaluate λ-terms.

Chapter 7

Conclusion

The languages available to programmers nowadays are not those of forty or even ten years ago. Programming languages have changed over time, sometimes to support new architectures [Nvi08] [SGS10] [MDMN12], sometimes not. New languages may adopt different programming paradigm not previously considered, or may combine different paradigms. Existing languages may incorporate some features previously not supported. Usually new functionalities are put together with existing ones, for backwards compatibility, thus making writing correct and efficient compilers or interpreters for languages a task more and more complex [AAB+13] [BL09].

Programming languages Even though the expressive power of two languages (or two versions or the same language) is the same (they can be used to compute the same functions, or solve the same problems), the programmer may have a number of reasons to prefer one over the other (depending on his taste and skills and possibly the particular task at hand or the time available to code). Some of the things we find in modern programming languages are inspired directly from logic or functional languages.

- *generators* (found in Haskell, Python, ...) are routines for on-demand-only computations which are useful for sequences which are either infinite or too big (which takes too much time to compute or too much space to store);

- *list comprehension* (F#, Python, ...) is a syntactic construct (deriving from *set-comprehension*) which maps all elements of a list (or other collection) which satisfy an optional condition to the list of the images of a given function on those values;

- *anonymous functions* (found in JavaScript, Python, ...) a.k.a. *lambda-functions* are functions which are defined without being bound to a name (this may be used in combination with *currying*, turning multiple-arguments functions into possibly more efficient parametric single-argument functions);

- *pattern matching* (Haskell, Scala, ...) is a way to do a structured and exhaustive case analysis (which is also used in some theorem provers to do a proof);

- *monads* (Scala, Scheme, ...) are used to do exception-handling and side-effects such as I/O (more in general, monads add imperative features to functional languages).

The above list is of course non exhaustive, but it is more than enough to prove that advances in programming-languages are of interest even for the end-user (i.e., the "casual" programmer) and that many appealing features have a sound logical foundation.

Domain-specific languages To work with larger and more complex programming languages, one would like to be able to incrementally add features to a simple primitive language [Sai12]. On one hand this is important for those domains in which typical problems are naturally phrased in a particular way and it is thus highly desirable (to avoid bugs, write efficient but maintainable code, etc...) that the programming language offers constructs to match the concepts of the domain; on the other hand having only the features that are actually needed simplifies the verification of programs [Dob05] [HPVD09].

Abstract machines Virtual machines are a way to have architecture-independent programs, since they *abstract* from the details of the particular implementation. Several programming languages are interpreted rather than compiled, or they are compiled to an intermediate architecture-independent language (Java being probably the most known example), although compilation to architecture-specific code typically offers more efficient code.

Analogously abstract machines (see Part III) are useful tools for studying the operational behavior of programming languages without thinking (too much) about implementation. In particular, we have seen as explicit-substitutions (see Section 6.2) help to understand even better the underlying calculus and that the Pointer Abstract Machine (which had been neglected for quite some time) can be tweaked to work with $\lambda\mu$-calculus, similarly to what had been done for the Krivine Machine.

Proof-assistants have been successfully employed to formally verify correctness and completeness of these evaluation mechanisms [Swi12].

Theorem provers Automatic and interactive theorem provers (a.k.a. proof-assistants) can help to check formal proofs, including complex ones that cannot be checked manually (e.g., proofs including a high number of cases to check individually) [Gon08]. Analogously, proof-assistants can be be employed to verify the correctness of programs, even non-trivial ones [Chl11].

> We would all like to have programs check that our programs are correct. Due in no small part to some bold but unfulfilled promises in the history of computer science, today most people who write software, practitioners and academics alike, assume that the costs of formal program verification outweigh the benefits. [...] the technology of program verification is mature enough today that it makes sense to use it in a support role in many kinds of research projects in computer science.

Functional languages, which spring from natural deduction of logical systems, are particularly suitable for this purpose for a number of reasons. Nevertheless, theorem

provers have found applications even to settings in which programs may have *side-effects* [NMS+08] or other imperative features [AGST10]; or in which programs may not terminate [BK08].

One might still object that even if we work with a suitable language the theorem prover we use to check programs is a program itself which may in turn contains bugs. That is not actually true, provided the theorem prover is designed in a certain way [ARCT11, §1].

> Similarly to Coq, Matita follows the so called *De Bruijn principle*, stating that proofs generated by the system should be verifiable by a small and trusted component, traditionally called *kernel*.

The above principle should be seen the equivalent of "Cogito ergo sum", i.e., I can doubt of the correctness of everything but the kernel. How can I trust it? If the kernel is of reasonable size (as in Coq and Matita) then I can check it manually.

Complexity analysis Type systems inspired by Linear Logic are powerful tool for the static verification of this kind of properties, and the work presented in Chapter 4 extends complexity-bounded logics to $\lambda\mu$-calculus. BLLP can probably be extended and improved upon, but it is undoubtedly a step in a new and interesting direction (e.g., we can dream of an "idealized" [Fel90] Ptime programming language with exception-handling facilities [dG95]).

When we write a program in a high-level language that we compile to a program written into a lower-level (possibly hardware-dependent) language we want to be sure that the compiler generates a code with the same semantics of the original (as done in project CompCert [Com] for an abstracted assembly language, which has resulted in a certified compiler that can in some cases generate code capable of outperforming the code obtained with `gcc`). Beside correctness, other interesting properties of programs we would like to be able to verify in a (semi-)automated way include the preservation of complexity. This is an issue that has been formally addressed only recently in project CerCo [Cer] for the MCS-51 microprocessors [MC12] (which is still used in embedded-systems development).

References

[A+03] Samson Abramsky et al. Sequentiality vs. concurrency in games and logic. *Mathematical Structures in Computer Science*, 13(4):531–565, 2003.

[AAB+13] RM Amadio, N Ayache, F Bobot, JP Boender, B Campbell, I Garnier, A Madet, J McKinna, DP Mulligan, M Piccolo, Randy Pollack, Yann Regis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (CerCo). In *Third International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, 2013.

[ABM14] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. Submitted to ICFP 2014, 2014.

[Abr94] Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

[Acc13a] Beniamino Accattoli. Evaluating functions as processes. *arXiv preprint arXiv:1302.6337*, 2013.

[Acc13b] Beniamino Accattoli. Linear logic and strong normalization. In *RTA*, pages 39–54, 2013.

[ADL12] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction (long version). *arXiv preprint arXiv:1202.1641*, 2012.

[AG98] Andrea Asperti and Stefano Guerrini. *The optimal implementation of functional programming languages*. Cambridge University Press, New York, NY, USA, 1998.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In *Interactive Theorem Proving*, pages 83–98. Springer, 2010.

[AH03] Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In *Automata, Languages and Programming*, pages 871–885. Springer, 2003.

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on*, pages 334–344. IEEE, 1998.

[AJ92] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 291–301. Springer, 1992.

[AK12] Beniamino Accattoli and Delia Kesner. The permutative λ-calculus. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 23–36. Springer, 2012.

[AL95] Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ-calculus. *Theoretical Computer Science*, 142(2):277–297, 1995.

[AL13] Andrea Asperti and Jean-Jacques Lévy. The cost of usage in the lambda-calculus. In *LICS*, pages 293–300, 2013.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[App07] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

[ARCT11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *Automated Deduction–CADE-23*, pages 64–69. Springer, 2011.

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992.

[BCDL11] Patrick Baillot, Paolo Coppola, and Ugo Dal Lago. Light logics and optimal reduction: Completeness and complexity. *Information and Computation*, 209(2):118–142, 2011.

[BDL12] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. In *CSL*, pages 62–76, 2012.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 90–101, New York, NY, USA, 2009. ACM.

[BK08] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in coq. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 89–96. ACM, 2008.

[BKMO08] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem. Recursion schemata for NC^k. In *Computer Science Logic*, pages 49–63. Springer, 2008.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[BM10] Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.

[BM12] Emmanuel Beffara and Virgile Mogbil. Proofs as executions. In *Theoretical Computer Science*, pages 280–294. Springer, 2012.

[BMdF12] Pierre Boudes, Damiano Mazza, and Lorenzo Tortora de Falco. An abstract approach to stratification in linear logic. *arXiv preprint arXiv:1206.6504*, 2012.

[Böh68] Corrado Böhm. Alcune proprieta delle forme β-η-normali nel λ-k-calcolo. *Pubblicazioni dell'Istituto per le Applicazioni del Calcolo*, 696, 1968.

[BP99] Patrick Baillot and Marco Pedicini. Elementary complexity and geometry of interaction. In *Typed Lambda Calculi and Applications*, pages 25–33. Springer, 1999.

[BST10] Michele Basaldella, Alexis Saurin, and Kazushige Terui. From focalization of logic to the logic of focalization. *Electronic Notes in Theoretical Computer Science*, 265:161–176, 2010.

[BT09] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. *Information and Computation*, 207(1):41–62, 2009.

[CCS09] M. Cimini, C. Sacerdoti Coen, and D. Sangiorgi. $\bar{\lambda}\mu\tilde{\mu}$ calculus, π-calculus, and abstract machines. *Proceedings of EXPRESS'09*, 2009.

[Cer] The CerCo project. `http://cerco.cs.unibo.it`.

[CH96] Pierre-Louis Curien and Hugo Herbelin. Computing with abstract Böhm trees. In *Proceedings of Fuji International Symposium on Functional and Logic Programming*, pages 20–39, 1996.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243. ACM, 2000.

[CHL96] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)*, 43(2):362–397, 1996.

[Chl11] Adam Chlipala. Certified programming with dependent types, 2011.

[Cla11] Pierre Clairambault. Estimation of the length of interactions in arena game semantics. In *Foundations of Software Science and Computational Structures*, pages 335–349. Springer, 2011.

[Cla13] Pierre Clairambault. Bounding skeletons, locally scoped terms and exact bounds for linear head reduction. In *Typed Lambda Calculi and Applications*, pages 109–124. Springer, 2013.

[CMM10] Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In *Theoretical Computer Science*, pages 165–181. Springer, 2010.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30, 1965.

[Com] The CompCert project. `http://pauillac.inria.fr/~xleroy/compcert`.

[Coq09] The Coq proof assistant reference manual, version 8.2, August 2009.

[Cré07] Pierre Crégut. Strongly reducing variants of the krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007.

[Cun12] Cunningham & Cunningham, Inc. Continuations are Gotos, last edited November 1, 2012. `http://c2.com/cgi/wiki?ContinuationsAreGotos`.

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[dBdBZ80] Jacobus W. de Bakker, Arie de Bruin, and Jeffrey Zucker. *Mathematical theory of program correctness*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1980.

[DC09] Daniel De Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *arXiv preprint arXiv:0905.4251*, 2009.

[DCK97] Roberto Di Cosmo and Delia Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 35–46. IEEE, 1997.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

[DF92] Oliver Danvy and Andrzex Filinski. Representing control: A study of the CPS transformation. *Mathematical structures in computer science*, 2(04):361–391, 1992.

[dG94a] Philippe de Groote. A CPS-translation of the $\lambda\mu$-calculus. In *CAAP*, volume 787 of *LNCS*, pages 85–99. Springer, 1994.

[dG94b] Philippe de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *Logic Programming and Automated Reasoning*, pages 31–43. Springer, 1994.

[dG95] Philippe de Groote. A simple calculus of exception handling. In *TLCA*, pages 201–215, 1995.

[dG98] Philippe de Groote. An environment machine for the $\lambda\mu$-calculus. *Mathematical Structures in Computer Science*, 8(6):637–669, 1998.

[dG01a] Philippe de Groote. Strong normalization of classical natural deduction with disjunction. In *Typed Lambda Calculi and Applications*, pages 182–196. Springer, 2001.

[DG01b] Philippe De Groote. Type raising, continuations, and classical logic. In *Proceedings of the thirteenth Amsterdam Colloquium*, pages 97–101, 2001.

[DHR96] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics and abstract machines. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 394–405. IEEE, 1996.

[Dij68] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[DLDG11] Ugo Dal Lago and Paolo Di Giamberardino. Soft session types (long version). *arXiv preprint arXiv:1107.4478*, 2011.

[DLG12] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2012.

[DLH09] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In *TLCA*, volume 5608 of *LNCS*, pages 80–94. Springer, 2009.

[DLL08] Ugo Dal Lago and Olivier Laurent. Quantitative game semantics for linear logic. In *Computer Science Logic*, pages 230–245. Springer, 2008.

[DLM08a] Ugo Dal Lago and Simone Martini. Proofs as efficient programs. In *Deduction, Computation, Experiment*, pages 141–157. Springer, 2008.

[DLM08b] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theoretical Computer Science*, 398(1):32–50, 2008.

[DLM09] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda-calculus. In *Automata, Languages and Programming*, pages 163–174. Springer, 2009.

[DLMS10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. *arXiv preprint arXiv:1011.6431*, 2010.

[DLMZ10] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[DLP12a] Ugo Dal Lago and Barbara Petit. The geometry of types (long version). *arXiv preprint arXiv:1210.6857*, 2012.

[DLP12b] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 115–126. ACM, 2012.

[DLP12c] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario (long version). *arXiv preprint arXiv:1207.5592*, 2012.

[DLP13a] Ugo Dal Lago and Giulio Pellitta. Complexity analysis in presence of control operators and higher-order functions. In *LPAR-19, Proceedings*, pages 258–273. Springer, 2013.

[DLP13b] Ugo Dal Lago and Giulio Pellitta. Complexity analysis in presence of control operators and higher-order functions (long version), 2013. http://arxiv.org/abs/1310.1763.

[DLP13c] Ugo Dal Lago and Giulio Pellitta. Polarized bounded linear logic. Fourth Workshop on Developments in Implicit Computational Complexity, DICE 2013, Held as part of ETAPS 2013, March 2013.

[DLPT12a] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In *Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2012.

[DLPT12b] Ugo Dal Lago and Paolo Parisen Toldin. An higher-order characterization of probabilistic polynomial time (long version). *arXiv preprint arXiv:1202.3317*, 2012.

[DLS] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space bounded functional programming. Submitted.

[DLS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012, page 205. Springer, 2010.

[DMVEP02] Maarten De Mol, Marko Van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. In *Implementation of Functional Languages*, pages 55–71. Springer, 2002.

[Dob05] Simon A. Dobson. The de Bruijn principle and the compositional design of programming languages. In *17th International Workshop on Implementation and Application of Functional Lan-*

guages, 2005. Available at `http://www.simondobson.org/softcopy/de-bruijn-iafl-05.pdf`.

[DP01] René David and Walter Py. $\lambda\mu$-calculus and Böhm's theorem. *Journal of Symbolic Logic*, pages 407–413, 2001.

[DR99] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1):79–97, 1999.

[DR04] Vincent Danos and Laurent Regnier. Head linear reduction. Technical report, Technical Report, Université Paris 7 and Université Aix-Marseille 2, 2004.

[Fag73] Ronald Fagin. *Contributions to the model theory of finite structures.* PhD thesis, University of California, Berkeley, 1973.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer, 1990.

[FFKD87] Matthias Felleisen, Daniel P Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical computer science*, 52(3):205–237, 1987.

[FGSW07] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293, 2007.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design*

and implementation, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[GAL92] Georges Gonthier, Martín Abadi, and J-J Lévy. Linear logic without boxes. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 223–234. IEEE, 1992.

[GDR07] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *Computer Science Logic*, pages 253–267. Springer, 2007.

[Ghi05] Dan R Ghica. Slot games: a quantitative model of computation. In *ACM SIGPLAN Notices*, volume 40, pages 85–97. ACM, 2005.

[Gil77] John Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[Gir87a] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Gir87b] Jean-Yves Girard. Linear logic and parallelism. In *Mathematical models for the semantics of parallelism*, pages 166–182. Springer, 1987.

[Gir89] Jean-Yves Girard. Towards a geometry of interaction. *Categories in Computer science and Logic*, 92:69–108, 1989.

[Gir91] Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

[Gir95a] Jean-Yves Girard. Light linear logic. In *Logic and computational complexity*, pages 145–176. Springer, 1995.

[Gir95b] Jean-Yves Girard. Linear logic: its syntax and semantics. *Advances in linear logic*, 222:1–42, 1995.

[Gir96] Jean-Yves Girard. Proof-nets: the parallel syntax for proof-theory. *Lecture Notes in Pure and Applied Mathematics*, pages 97–124, 1996.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. Proofs and types. *Cambridge Tracts in Theoretical Computer Science*, 7, 1989.

[GMRDR12] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of PSPACE. *ACM Transactions on Computational Logic (TOCL)*, 13(2):18, 2012.

[Goe92] Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1):45 – 66, 1992.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GRDR07] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for *lambda*-calculus. In *CSL*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.

[Gri90] Timothy Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58. ACM Press, 1990.

[GS13] Dan R Ghica and Alex Smith. From bounded affine types to automatic timing analysis. *arXiv preprint arXiv:1307.2473*, 2013.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Phil Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[Gul09] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *CAV*, volume 5643 of *LNCS*, pages 51–62. Springer, 2009.

[Hen65] F.C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553 – 578, 1965.

[HFW86] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer languages*, 11(3):143–153, 1986.

[HG08] Hugo Herbelin and Silvia Ghilezan. An approach to call-by-name delimited continuations. In *ACM SIGPLAN Notices*, volume 43, pages 383–394. ACM, 2008.

[HL10] Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theoretical computer science*, 411(22):2223–2238, 2010.

[HO00] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[Hof00] Martin Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[HPVD09] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *Model Driven Engineering Languages and Systems*, pages 423–437. Springer, 2009.

[HS04] Martin Hofmann and Philip J Scott. Realizability models for BLL-like languages. *Theoretical Computer Science*, 318(1):121–137, 2004.

[HS09] Hugo Herbelin and Alexis Saurin. $\lambda\mu$-calculus and $\Lambda\mu$-calculus: a capital difference, 2009.

[HYB04] Kohei Honda, Nobuko Yoshida, and Martin Berger. Control in the π-calculus. In *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04)*, 2004.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, Madrid, Spain, 2010. ACM Press.

[Jon97] Neil D. Jones. *Computability and complexity: from a programming perspective*, volume 21. MIT press, 1997.

[JYM00] Marion Jean-Yves and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *Logic for Programming and Automated Reasoning*, pages 25–42. Springer, 2000.

[Kes07] Delia Kesner. The theory of calculi with explicit substitutions revisited. In *Computer Science Logic*, pages 238–252. Springer, 2007.

[KHM+07] Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[KL07] Delia Kesner and Stéphane Lengrand. Resource operators for λ-calculus. *Information and Computation*, 205(4):419–473, 2007.

[Kri90] Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de Gödel. *Archive for Mathematical Logic*, 30(4):241–267, 1990.

[Kri94] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

[Kri96] Jean-Louis Krivine. About classical logic and imperative programming. *Annals of mathematics and Artificial Intelligence*, 16(1):405–414, 1996.

[Kri07] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1):163–180, 2004.

[Lam89] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30. ACM, 1989.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Lau02] Olivier Laurent. *Étude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.

[Lau03a] Olivier Laurent. Krivine's abstract machine and the $\lambda\mu$-calculus (an overview). Unpublished note, September 2003.

[Lau03b] Olivier Laurent. Polarized proof-nets and $\lambda\mu$-calculus. *Theoretical Computer Science*, 290(1):161–188, 2003.

[Lei93] Daniel Leivant. Stratified functional programs and computational complexity. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 325–333. ACM, 1993.

[Lip02] Sylvain Lippi. *Théorie et pratique des réseaux d'interaction*. PhD thesis, 2002.

[Lip07] Sylvain Lippi. The graphical Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):295–318, 2007.

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In *Typed Lambda Calculi and Applications*, pages 274–288. Springer, 1993.

[LM95] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In *Computer Science Logic*, pages 486–500. Springer, 1995.

[LM00] Julia L. Lawall and Harry G. Mairson. Sharing continuations: proofnets for languages with explicit control. In *Programming Languages and Systems*, pages 245–259. Springer, 2000.

[LM07] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *Computer Science Logic*, pages 451–465. Springer, 2007.

[LM09] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[Mac95] Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 198–208, New York, NY, USA, 1995. ACM.

[MC12] Dominic P Mulligan and Claudio Sacerdoti Coen. On the correctness of an optimising assembler for the intel MCS-51 microprocessor. In *Certified Programs and Proofs*, pages 43–59. Springer, 2012.

[MDMN12] Zigurd Mednieks, Laird Dornin, G Blake Meike, and Masumi Nakamura. *Programming Android: Java Programming for the New Generation of Mobile Devices*. O'Reilly, 2012.

[Mel95] Paul-André Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Typed Lambda Calculi and Applications*, pages 328–334. Springer, 1995.

[Mil82] Robin Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[Mil04] Dale Miller. Overview of linear logic programming. *Linear Logic in Computer Science*, 316:119–150, 2004.

[MM+06] Jean-Yves Marion, Jean-Yves Moyen, et al. Heap-size analysis for assembly programs. 2006. http://hal.archives-ouvertes.fr/hal-00067838.

[MP01] Richard Moot and Mario Piazza. Linguistic applications of first order intuitionistic linear logic. *Journal of Logic, Language and Information*, 10(2):211–232, 2001.

[MPT13] Jean-Yves Moyen and Paolo Parisen Toldin. A polytime complexity analyser for probabilistic polynomial time over imperative stack programs. *arXiv preprint arXiv:1304.3249*, 2013.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.

[MT03] Harry G. Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In *Theoretical Computer Science*, pages 23–36. Springer, 2003.

[MT10] Paul-André Mellies and Nicolas Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, 2010.

[Nee04] Peter Møller Neergaard. A functional language for logarithmic space. In *Programming Languages and Systems*, pages 311–326. Springer, 2004.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 229–240, New York, NY, USA, 2008. ACM.

[Nvi08] CUDA Nvidia. Programming guide, 2008.

[Ong96] C.-H.Luke Ong. A semantic view of classical proofs: Type-theoretic, categorical, and denotational characterizations. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 230–241. IEEE, 1996.

[OS97] C.-H. Luke Ong and Charles A. Stewart. A Curry-Howard foundation for functional computation with control. In *POPL*, pages 215–227. ACM Press, 1997.

[Par92] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *LPAR*, volume 624 of *LNCS*, pages 190–201. Springer, 1992.

[Par93] Michel Parigot. Classical proofs as programs. In *Computational logic and proof theory*, pages 263–276. Springer, 1993.

[PCM+05] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 216–227, New York, NY, USA, 2005. ACM.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[PQ07] Marco Pedicini and Francesco Quaglia. PELCR: Parallel environment for optimal lambda-calculus reduction. *ACM Transactions on Computational Logic (TOCL)*, 8(3):14, 2007.

[Pra65] Dag Prawitz. *Natural deduction: A proof-theoretical study*. PhD thesis, Stockholm, 1965.

[Reg92] Laurent Regnier. *Lambda-calcul et réseaux*. PhD thesis, 1992.

[Reg94] Laurent Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126(2):281–292, 1994.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[Ron12] Simona Ronchi Della Rocca. Linear logic and theoretical computer science in Italy (optimal reduction and implicit computational complexity). *INFLUXUS (`http://www.influxus.eu/`)*, Logique et Interaction: vers une geometrie de la cognition, 2012.

[Sai12] Luca Saiu. Gnu epsilon - an extensible programming language. *CoRR*, abs/1212.5210, 2012.

[Sau05] Alexis Saurin. Separation with streams in the $\lambda\mu$-calculus. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 356–365. IEEE, 2005.

[Sau08] Alexis Saurin. On the relations between the syntactic theories of $\lambda\mu$-calculi. In *Computer Science Logic*, pages 154–168. Springer, 2008.

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, pages 411–420, 2007.

[Sel01] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(02):207–260, 2001.

[Ses97] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997.

[SGS10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[Sha04] Chung-chieh Shan. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming*, pages 99–107, 2004.

[Shi97] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, pages 2–1, 1997.

[SR98] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of functional programming*, 8(6):543–572, 1998.

[Sta79] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[Swi12] Wouter Swierstra. From mathematics to abstract machine: A formal derivation of an executable Krivine machine. *arXiv preprint arXiv:1202.2924*, 2012.

[Ter08] K. Terui. On space efficiency of Krivine's abstract machine and pointer abstract machine, 2008. Workshop on Implicit Computational Complexity.

[UMK+03] Tomoharu Ugawa, Nobuhisa Minagawa, Tsuneyasu Komiya, Masahiro Yasugi, and Taiichi Yuasa. Lazy stack copying and stack copy sharing for the efficient implementation of continuations. In *Programming Languages and Systems: First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895, page 410. Springer, 2003.

[vEB90] Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. 1990.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 19–28. ACM, 1980.

Appendix A

Proofs of substitution lemmas for BLLP

Notation: for coinciseness, multiple substitution is denoted $\{x_1, \ldots, x_n/u_1, \ldots, u_n\}$, i.e., $t\{x_1, \ldots, x_n/u_1, \ldots, u_n\} := t\{x_1/u_1, \ldots x_n/u_n\}$; and a sequence of (labeled) formulas $[N_1]^{p_1}_{x_1}, \ldots, [N_n]^{p_n}_{x_n}$ is denoted $\{[N_i]^{p_i}_{x_i}\}^n_{i=1}$.

A.1 Auxiliary lemmas

Lemma A.1 (Weakening) *If* $\Gamma \vdash t : [N]^p_x \mid \Delta$ *is derivable, then* $\Gamma, y : {}^s_z[M]^q_u \vdash t : [N]^p_x \mid \Delta$ *and* $\Gamma \vdash t : [N]^p_x \mid \Delta, \alpha : [M]^q_u$*, where* x *(resp.* α*) does not appear in* Γ *(resp.* Δ*) are also derivable.*

Lemma A.2 *If* $\mathbf{N}$ *has the appropriate form then*

$$\sum_{x<p}\sum_{y<q}\mathbf{N} = \sum_{y<q}\sum_{x<p}\mathbf{N}.$$

Lemma A.3 *If N has the appropriate form then*

$$\otimes_{x<p}[N]^0_y = [N]^0_y.$$

Lemma A.4 *Let $\Gamma, x : {}^{s}_{z}[N]^p_y \vdash t : [M]^q_y|\Delta$ be a derivation. If $x \in \mathsf{FV}(t)$ then $p \sqsupseteq 1$.*

A.2 Proof of Lemma 4.12

By induction on the derivation of t

- If the last rule is var then the derivation has this form (here necessarily $l = 1$)

$$\frac{1 \sqsubseteq p, r[y/0] \sqsubseteq q, M \sqsubseteq N[y/0]}{x : {}^{r}_{z}[N]^p_y \vdash x : [M]^q_z|}\ \mathrm{var}$$

  The conclusion follows taking the derivation of u and (since $h \sqsupseteq 1$) using Lemma 4.2.

- If the last rule is either abs, μ-name, μ-abs, $?w^\mu$ or $?c^\mu$ then the conclusion follows by a straightforward application of the induction hypothesis.

- If the last rule is app then $t = (t_1[x_1, \ldots, x_l/u_1, \ldots, u_l])t_2[x_{l+1}, \ldots, x_{l+n}/u_{l+1}, \ldots, u_{l+n}]$. Hence the derivation has the following form

$$\dfrac{\begin{array}{ll}\pi \ \triangleright\ \Gamma_1, x_1 : {}^{s_1}_{z}[N_1]^{p_1}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} & \sigma \ \triangleright\ \Gamma_2, x_{l+1} : {}^{s_{l+1}}_{z}[N_{l+1}]^{p_{l+1}}_{y}, \ldots, x_{l+n} : {}^{s_{l+n}}_{z}[N_{l+n}]^{p_{l+n}}_{y} \\ \vdash t_1 : [O \multimap^r_z M]^{q_1}_a \mid \Delta_1 & \vdash t_2 : [O]^r_z \mid \Delta_2 m \sqsupseteq q_1, q \sqsupseteq q_1\end{array}}{\begin{array}{l}\Gamma_1, x_1 : {}^{s_1}_{z}[N_1]^{p_1}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y}, \sum_{d<m}\Gamma_2, x_{l+1} : \sum_{d<m} {}^{s_{l+1}}_{z}[N_{l+1}]^{p_{l+1}}_{y}, \\ \ldots, x_{l+n} : \sum_{d<m} {}^{s_{l+n}}_{z}[N_{l+n}]^{p_{l+n}}_{y} \vdash (t_1)t_2 : [M]^q_a \mid \Delta_1, \sum_{d<m}\Delta_2\end{array}}\ \mathsf{app}$$

Let $\pi^{IH} \ \triangleright\ \Gamma_1, \sum_{b<h_1}\Theta_1, \ldots, \sum_{b<h_l}\Theta_l \vdash t_1[x_1, \ldots, x_l/u_1, \ldots, u_l] : [O \multimap^r_z M]^{q_1}_a \mid \Delta_1, \sum_{b<h_1}\Xi_1, \ldots, \sum_{b<h_l}\Xi_l$ (resp. $\sigma^{IH} \ \triangleright\ \Gamma_2, \sum_{b<h_{l+1}}\Theta_{l+1}, \ldots, \sum_{b<h_{l+n}}\Theta_{l+n} \vdash t_2[x_{l+1}, \ldots, x_{l+n}/u_{l+1}, \ldots, u_{l+n}] : [O]^r_z \mid \Delta_2, \sum_{b<h_{l+1}}\Xi_{l+1}, \ldots, \sum_{b<h_{l+n}}\Xi_{l+n}$) the derivation obtained applying the induction hypothesis to π (resp. σ). The derivation we want is the following

$$\dfrac{\pi^{IH} \qquad \sigma^{IH} \qquad m \sqsupseteq q_1, q \sqsupseteq q_1}{\begin{array}{l}\Gamma_1, \{\sum_{b<h_i}\Theta_i\}^{l}_{i=1}, \{\sum_{d<m}\sum_{b<h_i}\Theta_i\}^{l+n}_{i=l+1}, \sum_{d<m}\Gamma_2 \\ \vdash (t_1[x_1, \ldots, x_l/u_1, \ldots, u_l])t_2[x_{l+1}, \ldots, x_{l+n}/u_{l+1}, \ldots, u_{l+n}] : [M]^q_a \\ \mid \Delta_1, \{\sum_{b<h_i}\Xi_i\}^{l}_{i=1}, \{\sum_{d<m}\sum_{b<h_i}\Xi_i\}^{l+n}_{i=l+1}, \sum_{d<m}\Delta_2\end{array}}\ \mathsf{app}$$

where $\{\sum_{d<m}\sum_{b<h_i}\Theta_i\}^{l+n}_{i=l+1} = \{\sum_{b<h_i}\sum_{d<m}\Theta_i\}^{l+n}_{i=l+1}$ and $\{\sum_{d<m}\sum_{b<h_i}\Xi_i\}^{l+n}_{i=l+1} = \{\sum_{b<h_i}\sum_{d<m}\Xi_i\}^{l+n}_{i=l+1}$ because of Lemma A.2.

- If the last rule is $?c^{\lambda}$ then $t = v[f/c][f/d]$. If $\forall i(f \neq x_i)$ the conclusion follows by a straightforward use of the induction hypothesis, so we can assume otherwise (thus, w.l.o.g., $t[x_1/u_1] = (v[c/x_1][d/x_1])[x_1/u_1] = v[c/u_1][d/u_1]$, where c and d have respectively k_c and k_d occurrences in v). The derivation of t has the following form

$$\frac{\begin{array}{l}\pi \ \triangleright\ \Gamma, c : {}^{s}_{z}[O]^{p_c}_{y}, d : {}^{s}_{z}[O]^{p_d}_{y}[b/b+h_c], \\ x_2 : {}^{s_2}_{z}[N_2]^{p_2}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} \vdash v : [M]^{q}_{a} \mid \Delta\end{array} \qquad {}^{s_1}_{z}[N_1]^{p_1}_{y} \sqsubseteq {}^{s}_{z}[O]^{p_c}_{y} \uplus {}^{s}_{z}[O]^{p_d}_{y}[b/b+h_c]}{\Gamma, x_1 : {}^{s_1}_{z}[N_1]^{p_1}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} \vdash v : [M]^{q}_{a} \mid \Delta}\ ?c^{\lambda}$$

(cf. Lemma 4.5 and 4.6) hence using Lemma 4.2 we get

$$\pi_S \ \triangleright\ \Gamma, c : {}^{s_1}_{z}[N_1]^{p_c}_{y}, d : {}^{s}_{z}[N_1]^{p_d}_{y}[b/b+h_c], x_2 : {}^{s_2}_{z}[N_2]^{p_2}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} \vdash v : [M]^{q}_{a} \mid \Delta$$

and applying the induction hypothesis

$$\frac{\begin{array}{l}\pi_S^{IH} \ \triangleright\ \Gamma, \sum_{b<h_c} \Theta_1, \sum_{b<h_d[b/b+h_c]} \Theta_1[b/b+h_c], \{\sum_{b<h_2} \Theta_2\}_{i=2}^{l} \\ \vdash v[{}^{u_1,u_1,u_2,\ldots,u_l}/_{c,d,x_2,\ldots,x_l}] : [M]^{q}_{a} \mid \Delta, \sum_{b<h_c} \Xi_1, \sum_{b<h_d[b/b+h_c]} \Xi_1[b/b+h_c], \{\sum_{b<h_2} \Xi_2\}_{i=2}^{l} \\ \qquad \sum_{b<h_1} \Theta_1 \sqsubseteq \sum_{b<h_c} \Theta_1 \uplus \sum_{b<h_d[b/b+h_c]} \Theta_1[b/b+h_c] \\ \qquad \sum_{b<h_1} \Xi_1 \sqsubseteq \sum_{b<h_c} \Xi_1 \uplus \sum_{b<h_d[b/b+h_c]} \Xi_1[b/b+h_c]\end{array}}{\Gamma, \sum_{b<h_1} \Theta_1, \ldots, \sum_{b<h_l} \Theta_l \vdash v[{}^{u_1,u_1,\ldots,u_l}/_{c,d,x_2,\ldots,x_l}] : [M]^{q}_{a} | \Delta, \sum_{b<h_1} \Xi_1, \ldots, \sum_{b<h_l} \Xi_l}\ ?c^{*}$$

- If the last rule of the derivation is $?w^{\lambda}$ then (assuming, w.l.o.g., that x_1 is introduced by $?w$, otherwise it is trivial) it is of the form

$$\frac{\pi \ \triangleright\ \Gamma, x_2 : {}^{s_2}_{z}[N_2]^{p_2}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} \vdash t : [M]^{q}_{y} \mid \Delta}{\Gamma, x_1 : {}^{s_1}_{z}[N_1]^{p_1}_{y}, x_2 : {}^{s_2}_{z}[N_2]^{p_2}_{y}, \ldots, x_l : {}^{s_l}_{z}[N_l]^{p_l}_{y} \vdash t : [M]^{q}_{y} \mid \Delta}\ ?w^{\lambda}$$

  Thus the derivation we want is obtained by π^{IH} by as many $?w^{*}$ rules as necessary.

A.3 Proof of Lemma 4.13

By induction on the derivation of t.

- Last rule cannot be var nor abs.
- If the last rule of the derivation is μ-name then $t = [\beta]v$, where, w.l.o.g., β may or may not be α_1. Suppose they are distinct variables, the derivation has the following form

$$\frac{\pi \ \triangleright \ \Gamma \vdash v : [O]^r_y \mid \Delta, \{\alpha_i : [N_i \multimap^{s_i}_z M_i]^{p_i}_y\}^l_{i=1}}{\Gamma \vdash [\beta]v : [\bot]^q_y \mid \Delta, \{\alpha_i : [N_i \multimap^{s_i}_z M_i]^{p_i}_y\}^l_{i=1}, \beta : [O]^r_y}\ \mu\text{-name}$$

  Let σ the sub-derivation of π which introduces α_1. We can assume α_1 appears free in t, otherwise the conclusion follows from Lemma A.1. Let σ^{IH} the derivation obtained applying the induction hypothesis to σ. The derivation we want is $\pi[\sigma/\sigma^{IH}]$. Suppose instead $\alpha_1 = \beta$, in this case the derivation has the following form

$$\frac{\pi \ \triangleright \ \Gamma \vdash v : [N_1 \multimap^{s_1}_z M_1]^{p_1}_y \mid \Delta, \{\alpha_i : [N_i \multimap^{s_i}_z M_i]^{p_i}_y\}^l_{i=2}}{\Gamma \vdash [\beta]v : [\bot]^q_y \mid \Delta, \{\alpha_i : [N_i \multimap^{s_i}_z M_i]^{p_i}_y\}^l_{i=1}}\ \mu\text{-name}$$

  In this case the derivation we seek is the following

$$\dfrac{\dfrac{\pi^{IH} \ \triangleright \ \Gamma, \left\{\sum_{b<h_i}\Theta_i\right\}^l_{i=2} \vdash v : [N_1 \multimap^{s_1}_z M_1]^{p_1}_y \mid \Delta, \left\{\alpha_i : [M_i]^{p_i}_y, \sum_{b<h_i}\Theta_i\right\}^l_{i=2} \qquad \rho_1 \ \triangleright \ \Theta_1 \vdash u_1 : [N_1]^{s_1}_z \mid \Xi, \ h_1 \sqsupseteq p_1}{\Gamma, \left\{\sum_{b<h_i}\Theta_i\right\}^l_{i=1} \vdash (v)u_1 : [M_1]^{p_1}_y \mid \Delta, \left\{\alpha_i : [M_i]^{p_i}_y, \sum_{b<h_i}\Xi_i\right\}^l_{i=2}, \sum_{b<h_1}\Xi_1}\ \text{app}}{\Gamma, \left\{\sum_{b<h_i}\Theta_i\right\}^l_{i=1} \vdash [\alpha_1](v)u_1 : [\bot]^q_y \mid \Delta, \left\{\alpha_i : [M_i]^{p_i}_y, \sum_{b<h_i}\Xi_i\right\}^l_{i=2}}\ \mu\text{-name}$$

- If the last rule of the derivation is $?w^{\mu}$ then, w.l.o.g., either the variable introduced is α_1, in which case it suffice to replace the type of α_1 and use Lemma A.1, or it is $\beta \neq \alpha_1$. In the latter case the conclusion follows applying the induction hypothesis to the sub-derivation above the $?w^{\mu}$ rule and then using a $?w^{\mu}$ introducing β with the type unchanged.
- If the last rule of the derivation is $?c^{\mu}$ then it suffices to proceed as in the λ-substitution Lemma, *mutatis mutandis*.
- If the last rule of the derivation is app, μ-abs, $?w^{\lambda}$ or $?c^{\lambda}$ the proof is trivial.