Introducing λ , a λ-calculus for effectful computation

Jirka Maršík, Maxime Amblard, Philippe de Groote

To cite this version:

Jirka Maršík, Maxime Amblard, Philippe de Groote. Introducing λ , a λ-calculus for effectful computation. Theoretical Computer Science, 2021, 869, pp.108-155. 10.1016/j.tcs.2021.02.038 . hal-03200474

HAL Id: hal-03200474

https://inria.hal.science/hal-03200474

Submitted on 16 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Introducing $(\!|\lambda|\!)$, a λ-calculus for Effectful Computation

Jirka Maršík[a], Maxime Amblard[b,c], Philippe de Groote[c]

[a] *Oracle Labs, Prague*
[b] *Université de Lorraine, CNRS, Inria-Nancy Grand Est*
[c] *Inria-Nancy Grand Est*

Abstract

We present $(\!|\lambda|\!)$, a calculus with special constructions for dealing with effects and handlers. This is an extension of the simply-typed λ-calculus (STLC). We enrich STLC with a type for representing effectful computations alongside with operations to create and process values of this type. The calculus is motivated by natural language modelling, and especially semantic representation. Traditionally, the meaning of a sentence is calculated using λ-terms, but some semantic phenomena need more flexibility. In this article we introduce the calculus and show that the calculus respects the laws of algebraic structures and it enjoys strong normalisation. To do so, confluence is proven using the Combinatory Reduction Systems (CRSs) of Klop and termination using the Inductive Data Type Systems (IDTSs) of Blanqui.

Keywords: side effects, monads, λ-calculus, handlers, CRS, IDTS

1 Introduction

λ-calculus is a widely used tool for semantic analysis of natural language [1]. λ terms serve as higher order logic formulas that give the truth conditions of sentences and they also describe how the meanings of the individual constituents compose together using abstraction and function application. The numerous linguistic phenomena in play lend themselves to analyses using computational side effects and monads [2, 3, 4, 5]. The idea of treating linguistic expressions as effectful actions or programs is also very relevant to dynamic semantics, which treats the meanings of sentences as instructions to update some common ground or other linguistic context [6, 7]. In order to study a fragment of natural language that encompasses several of the effectful phenomena, we need a framework that would help us manage the increasing complexity of the λ-terms involved. In this paper, we present an extension to the simply-typed λ-calculus for that exact purpose.

Taking stock of the different monadic structures of linguistic side effects, we can examine existing approaches that try to combine side effects to find a formalism that can talk about all the aspects of language at the same time. One such theoretical framework are effects and handlers. In this framework, programs are interpreted as sequences of instructions (or more generally as decision trees).[1] The instructions are symbols called *operations*, which stand for the different effects, the different ways that programs can interact with their contexts. The process of calculating the semantic representation of a linguistic expression is then expressed as a program using these operations. This is when handlers come into play. A *handler* is an interpreter that gives a definition to the operation symbols in a program. Handlers can be made modular [2] so that the interpreter for our vocabulary of context interactions can be defined as the composition of several smaller handlers, each treating a different aspect of language (dynamicity, implicatures, deixis. . .).

When using effects and handlers, we therefore start by enumerating the set of interactions that programs can have with their contexts. We then write handlers which implement these instructions and produce a suitable semantic representation. This approach thus closely follows the mantra given by Lewis:

> In order to say what a meaning is, we may first ask what a meaning does and then find something that does that.
>
> General Semantics, David Lewis [11]

[1] More precisely, we are interpreting programs in a free monad [8].
[2] In a similar way that monads can be turned into monad transformers (monad morphisms) and then composed [9, 2, 10].

We can trace the origins of effects and handlers to two strands of work. One is Cartwright and Felleisen's work on Extensible Denotational Language Specifications [12], in which a technique for building semantics is developed such that when a (programming) language is being extended with new constructions (and new side effects), the existing denotations remain compatible and can be reused.

The other precursor is Hyland, Plotkin and Power's work on algebraic effects [13], a categorical technique for studying effectful computations, which was later extended by Plotkin and Pretnar to include handlers [14, 15, 16]. The reader can refer to [17, 18] for an overview of computational effects. This was further developed in [19, 20].

The technique has gained in popularity in recent years (2012 and onward). It finds applications both in the encoding of effects in pure functional programming languages [21, 22, 23, 24] and in the design of programming languages [25, 26, 27, 28, 29], and also inspired extensions of other programming languages in Java [30] and in C [31].

Algebraic effects provide a strong and deep foundation for a theory of computation side effects. Their mathematical development is rooted in Lawvere theories [32, 33], with new generalizations [34] and extensions [35] of the context still being discovered. The use of algebraic theories to describe computation also lends itself to other fruitful applications, e.g. connecting the theories with algebraic specifications of programs.

The problem of the specification and verification of programs with algebraic effects has seen a lot of progress recently [36, 37, 38, 39, 40, 41, 42]. Other interesting developments include the introduction of dependent types [43], a generalization from monads to applicative functors and arrows [44], new mechanisms for abstracting effects [45] and novel implementation techniques [46, 47]. Algebraic effects and effect handlers provide a modular abstraction for effectful programming. They support user-defined effects, as in Haskell [48], in conjunction with direct-style effectful programming, as in ML[49, 50]. They also present a structured interface to programming with delimited continuations [51].

In this article, we will focus on defining a suitable calculus based on STLC which implement effects and handlers: $(\!|\lambda|\!)$.

We will be adding a new type constructor, $\mathcal{F}$, into our language. The type $\mathcal{F}(\alpha)$ will correspond to effectful *computations* that produce values of type α. The idea comes from the programming language Haskell and its use of monads [52, 53]. Our type constructor $\mathcal{F}$ will also stand in for a monad, one that has been already encoded in Haskell in several ways [21, 23]. The motivation behind $(\!|\lambda|\!)$ is to build a minimal language which directly gives us the primitive operations for working with this particular monad. This way, we end up with a language that:

- is smaller than Haskell (and thus more mananageable to analyse),
- is closer to the STLC (favored by semanticists),
- and which makes more evident the features that our proposal relies on.

The distinction between the type α and the type $\mathcal{F}(\alpha)$ will, in different analyses, align with dichotomies such as reference/sense or static/dynamic meaning.

The question then is, what form should our general $\mathcal{F}$ type constructor take? We want to have a construction that can combine all the existing ones. One can do the combining at the level of monads with the use of monad transformers, a technique pioneered by Moggi and very well-established in the Haskell programming community [52]. Simon Charlow has made the case that this technique can be exploited to great benefit in natural language semantics as well [5].

However, a competing technique has emerged in recent years. The technique goes by many names, "algebraic effects and handlers" and "extensible effects" being the most commonly used ones. This is in part due to the fact that it lies at the confluence of several research programs. This fact will allow us to present the theory from two different perspectives.

Algebraic Effects and Handlers

Hyland, Power and Plotkin have studied the problem of deriving denotational semantics of programming languages that combine different side effects [13]. In their approach, rather then modeling the individual effects using monads and combining the monads, every effect is expressed in terms of *operators* on computations. Computations thus become algebraic expressions with effects as operations and values as part of the generator set.

Let us take the example of nondeterminism. In the monadic framework, this effect is analyzed by shifting the type of denotations from α to the powerset $\mathcal{P}(\alpha)$. In the algebraic framework, a binary operator $+$

is introduced and is given meaning through a set of equations. In this case, these are the equations of a semilattice (stating the operator's associativity, commutativity and idempotence).

When the time comes to combine two effects, their signatures are summed together and their theories are combined through either a sum or a tensor (tensor differs from sum in that it adds commutativity laws for operators coming from the two different effects).

In order to fit exception handlers into their theory, Plotkin and Pretnar enriched the theory with a general notion of a *handler* [16]. A handler's purpose is to replace occurrences of an operator within a computation by another expression. This notion was shown to be very useful. Since using a handler on a computation is similar to interpreting its algebraic expression in a particular algebra, in many practical applications, the use of handlers has replaced equational theories altogether [25, 23, 24].

Extensible Effects

In the early 90's, Cartwright and Felleisen were working on the following problem. Imagine you have a simple programming language along with some denotational semantics or some other interpretation. In your simple language, numerical expressions might be interpreted as numbers. In that case, the literal number 3 would denote the number 3 and the application of the sum operator to two numerical expressions would denote the sum of their interpretations. Now imagine that you want to add mutable variables to your language. Numerical expressions no longer denote specific numbers, but rather functions from states of the variable store to both a number and an updated variable store (since expressions can now both read from and write to variables). The number 3 is thus no longer interpreted as the number 3 but as a combination of a constant function yielding the number 3 and an identity function. The addition operator now has to take care to thread the state of the memory through the evaluation of both of its arguments. In short, we are forced to give new interpretations for the entire language.

Cartwright and Felleisen proposed a solution to this problem [54, 12]. In their system, an expression can either yield a value or produce an effect. If it produces an effect, the effect percolates through the program all the way to the top, with the context that the effect projected from stored as a continuation. The effect and the continuation are then passed to an external "authority" that handles the effect, often by producing some output and passing it back to the continuation. When a new feature is added to the language, it often suffices to add a new kind of effect and introduce a new clause into the central "authority". The central authority then ends up being a collection of small modular interpreters for the various effect types. Denotation-wise, every expression can thus have a stable denotation which is either a pure value or an effect request coupled with a continuation.

Later on, this project was picked up by Kiselyov, Sabry and Swords, who, following Plotkin and Pretnar's work on handlers, proposed to break down the "authority" into the smaller constituent interpreters and have them be part of the language themselves [21].

Synthesis

In our language, values of type $\mathcal{F}(\alpha)$ can be seen either as algebraic expressions or as programs. Under the algebraic perspective, an expression is either a variable or an operator applied to some other expressions, whereas under the "extensible effects" perspective, a program is either a value or a request for an effect followed by some other programs (the continuation).

Our calculus will also have a special form for defining handlers. In the "algebraic effects and handlers" frame of mind, these can be thought of as algebras that interpret the operations within an algebraic expression. On the other hand, with "extensible effects", the intuition is more similar to that of an exception handler which intercepts requests of a certain type and decides how the computation should continue.

There is a way to avoid lifting when working with monad transformers. We characterize every monad transformer by some capabilities/operations it gives us and then we write abstract polymorphic terms which can be interpreted in different monads provided that they have enough structure to interpret the capability/operation. This is the method presented in [9, 55] and used in (Haskell) libraries implementing monad transformers [56]. However, formalizing this method already leads us half of the way towards effect and handlers (we write computations using abstract operations and the type of the computation indicates the operations that must be interpreted).

Article structure

The article is organised as follows: Section 2 gives the fundamental definitions of $(\!|\lambda|\!)$ and Section 3 shows some derivable rules. With all the equipment in place, we can now go deeper into the properties of the

calculus. We start in Section 4 with type soundness, which includes subject reduction and progress. Then we discuss the algebraic properties in Section 5, confluence in Section 6 and termination in Section 7.

2 Definitions

We start with the formal definitions of all the essential components of $(\!|\lambda|\!)$, starting with the syntax of terms, then the syntax of types. We then continue with the judgments that relate types to terms and the reduction semantics.

2.1 Terms

Let $\mathcal{X}$ be a set of variables, Σ a typed signature and $\mathcal{E}$ a set of operation symbols. The expressions of our language are comprised of the following:

variable x, where x is a variable from $\mathcal{X}$

constant c, where c is a constant from Σ

abstraction $\lambda x.\, M$, where x is a variable from $\mathcal{X}$ and M is an expression

application $M\, N$, where M and N are expressions

injection $\eta\, M$, where M is an expression

operation $\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$, where $\mathtt{op}$ is an operator from $\mathcal{E}$, x is a variable from $\mathcal{X}$ and M_{p} and M_{c} are expressions

handler $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta\, |\!)$ where $\mathtt{op}_i$ are operators from $\mathcal{E}$ and M_i and M_η are expressions

extraction $\downarrow$

exchange $\mathcal{C}$

The first four constructions — variables, constants, abstractions and applications — come directly from STLC with constants.

The next four deal with the algebraic expressions used to encode computations. Let us sketch the behaviors of these four kinds of expressions under the two readings outlined above.

Algebraic Expressions – The Denotational View

The set of algebraic expressions is generated by closing some generator set over the operations of the algebra. The η function serves to inject values from the generator set into the set of algebraic expressions. It is the constructor for the atomic algebraic expressions.

Next, for every symbol $\mathtt{op}$ in $\mathcal{E}$, we have a corresponding constructor $\mathtt{op}$ in our calculus. $\mathtt{op}$ is a constructor for algebraic expressions whose topmost operation is $\mathtt{op}$. The $\mathtt{op}$ constructor takes as argument a function that provides its operands, which are further algebraic expressions.

The banana brackets $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta\, |\!)$ contain algebras: interpretations of operators and constants. These components are combined into a catamorphism that can interpret algebraic expressions (hence the use of banana brackets [57]).[3]

The extraction function $\downarrow$, pronounced "cherry", takes an atomic algebraic expression (the kind produced by η) and projects out the element of the generator set.

[3]Since the banana brackets can contain an arbitrary number of operator clauses, we adopt the syntax of named parameters/records used in languages such as Ruby, Python or JavaScript.

Effectful Computations – The Operational View

We will now explain these constructions from the computational point of view.

The η function "returns" a given value. The result of applying it to a value x is a computation that immediately terminates and produces the value x.

The symbols from $\mathcal{E}$ become something like system calls. A computation can interrupt its execution and throw an exception with a request to perform a system-level operation. For every symbol `op` in $\mathcal{E}$, there is a constructor `op` that produces a computation which issues a request to perform the operation `op`. This constructor takes as an argument a continuation which yields the computation that should be pursued after the system-level operation `op` has been performed.

The banana brackets $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)$ describe handlers: they contain clauses for different kinds of interrupts (operation requests) and for successful computations (clause η). They behave very much like handlers in languages with resumable exceptions such as Common Lisp or Dylan.

Finally, the cherry function $\downarrow$ can take a computation that is guaranteed to be free of side effects and run it to capture its result.

The 9th construction in our calculus is the $\mathcal{C}$ operator. $\mathcal{C}$ serves as a link between the function type discussed by STLC (constructions 1–4) and the computation type introduced in our calculus (constructions 5–8). $\mathcal{C}$ is a (partial) function that takes a computation that produces a function and returns a function that yields computations. In a way, $\mathcal{C}$ makes abstracting over a variable and performing an operation commute together.[4]

We will see the utility of $\mathcal{C}$ later on. The idea came to us from a paper by Philippe de Groote [59] which tried to solve a similar problem. The name comes from the **C** combinator, which reorders the order of abstractions in a λ-term.

2.2 Types and Typing Rules

We now give a syntax for the types of $(\!|\lambda|\!)$ alongside with a typing relation. In the grammar below, ν ranges over atomic types from a set $\mathcal{T}$.

The types of our language consist of:

function $\alpha \to \beta$, where α and β are types

atom ν, where ν is an atomic type from $\mathcal{T}$

computation $\mathcal{F}_E(\alpha)$, where α is a type and E is an effect signature (defined next)

The only novelty here is the $\mathcal{F}_E(\alpha)$ computation[5] type. This type will be inhabited by effectful computations that have permission to perform the effects described in E and yield values of type α. The representation will be that of an algebraic expression with operators taken from the signature E and generators of type α.

In giving the typing rules, we will rely on the standard notion of a *context*. For us, specifically, a context is a partial mapping from the variables in $\mathcal{X}$ to the types defined above. We commonly write $\Gamma, x : \alpha$ for a context that assigns to x the type α and to other variables y the type $\Gamma(y)$. We also write $x : \alpha \in \Gamma$ to say that the context maps x to α. Note, however, that for $\Delta = \Gamma, x : \alpha, x : \beta$, we have $x : \beta \in \Delta$ while $x : \alpha \notin \Delta$.

Effect signatures are very much like contexts. They are partial mappings from the set of operation symbols $\mathcal{E}$ to pairs of types. We will write the elements of effect signatures the following way:
$\mathtt{op} : \alpha \rightarrowtail \beta \in E$ means that E maps `op` to the pair of types α and β.[6] When dealing with effect signatures, we will often make use of the disjoint union operator $\uplus$. The term $E_1 \uplus E_2$ serves as a constraint demanding that the domains of E_1 and E_2 be disjoint and at the same time it denotes the effect signature that is the union of E_1 and E_2.

The last kind of dictionary used by the type system is a standard *higher-order signature* for the constants (a map from names of constants to types). For those, we adopt the same conventions.

---

[4] This is *very* reminiscent of the idea behind Paul Blain Levy's call-by-push-value calculus [58], which treats abstracting over a variable as an effectful operation of popping a value from a stack. Using call-by-push-value could prove to be a rewarding way to refine our approach.

[5] Throughout this article, we will be using the term *computation* to mean values of type $\mathcal{F}_E(\alpha)$. Programs written in $(\!|\lambda|\!)$ are simply called terms and their normal forms are called values. To break it down, in $(\!|\lambda|\!)$, terms evaluate to values, some of which can be computations (those of an $\mathcal{F}$ type).

[6] The two types α and β are to be seen as the operation's *input* and *output* types, respectively.

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [\text{var}] \qquad \frac{c : \alpha \in \Sigma}{\Gamma \vdash c : \alpha}\ [\text{const}]$$

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.\, M : \alpha \to \beta}\ [\text{abs}] \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\, N : \beta}\ [\text{app}]$$

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash \eta\, M : \mathcal{F}_E(\alpha)}\ [\eta] \qquad \frac{\Gamma \vdash M_p : \alpha \qquad \Gamma, x : \beta \vdash M_c : \mathcal{F}_E(\gamma) \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathtt{op}\, M_p\, (\lambda x.\, M_c) : \mathcal{F}_E(\gamma)}\ [\mathtt{op}]$$

$$\frac{\Gamma \vdash M : \mathcal{F}_\emptyset(\alpha)}{\Gamma \vdash \between M : \alpha}\ [\between] \qquad \frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I} \uplus E_f \\ E' = E'' \uplus E_f \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)]_{i \in I} \\ \Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash (\!|\, (\mathtt{op}_i : M_i)_{i \in I},\ \eta : M_\eta \,|\!)\, N : \mathcal{F}_{E'}(\delta)}\ [(\!|\,|\!)]$$

$$\frac{\Gamma \vdash M : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash \mathcal{C}\, M : \mathcal{F}_E(\alpha \to \beta)}\ [\mathcal{C}]$$

Figure 1: The typing rules for $(\!|\lambda|\!)$.

In our typing judgments, contexts will appear to the left of the turnstile and they will hold information about the *statically* (lexically) bound variables, as in STLC. Effect signatures will appear as indices of computation types and they will hold information about the operations that are *dynamically* bound by handlers. Finally, there will be a single higher-order signature that will globally characterize all the available constants.

The typing judgments are presented in Figure 1. Metavariables M, $N\ldots$ stand for expressions, α, β, $\gamma\ldots$ stand for types, Γ, $\Delta\ldots$ stand for contexts, $\mathtt{op}$, $\mathtt{op}_i$ stand for operation symbols and E, $E'\ldots$ stand for effect signatures. Σ refers to the higher-order signature giving types to constants.

The typing rules mirror the syntax of expressions. Again, the first four rules come from STLC. The next four deal with introducing pure computations, enriching them with effectful operations, handling those operations away and finally eliminating pure computations. The $\mathcal{C}$ rule lets us start to see what we meant by saying that the $\mathcal{C}$ operator lets the function type and the computation type commute.

Let us ponder the types of the new constructions so as to get a grip on the interface that the calculus provides us for dealing with computations.

[η]

First off, we have the η operator. It takes a value of type α and injects it into the type $\mathcal{F}_E(\alpha)$. The metavariable E is free, meaning η can take values of type α to type $\mathcal{F}_E(\alpha)$ for any E. The algebraic intuition would say that elements of the generator set are valid algebraic expressions independent of the choice of signature. Computationally, returning a value is always an option, independently of the available permissions.

[op]

More complicated computations can be built up by extending existing computations using the *operation* construction. Let us have an effect signature E such that $\mathtt{op} : \alpha \rightarrowtail \beta \in E$. To use $\mathtt{op}$, we first apply it to a value of the input type α and to a continuation. The continuation is a function of type $\beta \to \mathcal{F}_E(\gamma)$ that accepts a value of the output type β (the result of performing the operation) and chooses in return a computation that should be pursued next. The return type of our new computation will thus be the return type γ of the computation provided by the continuation. The continuation's computation and the new extended computation will also share the same effect signature E. This means that all uses of the operation $\mathtt{op}$ within the created computation have the same input and output types.[7]

[7]In general, the same operation symbol can be used with different input and output types, in computations whose types are indexed by different effect signatures.

There is a parallel between the [var] rule and the [op] rule. The [var] rule lets us use a symbol x with type α provided $x : \alpha \in \Gamma$. The [op] rule lets us use a symbol op with type $\alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma)$ provided $\texttt{op} : \alpha \rightarrowtail \beta \in E$. The crucial difference is that contexts (Γ) are components of judgments whereas effect signatures (E) are components of types. The meaning of a variable is determined by inspecting the expression in which it occurs and finding the λ that binds it (this is known as *lexical* or *static binding*). On the other hand, the meaning of an operation in a computation is determined by evaluating the term in which the computation appears until the computation becomes the argument of a handler. This handler will then give meaning to the operation symbol by substituting it with a suitable interpretation (this kind of *late* binding is known as *dynamic binding*).

We have now seen how to construct pure computations using η and extend them by adding operations. However, before we go on and start talking about handlers, we would like to give the algebraic intuition behind [op], as the algebraic point of view makes explaining the handler rule [$(\!|\ |\!)$] easier.

We can see the effect signature as an algebraic signature. For every $\texttt{op} : \alpha \rightarrowtail \beta \in E$, we have an α-indexed family of operators of arity β. Let's unpack this statement.

- First, there is the matter of having an indexed family of operators. A common example of these is the case of scalar multiplication in the algebra of a vector space. A *single-sorted algebraic signature* is a set of operation symbols, each of which is given an arity (a natural number). For vector addition, the arity is 2, since vector addition acts on two vectors (two elements of the domain). Scalar multiplication acts on one scalar and one vector. However, neither arity 1 nor arity 2 adequately express this. We can get around the limitations of a single-sorted signature by introducing for every scalar k an operation of arity 1 that corresponds to multiplying the vector by k. Scalar multiplication is therefore not a single operator but a scalar-indexed family of operators.

  The very same strategy is applied here as well. A single operation symbol doesn't need to map to a single operator but can instead map to (possibly infinitely) many operators indexed by values of some type α. For example, writing messages to the program's output ($\texttt{print} : \mathit{string} \rightarrowtail 1$) can be seen as a string-indexed family of unary operators on computations. For every string s, we get an operator that maps computations c to computations that first print s and then continue as c.

- Next, we were speaking about operators of arity β. The use of a type in place of a numerical arity is due to a certain generalization. In set theory, natural numbers become sets that have the same cardinality as the number they represent ($|N| = N$). We can therefore conservatively generalize the idea of arity to a set by saying that an operator of arity X takes one operand per each element of the set X. It's a short step from there to using types as arities, wherein an operator of arity β takes one operand per possible value of type β.

  This will come in very handy in our system. We want our operator op to have as many operands as there are possible values in the output type β. Therefore, we simply say that the operator has arity β.

  How do we write down the application of an operator of arity β to its operands? We can no longer just list out all the operands, since types in $(\!|\lambda|\!)$ may have an unbounded number of inhabitants. We will organize operands in *operand clusters*,[8] arity-indexed families of operands. We will write them down as functions, using λ-abstraction, from the arity type β to some operand type, e.g., $\mathcal{F}_E(\gamma)$.

Now we can understand what it means to say that $\texttt{op} : \alpha \rightarrowtail \beta \in E$ gives rise to an α-indexed family of operators of arity β. We apply to op an index of type α to get an operator and then we apply that operator to an operand cluster of type $\beta \to \mathcal{F}_E(\gamma)$ to get a new expression of type $\mathcal{F}_E(\gamma)$.

We suggest visualizing these algebraic expressions as trees (see Section 2 of [60] for the original idea). Trees of type $\mathcal{F}_E(\alpha)$ consist of leafs containing values of type α and internal nodes labelled with operations and their parameters. Every internal node is labelled with some $\texttt{op} : \alpha \rightarrowtail \beta \in E$ and with a parameter of type α and it has a cluster of children indexed by β.

[$(\!|\ |\!)$]

Now we are ready to explain the handler rule. The typing rule for $(\!|\ |\!)$ is repeated in Figure 2.

[8] Our use of the word *cluster* is synonymous with the mathematical term *family*. We will be using the term *cluster* for families of computations passed to operations and handlers.

$$\frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I} \uplus E_f \\ E' = E'' \uplus E_f \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)]_{i \in I} \\ \Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash (\!| (\mathtt{op}_i : M_i)_{i \in I},\ \eta : M_\eta |\!)\, N : \mathcal{F}_{E'}(\delta)} \; [(\!|\,|\!)]$$

Figure 2: The typing rule for the handler construction.

To illustrate the constraints on the types of the components, M_i and M_η, of a handler, we will examine its semantics. The handler processes the algebraic expression N by recursive induction. Depending on the shape of the expression, one of the following will happen:

- If $N = \eta\, N'$, then N' is of type γ. This is where the M_η function comes in. It must take a value of type γ and produce a new tree of type $\mathcal{F}_{E'}(\delta)$, hence the fourth hypothesis of the $(\!|\,|\!)$ rule.
- If $N = \mathtt{op}_i\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})$ for some $i \in I$, then N_p must be of type α_i. Furthermore, for every $x : \beta_i$, we have an operand $N_\mathrm{c} : \mathcal{F}_E(\gamma)$. We know this since N is of type $\mathcal{F}_E(\gamma)$ and the first hypothesis tells us that $\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i \in E$.

  We will recursively apply our handler to the cluster of operands, changing their type from $\mathcal{F}_E(\gamma)$ to $\mathcal{F}_{E'}(\delta)$. We now need something which takes N_p, whose type is α_i, and the cluster of processed operands, type $\beta_i \to \mathcal{F}_{E'}(\delta)$, which is exactly the function M_i in the third hypothesis of the $(\!|\,|\!)$ rule.
- If $N = \mathtt{op}\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})$ and $\mathtt{op} : \alpha \rightarrowtail \beta \in E_f$ for some α and β, then we will ignore the node and process only its children. This means that the resulting expression will contain the operation symbol `op` from E_f,[9]. In order for such an expression to be of the desired type $\mathcal{F}_{E'}(\delta)$, E_f must be included in E', which is what the second hypothesis of the $(\!|\,|\!)$ rule guarantees.

We have covered the whole $(\!|\,|\!)$ rule, except for the presence of the effect signature E''. It serves two roles.

- First of all, it acts as a "free" variable over effect signatures. This means that we can give any effect signature E' to the type $\mathcal{F}_{E'}(\delta)$ of the resulting computation N' as long as E' contains E_f (E'' represents the relative complement of E_f in E'). This is in analogy to the free effect variable E in the [η] and [`op`] rules. This freedom of effect variables is a way of implementing the idea that a computation of type $\mathcal{F}_{E_1}(\alpha)$ can be used anywhere that a computation of type $\mathcal{F}_{E_2}(\alpha)$ is needed given that $E_1 \subseteq E_2$.
- In the previous paragraph, why did we put the word "free" in quotation marks? Because the effect variable E'' is not actually free. It is the complement of E_f in E' and E' is constrained by the types of M_i and M_η in the third and fourth hypotheses, respectively. The handler's clauses might themselves introduce new effects, which will in turn translate into constraints on E' and E''. This happens when a handler interprets an operation by making an appeal to some other operation (e.g. a handler could interpret computations using n-ary choice into computations using binary choice).

  As the simplest example, we can take a handler that replaces one operation symbol with another, $(\!|\, \mathtt{old} : (\lambda pc.\, \mathtt{new}\, p\, (\lambda y.\, c\, y)),\ \eta : (\lambda x.\, \eta\, x) \,|\!)$. The type scheme corresponding to the term is $\mathcal{F}_{\{\mathtt{old}:\alpha \rightarrowtail \beta\} \uplus E_f}(\gamma) \to \mathcal{F}_{\{\mathtt{new}:\alpha \rightarrowtail \beta\} \uplus E^+ \uplus E_f}(\gamma)$. In this scheme, α, β and γ are free meta-variables ranging over types and E_f and E^+ range over effect signatures. The E'' of the $(\!|\,|\!)$ rule corresponds to $\{\mathtt{new} : \alpha \rightarrowtail \beta\} \uplus E^+$ (i.e. E'' is not free, it must contain `new`). The handler has eliminated the `old` effect but it has also introduced the `new` effect.

This concludes our exploration of the $(\!|\,|\!)$ rule. We have explained it in terms of algebraic expressions and trees, using the denotational intution. We will develop the operational intuition, which talks about handlers in terms of computations and continuations, in Section 2.3, where we will give the semantics of our language using reduction rules.

[9] The f in E_f stands for *forwarded effects* since it refers to effects that the handler will not interpret but instead forward to some other interpreter. The notation comes from a similar rule in the λ_{eff} calculus [23].

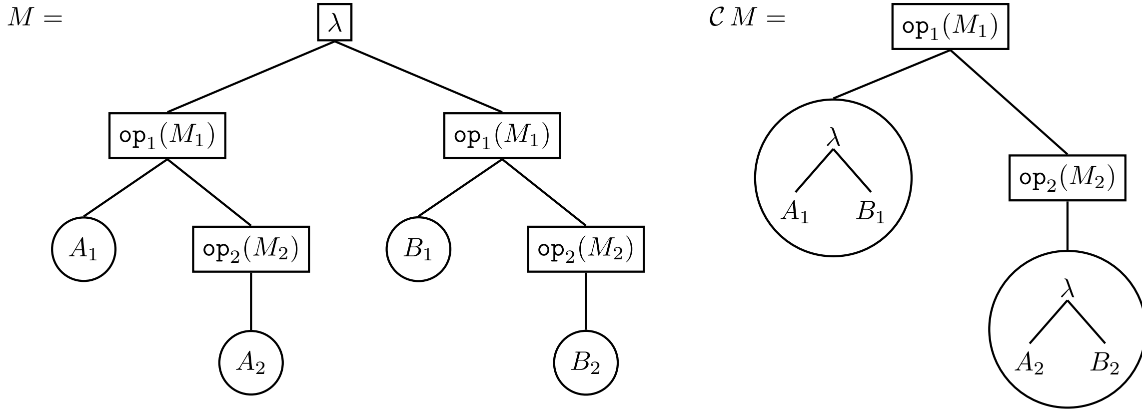

Figure 3: Example of applying the $\mathcal{C}$ operator to a term.

[cherry]

Next up is the cherry operator, $\between$. Its type is $\mathcal{F}_{\emptyset}(\alpha) \to \alpha$ and it serves as a kind of dual to the η opertator, an elimination for the $\mathcal{F}$ type.

The type $\mathcal{F}_{\emptyset}(\alpha)$ demands that the effect signature be empty. In such a case, the tree has no internal nodes and is composed of just a leaf containing a value of the type α. The $\between$ operator serves to extract that value.

Another way to look at it is to say that a computation of type $\mathcal{F}_{\emptyset}(\alpha)$ cannot perform any "unsafe" operations and it is therefore always safe to execute it and get the resulting value of type α.

[$\mathcal{C}$]

Finally, we take a look at the $\mathcal{C}$ operator. The type of the operator is $(\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\alpha \to \beta)$. Its input is an α-indexed family of computations and its output is a computation of α-indexed families. The operator applies only in the case when all the computations in the family share the same *internal structure*. By sharing the same internal structure, we mean that the trees can only differ in their leaves. What the $\mathcal{C}$ operator then does is to push the λ-binder down this common internal structure into the leaves. This way, we can evaluate/handle the common operations without commiting to a specific value of type α.

The action of the $\mathcal{C}$ operator is illustrated in Figure 3. Here, M is a function of some two-value type. It maps one value to the left subtree of λ (with leaves A_i) and the other value to the right subtree (leaves B_i). Both subtrees correspond to computations, in which a box is an operation and a circle is an atomic expression (i.e. a return value). Furthermore, $\mathtt{op}_1$ has a two-value output type (arity 2) and $\mathtt{op}_2$ has a one-value output type (arity 1).

Since the operations $\mathtt{op}_1$ and $\mathtt{op}_2$ and their arguments M_1 and M_2 are the same in both subtrees, and thus independent of the value passed to the λ, we can apply the $\mathcal{C}$ operator. The $\mathcal{C}$ pulls this common structure out of the λ and gives us a computation that produces a function.

We can also explain the action of $\mathcal{C}$ in operational terms. As in call-by-push-value [58], we can think of abstraction over α as some effectful operation that tries to pop a value $x : \alpha$ off a stack. The input of $\mathcal{C}$ can then be seen as a continuation waiting for this x and wanting to perform some further operations. $\mathcal{C}$ assumes that the continuation performs operations independently of x[10] and it can thus postpone popping x off the stack until after the operations dictated by the continuation have been evaluated. $\mathcal{C}$ is therefore a kind of commutativity law for operations and abstractions (the popping of a value off the operand stack): as long as one does not depend on the other, it does not matter whether we first perform an operation and then abstract over an argument or whether we do so the other way around.[11]

[10]Violating this assumption will yield terms which get stuck during evaluation (we will see the partial reduction rules in Section 2.3). Sam Lindley presented a refined type system for a similar calculus to track the use of variables [60]. A similar refinement should be possible in our case as well but it would obscure the already dense type notation.

[11]The other direction, typed $\mathcal{F}_E(\alpha \to \beta) \to (\alpha \to \mathcal{F}_E(\beta))$, is already possible without introducing a special operator since $\mathcal{F}_E$ is a functor.

2.3 Reduction Rules

We will now finally give a semantics to $(\!|\lambda|\!)$. The semantics will be given in the form of a reduction relation on terms. Even though the point of the calculus is to talk about effects, the reduction semantics will have no notion of order of evaluation; any reducible subexpression can be reduced in any context.

Before we dive into the reduction rules proper, we will first have to handle some formal paperwork, most of it due to the fact that we use variables and binders in our calculus. In order to quotient out the irrelevant distinction between terms that are the same up to variable names, we will introduce a series of definitions leading up to a notion of α-equivalence.

Definition 2.1. ***Evaluation contexts*** *are terms with a hole (written as* $[]$*) inside. They are formally defined by the following grammar.*

$$
\begin{aligned}
C ::=\ & [] \\
|\ & \lambda x.\, C \\
|\ & C\, N \\
|\ & M\, C \\
|\ & \mathtt{op}\, C\, (\lambda x.\, M_{\mathrm{c}}) \\
|\ & \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, C) \\
|\ & \eta\, C \\
|\ & (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)\, C \\
|\ & (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_i \colon C,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)\, N \\
|\ & (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_i \colon M_i,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon C \,|\!)\, N \\
|\ & \downarrow C \\
|\ & \mathcal{C}\, C
\end{aligned}
$$

Definition 2.2. *Let $\sim$ be a binary relation on the terms of $(\!|\lambda|\!)$. We define the relation $[\sim]$, called the* ***context closure*** *of $\sim$, as the smallest relation that contains $\sim$ and satisfies the following closure property for any evaluation context C:*

- *if $M\,[\sim]\,M'$, then $C[M]\,[\sim]\,C[M']$*

We will be defining relations on the terms of $(\!|\lambda|\!)$ that will correspond to different transformations (such as **swap** and the reduction rules). The notion of context closure will allow us to say that a term can be transformed by transforming any of its parts.

We are now at a point where we can easily lay down the reduction rules for $(\!|\lambda|\!)$. A reduction rule ξ will be a relation on terms. Most of the time, we will deal with their context closures, $[\xi]$, for which we will also adopt the notation $\to_\xi$. We will also use the notation $\to_{\xi_1,\ldots,\xi_n}$ for the composition $\to_{\xi_n} \circ \cdots \circ \to_{\xi_1}$. The *reduction relation* $\to$ of $(\!|\lambda|\!)$ is the union of the $\to_\xi$ relations for every reduction rule ξ. We will also use the symbols $\twoheadrightarrow$ and $\twoheadleftrightarrow$ to stand for the reflexive-transitive and reflexive-symmetric-transitive closures of $\to$, respectively. If $M \twoheadleftrightarrow N$, we will also say that M and N are *convertible*. A term which is not reducible to some other term is said to be in *normal form*.

We will now go through the reduction rules of $(\!|\lambda|\!)$, presented in Figure 4, one by one.

First off, we have the β and η rules. By no coincidence, they are the same rules as the ones found in STLC.

Next we have the three rules that govern the behavior of handlers. We recognize the three different rules as the three different cases in the informal denotational semantics given in Section 2.2.

- When the expression is just an atom (i.e. $\eta\, N$), rule $(\!|\eta|\!)$ applies the clause M_η to the value N contained within.
- When the expression is an operation $\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}}(x))$ with $j \in I$, we first recursively apply the handler to every child $N_{\mathrm{c}}(x)$. We then pass the parameter N_{p} stored in the node along with the cluster of the processed children to the clause M_j.
- When the expression is an operation $\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}}(x))$ but where $j \notin I$, we leave the node as it is and just recurse down on to the subexpressions (effectively using $\mathtt{op}_j$ as the handler clause for $\mathtt{op}_j$).

$$(\lambda x.\, M)\, N \to M[x \coloneqq N] \qquad \text{rule } \beta$$

$$\lambda x.\, M\, x \to M \qquad \text{rule } \eta \quad \text{where } x \notin \mathrm{FV}(M)$$

$$(\!|\, (\mathtt{op}_i {:}\, M_i)_{i \in I},\ \eta {:}\, M_\eta \,|\!)\, (\eta\, N) \to M_\eta\, N \qquad \text{rule } (\!|\, \eta \,|\!)$$

$$(\!|\, (\mathtt{op}_i {:}\, M_i)_{i \in I},\ \eta {:}\, M_\eta \,|\!)\, (\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) \to M_j\, N_{\mathrm{p}}\, (\lambda x.\, (\!|\, (\mathtt{op}_i {:}\, M_i)_{i \in I},\ \eta {:}\, M_\eta \,|\!)\, N_{\mathrm{c}}) \qquad \text{rule } (\!|\, \mathtt{op} \,|\!) \quad \text{where } j \in I \text{ and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta)$$

$$(\!|\, (\mathtt{op}_i {:}\, M_i)_{i \in I},\ \eta {:}\, M_\eta \,|\!)\, (\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) \to \mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, (\!|\, (\mathtt{op}_i {:}\, M_i)_{i \in I},\ \eta {:}\, M_\eta \,|\!)\, N_{\mathrm{c}}) \qquad \text{rule } (\!|\, \mathtt{op}' \,|\!) \quad \text{where } j \notin I \text{ and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta)$$

$$\text{\textcherry}\, (\eta\, M) \to M \qquad \text{rule \textcherry}$$

$$\mathcal{C}\, (\lambda x.\, \eta\, M) \to \eta\, (\lambda x.\, M) \qquad \text{rule } \mathcal{C}_\eta$$

$$\mathcal{C}\, (\lambda x.\, \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}})) \to \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}})) \qquad \text{rule } \mathcal{C}_{\mathtt{op}} \quad \text{where } x \notin \mathrm{FV}(M_{\mathrm{p}})$$

Figure 4: The reduction rules of $(\!|\, \lambda \,|\!)$.

By looking at these rules, we also notice that all they do is just traverse the continuations (N_{c}) and replace η with M_η and $\mathtt{op}_i$ with M_i. This justifies thinking of η and the operation symbols as special variables which are bound to a value when being passed through a handler. This substitutability is already hinted at by the types of M_η and M_i in the $(\!|\ |\!)$ typing rule and their correspondence with the typing rules $[\eta]$ and $[\mathtt{op}]$, respectively.

The next rule talks about the cherry operator. It does what we would expect it to do.[12] It expects its argument to always be an atomic algebraic expression, a pure computation, and it extracts the argument that was passed to the η constructor.

Finally, we have the two rules defining the behavior of $\mathcal{C}$. We remind ourselves that the goal of $\mathcal{C}$ is to make computations and abstractions commute by pushing λ below operation symbols and η.

- Rule $\mathcal{C}_\eta$ treats the base case where the computation that we try to push λ through is a pure computation. In that case, we just reorder the λ binder and the η operator.
- Rule $\mathcal{C}_{\mathtt{op}}$ deals with the case of the λ meeting an operation symbol. The solution is to push the $\mathcal{C}$ operator down through the continuation. The operation $\mathcal{C}\, (\lambda x.\, \dots)$ is applied recursively to every child $M_{\mathrm{c}}\, (y)$. However, this strategy is sound only when M_{p} has no free occurrence of x (which would have been bound by the λ in the redex but would become unbound in the contractum). We therefore have a constraint saying that x must not occur free in M_{p}. Unlike the other freshness constraints, this one cannot be fixed by a simple renaming of variables. If this constraint is not met, the $\mathcal{C}$ will not be able to reduce.

  When talking about the $\mathcal{C}$ operator in Section 2.2, we talked about how it applies only to families of computations that share the same internal structure (i.e. functions of x where the internal structure does not depend on x). This is reflected in the reduction rules in two ways:

  - Firstly, in order for $\mathcal{C}_{\mathtt{op}}$ to kick in, the body of the function must have already reduced to something

[12] Based on what we said about it in Section 2.2, not on its name.

of the form $\mathtt{op}\, M_\mathrm{p}\, M_\mathrm{c}$. This means that the next operation to be performed has already been determined to be $\mathtt{op}$ without needing to wait for the value of x.

- Secondly, the reduction can only proceed if M_p does not contain a free occurrence of x. This means that M_p is independent of x.

2.4 Common Combinators

Here we will introduce a collection of useful syntactic shortcuts and combinators for our calculus.

2.4.1 Composing Functions and Computations

First of all, to save some space and write functions in a terse "point-free" style, we introduce the composition operator (known as the **B** combinator in combinatory logic).

$$\begin{aligned} _ \circ _ &: (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) \\ f \circ g &= \lambda x.\, f\,(g\,x) \end{aligned}$$

We will also "functionalize" our term constructors (i.e. we will write η as a shortcut for $(\lambda x.\, \eta\, x)$). Our motive in not defining our symbols directly as function constants in the core calculus is due to the proofs of confluence and termination. A complete list of functionalized symbols is given below.[13]

$$\begin{aligned} &\eta : \alpha \to \mathcal{F}_E(\alpha) \\ &\eta = \lambda x.\, \eta\, x \\ &\mathtt{op} : \alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma) \\ &\mathtt{op} = \lambda pc.\, \mathtt{op}\, p\, (\lambda x.\, c\, x) \\ &(\!|\, (\mathtt{op}_i{:}\, M_i)_{i \in I},\ \eta{:}\, M_\eta \,|\!) : \mathcal{F}_E(\gamma) \to \mathcal{F}_{E'}(\delta) \\ &(\!|\, (\mathtt{op}_i{:}\, M_i)_{i \in I},\ \eta{:}\, M_\eta \,|\!) = \lambda x.\, (\!|\, (\mathtt{op}_i{:}\, M_i)_{i \in I},\ \eta{:}\, M_\eta \,|\!)\, x \\ &\downarrow : \mathcal{F}_\emptyset(\alpha) \to \alpha \\ &\downarrow = \lambda x.\, \downarrow x \\ &\mathcal{C} : (\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\alpha \to \beta) \\ &\mathcal{C} = \lambda f.\, \mathcal{C}\, f \end{aligned}$$

$$\begin{aligned} &\pi_1 : (\alpha \times \beta) \to \alpha \\ &\pi_1 = \lambda P.\, \pi_1\, P \\ &\pi_2 : (\alpha \times \beta) \to \beta \\ &\pi_2 = \lambda P.\, \pi_2\, P \\ &\mathrm{inl} : \alpha \to (\alpha + \beta) \\ &\mathrm{inl} = \lambda x.\, \mathrm{inl}\ x \\ &\mathrm{inr} : \beta \to (\alpha + \beta) \\ &\mathrm{inr} = \lambda x.\, \mathrm{inr}\ x \end{aligned}$$

Later on, in Section 5, we will see that our $\mathcal{F}_E$ is a functor which, combined with some other elements, forms a monad, or equivalently, a Kleisli triple. We use a star to denote the *extension* of a function from values to computations, as in [52], and we use $\gg\!=$ to denote the *bind* of a monad, as in Haskell.[14]

$$\begin{aligned} _^{*} &: (\alpha \to \mathcal{F}_E(\beta)) \to (\mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)) \\ f^{*} &= (\!|\, \eta{:}\, f \,|\!) \\ _ \gg\!= _ &: \mathcal{F}_E(\alpha) \to (\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\beta) \\ M \gg\!= N &= N^{*}\, M \end{aligned}$$

Finally, we will define a notation for applying infix operators to arguments wrapped inside computations. Let $_ \circledcirc _$ be an infix operator of type $\alpha \to \beta \to \gamma$. Then we define the following:

[13] The type and effect signature metavariables that appear in the types are bound by the typing constraints of the terms on the right-hand side.

[14] In the types of the operators given below, we use the same effect signature E everywhere. Technically, a more general type could be derived for these terms given our system. However, we will rarely need this extra flexibility and so we stick with these simpler types.

$$
\begin{aligned}
_ \ll\!\circledcirc _ &: \mathcal{F}_E(\alpha) \to \beta \to \mathcal{F}_E(\gamma) \\
X \ll\!\circledcirc\, y &= X \gg\!= (\lambda x.\, \eta\,(x \circledcirc y)) \\
_ \circledcirc\!\gg _ &: \alpha \to \mathcal{F}_E(\beta) \to \mathcal{F}_E(\gamma) \\
x \circledcirc\!\gg Y &= Y \gg\!= (\lambda y.\, \eta\,(x \circledcirc y)) \\
_ \ll\!\circledcirc\!\gg _ &: \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \to \mathcal{F}_E(\gamma) \\
X \ll\!\circledcirc\!\gg Y &= X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\,(x \circledcirc y)))
\end{aligned}
$$

In particular, we will be using this notation for the function application operator:

$$
\begin{aligned}
_ \cdot _ &: (\alpha \to \beta) \to \alpha \to \beta \\
f \cdot x &= f\, x
\end{aligned}
$$

which will yield the following combinators:

$$
\begin{aligned}
_ \ll\!\cdot _ &: \mathcal{F}_E(\alpha \to \beta) \to \alpha \to \mathcal{F}_E(\beta) \\
F \ll\!\cdot\, x &= F \gg\!= (\lambda f.\, \eta\,(f\, x)) \\
_ \cdot\!\gg _ &: (\alpha \to \beta) \to \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \\
f \cdot\!\gg X &= X \gg\!= (\lambda x.\, \eta\,(f\, x)) \\
_ \ll\!\cdot\!\gg _ &: \mathcal{F}_E(\alpha \to \beta) \to \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \\
F \ll\!\cdot\!\gg X &= F \gg\!= (\lambda f.\, X \gg\!= (\lambda x.\, \eta\,(f\, x)))
\end{aligned}
$$

The first of the three, $\ll\!\cdot$, is the inverse function to $\mathcal{C}$. The second, $\cdot\!\gg$, is the morphism component of the $\mathcal{F}_E$ functor (which we will demonstrate in Section 5). $\mathcal{F}_E$ is also an applicative functor [61] and the third operator in the list above, $\ll\!\cdot\!\gg$, is the operator for application within the functor.

We also need to introduce some shortcuts to allow us to proceed faster and at a higher level of abstraction. For exemple a regular pattern is a sequences of $\gg\!=$ and $*$. We note $\eta.\!\gg\!=$ this rule.

2.4.2 Operations and Handlers

Now we will look at syntactic sugar specific to $(\!|\,\lambda\,|\!)$. In 2.4.1, we have seen the bind operator $\gg\!=$ and other ways of composing computations. Since we now have a practical way to compose computations, we can simplify the way we write effectful operations.

$$
\mathsf{op!} = \lambda p.\, \mathsf{op}\, p\, (\lambda x.\, \eta\, x)
$$

The exclamation mark partially applies an operation by giving it the trivial continuation η. However, we can still recover op from $\mathsf{op!}$ using $\gg\!=$:

$$
\begin{aligned}
\mathsf{op!}\, p \gg\!= k &= (\lambda p.\, \mathsf{op}\, p\, (\lambda x.\, \eta\, x))\, p \gg\!= k \\
&\to_\beta \mathsf{op}\, p\, (\lambda x.\, \eta\, x) \gg\!= k \\
&= k^*\, (\mathsf{op}\, p\, (\lambda x.\, \eta\, x)) \\
&= (\!|\, \eta \colon k \,|\!)\, (\mathsf{op}\, p\, (\lambda x.\, \eta\, x)) \\
&\to_{(\!|\,\mathsf{op}'\,|\!)} \mathsf{op}\, p\, (\lambda x.\, (\!|\, \eta \colon k \,|\!)\, (\eta\, x)) \\
&\to_{(\!|\,\eta\,|\!)} \mathsf{op}\, p\, (\lambda x.\, k\, x)
\end{aligned}
$$

The exclamation mark streamlines the typing rule for operations, as it is presented on the pair of rules below:

$$
\frac{\mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathsf{op} : \alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma)}\ [\mathsf{op}]
\qquad\qquad
\frac{\mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathsf{op!} : \alpha \to \mathcal{F}_E(\beta)}\ [\mathsf{op!}]
$$

We can also see that the $\rightarrowtail$ arrow used in effect signatures gives rise to a Kleisli arrow since $\mathcal{F}_E$ is a monad.

Handlers

In Section 2.3, we have seen how the reduction rules treat unknown operation symbols: by leaving them intact. With some syntactic sugar, we can extend this behavior to the η operator as well. We will sometimes write a handler and omit giving the η clause. In that case, the η clause is presumed to be just η.[15] Schematically, we can define this piece of new syntax in the following way:

$$(\!| (\mathtt{op}_i \colon M_i)_{i \in I} |\!) = (\!| (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon \eta |\!)$$

Finally, we will introduce a special syntax for *closed handlers* [23]. A *closed handler* is a handler that interprets the entire computation that is given as its input (it must have a clause for every operator that appears within). Since all effects are handled and none are forwarded, the codomain of the handler can be something else than a computation. However, trying to write handlers that want to exploit this possibility, implies that there is a lot of translating between α types and $\mathcal{F}_\emptyset(\alpha)$ types that needs to be done and that clouds the inherent simplicity of a closed handler. We introduce syntax for closed handlers which takes care of this problem.

$$\boldsymbol{(\!|} (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \boldsymbol{|\!)}\, N = \downharpoonleft ((\!| (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downharpoonleft \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, (M_\eta\, x)) |\!)\, N)$$

As we can see, the only material added by the closed handler brackets are the functions η and $\downharpoonleft$,[16] which simply translate between the types α and $\mathcal{F}_\emptyset(\alpha)$. Rather than closely studying the definition and scrutinizing the etas and the cherries, the idea of a closed handler is better conveyed by giving its typing rule. The following rule will be proven sound in 3.3:

$$\frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I} \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta]_{i \in I} \\ \Gamma \vdash M_\eta : \gamma \to \delta \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash \boldsymbol{(\!|} (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \boldsymbol{|\!)}\, N : \delta}\ [\boldsymbol{(\!|\,|\!)}]$$

The lack of multiple effect signatures to implement effect forwarding makes the $\boldsymbol{(\!|\,|\!)}$ rule simpler than the one of $(\!|\,|\!)$:

- M_η gives δ-typed interpretations to the terminal values of type γ
- M_i maps the parameter of type α_i and the δ-typed interpretations of the β_i-indexed family of children to a δ-typed interpretation of an internal node

The η clause in a closed handler is optional. Similarly to open handlers, we will assume that $\gamma = \delta$ and that M_η is the identity function. This is the same as saying that a closed handler without an η clause is translated into an open handler without an η clause.

$$\begin{aligned}
\boldsymbol{(\!|} (\mathtt{op}_i \colon M_i)_{i \in I} \boldsymbol{|\!)}\, N &= \downharpoonleft ((\!| (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downharpoonleft \circ k))))_{i \in I} |\!)\, N) \\
&= \downharpoonleft ((\!| (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downharpoonleft \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, x) |\!)\, N) \\
&= \downharpoonleft ((\!| (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downharpoonleft \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, ((\lambda x.\, x)\, x)) |\!)\, N) \\
&= \boldsymbol{(\!|} (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon (\lambda x.\, x) \boldsymbol{|\!)}\, N
\end{aligned}$$

3 Derived Rules

At the end of Section 2, in 2.4, we have introduced some new syntax for $(\!| \lambda |\!)$ terms and we have translated that syntax into terms of the core $(\!| \lambda |\!)$ calculus. In this section, we will give typing rules and reduction rules to these new constructions and prove their correctness.

[15] This is not a part of the core calculus as this is only sound when the type of the handler is of the shape $\mathcal{F}_E(\gamma) \to \mathcal{F}_{E'}(\gamma)$ (i.e. when the handler preserves the type γ of values returned by the computation).

[16] Composing the $\downharpoonleft$ operator with a $(\!|\,|\!)$ handler is an identifying characteristic of closed handlers: a closed handler is a banana with a cherry on the top.

3.1 Function Composition ($\circ$)

The first piece of syntactic sugar we have introduced was an infix symbol for function composition.

$$f \circ g = \lambda x.\, f\,(g\,x)$$

In order to type terms containing this symbol, it will be useful to have a typing rule.

Proposition 3.1. *The following typing rule is derivable in* $(\!|\lambda|\!)$*:*

$$\frac{\Gamma \vdash M : \beta \to \gamma \qquad \Gamma \vdash N : \alpha \to \beta}{\Gamma \vdash M \circ N : \alpha \to \gamma}\,[\circ]$$

Proof. Since $M \circ N = \lambda x.\, M\,(N\,x)$, we can prove the validity of this rule with the typing rule below:

$$\dfrac{\dfrac{\Gamma, x : \alpha \vdash M : \beta \to \gamma \qquad \dfrac{\Gamma, x : \alpha \vdash N : \alpha \to \beta \qquad \Gamma, x : \alpha \vdash x : \alpha}{\Gamma, x : \alpha \vdash N\,x : \beta}\,[\text{app}]}{\Gamma, x : \alpha \vdash M\,(N\,x) : \gamma}\,[\text{app}]}{\Gamma \vdash \lambda x.\, M\,(N\,x) : \alpha \to \gamma}\,[\text{abs}]$$

x is presumed to be fresh for M and N and so we can equate $\Gamma, x : \alpha \vdash M : \beta \to \gamma$ with $\Gamma \vdash M : \beta \to \gamma$ and the same for N. □

The result of function composition is another function and functions can be applied to arguments. We can derive a reduction rule for this kind of function.

Proposition 3.2. *The following reduction is derivable in* $(\!|\lambda|\!)$*:*

$$(M_1 \circ M_2)\,N \quad \to_\circ \quad M_1\,(M_2\,N)$$

Proof.

$$\begin{aligned}(M_1 \circ M_2)\,N &= (\lambda x.\, M_1\,(M_2\,x))\,N \\ &\to_\beta M_1\,(M_2\,N)\end{aligned}$$

□

3.2 Monadic Bind ($\gg\!=$)

As a reminder, we give the definition of $\gg\!=$ from 2.4.1.

$$\begin{aligned}M \gg\!= N &= N^*\,M \\ &= (\!|\,\eta\!: N\,|\!)\,M\end{aligned}$$

First, we will prove the correct typing for $\gg\!=$.

Proposition 3.3. *The following typing rule is derivable in* $(\!|\lambda|\!)$*:*

$$\frac{\Gamma \vdash M : \mathcal{F}_E(\alpha) \qquad \Gamma \vdash N : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash M \gg\!= N : \mathcal{F}_E(\beta)}\,[\gg\!=]$$

Proof. We note that $M \gg\!= N = (\!|\,\eta\!: N\,|\!)\,M$ and construct the following typing derivation in $(\!|\lambda|\!)$:

$$\dfrac{\dfrac{\Gamma \vdash N : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash (\!|\,\eta\!: N\,|\!) : \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)}\,[(\!|\;|\!)] \qquad \Gamma \vdash M : \mathcal{F}_E(\alpha)}{\Gamma \vdash (\!|\,\eta\!: N\,|\!)\,M : \mathcal{F}_E(\beta)}\,[\text{app}]$$

□

Next, we will prove the validity of two reduction rules for $\gg\!=$.

Proposition 3.4. *The following reductions are derivable in* $(\!|\lambda|\!)$*:*

$$
\begin{aligned}
\eta\, M \gg\!= N &\quad \to_{\eta.\gg=} \quad N\, M \\
\mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, M_{\mathrm{c}}) \gg\!= N &\quad \to_{\mathtt{op}.\gg=} \quad \mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, M_{\mathrm{c}} \gg\!= N)
\end{aligned}
$$

Proof.

$$
\begin{aligned}
\eta\, M \gg\!= N &= (\!|\, \eta{:}\, N \,|\!)\, (\eta\, M) \\
&\to_{(\!|\eta|\!)} N\, M
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, M_{\mathrm{c}}) \gg\!= N &= (\!|\, \eta{:}\, N \,|\!)\, (\mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, M_{\mathrm{c}})) \\
&\to_{(\!|\mathtt{op}'|\!)} \mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, (\!|\, \eta{:}\, N \,|\!)\, M_{\mathrm{c}}) \\
&= \mathtt{op}\, M_{\mathrm{p}}\,(\lambda x.\, M_{\mathrm{c}} \gg\!= N)
\end{aligned}
$$

□

3.3 Closed Handlers

In 2.4.2, we introduced a notation for closed handlers. Even though we define closed handlers in terms of (open) handlers, their typing and reduction rules are actually simpler, since they do not have to go out of their way to support openness (i.e. passing through uninterpreted operations).

$$
\boldsymbol{(\!|}\, (\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta \,\boldsymbol{|\!)}\, N = \downarrow\, ((\!|\, (\mathtt{op}_i{:}\, (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k))))_{i\in I},\ \eta{:}\, (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N)
$$

We will first go through the typing rule.

Proposition 3.5. *The following typing rule is derivable in* $(\!|\lambda|\!)$*:*

$$
\frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta]_{i\in I} \\ \Gamma \vdash M_\eta : \gamma \to \delta \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash \boldsymbol{(\!|}\, (\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta \,\boldsymbol{|\!)}\, N : \delta}\ [\boldsymbol{(\!|\,|\!)}]
$$

Proof. We have $\boldsymbol{(\!|}\, (\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta \,\boldsymbol{|\!)}\, N = \downarrow\, ((\!|\, (\mathtt{op}_i{:}\, (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k))))_{i\in I},\ \eta{:}\, (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N)$ and we will proceed by building a typing derivation for this term.

$$
\frac{\dfrac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \\ [\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)]_{i\in I} \\ \Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_\emptyset(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash (\!|\, (\mathtt{op}_i{:}\, (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k))))_{i\in I},\ \eta{:}\, (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N : \mathcal{F}_\emptyset(\delta)}\ [(\!|\,|\!)]}{\Gamma \vdash \downarrow\, ((\!|\, (\mathtt{op}_i{:}\, (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k))))_{i\in I},\ \eta{:}\, (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N) : \delta}\ [\downarrow]
$$

We still need to prove both $\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)$ for every $i \in I$ and $\Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_\emptyset(\delta)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta \qquad \Gamma, x : \alpha_i \vdash x : \alpha_i}{\Gamma, x : \alpha_i \vdash M_i\, x : (\beta_i \to \delta) \to \delta}\ [\mathrm{app}] \qquad \dfrac{\Gamma \vdash\, \downarrow\, : \mathcal{F}_\emptyset(\delta) \to \delta \qquad \Gamma, k : \beta_i \to \mathcal{F}_\emptyset(\delta) \vdash k : \beta_i \to \mathcal{F}_\emptyset(\delta)}{\Gamma, k : \beta_i \to \mathcal{F}_\emptyset(\delta) \vdash\, \downarrow \circ k : \beta_i \to \delta}\ [\circ]}{\Gamma, x : \alpha_i, k : \beta_i \to \mathcal{F}_\emptyset(\delta) \vdash M_i\, x\, (\downarrow \circ k) : \delta}\ [\mathrm{app}]}{\Gamma, x : \alpha_i, k : \beta_i \to \mathcal{F}_\emptyset(\delta) \vdash \eta\, (M_i\, x\, (\downarrow \circ k)) : \mathcal{F}_\emptyset(\delta)}\ [\eta]}{\Gamma, x : \alpha_i \vdash \lambda k.\, \eta\, (M_i\, x\, (\downarrow \circ k)) : (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)}\ [\mathrm{abs}]}{\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)}\ [\mathrm{abs}]
$$

x and k are assumed to be fresh for M_i.

$$
\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M_\eta : \gamma \to \delta \qquad \Gamma, x : \gamma \vdash x : \gamma}{\Gamma, x : \gamma \vdash M_\eta\, x : \delta}\ [\mathrm{app}]}{\Gamma, x : \gamma \vdash \eta\, (M_\eta\, x) : \mathcal{F}_\emptyset(\delta)}\ [\eta]}{\Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_\emptyset(\delta)}\ [\mathrm{abs}]
$$

x is assumed to be fresh for M_η. ☐

We can also have reduction rules for closed handlers, which work exactly the same way as the open handler reduction rules (only they do not include cases for uninterpreted operations).

Proposition 3.6. *The following reductions are derivable in* $\llparenthesis\,\lambda\,\rrparenthesis$*:*

$$\begin{array}{rcl}
\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\,(\eta\, N) & \to_{\boldsymbol{\llparenthesis}\eta\boldsymbol{\rrparenthesis}} & M_\eta\, N \\
\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\,(\mathtt{op}_i\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})) & \to_{\boldsymbol{\llparenthesis}\mathtt{op}\boldsymbol{\rrparenthesis}} & M_i\, N_\mathrm{p}\, (\lambda x.\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\, N_\mathrm{c})
\end{array}$$

Proof.

$$\begin{aligned}
\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\,(\eta\, N) &= \downarrow (\llparenthesis\, (\mathtt{op}_i\!: (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ\, k))))_{i \in I},\ \eta\!: (\lambda x.\, \eta\, (M_\eta\, x))\, \rrparenthesis\, (\eta\, N)) \\
&\to_{\llparenthesis\,\eta\,\rrparenthesis} \downarrow ((\lambda x.\, \eta\, (M_\eta\, x))\, N) \\
&\to_\beta \downarrow (\eta\, (M_\eta\, N)) \\
&\to_{\downarrow} M_\eta\, N
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\,(\mathtt{op}_i\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})) &= \downarrow (\llparenthesis\, (\mathtt{op}_i\!: (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ\, k))))_{i \in I},\ \eta\!: (\lambda x.\, \eta\, (M_\eta\, x))\, \rrparenthesis\, (\mathtt{op}_i\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c}))) \\
&\to_{\llparenthesis\,\mathtt{op}\,\rrparenthesis} \downarrow ((\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ\, k)))\, N_\mathrm{p}\, (\lambda x.\, \llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})) \\
&\to_\beta \downarrow ((\lambda k.\, \eta\, (M_i\, N_\mathrm{p}\, (\downarrow \circ\, k)))\, (\lambda x.\, \llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})) \\
&\to_\beta \downarrow (\eta\, (M_i\, N_\mathrm{p}\, (\downarrow \circ\, (\lambda x.\, \llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})))) \\
&\to_{\downarrow} M_i\, N_\mathrm{p}\, (\downarrow \circ\, (\lambda x.\, \llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})) \\
&= M_i\, N_\mathrm{p}\, (\lambda x.\, \downarrow ((\lambda x.\, \llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})\, x)) \\
&\to_\beta M_i\, N_\mathrm{p}\, (\lambda x.\, \downarrow (\llparenthesis \ldots \rrparenthesis\, N_\mathrm{c})) \\
&= M_i\, N_\mathrm{p}\, (\lambda x.\boldsymbol{\llparenthesis}(\mathtt{op}_i\!: M_i)_{i \in I},\ \eta\!: M_\eta \boldsymbol{\rrparenthesis}\, N_\mathrm{c})
\end{aligned}$$

In the above, $\llparenthesis \ldots \rrparenthesis$ is taken to be a shortcut for $\llparenthesis\, (\mathtt{op}_i\!: (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow \circ\, k))))_{i \in I},\ \eta\!: (\lambda x.\, \eta\, (M_\eta\, x))\, \rrparenthesis$. ☐

4 Type Soundness

In Section 2, we have introduced both a type system and a reduction semantics for $\llparenthesis\,\lambda\,\rrparenthesis$. Now we will give more substance to these two definitions by proving properties which outline the relationship between them. Types can give us two guarantees: typed terms do not get stuck and typed terms always terminate. The former property is known as *progress* and, in Subsection 4.2, we will show that it holds for $\llparenthesis\,\lambda\,\rrparenthesis$ as long as we abstain from using the partial function $\mathcal{C}$. The latter is known as *termination* and its proof is more involved, so we will delay it until section 7. For both of these properties to hold, it will be essential to prove that a typed term stays typed after performing a reduction. This will be the object of the next subsection.

4.1 Subject Reduction

We now turn our attention to the subject reduction property. We can summarize subject reduction with the slogan "reduction preserves types". The rest of this section will consider a formal proof of this property for $\llparenthesis\,\lambda\,\rrparenthesis$, but before we begin, we present a the definition of substitution and a lemma.

Definition 4.1. *Let M and N be terms and x a variable. We define the* ***(capture-resolving) substitution*** *of N for x in M, written as $M[x := N]$,*[17] *using the following equations:*

[17] From now on, this notation will be used for capture-resolving substitution only.

$$
\begin{aligned}
(\lambda y.\, M)[x := N] &= \lambda y.\, (M[x := N]) \textit{ assuming that } y \neq x \textit{ and } y \textit{ is fresh for } N^{18} \\
(M\, K)[x := N] &= (M[x := N])\, (K[x := N]) \\
x[x := N] &= N \\
y[x := N] &= y \textit{ given that } x \neq y^{19} \\
c[x := N] &= c \\
(\mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}))[x := N] &= \mathtt{op}\, (M_{\mathrm{p}}[x := N])\, (\lambda y.\, M_{\mathrm{c}}[x := N]) \textit{ assuming that } y \neq x \textit{ and } y \textit{ is fresh for } N \\
(\eta\, M)[x := N] &= \eta\, (M[x := N]) \\
(\llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, N')[x := N] &= \llparenthesis\, (\mathtt{op}_i \colon (M_i[x := N]))_{i \in I},\ \eta \colon (M_\eta[x := N]) \,\rrparenthesis\, (N'[x := N]) \\
(\downarrow M)[x := N] &= \downarrow (M[x := N]) \\
(\mathcal{C}\, M)[x := N] &= \mathcal{C}\, (M[x := N])
\end{aligned}
$$

We are now at a point where we can easily lay down the reduction rules for $\llparenthesis \lambda \rrparenthesis$. A reduction rule ξ will be a relation on terms. Most of the time, we will deal with their context closures, $[\xi]$, for which we will also adopt the notation $\to_\xi$. We will also use the notation $\to_{\xi_1,\ldots,\xi_n}$ for the composition $\to_{\xi_n} \circ \cdots \circ \to_{\xi_1}$. The *reduction relation* $\to$ of $\llparenthesis \lambda \rrparenthesis$ is the union of the $\to_\xi$ relations for every reduction rule ξ. We will also use the symbols $\twoheadrightarrow$ and $\twoheadleftrightarrow$ to stand for the reflexive-transitive and reflexive-symmetric-transitive closures of $\to$, respectively. If $M \twoheadleftrightarrow N$, we will also say that M and N are *convertible*. A term which is not reducible to some other term is said to be in *normal form*.

Lemma 4.2. ***Substitution and types***

Whenever we have $\Gamma, x : \alpha \vdash M : \tau$ and $\Gamma \vdash N : \alpha$, we also have $\Gamma \vdash M[x := N] : \tau$ (i.e. we can substitute in M while preserving the type).

Proof. The proof is carried out by induction on the structure of M (or rather the structure of the type derivation $\Gamma \vdash M : \tau$).

- $M = y$
  - If $y = x$, then $M[x := N] = N$ and $\alpha = \tau$. We immediately have $\Gamma \vdash M[x := N] : \tau$ from the assumption that $\Gamma \vdash N : \alpha$.
  - If $y \neq x$, then $M[x := N] = x$ and we get $\Gamma \vdash M[x := N] : \tau$ from the assumption that $\Gamma, x : \alpha \vdash M : \tau$ and the fact that $x \notin \mathrm{FV}(M)$.
- All the other cases end up being trivial. We follow the definition of substitution (Definition 4.1) which just applies substitution to all of the subterms. For every such subterm, we make appeal to the induction hypothesis and construct the new typing derivation.

□

Property 4.3. ***Subject reduction***

If $\Gamma \vdash M : \tau$ and $M \to N$, then $\Gamma \vdash N : \tau$.

Proof. We prove this by induction on the reduction rule used in $M \to N$.

- $\boxed{M \to_\beta N}$ It must be the case that $M = (\lambda x.\, M')\, M''$ and $N = M'[x := M'']$. Since, $\Gamma \vdash M : \tau$, we must have the following typing derivation:

$$
\dfrac{\dfrac{\Gamma, x : \alpha \vdash M' : \tau}{\Gamma \vdash \lambda x.\, M' : \alpha \to \tau}\,[\mathrm{abs}] \qquad \Gamma \vdash M'' : \alpha}{\Gamma \vdash (\lambda x.\, M')\, M'' : \tau}\,[\mathrm{app}]
$$

[18] Here, y is a bound variable and we can simply *assume* that it is different from x and proceed...

[19] ...whereas here, y is a free variable and therefore, we have to *examine* whether it is different from x or not.

We apply Lemma 4.2 to $\Gamma, x : \alpha \vdash M' : \tau$ and $\Gamma \vdash M'' : \alpha$ to get a typing derivation for $\Gamma \vdash M'[x := M''] : \tau$.

- $\boxed{M \to_\eta N}$ We have $M = \lambda x.\, M'\, x$ with x fresh for M', $N = M'$ and $\tau = \tau_1 \to \tau_2$. Since $\Gamma \vdash M : \tau$, we have the following:

$$\dfrac{\dfrac{\Gamma, x : \tau_1 \vdash M' : \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash x : \tau_1}{\Gamma, x : \tau_1 \vdash M'\, x : \tau_2}\,[\text{app}]}{\Gamma \vdash \lambda x.\, M'\, x : \tau_1 \to \tau_2}\,[\text{abs}]$$

From the above derivation, we can extract $\Gamma, x : \tau_1 \vdash M' : \tau_1 \to \tau_2$. However, since x is fresh for M', we can strengthen this to $\Gamma \vdash M' : \tau_1 \to \tau_2$, which is what we wanted to prove.

- $\boxed{M \to_{\llparenthesis \eta \rrparenthesis} N}$

We have $M = \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\eta\, M')$, $N = M_\eta\, M'$, $\tau = \mathcal{F}_{E'}(\delta)$ and the following typing derivation for M:

$$\dfrac{\Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \qquad \dfrac{\Gamma \vdash M' : \gamma}{\Gamma \vdash \eta\, M' : \mathcal{F}_E(\gamma)} \qquad \ldots}{\Gamma \vdash \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\eta\, M') : \mathcal{F}_{E'}(\delta)}\,[\llparenthesis\,\rrparenthesis]$$

From the inferred typing judgments for M_η and M', we can build the typing derivation for $M_\eta\, M'$.

$$\dfrac{\Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \qquad \Gamma \vdash M' : \gamma}{\Gamma \vdash M_\eta\, M' : \mathcal{F}_{E'}(\delta)}\,[\text{app}]$$

- $\boxed{M \to_{\llparenthesis \mathtt{op} \rrparenthesis} N}$

We have $M = \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\mathtt{op}_j\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c}))$, $N = M_j\, M_\mathrm{p}\, (\lambda x.\, \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c})$ and $\tau = \mathcal{F}_{E'}(\delta)$.

$$\dfrac{\Gamma \vdash M_j : \alpha_j \to (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta) \qquad \dfrac{\Gamma \vdash M_\mathrm{p} : \alpha_j \qquad \Gamma, x : \beta_j \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \mathtt{op}_j : \alpha_j \rightarrowtail \beta_j \in E}{\Gamma \vdash \mathtt{op}_j\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c}) : \mathcal{F}_E(\gamma)}\,[\mathtt{op}] \qquad \ldots}{\Gamma \vdash \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\mathtt{op}_j\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c})) : \mathcal{F}_{E'}(\delta)}\,[\llparenthesis\,\rrparenthesis]$$

From the types of M_p, M_c and M_j, we can calculate the type of our redex, $M_j\, M_\mathrm{p}\, (\lambda x.\, \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c})$.

$$\dfrac{\dfrac{\Gamma \vdash M_j : \alpha_j \to (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta) \qquad \Gamma \vdash M_\mathrm{p} : \alpha_j}{\Gamma \vdash M_j\, M_\mathrm{p} : (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)}\,[\text{app}] \qquad \dfrac{\dfrac{\Gamma, x : \beta_j \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \ldots}{\Gamma, x : \beta_j \vdash \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c} : \mathcal{F}_{E'}(\delta)}\,[\llparenthesis\,\rrparenthesis]}{\Gamma \vdash \lambda x.\, \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c} : \beta_j \to \mathcal{F}_{E'}(\delta)}\,[\text{abs}]}{\Gamma \vdash M_j\, M_\mathrm{p}\, (\lambda x.\, \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c}) : \mathcal{F}_{E'}(\delta)}\,[\text{app}]$$

- $\boxed{M \to_{\llparenthesis \mathtt{op}' \rrparenthesis} N}$

We have $M = \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\mathtt{op}\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c}))$, $N = \mathtt{op}\, M_\mathrm{p}\, (\lambda x.\, \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_\mathrm{c})$ and $\tau = \mathcal{F}_{E'}(\delta)$.

$$\dfrac{\dfrac{\Gamma \vdash M_\mathrm{p} : \alpha \qquad \Gamma, x : \beta \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathtt{op}\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c}) : \mathcal{F}_E(\gamma)}\,[\mathtt{op}] \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E' \qquad \ldots}{\Gamma \vdash \llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, (\mathtt{op}\, M_\mathrm{p}\, (\lambda x.\, M_\mathrm{c})) : \mathcal{F}_{E'}(\delta)}\,[\llparenthesis\,\rrparenthesis]$$

From the inferred judgments, we can build a typing derivation for the redex.

$$\dfrac{\Gamma \vdash M_{\mathrm{p}} : \alpha \qquad \dfrac{\Gamma, x : \beta \vdash M_{\mathrm{c}} : \mathcal{F}_E(\gamma) \qquad \ldots}{\Gamma, x : \beta \vdash \llparenthesis\, (\mathsf{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_{\mathrm{c}} : \mathcal{F}_{E'}(\delta)}\,[\llparenthesis\,\rrparenthesis] \qquad \mathsf{op} : \alpha \rightarrowtail \beta \in E'}{\Gamma \vdash \mathsf{op}\, M_{\mathrm{p}}\, (\lambda x.\, \llparenthesis\, (\mathsf{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, M_{\mathrm{c}}) : \mathcal{F}_{E'}(\delta)}\,[\mathsf{op}]$$

- $\boxed{M \to_{⫰} N}$

  In this case, $M = ⫰\, (\eta\, M')$ and $N = M'$.

$$\dfrac{\dfrac{\Gamma \vdash M' : \tau}{\Gamma \vdash \eta\, M' : \mathcal{F}_{\emptyset}(\tau)}\,[\eta]}{\Gamma \vdash ⫰\, (\eta\, M') : \tau}\,[⫰]$$

  We immediately get $\Gamma \vdash M' : \tau$, which is the sought after typing derivation of the redex.

- $\boxed{M \to_{\mathcal{C}_\eta} N}$

  $M = \mathcal{C}\, (\lambda x.\, \eta\, M)$, $N = \eta\, (\lambda x.\, M)$ and $\tau = \mathcal{F}_E(\gamma \to \delta)$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M : \delta}{\Gamma, x : \gamma \vdash \eta\, M : \mathcal{F}_E(\delta)}\,[\eta]}{\Gamma \vdash \lambda x.\, \eta\, M : \gamma \to \mathcal{F}_E(\delta)}\,[\mathrm{abs}]}{\Gamma \vdash \mathcal{C}\, (\lambda x.\, \eta\, M) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}]$$

  From these judgments, we build a type for the redex.

$$\dfrac{\dfrac{\Gamma, x : \gamma \vdash M : \delta}{\Gamma \vdash \lambda x.\, M : \gamma \to \delta}\,[\mathrm{abs}]}{\Gamma \vdash \eta\, (\lambda x.\, M) : \mathcal{F}_E(\gamma \to \delta)}\,[\eta]$$

- $\boxed{M \to_{\mathcal{C}_{\mathsf{op}}} N}$

  $M = \mathcal{C}\, (\lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}))$, $N = \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}}))$ and $\tau = \mathcal{F}_E(\gamma \to \delta)$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M_{\mathrm{p}} : \alpha \qquad \Gamma, x : \gamma, y : \beta \vdash M_{\mathrm{c}} : \mathcal{F}_E(\delta) \qquad \mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma, x : \gamma \vdash \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}) : \mathcal{F}_E(\delta)}\,[\mathsf{op}]}{\Gamma \vdash \lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}) : \gamma \to \mathcal{F}_E(\delta)}\,[\mathrm{abs}]}{\Gamma \vdash \mathcal{C}\, (\lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}})) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}]$$

  With the judgments above, we build the derivation below.

$$\dfrac{\Gamma \vdash M_{\mathrm{p}} : \alpha \qquad \dfrac{\dfrac{\Gamma, y : \beta, x : \gamma \vdash M_{\mathrm{c}} : \mathcal{F}_E(\delta)}{\Gamma, y : \beta \vdash \lambda x.\, M_{\mathrm{c}} : \gamma \to \mathcal{F}_E(\delta)}\,[\mathrm{abs}]}{\Gamma, y : \beta \vdash \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}}) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}] \qquad \mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathsf{op}\, M_{\mathrm{p}}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}})) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathsf{op}]$$

  In the above we get $\Gamma \vdash M_{\mathrm{p}} : \alpha$ from $\Gamma, x : \gamma \vdash M_{\mathrm{p}} : \alpha$ and the rule's condition that $x \notin \mathrm{FV}(M_{\mathrm{p}})$.

- $\boxed{C[M'] \to C[N']}$

  The reduction relation of $(\!|\lambda|\!)$ is defined as the context closure of the individual reduction rules. We have covered the rules themselves, we now address the context closure. By induction hypothesis, we know that the reduction from $M' \to N'$ preserves types, i.e. for any Δ and α such that $\Delta \vdash M' : \alpha$, we have $\Delta \vdash N' : \alpha$. We observe that the typing rules of $(\!|\lambda|\!)$ (Figure 1) are compositional, meaning that the type of a term depends only on the types of its subterms, not on their syntactic form. We can check this easily by looking at the premises of all of the typing rules. For every immediate subterm T, there is a premise $\Delta \vdash T : \alpha$ where T is a metavariable. We can therefore replace T and its typing derivation by some other T' with $\Delta \vdash T' : \alpha$. Since the typing rules of $(\!|\lambda|\!)$ are compositional, we can replace the $\Delta \vdash M' : \alpha$ in $\Gamma \vdash C[M'] : \tau$ by $\Delta \vdash N' : \alpha$ and get $\Gamma \vdash C[N'] : \tau$.

□

We have proven subject reduction for core $(\!|\lambda|\!)$. The syntax, semantics and types that we have introduced for sums and products are standard. Their proofs of subject reduction carry over into our setting as well.

4.2 Progress

Progress means that typed terms are never stuck. Among the terms of $(\!|\lambda|\!)$, we will have to identify terms which are acceptable stopping points for reduction. Progress will mean that if a term is not in one of these acceptable positions, then there must be a way to continue reducing. The term we will use for these acceptable results is *value*.

Definition 4.4. *A $(\!|\lambda|\!)$ term is a **value** if it can be generated by the following grammar:*

$$\begin{aligned} V ::= &\ \lambda x.\, M \\ \mid &\ \mathtt{op}\, V\, (\lambda x.\, M) \\ \mid &\ \eta\, V \end{aligned}$$

where M ranges over $(\!|\lambda|\!)$ terms.

The above definition reflects the intuition that $(\!|\lambda|\!)$ consists of functions and computations, where functions are built using λ and computations using $\mathtt{op}$ and η. The other syntactic constructions (application, $(\!|\ |\!)$, $\downarrow$ and $\mathcal{C}$) all have rules which are supposed to eventually replace them with other terms. As with subject reduction, before we proceed to the main property, we start with a lemma.

Lemma 4.5. ***Value classification*** *Let V be a closed well-typed value (i.e. $\emptyset \vdash V : \tau$). Then the following hold:*

- *if $\tau = \alpha \to \beta$, then $V = \lambda x.\, M$*
- *if $\tau = \mathcal{F}_E(\alpha)$, then either $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$*

Proof.

- Assume $\tau = \alpha \to \beta$. If $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$, then τ must be a computation type $\mathcal{F}_E(\gamma)$, which is a contradiction. The only remaining possibility is therefore $V = \lambda x.\, M$.
- Assume $\tau = \mathcal{F}_E(\alpha)$. If $V = \lambda x.\, M$, then τ must be a function type $\beta \to \gamma$, which is a contradiction. The only remaining possibilities are therefore $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$.

□

Property 4.6. ***Progress*** *Every closed well-typed term M from $(\!|\lambda|\!)$ without $\mathcal{C}$ and constants*[20] *is either a value or is reducible to some other term.*

Proof. We will proceed by induction on M.

[20] Constants are assumed to be reduced away by some external rule. In our case, this will be the application of an ACG lexicon (**??**).

- $M = \lambda x.\, M'$ Then M is already a value.
- $M = x$ Impossible, since M must be a closed term.
- $M = M_1\, M_2$ By induction hypothesis, M_1 and M_2 are either values or reducible terms. If either one is reducible, then our term is reducible as well and we are done. If neither is reducible, then they are both values. Since M is a closed well-typed term (i.e. $\emptyset \vdash M : \tau$), then $\vdash M_1 : \alpha \to \tau$ for some α. Thanks to Lemma 4.5, we have that $M_1 = \lambda x.\, M_1'$. This means that $M = (\lambda x.\, M_1')\, M_2$ and M is therefore reducible with β.
- $M = \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ By induction hypothesis, M_{p} is either reducible or a value. If M_{p} is reducible, then so is M. If it is a value, then so is M as well.
- $M = \eta\, N$ The same argument as for $\mathtt{op}$. By induction hypothesis N is reducible or a value and therefore the same holds for M.
- $M = (\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\, N$ By induction hypothesis, N is either a value or it is itself reducible. If it is reducible, then so is M. If it is not, then it must be a (closed) value. The type of N is a computation type $\mathcal{F}_E(\alpha)$ and so by Lemma 4.5, it must either be $\mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $\eta\, V$. If $N = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$, then $(\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\, (\mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}))$ is reducible by $(\!|\, \mathtt{op} \,|\!)$ or $(\!|\, \mathtt{op}' \,|\!)$ (depending on whether or not $\mathtt{op} \in \{\mathtt{op}_i\}_{i\in I}$). Otherwise, if $N = \eta\, V$, then $(\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\, (\eta\, V)$ is reducible by $(\!|\, \eta \,|\!)$.
- $M = \downarrow N$ By induction hypothesis, N is either reducible or a value. As before, we only have to focus on the case when N is a value. From Lemma 4.5, we know that $N = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $N = \eta\, V$. However, we can rule out the former since we know that $\emptyset \vdash N : \mathcal{F}_\emptyset(\alpha)$, meaning that $\mathtt{op}$ is not in the empty effect signature $\emptyset$. We therefore end up with $\downarrow (\eta\, V)$, which is reducible by $\downarrow$.

□

We have shown progress for $(\!|\,\lambda\,|\!)$ without $\mathcal{C}$. It is easy to see that we cannot do better, as the $\mathcal{C}$ operator can violate progress and get us stuck quite easily.

Observation 4.7. *There exists a closed well-typed term M from $(\!|\,\lambda\,|\!)$ without constants that is neither a value nor reducible to some other term.*

Proof. The most trivial example is $\mathcal{C}\, (\lambda x.\, x)$. The computation that is performed by the body of the function $\lambda x.\, x$ is entirely determined by the parameter x. It is therefore not possible to pull out this structure outside of the function. Therefore, applying the $\mathcal{C}$ operator to this function is undefined and evaluation gets stuck. □

5 Algebraic Properties

In this section, we will clarify what we mean when we say that the $\mathcal{F}_E(\alpha)$ computation types form a functor/applicative functor/monad and we will prove that the constructions in $(\!|\,\lambda\,|\!)$ conform to the laws of these algebraic structures. The object on which we will build these mathematical structures will be the meanings of $(\!|\,\lambda\,|\!)$ terms. We will therefore start by building an interpretation for $(\!|\,\lambda\,|\!)$, a denotational semantics. Then we will be in measure to define the algebraic structures mentioned above and verify that their laws are satisfied.

5.1 Denotational Semantics

We start by identifying the domains of interpretation. For each type, we designate a set such that all terms having that type will be interpreted in that set. Before we do so, we introduce some notation on sets.

Notation 5.1. *Let A and B be sets. Then:*

- A^B *is the set of functions from B to A*
- $A \times B$ *is the cartesian product of A and B*
- $A \sqcup B$ *is the disjoint union of A and B*[21]

[21]Note that this disjoint union operator $\sqcup$ is different from the $\uplus$ one from Section 2. $A \sqcup B$ is defined as $\{(x, 0) \mid x \in A\} \cup \{(x, 1) \mid x \in B\}$.

- $A_\bot$ *is the disjoint union of* A *and* $\{\bot\}$

Definition 5.2. *Given a set* A_ν *for every atomic type* ν*, the* ***interpretation of a type*** τ *is a set* $[\![\tau]\!]$ *defined inductively by:*

$$\begin{aligned} [\![\nu]\!] &= (A_\nu)_\bot \\ [\![\alpha \to \beta]\!] &= ([\![\alpha]\!] \to [\![\beta]\!])_\bot \\ [\![\mathcal{F}_E(\gamma)]\!] &= ([\![\gamma]\!] \sqcup \bigsqcup_{\mathtt{op}:\alpha \rightarrowtail \beta \in E} [\![\alpha]\!] \times [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta]\!]})_\bot \end{aligned}$$

Note that $[\![\mathcal{F}_E(\gamma)]\!]$ *is recursively defined not only by induction on the type itself but also by its use of* $[\![\mathcal{F}_E(\gamma)]\!]$ *on the right hand side. Formally, we take* $[\![\mathcal{F}_E(\gamma)]\!]$ *to be the least fixed point of the monotone functional* $F(X) = ([\![\gamma]\!] \sqcup \bigsqcup_{\mathtt{op}:\alpha \rightarrowtail \beta \in E} [\![\alpha]\!] \times X^{[\![\beta]\!]})_\bot$*, whose existence is guaranteed by the Knaster-Tarski theorem [62, 63].*

Notation 5.3. *We will use* λ *notation to write down elements of* $[\![\alpha \to \beta]\!]$*:*

- $\lambda x.\, F(x) \in [\![\alpha \to \beta]\!]$ *when* $F(x) \in [\![\beta]\!]$ *for every* $x \in [\![\alpha]\!]$
- $\bot \in [\![\alpha \to \beta]\!]$

We will use the following syntax to write down elements of $[\![\mathcal{F}_E(\gamma)]\!]$*:*

- $\eta(x) \in [\![\mathcal{F}_E(\gamma)]\!]$ *with* $x \in [\![\gamma]\!]$
- $\mathtt{op}(p, c) \in [\![\mathcal{F}_E(\gamma)]\!]$ *with* $\mathtt{op} : \alpha \rightarrowtail \beta \in E$*,* $p \in [\![\alpha]\!]$ *and* $c \in [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta]\!]}$
- $\bot \in [\![\mathcal{F}_E(\gamma)]\!]$

The definition of $[\![\tau]\!]$ follows the definition of a value (Definition 4.4): function types denote functions and computation types either denote atomic algebraic expressions (η) or applications of algebraic operations ($\mathtt{op}$). In the denotational semantics, we also take care of the fact that terms can get stuck and fail to yield the expected value. We represent this by adding the element $\bot$ to the interpretation of every type.

Definition 5.4. *We define the* ***interpretation of a typing context*** Γ *as the set* $[\![\Gamma]\!]$ *of functions that map every* $x : \alpha \in \Gamma$ *to an element of* $[\![\alpha]\!]$*.*

We will call these functions ***valuations****. We will use the notation* $e[x := f]$ *to stand for the* ***extension*** *of* e *with* $x \mapsto f$*. The domain of the extension is* $\mathrm{dom}(e) \cup x$*. The extension maps* x *to* f *and every other variable in its domain to* $e(x)$*.*

Definition 5.5. *Assume given* $\mathcal{I}(c) \in [\![\alpha]\!]$ *for every constant* $c : \alpha \in \Sigma$*. For a well-typed term* M *with* $\Gamma \vdash M : \tau$*, we define the* ***interpretation of term*** M *as a function* $[\![M]\!]$ *from* $[\![\Gamma]\!]$ *to* $[\![\tau]\!]$*. The definition proceeds by induction on* M*:*[22]

$$\begin{aligned} [\![\lambda x.\, M]\!](e) &= \lambda X.\, ([\![M]\!](e[x := X])) \\ [\![x]\!](e) &= e(x) \\ [\![M\,N]\!](e) &= \begin{cases} [\![M]\!](e)([\![N]\!](e)), & \textit{if } [\![M]\!](e) \textit{ is a function} \\ \bot, & \textit{if } [\![M]\!](e) \textit{ is } \bot \end{cases} \\ [\![c]\!](e) &= \mathcal{I}(c) \\ [\![\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})]\!](e) &= \mathtt{op}([\![M_{\mathrm{p}}]\!](e), \lambda X.\, ([\![M_{\mathrm{c}}]\!](e[x := X]))) \\ [\![\eta\, M]\!](e) &= \eta([\![M]\!](e)) \\ [\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N]\!](e) &= [\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)]\!](e)([\![N]\!](e)) \\ [\![\downarrow M]\!](e) &= \begin{cases} x, & \textit{if } [\![M]\!](e) = \eta(x) \\ \bot, & \textit{otherwise} \end{cases} \\ [\![\mathcal{C}\, M]\!](e) &= [\![\mathcal{C}]\!]([\![M]\!](e)) \end{aligned}$$

[22] In the definition, we make use of $[\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)]\!](e)$ and $[\![\mathcal{C}]\!]$. This notation is introduced right after this definition.

Definition 5.6. *The* ***interpretation of a handler*** $(\!|\, (\mathsf{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)$ *within a valuation* e *(also written as* $[\![(\!|\, (\mathsf{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)]\!](e)$*) is the function* h *defined inductively by:*

$$h(\eta(x)) = \begin{cases} [\![M_\eta]\!](e)(x), & \text{if } [\![M_\eta]\!](e) \text{ is a function} \\ \bot, & \text{otherwise} \end{cases}$$

$$h(\mathsf{op}_j(p,c)) = \begin{cases} [\![M_j]\!](e)(p)(\lambda x.\, h(c(x))), & \text{if } j \in I, \text{ and } [\![M_j]\!](e) \text{ and } [\![M_j]\!](e)(p) \text{ are both functions} \\ \mathsf{op}_j(p, \lambda x.\, h(c(x))), & \text{if } j \notin I \\ \bot, & \text{otherwise} \end{cases}$$

$$h(\bot) = \bot$$

The equations defining h *use* h *on the right-hand side. Nevertheless,* h *is well-defined since we can rely on induction. There is a* ***well-founded ordering on the elements of*** $[\![\mathcal{F}_E(\gamma)]\!]$*, where* $\forall x.\ \mathsf{op}(p,c) > c(x)$*. The monotonic functional* $F(X) = ([\![\gamma]\!] \sqcup \bigsqcup_{\mathsf{op}:\alpha\rightarrowtail\beta\in E} [\![\alpha]\!] \times X^{[\![\beta]\!]})_\bot$ *used in defining* $[\![\mathcal{F}_E(\gamma)]\!]$ *(Definition 5.2) is also Scott-continuous (i.e. it is both monotonic and it preserves suprema). By Kleene fixed-point theorem [64], we have that the least fixed point of* F *is the supremum of the series* $\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$*. Let the* ***rank*** *of* x *be the smallest* n *such that* $x \in F^n(\emptyset)$*. The ordering* $<_{\mathrm{r}}$*, defined as* $x <_{\mathrm{r}} y$ *whenever* $\mathrm{rank}(x) < \mathrm{rank}(y)$*, is a well-founded ordering. It is also the inductive ordering that we were looking for. Whenever* $\mathrm{rank}(\mathsf{op}(p,c)) = n$*, then* c *is a function whose codomain is* $F^{n-1}(\emptyset)$ *and therefore* $\forall x.\ \mathsf{op}(p,c) >_{\mathrm{r}} c(x)$*.*

Definition 5.7. *The* ***interpretation of the*** $\mathcal{C}$ ***operator*** *is a function* g *defined inductively by:*

$$g(f) = \begin{cases} \eta(h), & \text{if } f \text{ is a function and } \exists h.\ \forall x.\ f(x) = \eta(h(x)) \\ \mathsf{op}(p, \lambda y.\, g(\lambda x.\, c(x)(y))), & \text{if } f \text{ is a function and } \exists \mathsf{op}, p, c.\ \forall x.\ f(x) = \mathsf{op}(p, c(x)) \\ \bot, & \text{otherwise} \end{cases}$$

As with Definition 5.6, we have to show that this is actually a valid definition since we are using g *on the right-hand side of an equation defining* g*. This time around, the arguments to* g *are functions whose codomain is the interpretation of some computation type* $\mathcal{F}_E(\beta)$*. We can extend a well-founded ordering on the set* $[\![\mathcal{F}_E(\beta)]\!]$ *to a* ***well-founded ordering on*** $[\![\alpha \to \mathcal{F}_E(\beta)]\!]$ *by stating that* $f < g$ *whenever* f *and* g *are both functions (not* $\bot$*) and* $\forall x \in [\![\alpha]\!].\ f(x) < g(x)$*.*

We have to show that the recursive call to g *in the definition above is performed on an argument which is smaller than the original function. Let* $f' = \lambda x.\, c(x)(y)$ *be the function to which we recursively apply* g*. We have that* $f(x) = \mathsf{op}(p, c(x))$ *and* $f'(x) = c(x)(y)$*. We know that* $\forall y.\ \mathsf{op}(p, c(x)) > c(x)(y)$*, since that is the property of the well-founded ordering on the elements of* $[\![\mathcal{F}_E(\gamma)]\!]$ *established above. Therefore, we have that* $\forall x \in [\![\alpha]\!].\ f(x) > f'(x)$ *and so* $f > f'$*.*

This was the entire definition of our denotational semantics.[23] We will now compare it to the reduction semantics introduced in 2.3.

Property 5.8. ***Soundness of reduction w.r.t. denotations*** *Whenever* $M \to N$ *in* $(\!|\lambda|\!)$*, then* $[\![M]\!] = [\![N]\!]$*.*

Proof. The property relies on two facts: that our denotational semantics is compositional, which means that the context closure of reduction rules preserves denotations, and that every individual reduction preserves denotations. To prove so for the β rule is a matter of proving a lemma stating that $[\![M]\!](e[x := [\![N]\!](e)]) = [\![M[x := N]]\!](e)$, which follows from the compositionality of the denotational semantics. For all the other rules, it suffices to use the definition of interpretation (Definition 5.5) to calculate the denotation of both the left-hand side and the right-hand side and verify that they are the same object. □

We see that equalities from the reduction semantics are carried over to the denotational semantics. The converse, however, is not the case.

Observation 5.9. ***Incompleteness of reduction w.r.t. denotations***

There exist terms M *and* N *in* $(\!|\lambda|\!)$ *such that* $[\![M]\!] = [\![N]\!]$ *but* M *and* N *are not convertible.*

---

[23] We could also extend this interpretation to sums and products. The types would be interpreted by $[\![\alpha \times \beta]\!] = ([\![\alpha]\!] \times [\![\beta]\!])_\bot$ and $[\![\alpha + \beta]\!] = ([\![\alpha]\!] \sqcup [\![\beta]\!])_\bot$. The term level definitions would be the standard definitions one would expect for pairs and variants (modulo the treatment of $\bot$).

Proof. Consider a stuck term such as $M = \mathcal{C}\,(\lambda x.\, x)$ and another term $N = (\!|\,|\!)\, M = (\!|\,|\!)\,(\mathcal{C}\,(\lambda x.\, x))$. Neither one of these two terms is reducible and neither one is a value. They are stuck and the denotational semantics assigns the value $\bot$ to both of them, therefore $[\![M]\!] = [\![N]\!]$. However, as a consequence of confluence (coming up in section 6), a pair of different normal terms is never convertible, and therefore M and N are not convertible. □

And this concludes the definition of the denotational semantics of $(\!|\,\lambda\,|\!)$. Throughout most of the article, we will be using the reduction semantics introduced in 2.3, even though it is incomplete, since it allows us to simplify terms in a mechanical and transparent step-by-step manner. However, the denotational semantics will be useful to us in the rest of this section since it will let us access extra equalities needed to prove some general laws.

5.2 *Category*

We aim to show that the computation types in $(\!|\,\lambda\,|\!)$ form a monad. All these terms are defined w.r.t. some category and so we will start by introducing the category underlying $(\!|\,\lambda\,|\!)$.

We will be working with a particular category, which we will call $(\!|\,\lambda\,|\!)$. The $(\!|\,\lambda\,|\!)$ category consists of:

objects: the types of the $(\!|\,\lambda\,|\!)$ calculus

arrows: for any two types α and β, the arrows from α to β are the functions from $[\![\alpha]\!]$ to $[\![\beta]\!]$

composition: composition of arrows is defined as composition of functions

identities: for every type α, we define id_α as the identity function with domain $[\![\alpha]\!]$

Since the arrows in our category are functions, the three laws of a category (associativity, left identity and right identity) fall out of the same properties for functions.

Since the arrows in our category are functions, the three laws of a category (associativity (**??**), left identity (**??**) and right identity (**??**)) fall out of the same properties for functions.

5.3 *The Three Laws*

Monads form a subset of applicative functors which in turn is a subset of functors. Instead of incrementally building up from a functor all the way to a monad, it will end up being more practical to first prove the monad laws and then illustrate how they let us verify the functor and applicative functor laws. Therefore, we first define our monadic bind operator and prove the three monad laws.

Definition 5.10. *Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$ and f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$. We define $X \gg\!= f$ inductively on the structure of X:*

$$
\begin{aligned}
\mathtt{op}(p, c) \gg\!= f &= \mathtt{op}(p, \lambda x.\, c(x) \gg\!= f) \\
\eta(x) \gg\!= f &= f(x) \\
\bot \gg\!= f &= \bot
\end{aligned}
$$

Note that $X \gg\!= f$ is equivalent to $[\![x \gg\!= y]\!]([x \mapsto X, y \mapsto f])$.

Law 5.11. *(Associativity of $\gg\!=$)*

Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$, f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$ and g be a function from $[\![\beta]\!]$ to $[\![\mathcal{F}_E(\gamma)]\!]$. Then the following equation holds:

$$(X \gg\!= f) \gg\!= g \;=\; X \gg\!= (\lambda x.\, f(x) \gg\!= g)$$

Proof. Proof by induction on the well-founded structure of X:

- $X = \bot$
$$
\begin{aligned}
(\bot \gg\!= f) \gg\!= g &= \bot \gg\!= g \\
&= \bot \\
&= \bot \gg\!= (\lambda x.\, f(x) \gg\!= g)
\end{aligned}
$$

- $X = \eta(x)$
$$
\begin{aligned}
(\eta(x) \gg\!= f) \gg\!= g &= f(x) \gg\!= g \\
&= (\lambda x.\, f(x) \gg\!= g)(x) \\
&= \eta(x) \gg\!= (\lambda x.\, f(x) \gg\!= g)
\end{aligned}
$$

- $X = \mathtt{op}(p, c)$

$$\begin{aligned}(\mathtt{op}(p,c) \gg\!= f) \gg\!= g &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= f) \gg\!= g \\ &= \mathtt{op}(p, \lambda y.\, (c(y) \gg\!= f) \gg\!= g) \\ &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= (\lambda x.\, (f(x) \gg\!= g))) \\ &= \mathtt{op}(p, c) \gg\!= (\lambda x.\, f(x) \gg\!= g)\end{aligned}$$

□

Law 5.12. *(Left identity for $\gg\!=$)*
Let $\eta(x)$ be from $[\![\mathcal{F}_E(\alpha)]\!]$ and f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$. Then the following holds:

$$\eta(x) \gg\!= f = f(x)$$

Proof. Follows immediately from the definition of $\gg\!=$ (Definition 5.10). □

Law 5.13. *(Right identity for $\gg\!=$)*
Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$. Then the following holds:

$$X \gg\!= (\lambda x.\, \eta(x)) = X$$

Proof. By induction on the structure of X:

- $X = \bot$

$$\bot \gg\!= (\lambda x.\, \eta(x)) = \bot$$

- $X = \eta(x)$

$$\begin{aligned}\eta(x) \gg\!= (\lambda x.\, \eta(x)) &= (\lambda x.\, \eta(x))(x) \\ &= \eta(x)\end{aligned}$$

- $X = \mathtt{op}(p, c)$

$$\begin{aligned}\mathtt{op}(p,c) \gg\!= (\lambda x.\, \eta(x)) &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= (\lambda x.\, \eta(x))) \\ &= \mathtt{op}(p, \lambda y.\, c(y)) \\ &= \mathtt{op}(p, c)\end{aligned}$$

□

5.4 Monad

Definition 5.14. *A **monad** is a functor F and two combinators, $\eta : \alpha \to F(\alpha)$ and $\gg\!= : F(\alpha) \to (\alpha \to F(\beta)) \to F(\beta)$, polymorphic in α and β. These objects must also satisfy the following laws:*

$$(X \gg\!= f) \gg\!= g = X \gg\!= (\lambda x.\, f(x) \gg\!= g) \qquad \textit{(Associativity)} \qquad (1)$$
$$\eta(x) \gg\!= f = f(x) \qquad \textit{(Left identity)} \qquad (2)$$
$$X \gg\!= \eta = X \qquad \textit{(Right identity)} \qquad (3)$$

To understand why the laws look the way they do, we will consider functions of type $\alpha \to F(\beta)$. These are the kinds of functions one might use to model call-by-value [52, 65]: we take a value α and then yield some computation $F(\beta)$. Now assume we would use this type of functions to model procedures of input type α and output type β and we would want these procedures to form a category. For every type α, we would like an identity procedure with input type and output type α, therefore a function of type $\alpha \to F(\alpha)$. The polymorphic η combinator will be this identity procedure. We would also like to be able to compose a procedure from α to β with a procedure from β to γ, i.e. compose functions of types $\alpha \to F(\beta)$ and $\beta \to F(\gamma)$. When composing $f : \alpha \to F(\beta)$ with $g : \beta \to F(\gamma)$, we run into the problem of having some $f(x) : F(\beta)$ and $g : \beta \to F(\gamma)$ that we cannot compose. This is where $\gg\!=$ comes in and composes these two values for us. Let $f > \!\!=\!\!> g = \lambda x.\, f(x) \gg\!= g$ be the resulting composition operator. In order for this structure to be a category, it needs to satisfy the following:

$$\begin{aligned}(f > \!\!=\!\!> g) > \!\!=\!\!> h &= f > \!\!=\!\!> (g > \!\!=\!\!> h) \\ \eta > \!\!=\!\!> f &= f \\ f > \!\!=\!\!> \eta &= f\end{aligned}$$

By taking f to be the constant function that returns X, we end up with the laws of the monad. Conversely, with the principle of extensionality [66], we can derive these laws from the monad laws. Therefore $\langle F, \eta, \gg\!= \rangle$ forms a monad whenever the derived $>\!=\!\!>$ and η form a category. This kind of category is called a *Kleisli category* and the particular presentation of a monad that we have given here is known as a *Kleisli triple*. To prove that $\mathcal{F}_E$ is a monad will be trivial: we have already done so! The η combinator is of course our η and $\gg\!=$ is our $\gg\!=$. The three laws that we need to verify are three laws that we have introduced in 5.3 and have been using throughout this section. Monads have been introduced to natural language semantics by Chung-chieh Shan in 2002 [2]. Since then, they have seen occasional use, mostly to handle dynamics without burdening the semantics with context/continuation passing [67, 68], but also other phenomena such as conventional implicature/opacity [69, 70, 71]. The challenge of combining different phenomena which rely on different monads has been tackled from two angles: using distributive laws for monads [72] and using monad transformers [5, 73]. In the $(\!|\lambda|\!)$ calculus, the monadic operations η and $\gg\!=$ are available as the η constructor and the $\gg\!=$ combinator introduced in 2.4.1.

6 Confluence

The object of our study during this section will be the proof of the *confluence property* of $(\!|\lambda|\!)$. Informally, it means that a single term cannot reduce to two or more different results. Together with the termination from Section 7, this will give us the property that every term yields exactly one result and does so in a finite amount of steps (a property known as *strong normalization*). Confluence also gives us a strong tool to prove an inequality on terms. If two terms reduce to different normal forms, confluence guarantees us that they are not convertible.

Definition 6.1. *A reduction relution $\to$ on a set A is said to be* ***confluent*** *whenever for each $a, b, c \in A$ such that $a \to b$ and $a \to c$ there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.*

Proofs of this property are often mechanical and follow the same pattern. Our strategy will be to reuse a general result which applies one such proof for a general class of rewriting systems. Our rewriting system is a system of reductions on terms and the reductions have side conditions concerning the binding of free variables. A good fit for this kind of system are the Combinatory Reduction Systems (CRSs) of Klop [74]. The main result about CRSs that we will make use of is the following (Corollary 13.6 in [74]).

Theorem 6.2. ***Confluence of orthogonal CRSs*** *Every orthogonal CRS is confluent.*

We will model $(\!|\lambda|\!)$ as a CRS. However, η-reduction will deny us orthogonality. We will therefore first prove confluence of $(\!|\lambda|\!)$ without η-reduction and then we will manually show that confluence is preserved on adding η-reduction back.

Notation 6.3. *The* ***intensional*** *$(\!|\lambda|\!)$* ***calculus*** *$(\!|\lambda|\!)_{-\eta}$ is the $(\!|\lambda|\!)$ calculus without the η-reduction rule.*

The rest of this section will go like this:

- CRS: a formalism for higher-order rewriting (6.1)
- $(\!|\lambda|\!)$ is a CRS (6.2)
- Klop et al [93]: Every orthogonal CRS is confluent (6.3)
  - $(\!|\lambda|\!)_{-\eta}$ is an orthogonal CRS $\Rightarrow$ $(\!|\lambda|\!)_{-\eta}$ is confluent (Lemma 6.12)
  - η is an orthogonal CRS $\Rightarrow$ η is confluent (Lemma 6.13)
- $(\!|\lambda|\!)_{-\eta} + \eta$ is confluent (6.4, Theorem 6.18)
  - because $(\!|\lambda|\!)_{-\eta}$ and η commute (Lemma 6.17)

6.1 Combinatory Reduction Systems

A Combinatory Reduction System is defined by an alphabet and a set of rewriting rules. We will first cover the alphabet.

Definition 6.4. *A* ***CRS alphabet*** *consists of:*

- *a set* Var *of* variables *(written lower-case as x, y, z,...)*
- *a set* MVar *of* metavariables *(written upper-case as M, N, ...), each with is own arity*
- *a set of* function symbols*, each with its own arity*

Let us sketch the difference between the variables in Var and the metavariables in MVar. The variables in Var are the variables of the object-level terms, in our case it will be the variables of $(\!|\lambda|\!)$. The variables in MVar are the metavariables that will occur in our reduction rules and which we will have to instantiate in order to derive specific application of those rules. In other words, the variables in Var are there to express the binding structure within the terms being reduced and the metavariables in MVar are there to stand in for specific terms when applying a reduction rule.

Definition 6.5. *The* ***metaterms*** *of a CRS are given inductively:*

- *variables are metaterms*
- *if t is a metaterm and x a variable, then $[x]t$ is a metaterm called* ***abstraction***
- *if F is an n-ary function symbol and t_1,...,t_n are metaterms, then $F(t_1, \ldots, t_n)$ is a metaterm*
- *if M is an n-ary metavariable and t_1,...,t_n are metaterms, then $M(t_1, \ldots, t_n)$ is a metaterm*

Definition 6.6. *The* ***terms*** *of a CRS are its metaterms which do not contain any metavariables.*

To finish the formal introduction of CRSs, we give the definition of a CRS reduction rule.

Definition 6.7. *A* ***CRS reduction rule*** *is a pair of metaterms $s \to t$ such that:*

- *s and t are both closed, i.e. all variables are bound using the $[_]_$ abstraction binder*
- *s is of the form $F(t_1, \ldots, t_n)$*
- *all the metavariables that occur in t also occur in s*
- *any metavariable M that occurs in s only occurs in the form $M(x_1, \ldots, x_k)$, where x_i are pairwise distinct variables*

Definition 6.8. *A* ***Combinatory Reduction System (CRS)*** *is a pair of a CRS alphabet and a set of CRS reduction rules.*

We will only sketch the way that a CRS gives rise to a reduction relation and we will direct curious readers to Sections 11 and 12 of [74]. When we instantiate the metavariables in a CRS rule, we use a *valuation* that assigns to every n-ary metavariable a term with holes labelled from 1 to n. The instantiation of $M(t_1, \ldots, t_n)$ then replaces the metavariable M using the valuation and then fills the holes labelled $1, \ldots, n$ with the terms $t_1, \ldots, t_n$ respectively. The crucial detail is that in a particular context, a metavariable can only be instantiated with terms M that do not contain any free variables bound in that context. This means that for the instantiation of M to contain a variable bound in its context, M must explicitly take that variable as an argument. All other variables not explicitly declared can therefore be safely assumed to not occur freely within M. Consider the following examples of β and η-reduction.

$$(\lambda x.\, M(x))\, N \to M(N)$$
$$\lambda x.\, N\, x \to N$$

More formally written as:

$$@(\lambda([x]M(x)), N) \to M(N)$$
$$\lambda([x]@(N, x)) \to N$$

where λ is a unary function symbol and @ is a binary function symbol. In both of the versions, M is a unary metavariable and N is a nullary metavariable. In the rule for β-reduction, we can observe how the idea of instantiating metavariables by terms with holes lets us express the same idea for which we had to introduce the meta-level operation of substitution. In the rule for η-reduction, we see that N appears in a context where x is bound but it does not have x as one of its arguments. Therefore, it will be impossible to instantiate N in such a way that it contains a free occurrence of x. In both of those rules, we were able to get rid of meta-level operations (substitution) and conditions ($x \notin FV(N)$) and have them both implemented by the formalism itself.

6.2 $\llparenthesis \lambda \rrparenthesis$ as a CRS

We will now see how to rephrase the reduction rules of $\llparenthesis \lambda \rrparenthesis$ in order to fit in to the CRS framework. We have already seen how to translate the β and η rules in the previous subsection. The next rules to address are the rules defining the semantics of the $\llparenthesis\ \rrparenthesis$ handlers. We will repeat the rules for handlers to make the issue at hand clear.

$$\begin{aligned}
&\llparenthesis (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \rrparenthesis\, (\eta\, N) \to && \text{rule } \llparenthesis \eta \rrparenthesis \\
&M_\eta\, N && \\[1em]
&\llparenthesis (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \rrparenthesis\, (\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) \to && \text{rule } \llparenthesis \mathtt{op} \rrparenthesis \\
&M_j\, N_{\mathrm{p}}\, (\lambda x.\, \llparenthesis (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \rrparenthesis\, N_{\mathrm{c}}) && \text{where } j \in I \\
& && \text{and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta) \\[1em]
&\llparenthesis (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \rrparenthesis\, (\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) \to && \text{rule } \llparenthesis \mathtt{op}' \rrparenthesis \\
&\mathtt{op}_j\, N_{\mathrm{p}}\, (\lambda x.\, \llparenthesis (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \rrparenthesis\, N_{\mathrm{c}}) && \text{where } j \notin I \\
& && \text{and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta)
\end{aligned}$$

The syntax of CRSs does not allow us to use the $(\mathtt{op}_i \colon M_i)_{i \in I}$ notation nor capture the $j \in I$ or $j \notin I$ conditions. The symbols $\mathtt{op}_i$ are problematic as well, since technically, they are not concrete $\llparenthesis \lambda \rrparenthesis$ syntax but metavariables standing in for operation symbols. We do away with all of the above problems by expanding these meta-notations and adding a separate rule for every possible instantiation of the schema. This means that for each sequence of distinct operation symbols $\mathtt{op}_1,\ldots,\mathtt{op}_n$, we end up with:

- a special rewriting rule $\llparenthesis \mathtt{op}_1 \colon M_1, \ldots, \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \rrparenthesis\, (\eta\, N) \to M_\eta\, N$
- for every $1 \leq i \leq n$, a special rewriting rule
  $\llparenthesis \mathtt{op}_1 \colon M_1, \ldots, \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \rrparenthesis\, (\mathtt{op}_i\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}}(x)))$
  $\to M_i\, N_{\mathrm{p}}\, (\lambda x.\, \llparenthesis \mathtt{op}_1 \colon M_1, \ldots, \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \rrparenthesis\, N_{\mathrm{c}}(x))$
- for every $\mathtt{op}' \in \mathcal{E} \setminus \{\mathtt{op}_i \| 1 \leq i \leq n\}$, a special rewriting rule
  $\llparenthesis \mathtt{op}_1 \colon M_1, \ldots, \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \rrparenthesis\, (\mathtt{op}'\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}}(x)))$
  $\to \mathtt{op}'\, N_{\mathrm{p}}\, (\lambda x.\, \llparenthesis \mathtt{op}_1 \colon M_1, \ldots, \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \rrparenthesis\, N_{\mathrm{c}}(x))$

The rule for the cherry $\downarrow$ extraction operator is already in CRS form, so all we have to do is address the rules for the $\mathcal{C}$ operator. We present them side-by-side in their original form and in CRS-style. Original:

$$\begin{aligned}
\mathcal{C}\, (\lambda x.\, \eta\, M) &\to \eta\, (\lambda x.\, M) \\
\mathcal{C}\, (\lambda x.\, \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}})) &\to \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}})) \\
&\quad \text{where } x \notin \mathrm{FV}(M_{\mathrm{p}})
\end{aligned}$$

CRS-style:

$$\begin{aligned}
\mathcal{C}\, (\lambda x.\, \eta\, (M(x))) &\to \eta\, (\lambda x.\, M(x)) \\
\mathcal{C}\, (\lambda x.\, \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}(x, y))) &\to \mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_{\mathrm{c}}(x, y)))
\end{aligned}$$

We can see that the only difference is to replace "simple" metavariables M, M_{p} and M_{c} with their higher-order versions: the unary M, nullary M_{p} and binary M_{c}. We see that every CRS metavariable is applied to the variables in its scope, except for M_{p}, which thus loses access to the variable x. This way, the condition that x must not appear free in M_{p} is now encoded directly in the reduction rule itself. In 6.1, we have said that a CRS is formed by a set of reduction rules and by an alphabet. We have already seen all of the rules of our CRS (β and η were given at the end of 6.1 and the $\downarrow$ rule is the same as the original one in 2.3). In order to have a complete definition, all that remains is to identify the alphabet. The set of variables Var is exactly

the set of variables $\mathcal{X}$ used in the definition of $(\!|\lambda|\!)$. The set of metavariables $MVar$ consists of the unary M, nullary N, nullary N_p, unary N_c, nullary M_p, binary M_c, nullary M_i and nullary M_η. The set of function symbols is composed of the following:

- the binary symbol @ for function application
- the unary symbol λ for function abstraction
- a nullary symbol for every constant in the signature Σ
- the unary symbol η for the injection operator
- a binary symbol $\mathtt{op}$ for every $\mathtt{op} \in \mathcal{E}$
- a $(n+2)$-ary symbol $((\!|\,\mathtt{op}_1{:}\,_,\ \ldots,\ \mathtt{op}_n{:}\,_,\ \eta{:}\,_\,|\!)\,_)$ for every sequence $\mathtt{op}_1, \ldots, \mathtt{op}_n$ of distinct symbols from $\mathcal{E}$ of length n
- the unary symbol $\downharpoonleft$ for the extraction operator
- the unary symbol $\mathcal{C}$ for the $\mathcal{C}$ operator

In giving the CRS-style reduction rules above, we have used the "native" syntax of $(\!|\lambda|\!)$ instead of writing out everything in terms of function symbols. For clarity, we give the rules governing the relationship of the two. We write:

- $@(t, u)$ as $t\,u$
- $\lambda([x]t)$ as $\lambda x.\, t$
- $\eta(t)$ as $\eta\, t$
- $\mathtt{op}(t_\mathrm{p}, [x]t_\mathrm{c})$ as $\mathtt{op}\, t_\mathrm{p}\, (\lambda x.\, t_\mathrm{c})$[24]
- $((\!|\,\mathtt{op}_1{:}\,_,\ \ldots,\ \mathtt{op}_n{:}\,_,\ \eta{:}\,_\,|\!)\,_)(t_1, \ldots, t_n, t_\eta, u)$ as $(\!|\,\mathtt{op}_1{:}\,t_1,\ \ldots,\ \mathtt{op}_n{:}\,t_n,\ \eta{:}\,t_\eta\,|\!)\, u$
- $\downharpoonleft(t)$ as $\downharpoonleft t$
- $\mathcal{C}(t)$ as $\mathcal{C}\, t$

We have connected the terms of $(\!|\lambda|\!)$ with CRS terms and we have also expressed all of our reduction rules in terms of CRS reduction rules. As in $(\!|\lambda|\!)$, CRS then proceeds to take a context closure of this redex-contractum relation. Our translation from $(\!|\lambda|\!)$ to a CRS also preserves subterms[25] and so we end up constructing the same reduction relation.

6.3 Orthogonal CRSs

In order to use Theorem 6.2, we need to show that our CRS is orthogonal, so let us start us by looking at what "orthogonal" means in the context of CRSs.

Definition 6.9. *A CRS is **orthogonal** if it is non-overlapping and left-linear.*

We will need to satisfy two criteria: no overlaps and left linearity. We will start with the latter.

Definition 6.10. *A CRS is **left-linear** if the left-hand sides of all its reduction rules are linear. A CRS metaterm is **linear** if no metavariable occurs twice within it.*

By going through the rules we have given in 6.2, we can see at a glance that no rule uses the same metavariable twice in its left-hand side and so our CRS is indeed left-linear.

[24] Note that with this translation, $\mathtt{op}\, t_\mathrm{p}\, (\lambda x.\, t_\mathrm{c})$ does not contain $\lambda x.\, t_\mathrm{c}$ as a subterm. This is the same as in $(\!|\lambda|\!)$, where the notion of evaluation context (see 2.3) does not identify $\lambda x.\, t_\mathrm{c}$, but rather t_c, as a subterm of $\mathtt{op}\, t_\mathrm{p}\, (\lambda x.\, t_\mathrm{c})$. This becomes important in our discussion of confluence since it makes it impossible to make the λ disappear by something like η-reduction.

[25] More precisely, if a is a subterm of b in $(\!|\lambda|\!)$ then the CRS version of a is a subterm of the CRS version of b. In the other direction, whenever a is a variable or a function-headed term which is a subterm of b in the CRS version of $(\!|\lambda|\!)$, then the corresponding a in $(\!|\lambda|\!)$ is a subterm of the corresponding b.

Definition 6.11. *A CRS is **non-overlapping** if:*

- *Let $r = s \to t$ be some reduction rule of the CRS and let $M_1, \ldots, M_n$ be all the metavariables occurring in the left-hand side s. Whenever we can instantiate the metavariables in s such that the resulting term contains a redex for some other rule r', then said redex must be contained in the instantiation of one of the metavariables M_i.*
- *Similarly, whenever we can instantiate the metavariables in s such that the resulting term* properly contains *a redex* for the same rule r, *then that redex as well must be contained in the instantiation of one of the metavariables M_i.*

In simpler words, no left-hand side of any rule can contain bits which look like the top of the left-hand side of some other rule. Let us try and verify this property in $\llparenthesis \lambda \rrparenthesis$:

- The $\llparenthesis\ \rrparenthesis$ rules have no overlaps with any of the other rules. Their left-hand sides are constructed only of the $\llparenthesis\ \rrparenthesis$ symbols and the `op` and η constructors. Since there is no reduction rule headed by `op` and η, they have no overlap with any of the other rules. Furthermore, the three $\llparenthesis\ \rrparenthesis$ rules are mutually exclusive, so there is no overlap between themselves.
- The ⫰ rule does not overlap with any of the other neither, since the left-hand side contains only ⫰ and η, and there is no reduction rule headed by η.
- The $\mathcal{C}$ rules are both mutually exclusives, so there is no overlap between the two. However, their left-hand sides are built not only out of $\mathcal{C}$, `op` and η, but also of λ, for which there is the η-reduction rule. Fortunately, in this case, the $\mathcal{C}$ rules only apply when the λ-abstraction's body is an η expression or an `op` expression, whereas the η rule applies only when the body is an application expression.[26] Therefore, there is no overlap.

We have established that all the reduction rules in our system are pairwise non-overlapping *except* for β and η. However, these two have a notorious overlap. We can instantiate the metavariables in the left-hand side of the β rule to get a term which contains an η-redex which shares the λ-abstraction with the β-redex.

$$(\lambda x.\, y\, x)\, z$$

We can also instantiate the metavariables in the left-hand side of the η rule to create a β-redex which shares the application with the η-redex.

$$\lambda x.\, (\lambda z.\, z)\, x$$

Because of these overlaps, the $\llparenthesis \lambda \rrparenthesis$ CRS is therefore *not* orthogonal. However, we can still make good use of Theorem 6.2.

Lemma 6.12. ***Confluence of*** $\llparenthesis \lambda \rrparenthesis_{-\eta}$ *The $\llparenthesis \lambda \rrparenthesis$ reduction system without the η rule is confluent.*

Proof. If we exclude the η rule, we have a CRS which is left-linear and also non-overlapping.[27] Therefore, it is orthogonal and thanks to Theorem 6.2, also confluent. □

Lemma 6.13. ***Confluence of** η**-reduction*** *The reduction system on $\llparenthesis \lambda \rrparenthesis$ terms containing only the η-reduction rule is confluent.*[28]

Proof. We have seen that η is a valid left-linear CRS rule. It also does not overlap itself since its left-hand side does not contain any λ subexpression. The CRS consisting of just the η rule is therefore orthogonal and confluent. □

[26]This is not so much a fortunate conincidence but rather a deliberate choice in the design of the calculus. For example, it is one of the reasons why, in $\llparenthesis \lambda \rrparenthesis$, $\eta\, x$ is not decomposed as an application of the built-in function η to x, but is treated as a special form.

[27]We know that β does not overlap any of the other rules. Neither does it overlap itself since its left-hand side does not have an application subexpression.

[28]This also holds for $\llparenthesis \lambda \rrparenthesis$ with sums and products since their rules are left-linear and do not overlap with the $\llparenthesis \lambda \rrparenthesis$ rules.

6.4 Putting η Back in $(\!|\lambda|\!)$

We have shown that both $(\!|\lambda|\!)_{-\eta}$ and η are confluent. The reduction relation of the complete $(\!|\lambda|\!)$ calculus is the union of these two reduction relations. Using the Lemma of Hindley-Rosen (1.0.8.(2) in [75]), we can show that this union is confluent by showing that the two reduction relations commute together.

Definition 6.14. *Let $\to_1$ and $\to_2$ be two reduction relations on the same set of terms A. $\to_1$ and $\to_2$* ***commute*** *if for every $a, b, c \in A$ such that $a \twoheadrightarrow_1 b$ and $a \twoheadrightarrow_2 c$, there exists a $d \in A$ such that $b \twoheadrightarrow_2 d$ and $c \twoheadrightarrow_1 d$.*

Lemma 6.15. ***Lemma of Hindley-Rosen*** *[75]*

Let $\to_1$ and $\to_2$ be two confluent reduction relations on the same set of terms. If $\to_1$ and $\to_2$ commute, then the reduction relation $\to_1 \cup \to_2$ is confluent.

We will not be proving the commutativity directly from the definition. Instead, we will use a lemma due to Hindley (1.0.8.(3) in [75]).

Lemma 6.16. *Let $\to_1$ and $\to_2$ be two reduction relations on the same set of terms A. Suppose that whenever there are $a, b, c \in A$ such that $a \to_1 b$ and $a \to_2 c$, there is also some $d \in A$ such that $b \twoheadrightarrow_2 d$ and $c \to_1^= d$ (meaning $c \to_1 d$ or $c = d$). In that case, $\to_1$ commutes with $\to_2$. [75]*

We can use this to prove that $(\!|\lambda|\!)_{-\eta}$ commutes with the η-reduction rule.

Lemma 6.17. ***Commutativity of*** η ***and*** $(\!|\lambda|\!)_{-\eta}$ *The reduction relations induced by η and by the rest of the $(\!|\lambda|\!)$ rules commute.*

Proof. We will prove this lemma by an appeal to Lemma 6.16. Let $\to_\eta$ be the reduction relation induced by the rule η and $\to_{(\!|\lambda|\!)_{-\eta}}$ the reduction relation induced by all the other reduction rules in $(\!|\lambda|\!)$. We need to prove that for all terms a, b and c where $a \to_{(\!|\lambda|\!)_{-\eta}} b$ and $a \to_\eta c$, we have a term d such that $b \twoheadrightarrow_\eta d$ and $c \to^=_{(\!|\lambda|\!)_{-\eta}} d$. This will turn out to be a routine proof by induction on the structure of the term a. The base cases are trivial since terms without any proper subterms happen to have no redexes in $(\!|\lambda|\!)$ and therefore trivially satisfy the criterion. In the inductive step, we will proceed by analyzing the relative positions of the redexes which led to the reductions $a \to_{(\!|\lambda|\!)_{-\eta}} b$ and $a \to_\eta c$.

- If both reductions occurred within a common subterm of a, i.e. $a = C[a']$, $b = C[b']$ and $c = C[c']$ while at the same time $a' \to_{(\!|\lambda|\!)_{-\eta}} b'$ and $a' \to_\eta c'$, we can use the induction hypothesis for a'. This gives us a d' such that $b' \twoheadrightarrow_\eta d'$ and $c' \to^=_{(\!|\lambda|\!)_{-\eta}} d'$ and therefore we also have $d = C[d']$ with $b \twoheadrightarrow_\eta d$ and $c \to^=_{(\!|\lambda|\!)_{-\eta}} d$.
- If both reductions occurred within non-overlapping subterms of a, i.e. $a = C[a_1, a_2]$, $b = C[b', a_2]$ and $c = C[a_1, c']$ with $a \to_{(\!|\lambda|\!)_{-\eta}} b$ and $a \to_\eta c$: We can take $d = C[b', c']$ since we have $b \twoheadrightarrow_\eta d$ in one step and $c \to^=_{(\!|\lambda|\!)_{-\eta}} d$ in one step too.
- If the redex in $a \to_{(\!|\lambda|\!)_{-\eta}} b$ is the entire term a, but the redex in $a \to_\eta c$ is a proper subterm of a: We will solve this by case analysis on the form of a:
  - If a is an application: Since a is an application and also a $(\!|\lambda|\!)_{-\eta}$-redex, it must match the left-hand side of the β rule, $(\lambda x.\, M(x))\, N$, and b must be $M(N)$.
    * We will first deal with the case when the η-redex which lead to c originated in $M(x)$. In that case $M(x) \to_\eta M'(x)$ and $c = (\lambda x.\, M'(x))\, N$. Our sought-after d is then $M'(N)$, since $c \to^=_{(\!|\lambda|\!)_{-\eta}} d$ via β in one step and $b = M(N) \twoheadrightarrow_\eta d = M'(N)$.
    * Now we get to one of the two interesting cases which necessitated this whole lemma: the overlap between β and η, with β on the top. If the η-redex did not originate in $M(x)$, then the η-redex must be $\lambda x.\, M(x)$. Therefore, $M = T\,x$ and $a = (\lambda x.\, T\,x)\, N$. Performing the η-reduction yields $c = T\,N$. In this case, both b and c are equal to $T\,N$ and so we can choose $T\,N$ as our d.
  - If a is any other kind of term: Let $l \to r$ be the rule used in $a \to_{(\!|\lambda|\!)_{-\eta}} b$. Not counting β, which only acts on applications and which we dealt with just above, the rules of $(\!|\lambda|\!)_{-\eta}$ do not overlap with the η rule. This means the η-redex which led to c must lie entirely inside a part of l which corresponds to a metavariable. Let M be that metavariable, then we will decompose l into $L(M)$

and r into $R(M)$. We have $a = L(a')$ for some a', $b = R(a')$ and $c = L(a'')$[29]. Our d will be $R(a'')$ and we have $b = R(a') \twoheadrightarrow_\eta d = R(a'')$ in several steps[30] and $c = L(a'') \to^{=}_{(\!|\lambda|\!)_{-\eta}} d = R(a'')$ in one step of $l \to r$.

- If the redex in $a \to_\eta c$ is the entire term a, but the redex in $a \to_{(\!|\lambda|\!)_{-\eta}} b$ is a proper subterm of a: In this case, a must be an abstraction that matches the left-hand side of the η rule, i.e. $a = \lambda x.\, N\, x$. Also, we have $c = N$.
  - As before, we will first deal with the case when the $(\!|\lambda|\!)_{-\eta}$-redex is contained completely within N. Then $N \to_{(\!|\lambda|\!)_{-\eta}} N'$ and $b = \lambda x.\, N'\, x$. The common reduct d is N' since $b \twoheadrightarrow_\eta d$ in one step and $c = N \to^{=}_{(\!|\lambda|\!)_{-\eta}} d = N'$ as established before.
  - Now this is where we deal with the second overlap between β and η in our reduction system, the one with η on top. The $(\!|\lambda|\!)_{-\eta}$-redex in a must be $N\, x$ and the reduction rule in question must therefore be β. Therefore, $N = \lambda y.\, T(y)$ and $a = \lambda x.\, (\lambda y.\, T(y))\, x$. Performing the β-reduction gives us $b = \lambda x.\, T(x)$ which is, however, equal to $c = N = \lambda y.\, T(y)$. So we can choose $d = b$ and we are done.
- If a is the redex for both reductions $a \to_{(\!|\lambda|\!)_{-\eta}} b$ and $a \to_\eta c$, then a must match the left-hand side of a $(\!|\lambda|\!)_{-\eta}$ rule and the η rule. However, this is impossible since the left-hand side of the η rule is headed by abstraction, which is the case for none of the rules of $(\!|\lambda|\!)_{-\eta}$.

□

Equipped with this lemma, we can go on to prove our main result, Theorem 6.18, the confluence of $(\!|\lambda|\!)$.

Theorem 6.18. ***Confluence of*** $(\!|\lambda|\!)$
The reduction relation $\to$ on the set of $(\!|\lambda|\!)$ terms, defined by the reduction rules in 2.3, is confluent.

Proof. From Lemma 6.12, we know that the $(\!|\lambda|\!)_{-\eta}$ system is confluent and from Lemma 6.13, we know that the η-reduction rule is confluent as well. Lemma 6.17 tells us that these two reduction systems commute and therefore, by Lemma 6.15, their union, which is the $(\!|\lambda|\!)$ reduction system, commutes as well. □

7 Termination

Definition 7.1. *A reduction relation is* ***terminating*** *if there is no infinite chain $M_1 \to M_2 \to \ldots$.*

In this section, we will prove termination with a similar strategy as the one we employed for confluence. $(\!|\lambda|\!)$ is an extension of the λ-calculus with computation types and some operations on computations. Our computations can be thought of as algebraic expressions, i.e. they have a tree-like inductive structure. The reason that all computations in $(\!|\lambda|\!)$ terminate is that the operations defined on computations rely on well-founded recursion. However, it is quite tricky to go from this intuition to a formal proof of termination. Fortunately, we can rely on existing results. Blanqui, Jouannaud and Okada have introduced Inductive Data Type Systems (IDTSs) [76, 77]. Like CRSs, IDTSs are a class of rewriting systems for which we can prove certain interesting general results. In this section, we will start by examining the definition of an IDTS and fitting $(\!|\lambda|\!)$ into that definition. The theory of IDTSs comes with a sufficient condition for termination known as the General Schema. $(\!|\lambda|\!)$ will not satisfy this condition and so we will first transform it using Hamana's technique of higher-order semantic labelling [78]. As with our proof of confluence, we will first consider the case of $(\!|\lambda|\!)$ without η-reduction and then add η manually while preserving termination. The plan will look like this:

- IDTS = Typed CRS (7.1)
- The $(\!|\lambda|\!)_\tau$ IDTS (7.2)

[29] Since our rules are left-linear, M is guaranteed to appear in $L(M)$ at most once. Therefore, if $a' \to_\eta a''$ in one step, then also $L(a') \to_\eta L(a'')$ in one step as well.

[30] a' can occur multiple times in $R(a')$ when the rule $l \to r$ is duplicating (which is actually the case for the $(\!|\texttt{op}|\!)$ rules). However, we are able to go from $R(a')$ to $R(a'')$ in multiple steps. NB: This is why we use Lemma 6.16 instead of trying to prove commutativity directly.

- if $(\!|\lambda|\!)_\tau$ terminates, then $(\!|\lambda|\!)_{-\eta}$ terminates (Lemma 7.11)

- Blanqui [77]: General Schema $\Rightarrow$ termination (7.3)
- Hamana [78]: IDTS R terminates iff the labelled IDTS $\overline{R}$ terminates (7.4)
  - Theorem 7.40: $\overline{(\!|\lambda|\!)_\tau}$ terminates (via Blanqui [77])
  - Corollary 7.41: $(\!|\lambda|\!)_\tau$ terminates (via Hamana [78])
  - Corollary 7.42: $(\!|\lambda|\!)_{-\eta}$ terminates (via Lemma 7.11)
- $(\!|\lambda|\!)_{-\eta} + \eta$ terminates (7.5, Theorem 7.46)
  - because $(\!|\lambda|\!)_{-\eta}$ and η are exchangeable (Lemma 7.44)
  - and therefore $(\!|\lambda|\!)$ is strongly normalizing (Theorem7.47)

7.1 Inductive Data Type Systems

We will go by the revised definition of Inductive Data Type Systems that figures in [77] and [78]. This formulation extends IDTSs to higher-order rewriting and does so using the CRS formalism that we introduced earlier.

Definition 7.2. *An* ***Inductive Data Type System (IDTS)*** *is a pair of an IDTS alphabet and a set of IDTS rewrite rules.*

Just like a CRS, an IDTS is an alphabet coupled with some rewrite rules. Let us first look at the alphabet and the rules for building terms out of the elements of the alphabet; the rewrite rules will follow.

Definition 7.3. *The set of types $T(\mathcal{B})$ contains:*

- *all the types from $\mathcal{B}$*
- *a type $\alpha \Rightarrow \beta$ for every α and β in $T(\mathcal{B})$*

Definition 7.4. *An* ***IDTS alphabet*** *consists of:*

- $\mathcal{B}$*, a set of* base types
- $\mathcal{X}$*, a family $(X_\tau)_{\tau \in T(\mathcal{B})}$ of sets of* variables
- $\mathcal{F}$*, a family $(F_{\alpha_1,\ldots,\alpha_n,\beta})_{\alpha_1,\ldots,\alpha_n,\beta \in T(\mathcal{B})}$ of sets of* function symbols
- $\mathcal{Z}$*, a family $(Z_{\alpha_1,\ldots,\alpha_n,\beta})_{\alpha_1,\ldots,\alpha_n,\beta \in T(\mathcal{B})}$ of sets of* metavariables

The distinction between a CRS-alphabet and an IDTS alphabet is that the IDTS alphabet comes equipped with a set of types. Furthermore, all the other symbols in the alphabet are indexed by types, so we end up with typed variables, typed function symbols and typed metavariables. When we consider IDTS metaterms, we admit only well-typed terms. The definition of IDTS metaterms refines the definition of CRS metaterms by restraining term formation in accordance with the types.

Definition 7.5. *The* ***typed metaterms*** *of an IDTS are given inductively:*

- *variables from X_τ are metaterms of type τ*
- *if t is a metaterm of type β and x a variable from X_α, then $[x]t$ is a metaterm of type $\alpha \Rightarrow \beta$ called* ***abstraction***
- *if F is an function symbol from $F_{\alpha_1,\ldots,\alpha_n,\beta}$ and $t_1,\ldots,t_n$ are metaterms of types $\alpha_1,\ldots,\alpha_n$, respectively, then $F(t_1,\ldots,t_n)$ is a metaterm of type β*
- *if M is a metavariable from $Z_{\alpha_1,\ldots,\alpha_n,\beta}$ and $t_1,\ldots,t_n$ are metaterms of types $\alpha_1,\ldots,\alpha_n$, respectively, then $M(t_1,\ldots,t_n)$ is a metaterm of type β*

Definition 7.6. *The* ***terms*** *of an IDTS are its metaterms which do not contain any metavariables.*

The definition of an IDTS rewrite rule is almost identical to the one for CRS reduction rules. The only difference is the extra condition stating that the redex and contractum must have identical types.

Definition 7.7. *An **IDTS rewrite rule** is a pair of metaterms $s \to t$ such that:*

- *s and t are both closed, i.e. all variables are bound using the $[_]_$ abstraction binder*
- *s is of the form $F(t_1, \ldots, t_n)$*
- *all the metavariables that occur in t also occur in s*
- *any metavariable M that occurs in s only occurs in the form $M(x_1, \ldots, x_k)$, where x_i are pairwise distinct variables*
- *s and t are both of the same type*

As stated above, an IDTS is just an alphabet along with a set of rewrite rules. An IDTS induces a rewriting relation in exactly the same way as a CRS does, see [77] for more details.

7.2 $(\!|\lambda|\!)$ as an IDTS

Now we will link $(\!|\lambda|\!)$ to the IDTS framework in order to benefit from its general termination results. The biggest obstacle will be that IDTS assigns a fixed type to every symbol. In $(\!|\lambda|\!)$, symbols are polymorphic: the η constructor can produce expressions like $\eta \star : \mathcal{F}_E(1)$ or $\eta\,(\lambda x.\, x) : \mathcal{F}_E(\alpha \to \alpha)$ and that for any choice of E. We would therefore like to replace function symbols such as η with specialized symbols $\eta_{\mathcal{F}_E(\alpha)}$. For a given type α and effect signature E, the symbol $\eta_{\mathcal{F}_E(\alpha)}$ would have the type $\alpha \to \mathcal{F}_E(\alpha)$, i.e. it would belong to $F_{\alpha,\mathcal{F}_E(\alpha)}$. We will call this calculus with specialized symbols $(\!|\lambda|\!)_\tau$. There will not be a bijection between $(\!|\lambda|\!)$ and $(\!|\lambda|\!)_\tau$ since a single term in $(\!|\lambda|\!)$ will generally correspond to a multitude of specialized versions in $(\!|\lambda|\!)_\tau$ (think of $\lambda x.\, x$ in $(\!|\lambda|\!)$ versus $\lambda x_\iota.\, x_\iota, \lambda x_o.\, x_o \ldots$ in $(\!|\lambda|\!)_\tau$). Therefore, the results we prove for $(\!|\lambda|\!)_\tau$ will not automatically transfer to $(\!|\lambda|\!)$. In the rest of this subsection, we will elaborate the definition of $(\!|\lambda|\!)_\tau$ and show why termination carries over from $(\!|\lambda|\!)_\tau$ to $(\!|\lambda|\!)_{-\eta}$.[31]

7.2.1 Defining $(\!|\lambda|\!)_\tau$

$(\!|\lambda|\!)_\tau$ will be defined as an IDTS. This means we need to first identify the alphabet. The base types $\mathcal{B}$ of $(\!|\lambda|\!)_\tau$ will be the set of types of $(\!|\lambda|\!)$. [32] Note that both $(\!|\lambda|\!)$ and IDTS have a notion of function type, but the notation is different. Contrary to common practice, in our exposition of IDTS we use $\alpha \Rightarrow \beta$ for the IDTS function type. This allows us to keep using the $\alpha \to \beta$ notation for $(\!|\lambda|\!)$ types, as we do in the rest of the article. Next, we will introduce function symbols for all the syntactic constructions of $(\!|\lambda|\!)$, except for abstraction, which is handled by the $[_]_$ binder construct already found in IDTSs:

- $\mathsf{ap}_{\alpha,\beta} \in F_{\alpha\to\beta,\alpha,\beta}$ (i.e. for every pair of types α and β, there will be a function symbol $\mathsf{ap}_{\alpha,\beta}$ of type $(\alpha \to \beta) \Rightarrow \alpha \Rightarrow \beta$ in our alphabet)
- $\lambda_{\alpha,\beta} \in F_{\alpha\Rightarrow\beta,\alpha\to\beta}$
- $c \in F_\alpha$ for any constant $c : \alpha \in \Sigma$
- $\eta_{\alpha,E} \in F_{\alpha,\mathcal{F}_E(\alpha)}$
- $\mathtt{op}_{\gamma,E} \in F_{\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}$ for any operation symbol $\mathtt{op}$ from $\mathcal{E}$ and any E such that $\mathtt{op} : \alpha \rightarrowtail \beta \in E$
- $\downarrow_\alpha \in F_{\mathcal{F}_\emptyset(\alpha),\alpha}$
- $(\!|\,|\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1\to(\beta_1\to\mathcal{F}_{E'}(\delta)),\ldots,\alpha_n\to(\beta_n\to\mathcal{F}_{E'}(\delta)),\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$ where:
  - $\mathtt{op}_1 : \alpha_1 \rightarrowtail \beta_1 \in E,\ \ldots,\ \mathtt{op}_n : \alpha_n \rightarrowtail \beta_n \in E$

[31] In the sequel, we will ignore the η-reduction and use IDTSs to prove the termination of $(\!|\lambda|\!)$ without η-reduction, $(\!|\lambda|\!)_{-\eta}$.

[32] Note that throughout this section, we will make a distinction between two notions of "basic" types: atomic types and base types. Atomic types are the basic types of $(\!|\lambda|\!)$. Base types are the basic types of IDTSs. In our particular IDTS, the base types consist of *all* the types of $(\!|\lambda|\!)$, i.e. the atomic types, the $(\!|\lambda|\!)$ function types $\alpha \to \beta$ and the computation types. This means that, from the point of view of the IDTS, $(\!|\lambda|\!)$ function types and computation types are just another base type.

- $E \setminus \{\mathtt{op}_1, \ldots, \mathtt{op}_n\} \subseteq E'$

- $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha \to \mathcal{F}_E(\beta), \mathcal{F}_E(\alpha \to \beta)}$

The list above is based on the typing rules of $(\!|\lambda|\!)$ found on Figure 1. We convert the typing rules of $(\!|\lambda|\!)$ into the typed function symbols of $(\!|\lambda|\!)_\tau$ with the following process:

- We take a typing rule of $(\!|\lambda|\!)$, other than [var] (since variables are already present in the language of IDTS terms).
- We identify all the type-level metavariables. That is, metavariables $\alpha, \beta, \gamma \ldots$ ranging over types, metavariables $E, E' \ldots$ ranging over effect signatures and metavariables $\mathtt{op}$ ranging over operation symbols.
- We strip these metavariables down to a minimal non-redundant set (e.g. in the [$\mathtt{op}$] rule, we have that $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, therefore E and $\mathtt{op}$ determine α and β and α and β are redundant).
- We introduce a family of symbols: for every possible instantiation of the metavariables mentioned above, we will have a different symbol. The arity of the symbol will correspond to the number of typing judgments that serve as hypotheses to the typing rule. The types of the arguments and of the result will be derived from the types of the judgments of the hypotheses and the conclusion respectively. If a variable of type α is bound in a premise of type β, then that will correspond to the IDTS function type $\alpha \Rightarrow \beta$.
  - Example: In the [η] rule, we have two metavariables: α standing in for a type and E standing in for an effect signature. The rule has one typing judgment hypothesis. For every type α and every effect signature E, we will therefore have a unary symbol $\eta_{\alpha,E}$ of type $\alpha \Rightarrow \mathcal{F}_E(\alpha)$ (i.e. belonging to $F_{\alpha,\mathcal{F}_E(\alpha)}$).

A specifically-typed symbol in $(\!|\lambda|\!)_\tau$ then corresponds to an instantiation of the type metavariables in a $(\!|\lambda|\!)$ typing rule. We can follow this correspondence further and see that $(\!|\lambda|\!)_\tau$ IDTS terms, written using the above function symbols, correspond to typing derivations in $(\!|\lambda|\!)$. Our alphabet now has types and function symbols. We also need to specify the sets of variables and metavariables and so we will take some arbitrary sets with $x_\tau, y_\tau, \ldots \in X_\tau$ and $M_{\alpha_1,\ldots,\alpha_n,\beta}, N_{\alpha_1,\ldots,\alpha_n,\beta}, \ldots \in Z_{\alpha_1,\ldots,\alpha_n,\beta}$. To complete our IDTS, we have to give the rewrite rules. The rules for $(\!|\lambda|\!)_\tau$ are given in Figure 5. An important property of an IDTS rewrite rule is that both its left-hand and right-hand side are well-typed and that they have the same type. In order to facilitate the reader's verification that this is indeed the case, we have used a different labelling scheme for function symbols. When we write $f_{\alpha_1,\ldots,\alpha_n,\beta}$, we are referring to the instance of symbol f which has the type $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$ (i.e. belongs to $F_{\alpha_1,\ldots,\alpha_n,\beta}$). This way, instead of using a symbol name like $\eta_{\alpha,E}$, forcing to look up its type $\alpha \Rightarrow \mathcal{F}_E(\alpha)$, we will refer to this symbol directly as $\eta_{\alpha,\mathcal{F}_E(\alpha)}$. In Figure 5 presents the rewrite rules with all the subscripts removed. This allows to get a high-level look at the term without any of the type annotation noise. By removing the type indices from the $(\!|\lambda|\!)_\tau$ IDTS rewrite rules, we get the $(\!|\lambda|\!)$ CRS-reduction rules of Section 6 (modulo the renaming of @ to $\mathtt{ap}$). When describing the rewrite rules for handlers, we introduce a shortcut $(\!|\,|\!)^{(\mathtt{op}_i)_{i \in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N_{\mathcal{F}_E(\gamma)})$, which stands for $(\!|\,|\!)$ partially applied to the clauses M^i and M^η. We then reuse this shortcut in all of the $(\!|\,|\!)$ rules.

7.2.2 *Connecting $(\!|\lambda|\!)_\tau$ to $(\!|\lambda|\!)$*

We have given a complete formal definition of $(\!|\lambda|\!)_\tau$. This will let us find a proof of termination for $(\!|\lambda|\!)_\tau$ using the theory of IDTSs. However, in order to carry over this result to our original calculus, we will need to formalize the relationship between the two.

Definition 7.8. ***Term*** *is a (partial) function from $(\!|\lambda|\!)_\tau$ terms to $(\!|\lambda|\!)$ terms which removes any type annotations (the subscripts on function symbols, variables and metavariables) and translates $(\!|\lambda|\!)_\tau$ syntax to $(\!|\lambda|\!)$ syntax using the following equations:*

$$\mathsf{ap}(\lambda([x]M(x)), N) \to \qquad \text{rules } \beta_{\alpha,\gamma}$$
$$M(N)$$
$$\mathsf{ap}_{\alpha\to\gamma,\alpha,\gamma}(\lambda_{\alpha\Rightarrow\gamma,\alpha\to\gamma}([x_\alpha]M_{\alpha,\gamma}(x_\alpha)), N_\alpha) \to$$
$$M_{\alpha,\gamma}(N_\alpha)$$

$$\text{let } (\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N_{\mathcal{F}_E(\gamma)})$$
$$= (\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{(\alpha_i\to(\beta_i\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta))_{i\in I},\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}((M^i_{\alpha_i\to(\beta_i\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)})_{i\in I}, M^\eta_{\gamma\to\mathcal{F}_{E'}(\delta)}, N_{\mathcal{F}_E(\gamma)})$$

$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}(\eta(N)) \to \qquad \text{rules } (\!|\,\eta\,|\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$$
$$\mathsf{ap}(M^\eta, N)$$
$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\eta_{\gamma,\mathcal{F}_E(\gamma)}(N_\gamma)) \to$$
$$\mathsf{ap}_{\gamma\to\mathcal{F}_{E'}(\delta),\gamma,\mathcal{F}_{E'}(\delta)}(M^\eta_{\gamma\to\mathcal{F}_{E'}(\delta)}, N_\gamma)$$

$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}(\mathsf{op}_j(N^{\mathrm{p}}, [y]N^{\mathrm{c}}(y))) \to \qquad \text{rules } (\!|\,\mathsf{op}\,|\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$$
$$\mathsf{ap}(\mathsf{ap}(M^j, N^{\mathrm{p}}), \lambda([y](\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}(N^{\mathrm{c}}(y)))) \qquad \text{where } j \in I$$
$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\mathsf{op}_{j\,\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}(N^{\mathrm{p}}_\alpha, [y_\beta]N^{\mathrm{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))) \to$$
$$\mathsf{ap}_{(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta),\beta\to\mathcal{F}_{E'}(\delta),\mathcal{F}_{E'}(\delta)}(\mathsf{ap}_{\alpha\to(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta),\alpha,(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)}(M^j_{\alpha\to(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)}, N^{\mathrm{p}}_\alpha), \lambda_{\beta\Rightarrow\mathcal{F}_{E'}(\delta),\beta\to\mathcal{F}_{E'}(\delta)}([y_\beta](\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N^{\mathrm{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))))$$

$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}(\mathsf{op}_j(N^{\mathrm{p}}, [y]N^{\mathrm{c}}(y))) \to \qquad \text{rules } (\!|\,\mathsf{op}'\,|\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$$
$$\mathsf{op}_j(N^{\mathrm{p}}, [y](\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}(N^{\mathrm{c}}(y))) \qquad \text{where } j \notin I$$
$$(\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\mathsf{op}_{j\,\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}(N^{\mathrm{p}}_\alpha, [y_\beta]N^{\mathrm{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))) \to$$
$$\mathsf{op}_{j\,\alpha,\beta\Rightarrow\mathcal{F}_{E'}(\delta),\mathcal{F}_{E'}(\delta)}(N^{\mathrm{p}}_\alpha, [y_\beta](\!|\ |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N^{\mathrm{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta)))$$

$$\downarrow\!\!\circ(\eta(N)) \to \qquad \text{rules } \downarrow\!\!\circ_\alpha$$
$$N$$
$$\downarrow\!\!\circ_{\mathcal{F}_\emptyset(\alpha),\alpha}(\eta_{\alpha,\mathcal{F}_\emptyset(\alpha)}(N_\alpha)) \to$$
$$N_\alpha$$

$$\mathcal{C}(\lambda([x]\eta(M(x)))) \to \qquad \text{rules } \mathcal{C}^\eta_{\alpha,\beta,E}$$
$$\eta(\lambda([x]M(x)))$$
$$\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]\eta_{\beta,\mathcal{F}_E(\beta)}(M_{\alpha,\beta}(x_\alpha)))) \to$$
$$\eta_{(\alpha\to\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\beta,\alpha\to\beta}([x_\alpha]M_{\alpha,\beta}(x_\alpha)))$$

$$\mathcal{C}(\lambda([x]\mathsf{op}(M^p, [y]M^c(x,y)))) \to \qquad \text{rules } \mathcal{C}^{\mathsf{op}}_{\alpha,\beta,E}$$
$$\mathsf{op}(M^p, [y]\mathcal{C}(\lambda([x]M^c(x,y))))$$
$$\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]\mathsf{op}_{\gamma,\delta\Rightarrow\mathcal{F}_E(\beta),\mathcal{F}_E(\beta)}(M^p_\gamma, [y_\delta]M^c_{\alpha,\delta,\mathcal{F}_E(\beta)}(x_\alpha, y_\delta)))) \to$$
$$\mathsf{op}_{\gamma,\delta\Rightarrow\mathcal{F}_E(\alpha\to\beta),\mathcal{F}_E(\alpha\to\beta)}(M^p_\gamma, [y_\delta]\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]M^c_{\alpha,\delta,\mathcal{F}_E(\beta)}(x_\alpha, y_\delta))))$$

Figure 5: The IDTS rewrite rules for $(\!|\,\lambda\,|\!)_\tau$, shown in parallel with the CRS rules for $(\!|\,\lambda\,|\!)$.

$$
\begin{array}{ll}
\mathrm{Term}(x) & = x \\
\mathrm{Term}(\lambda([x]M)) & = \lambda x.\ \mathrm{Term}(M) \\
\mathrm{Term}(\mathsf{ap}(M, N)) & = (\mathrm{Term}(M))\,(\mathrm{Term}(N)) \\
\mathrm{Term}(c) & = c \\
\mathrm{Term}(\eta(M)) & = \eta\,(\mathrm{Term}(M)) \\
\mathrm{Term}(\mathsf{op}(M^{\mathrm{p}}, [x]M^{\mathrm{c}})) & = \mathsf{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x.\ \mathrm{Term}(M^{\mathrm{c}})) \\
\mathrm{Term}(⫰(M)) & = ⫰\,(\mathrm{Term}(M)) \\
\mathrm{Term}(\llparenthesis\,\rrparenthesis_{\mathsf{op}_1,\ldots,\mathsf{op}_n}(M_1, \ldots, M_n, M_\eta, N)) & = \llparenthesis\, \mathsf{op}_1\!: \mathrm{Term}(M_1),\ \ \mathsf{op}_n\!: \mathrm{Term}(M_n),\ \ \eta\!: \mathrm{Term}(M_\eta)\,\rrparenthesis\,(\mathrm{Term}(N)) \\
\mathrm{Term}(\mathcal{C}(M)) & = \mathcal{C}\,(\mathrm{Term}(M))
\end{array}
$$

Definition 7.9. ***Types*** *is a function from* $\llparenthesis\,\lambda\,\rrparenthesis$ *terms to sets of* $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ *terms, defined by the equation below.*

$$\mathrm{Types}(M) = \{m \mid \mathrm{Term}(m) = M\}$$

Lemma 7.10. *Let* M *and* N *be* $\llparenthesis\,\lambda\,\rrparenthesis$ *terms. Then,*

$$M \to_{\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}} N \quad \Rightarrow \quad \forall m \in \mathrm{Types}(M).\ \exists n \in \mathrm{Types}(N).\ m \to n$$

In the above, upper-case letters stand for $\llparenthesis\,\lambda\,\rrparenthesis$ *terms, while lower-case letters stand for* $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ *terms.*

Proof. This property is essentially a stronger kind of subject reduction for $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$. In proofs of subject reduction, we examine every reduction rule and we show how a typing derivation of the redex can be transformed into a typing derivation of the contractum. We can think of $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ terms as $\llparenthesis\,\lambda\,\rrparenthesis$ typing derivations. The reduction rules in Figure 5 are the rules which tell us how to take a typing of the redex and transform it into a typing of the contractum.

In order to prove this property, we will need to check the following:

- The redexes and contracta in Figure 5 are well-formed (i.e. well-typed). For that reason, we have included the type of every variable, metavariable and function symbol as a subscript.
- Applying Term to the left-hand and right-hand sides of the $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ rules yields the left-hand and right-hand sides of all the $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$ rules (and therefore the left-hand and right-hand sides of $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ rules belong to the Types image of the left-hand and right-hand sides of the $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$ rules). Since in Figure 5, we have included the terms with their type annotations removed, we can see at a glance that the stripped rules align with the CRS formulation of $\llparenthesis\,\lambda\,\rrparenthesis$.
- Finally, we have to check whether the rewriting rules in Figure 5 actually apply to *all* the $m \in \mathrm{Types}(M)$. In other words, we need to check whether the type annotation scheme used for the left-hand sides is the most general and covers all possible typings of the left-hand side. This is the case because we have followed the typing rule constraints and given the most general type annotations.

Given a reduction in $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$ from M to N, we can find the untyped reduction rule used in Figure 5. We know that if $m \in \mathrm{Types}(M)$, then m then matches the left-hand side of the corresponding typed rule. We also know that the right-hand side of the typed rule belongs to $\mathrm{Types}(N)$ and therefore, the property holds. Furthermore, if we were to formalize the correspondence between $\llparenthesis\,\lambda\,\rrparenthesis$ typing derivations and $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ terms, we would get another proof of subject reduction for $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$. □

Lemma 7.11. *If the reduction relation of* $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ *is terminating, then so is the* $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$ *reduction relation on well-typed terms.*

Proof. Consider the contrapositive: if there exists an infinite $\llparenthesis\,\lambda\,\rrparenthesis_{-\eta}$ chain of well-typed $\llparenthesis\,\lambda\,\rrparenthesis$ terms, we can use Lemma 7.10 to translate it, link by link, to an infinite $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ chain. However, infinite $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ reduction chains do not exist since $\llparenthesis\,\lambda\,\rrparenthesis_\tau$ is terminating. □

7.3 Termination for IDTSs

So far, we have introduced an IDTS and have shown that if this IDTS is terminating, then so is $(\!|\lambda|\!)_{-\eta}$. We will now look at a general result for IDTSs that we will make use of.

Theorem 7.12. ***Strong normalization*** *[77] Let $\mathcal{I} = (\mathcal{A}, \mathcal{R})$ be a β-IDTS satisfying the assumptions (A). If all the rules of $\mathcal{R}$ satisfy the General Schema, then $\rightarrow_{\mathcal{I}}$ is terminating.*

The theorem was lifted verbatim[33] from [77] and parts of it deserve explaining:

- What is a β-IDTS?
- What are the assumptions (A)?
- What is the General Schema?

We will deal with these in order. A β-IDTS is an IDTS which, for every two types α and β, has a function symbol $@_{\alpha,\beta} \in F_{\alpha\Rightarrow\beta,\alpha,\beta}$ and a rule $@_{\alpha,\beta}([x_\alpha]M_{\alpha,\beta}(x_\alpha), N_\alpha) \rightarrow M_{\alpha,\beta}(N_\alpha)$. Furthermore, there must be no other rules whose left-hand side is headed by @. We can turn our IDTS from 7.2 into a β-IDTS by extending it with these function symbols and reduction rules.[34] Termination in a larger system will still imply termination in our system.

7.3.1 Checking Off the Assumptions

Next, we will deal with the assumptions (A).

Definition 7.13. *The* ***Assumptions (A)*** *are defined as the following four conditions:*

1. *every constructor is positive*
2. *no left-hand side of rule is headed by a constructor*
3. *both $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ are well-founded*
4. *$stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$*

For these to make sense to us, we will need to identify some more structure on top of our IDTS: the notion of a constructor and the $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ relations. We will need to designate for every base type γ a set $C_\gamma \subseteq \cup_{p\geq 0,\alpha_1,\ldots,\alpha_p\in T(\mathcal{B})} F_{\alpha_1,\ldots,\alpha_p,\gamma}$ (i.e. a set of function symbols with result type γ). We will call the elements of these sets *constructors* of γ. The base types of our IDTS consist of atomic types, function types and computation types. We will have no constructors for atomic types. On the other hand, every function type $\alpha \rightarrow \beta$ will have a constructor $\lambda_{\alpha,\beta}$ ($\in F_{\alpha\Rightarrow\beta,\alpha\rightarrow\beta}$) and every computation type $\mathcal{F}_E(\gamma)$ will have constructors $\eta_{\gamma,E}$ ($\in F_{\gamma,\mathcal{F}_E(\gamma)}$) and $\mathtt{op}_{\gamma,E}$ ($\in F_{\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}$) for every $\mathtt{op} : \alpha \rightarrowtail \beta \in E$. We can now check assumption (A.2). Since the only constructors in our IDTS are η, op and λ, we validate this assumption.[35] Our choice of constructors induces a binary relation on the base types.

Definition 7.14. *The base type α* ***depends on*** *the base type β if there is a constructor $c \in C_\alpha$ such that β occurs in the type of one of the arguments of c.*

We will use $\geq_{\mathcal{B}}$ to mean the reflexive-transitive closure of this relation. Furthermore, we will use $=_{\mathcal{B}}$ and $>_{\mathcal{B}}$ to mean the associated equivalence and strict ordering, respectively.

Observation 7.15. *If $\tau_1 \leq_{\mathcal{B}} \tau_2$, then τ_1 is a subterm of τ_2.*

Proof. We will prove this by induction on the structure of the base type τ_2. If τ_2 is an atomic type, then τ_2 has no constructors, so it does not depend on any other type. If we look at the reflexive-transitive closure of that, $\geq_{\mathcal{B}}$, then the only type α such that $\tau_2 \geq_{\mathcal{B}} \alpha$ is, by reflexivity, τ_2 itself, which is a subterm of τ_2.

If τ_2 is the computation type $\mathcal{F}_E(\gamma)$, then we will have several constructors. We have $\eta_{\gamma,E}$ with a single argument of type γ. We thus know that $\mathcal{F}_E(\gamma)$ depends on γ. For every $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, we have a

[33] In fact, the actual Theorem in [77] states that the system is *strongly normalizing*. However, by strongly normalizing they mean that every term is computable, i.e. that there is no infinite reduction chain.

[34] These β rules and application operators are different from the ones already in our IDTS. ap is defined for the $\alpha \rightarrow \beta$ function type from $(\!|\lambda|\!)$ whereas @ serves the $\alpha \Rightarrow \beta$ type of IDTS.

[35] This is why we have to prove termination with η reduction separately

constructor $\mathtt{op}_{\gamma,E}$ with arguments of types α and $\beta \Rightarrow \mathcal{F}_E(\gamma)$. This tells us that $\mathcal{F}_E(\gamma)$ also depends on α, β and $\mathcal{F}_E(\gamma)$. $\mathcal{F}_E(\gamma)$ does not have any more constructors, so those are all the types it depends on. The $\geq_\mathcal{B}$ relation, which is the subject of this observation, is the reflexive-transitive closure of the dependency relation between base types. This means that $\tau_2 \geq_\mathcal{B} \tau_1$ if either $\tau_2 = \tau_1$ or τ_2 depends on some $\tau_2' \neq \tau_2$ such that $\tau_2' \geq_\mathcal{B} \tau_1$.

- If $\tau_2 = \tau_1$, then trivially τ_1 is a subterm of τ_2 and we are done.
- If τ_2 depends on some $\tau_2' \neq \tau_2$, then τ_2' must be either γ or one of the α or β from E since $\tau_2 = \mathcal{F}_E(\gamma)$. In all these cases, we can apply the induction hypothesis for τ_2'. We know that $\tau_2' \geq_\mathcal{B} \tau_1$ and by the induction hypothesis, we now know that τ_1 is a subterm of τ_2'. Since τ_2' is a subterm of τ_2, we have that τ_1 is a subterm of τ_2.

□

Corollary 7.16. *If $\tau_1 =_\mathcal{B} \tau_2$, then $\tau_1 = \tau_2$.*

Corollary 7.17. *If $\tau_1 <_\mathcal{B} \tau_2$, then τ_1 is a proper subterm of τ_2.*

We can now check assumption (A.3). Since the proper subterm relation is well-founded (i.e. has no infinite descending chains) and $>_\mathcal{B}$ is a subset of the proper subterm relation, then $>_\mathcal{B}$ must be well-founded as well. We can also check assumption (A.1) once we explain what a strictly positive type is.

Definition 7.18. *A constructor $c \in C_\beta$ is **positive** if every base type $\alpha =_\mathcal{B} \beta$ occurs only at positive positions in the types of the arguments of c.*

Definition 7.19. *The base types occurring in **positive positions*** (Pos) *and the base types occurring in **negative positions*** (Neg) *within a type are defined by the following mutually recursive equations:*

$$\begin{aligned}
\mathrm{Pos}(\alpha \Rightarrow \beta) &= \mathrm{Neg}(\alpha) \cup \mathrm{Pos}(\beta) \\
\mathrm{Neg}(\alpha \Rightarrow \beta) &= \mathrm{Pos}(\alpha) \cup \mathrm{Neg}(\beta) \\
\mathrm{Pos}(\nu) &= \{\nu\} \quad \textit{with } \nu \textit{ an atomic type} \\
\mathrm{Neg}(\nu) &= \emptyset \quad \textit{with } \nu \textit{ an atomic type}
\end{aligned}$$

In our IDTS, $\alpha =_\mathcal{B} \beta$ is true only when $\alpha = \beta$. The only time a base type occurs in the type of one of its constructor's arguments is in the case of the $\mathtt{op}$ constructors. Given $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, $\mathtt{op}_{\gamma,E}$ is a constructor of $\mathcal{F}_E(\gamma)$; the type of its second argument is $\beta \Rightarrow \mathcal{F}_E(\gamma)$. This occurrence is positive and so we validate assumption (A.1). To validate the second half of (A.3), we will need to introduce the $>_\mathcal{F}$ relation. As $>_\mathcal{B}$ was induced by the structure of constructors, $>_\mathcal{F}$ will be induced by the structure of the rewriting rules $\mathcal{R}$ of our IDTS.

Definition 7.20. *A function symbol f **depends on** a function symbol g if there is a rule defining f (i.e. whose left-hand side is headed by f) and in the right-hand side of which g occurs. We will use $\geq_\mathcal{F}$ as the name for the reflexive-transitive closure of this relation. We will also write $=_\mathcal{F}$ and $>_\mathcal{F}$ for the associated equivalence and strict ordering, respectively.*

If we scan the rules of $(\!|\lambda|\!)$, we will see that the $(\!|\ |\!)$ symbols depend on $\mathtt{op}$ (for when there is no handler and the $\mathtt{op}$ is copied), $\mathtt{ap}$ (for applying the handler clauses to their arguments), λ (for the continuation) and on $(\!|\ |\!)$ (for recursion). The $\mathcal{C}$ symbols depend on $\mathtt{op}$ (when passing the λ through an $\mathtt{op}$), η (when switching the λ with the η), λ (for the argument) and $\mathcal{C}$ (for recursion). There is no other dependency in our IDTS. This means we can check off the second part of assumption (A.3) since $>_\mathcal{F}$ is well-founded (it contains only $(\!|\ |\!) >_\mathcal{F} \mathtt{op}$, $(\!|\ |\!) >_\mathcal{F} \mathtt{ap}$, $(\!|\ |\!) >_\mathcal{F} \lambda$, $\mathcal{C} >_\mathcal{F} \mathtt{op}$, $\mathcal{C} >_\mathcal{F} \eta$ and $\mathcal{C} >_\mathcal{F} \lambda$). Assumption (A.4) is trivial in our case since, within our IDTS, $f =_\mathcal{F} g$ only when $f = g$. This assumption comes into play only in the general theory of IDTSs when one exploits mutual recursion with functions of multiple arguments. The $stat_f$ values mentioned in the assumption (A.4) describe the way in which a function's arguments should be ordered to guarantee that recursive calls are always made to smaller arguments. In the case of mutual recursion, both functions must agree on the order according to which they will decrease their arguments. Since we do not deal with mutual recursion in $(\!|\lambda|\!)$, we will not go into any more detail into this.

7.3.2 General Schema

There is one last obstacle in our way towards proving termination of $(\!|\lambda|\!)_\tau$. We will need to verify that the rewrite rules that we have given in Figure 5 follow the General Schema.

Definition 7.21. *A rewrite rule $f(l_1, \ldots, l_n) \to r$ follows the **General Schema** if $r \in \mathcal{CC}_f(l_1, \ldots, l_n)$.*

$\mathcal{CC}_f(l_1, \ldots, l_n)$ refers to the so-called *computable closure* of the left-hand side $f(l_1, \ldots, l_n)$. The idea behind the computable closure is that the left-hand side of a rewrite rule can tell us what are all the possible right-hand sides that still lead to a correct proof of termination.[36] A formal definition of computable closure is given in [77, p. 8]. Informally, $r \in \mathcal{CC}_f(l_1, \ldots, l_n)$ if:

- Every metavariable used in r is accessible in one of $l_1, \ldots, l_n$.
- Recursive function calls (i.e. uses of function symbols $g =_{\mathcal{F}} f$) are made to arguments smaller than the arguments $l_1, \ldots, l_n$.

A metavariable is **accessible** in a term if it appears at the top of the term or under abstractions or constructors. If a metavariable occurs inside an argument of a function symbol which is not a constructor, then there are some technical constraints on whether it is accessible. However, in every rewrite rule of our IDTS, the arguments of the function symbol being defined contain only constructors as function symbols. Finally, we will need to show that the arguments being recursively passed to $(\!|\ |\!)$ and $\mathcal{C}$ are smaller than the original arguments and therefore the recursion is well-founded and terminating. However, the General Schema presented in [77] uses a notion of "smaller than" which is not sufficient to capture the decrease of our arguments. On the other hand, when we were defining a denotational semantics for $(\!|\lambda|\!)$, we gave a well-founded ordering showing that the successive arguments to these operations are in fact decreasing. We will therefore make use of a technique which will allows us to incorporate this semantic insight into the IDTS so that the General Schema will be able to recognize the decreasing nature of the arguments.

7.4 Higher-Order Semantic Labelling

We will make use of the higher-order semantic labelling technique presented by Makoto Hamana in [78]. The idea behind the semantic labelling technique is to label function symbols with the denotations of their arguments. Whereas before, a function symbol was rewritten to the same function symbol on a smaller argument, in the labelled IDTS, a labelled symbol will be rewritten to a different *smaller* symbol (i.e. one with a smaller label). The theory in [78] is expressed in terms of category theory. This results in a very elegant and concise formulation of the theorems and their proofs. In this article, we only care about the applications of the theory and so we will try to introduce the technique without presupposing the reader's familiarity with category theory.

7.4.1 Presheaves and Binding Algebras

We will nevertheless introduce a few terms from category theory. When dealing with binding and types, it is usually not so useful to consider a mixed set of terms or denotations of different types. It is much more pertinent to speak of families of terms having the same type in the same typing context, i.e. $T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}$. In this example, T is a family of sets, indexed first by type and second by context. We can therefore say that $T \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$, where $\mathcal{B}$ is the set of base types of our IDTS (i.e. the set of $(\!|\lambda|\!)$ types) and $\mathbb{F}\downarrow\mathcal{B}$ is the set of $(\!|\lambda|\!)$ typing contexts (functions from finite sets to $\mathcal{B}$). The category-theoretical presentation of abstract syntax and binding originating in [80] relies on a similar notion known as *presheaf*. Presheaf can be seen as a synonym for functor, usually going from some kind of "index category" to some other category. In the above example, $\mathcal{B}$, $\mathbb{F}\downarrow\mathcal{B}$ and **Set** can be seen as categories:

- $\mathcal{B}$ has base types as objects and no arrows besides the mandatory identities
- $\mathbb{F}\downarrow\mathcal{B}$ has typing contexts as objects and renamings of contexts (exchanges, weakenings, contractions) as arrows
- **Set** is the standard category with sets as objects and functions as arrows

[36] Theorem 7.12 is proven using Tait's method of computability predicates [79]. The term computable closure comes from the fact that the admissible right-hand sides are the metavariables of the left-hand side closed on operations that preserve computability.

$\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ is the category of functors from $\mathbb{F}\downarrow\mathcal{B}$ to $\mathbf{Set}$. The object component of such a functor maps contexts to sets (usually sets of objects having some type within the given context). The arrow component translates the renamings of contexts into renamings of variables in these objects. The functors in the category $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ map types to the objects of $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$; their arrow component is trivial since $\mathcal{B}$ has only trivial arrows. We will call the objects of $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ presheaves (sometimes, we will also call the objects in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ presheaves). In our presentation, we will care only about the object level, meaning that we will identify a presheaf with a family of sets. We will now consider some presheaves that will come into play:

- The key presheaf will be the presheaf T of $(\!|\lambda|\!)$ terms, $T_{\tau,\Gamma} = \{M \mid \Gamma \vdash M : \tau\}$. Every element of $T_{\tau,\Gamma}$ is a well-typed $(\!|\lambda|\!)$ term.
- Another useful presheaf is the presheaf V of variables where $V_{\tau,\Gamma} = \{x \mid x : \tau \in \Gamma\}$.
- Z is the presheaf of the IDTS metavariables from $\mathcal{Z}$, $Z_{\tau,(x_1:\alpha_1,\ldots,x_n:\alpha_n)} = \{M \mid M \in Z_{\alpha_1,\ldots,\alpha_n,\tau}\}$.
- $T_\Sigma V$ is the presheaf of IDTS terms with alphabet Σ.
- $M_\Sigma Z$ is the presheaf of IDTS metaterms with alphabet Σ and typed metavariables Z.

Now we will define some endofunctors on the category of presheaves. As before, we will ignore the arrow component and give only a mapping from one family to another.

- First, we introduce an endofunctor on the category of presheaves in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$. For every base type τ, we have a functor $\delta_\tau : \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}} \to \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$. For $A \in \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$, we define $(\delta_\tau A)(\Gamma) = A(\Gamma + \tau)$ where $\Gamma + \tau$ is the extension of context Γ by a variable of type τ.[37] The idea behind this operation is to model binders, i.e. the arrow type $\Rightarrow$ of IDTS. If the presheaf $A_\beta \in \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ models the type β, then the presheaf $\delta_\alpha A_\beta$ models the type $\alpha \Rightarrow \beta$.
- The alphabet of our IDTS, Σ, induces an endofunctor on the category $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ mapping presheaves A to presheaves ΣA.
$$(\Sigma A)_\gamma = \coprod_{f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_l}\Rightarrow\beta_l,\gamma}} \prod_{1\leq i\leq l} \delta_{\vec{\alpha_i}} A_{\beta_i}$$
In the above, we use the vector notation $\vec{\alpha} \Rightarrow \beta$ for $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$ and $\delta_{\vec{\alpha}}$ for $\delta_{\alpha_1} \circ \ldots \circ \delta_{\alpha_n}$. Note that the above definition assumes that $\vec{\alpha_i}$, β_i and γ are all base types. Since in our encoding of $(\!|\lambda|\!)$, we use a function constructor $\to$ on the level of base types, this is the case.

We are now ready to define the notion of a Σ-binding algebra.

Definition 7.22. *A Σ-**(binding) algebra** $\mathcal{A}$ is a pair of a presheaf A and a natural transformation*[38] *$\alpha : \Sigma A \to A$. The presheaf is the **carrier** and the natural transformation interprets the **operations**. Since ΣA is a coproduct over all the $f \in \mathcal{F}$, we can also see α as the copair $[f^{\mathcal{A}}]_{f\in\mathcal{F}}$, where $f^{\mathcal{A}}$ is the interpretation in algebra $\mathcal{A}$ of operation f.*

We will need to construct a $(V + \Sigma)$-algebra in order to proceed, where $(V + \Sigma)(A)$ is defined as $V + \Sigma(A)$. Our algebra will be a term algebra, the carrier will be the presheaf T. We will need to give an interpretation to variables and to every function symbol defined in the alphabet Σ. This interpretation must be given as an $\alpha : V + \Sigma T \to T$, meaning that the interpretation of every function symbol must be compositional. A value from $(\Sigma T)_{\gamma,\Gamma}$ is composed of some function symbol f of result type γ together with the interpretation of all of the symbol's arguments. If the i-th argument of the function symbol f has type β_i and binds variables $\vec{\alpha_i}$, then the interpretation of the argument in our term model will be a $(\!|\lambda|\!)$ term whose type in the context $\Gamma + \vec{\alpha_i}$ is β_i. The translation Term from $(\!|\lambda|\!)_\tau$ terms to $(\!|\lambda|\!)$ terms that we have given in 7.2 gives us the operations

[37]When we extend a context, we usually extend it with a pair of a variable name and a type, e.g. $\Gamma, x : \tau$. However, the theory of binding algebras uses Bruijn levels [81], where the names of variables in a context are always integers from 1 to some n. Extending a context $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ with a type τ then yields a context $x_1 : \alpha_1, \ldots, x_n : \alpha_n, x_{n+1} : \tau$.

[38]Natural transformation is the name for an arrow between two functors (presheaves). In our particular setting, naturality boils down to A_γ being a function of $(\Sigma A)_\gamma$ for every γ.

of the term model algebra. For example, the line defining the translation of the expression $\mathtt{op}(M^{\mathrm{p}}, [x]M^{\mathrm{c}})$ can be transformed into an interpretation for the function symbol $\mathtt{op}$ the following way:

$$\mathrm{Term}(\mathtt{op}(M^{\mathrm{p}}, [x]M^{\mathrm{c}})) = \mathtt{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x.\ \mathrm{Term}(M^{\mathrm{c}}))$$
$$\mathtt{op}_{\Gamma}(M^{\mathrm{p}}, M^{\mathrm{c}}) = \mathtt{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x_{n+1}.\ \mathrm{Term}(M^{\mathrm{c}}))$$

where $n = |\Gamma|$. Therefore, the Term translation function from 7.2 gives us a $(V + \Sigma)$-algebra $\mathcal{T}$ with carrier T.

7.4.2 Building a Quasi-Model

We will now deal with presheaves equipped with partial orders.

Definition 7.23. *A* ***presheaf equipped with a partial order*** *is a pair of a presheaf A and a family of partial orders $\geq_A$ such that $\geq_{A_{\tau,\Gamma}}$ is a partial order on the set $A_{\tau,\Gamma}$.*

Definition 7.24. *An arrow $f : A^1 \times \cdots \times A^n \to B$ in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ is* ***weakly monotonic*** *if for all Γ and $a_1, b_1 \in A^1_\Gamma$, $\ldots$, $a_n, b_n \in A^n_\Gamma$ with $a_k \geq_{A_{k,\Gamma}} b_k$ for some k and $a_j = b_j$ for all $j \neq k$, we have that $f(\Gamma)(a_1, \ldots, a_n) \geq_{B_\Gamma} f(\Gamma)(b_1, \ldots, b_n)$.*

Definition 7.25. *A* ***weakly monotonic*** $(V+\Sigma)$***-algebra*** *is a $(V+\Sigma)$-algebra $\mathcal{A}$ whose carrier A is equipped with a partial order $\geq_A$ such that every operation of $\mathcal{A}$ is weakly monotonic.*

We want to equip our $(V + \Sigma)$-algebra $\mathcal{T}$ with the $\twoheadrightarrow$ order. However, while we know that $\twoheadrightarrow$ is by definition a preorder, reflexive and transitive, we do not know whether it is antisymmetric and therefore whether it forms a partial order. Because of this, we will build a partial order *on top of* the $\twoheadrightarrow$ preorder.

Definition 7.26. *We say that terms M and N are* ***interreducible****, $M \rightleftarrows N$, if $M \twoheadrightarrow N$ and $N \twoheadrightarrow M$.*

The interreducibility relation defined above is an equivalence relation and we can use it to quotient sets of terms. We define the $T^{/\rightleftarrows}$ presheaf as the presheaf with $T^{/\rightleftarrows}_{\tau,\Gamma} = \{\{N \mid M \rightleftarrows N\} \mid \Gamma \vdash M : \tau\}$, i.e. $T^{/\rightleftarrows}$ is the quotient of the T presheaf w.r.t. the interreducibility relation. The elements of $T^{/\rightleftarrows}_{\tau,\Gamma}$ are interreducibility classes of terms having the type τ in the typing context Γ. We will use the metavariables $\mathcal{M}$ and $\mathcal{N}$ for these equivalence classes. The preorder $\twoheadrightarrow$ on $(\!|\lambda|\!)$ terms can be extended to interreducibility classes of $(\!|\lambda|\!)$ terms. Formally, we have $\mathcal{M} \twoheadrightarrow \mathcal{N}$ if there exists $M \in \mathcal{M}$ and $N \in \mathcal{N}$ such that $M \twoheadrightarrow N$. This preorder is antisymmetric and the $\twoheadrightarrow$ relation on interreducibility classes therefore forms a partial order. This means that $(T^{/\rightleftarrows}, \twoheadrightarrow)$ is a presheaf equipped with a partial order. We can verify that $\rightleftarrows$ is a congruence on the $(V + \Sigma)$-algebra $\mathcal{T}$. All of the operations in the algebra $\mathcal{T}$ construct new $(\!|\lambda|\!)$ terms with the operands as subterms of the constructed term. Let f be an operation of $\mathcal{T}$ and $M_1, \ldots, M_k$, and $N_1, \ldots, N_k$ be $(\!|\lambda|\!)$ terms such that $\forall i.\, M_i \rightleftarrows N_i$. Then we have $f(M_1, \ldots, M_k) \twoheadrightarrow f(N_1, \ldots, N_k)$ because $\forall i.\, M_i \twoheadrightarrow N_i$ and $f(N_1, \ldots, N_k) \twoheadrightarrow f(M_1, \ldots, M_k)$ because $\forall i.\, N_i \twoheadrightarrow M_i$. Therefore, we have $f(M_1, \ldots, M_k) \rightleftarrows f(N_1, \ldots, N_k)$. Since $\rightleftarrows$ is a congruence on $(V + \Sigma)$-algebra, we can quotient it and get a $(V + \Sigma)$-algebra $\mathcal{T}^{/\rightleftarrows}$ whose carrier is the $T^{/\rightleftarrows}$ presheaf. We now have a $(V + \Sigma)$-algebra, $\mathcal{T}^{/\rightleftarrows}$, whose carrier is equipped with a partial order, $\twoheadrightarrow$. Because the reduction relation $\twoheadrightarrow$ of $(\!|\lambda|\!)$ is closed on contexts, the operations of $\mathcal{T}^{/\rightleftarrows}$ are weakly monotonic: if we replace one of the arguments $\mathcal{M}_i$ in $f(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ with an $\mathcal{M}'_i$ such that $\mathcal{M}_i \twoheadrightarrow \mathcal{M}'_i$, then we will also have $f(\mathcal{M}_1, \ldots, \mathcal{M}_i, \ldots, \mathcal{M}_n) \twoheadrightarrow f(\mathcal{M}_1, \ldots, \mathcal{M}'_i, \ldots, \mathcal{M}_n)$. Therefore, we have a weakly monotonic $(V + \Sigma)$-algebra $\mathcal{T}^{/\rightleftarrows}$.

Definition 7.27. *For a given $(V+\Sigma)$-algebra $\mathcal{A}$, a* ***term-generated assignment*** *ϕ is an arrow in $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ from the presheaf Z to the presheaf A such that $\phi = {!} \circ \theta$, where:*

- *θ is an IDTS valuation,[39] i.e. an arrow from Z to $T_\Sigma V$.*
- *$!$ is the unique homomorphism from the initial $(V + \Sigma)$-algebra $T_\Sigma V$ to A.[40]*

[39] Same as the CRS valuations introduced in 6.1, but typed.

[40] Homomorphisms between Σ-algebras are defined in the same way as homomorphisms for first-order algebras. The term algebra $T_\Sigma V$ is called an initial algebra because we can find a (unique) homomorphism from $T_\Sigma V$ to any other algebra $\mathcal{A}$ that works by interpreting terms from $T_\Sigma V$ using the operations of $\mathcal{A}$.

To clarify the nomenclature: valuations replace metavariables with terms, assignments replace metavariables with interpretations in some algebra and term-generated assignments are assignments that can only assign an interpretation x if x can be computed as the interpretation of some term.

Definition 7.28. *A weakly monotonic $(V+\Sigma)$-algebra $(\mathcal{A}, \geq_A)$* ***satisfies an IDTS rewrite rule*** *$l \to r$, with l and r of type τ, if for all term-generated assignments ϕ of the free metavariables Z in l and r, we have:*

$$!\,\theta^*_{\tau,\Gamma}(l) \geq_{A_{\tau,\Gamma}} !\,\theta^*_{\tau,\Gamma}(r)$$

where $\phi = !\circ\theta$, θ^* is the extension of the valuation θ to meta-terms and Γ is the context regrouping all the free variables exposed by θ.

Definition 7.29. *A weakly monotonic $(V+\Sigma)$-algebra $(\mathcal{A}, \geq_A)$ is a quasi-model for the IDTS $(\Sigma, \mathcal{R})$ if $(\mathcal{A}, \geq_A)$ satisfies every rule in $\mathcal{R}$.*

Our weakly monotonic algebra $(\mathcal{T}^{/\rightleftarrows}, \twoheadrightarrow)$ is a quasi-model for the IDTS $(\!|\,\lambda\,|\!)_\tau$. The expressions $L = !\,\theta^*_{\tau,\Gamma}(l)$ and $R = !\,\theta^*_{\tau,\Gamma}(r)$ are instances of the left-hand and right-hand side, respectively, of the $(\!|\,\lambda\,|\!)$ reduction rule $l \to r$. Therefore, we always have $L \twoheadrightarrow R$.

7.4.3 Labelling Our System

We will now decide how to label the $(\!|\,|\!)$ and $\mathcal{C}$ symbols. The labels we will choose will be the $(\!|\,\lambda\,|\!)$ denotations introduced in 5.1. We will build up some orders on the denotations that will become crucial later.

Definition 7.30. *For each $(\!|\,\lambda\,|\!)$ type τ, we define a well-founded strict partial order $>_{[\![\tau]\!]}$ on the set of denotations $[\![\tau]\!]$ by induction on τ.*

- *τ is an atomic type Then $>_{[\![\tau]\!]}$ is the empty relation.*
- *$\tau = \alpha \to \beta$ $f >_{[\![\tau]\!]} g$ if and only if f and g are both functions (i.e. not $\bot$) and $\forall x \in [\![\alpha]\!].\ f(x) >_{[\![\beta]\!]} g(x)$. The new order is well-founded: any hypothetical descending chain $f_1 >_{[\![\tau]\!]} f_2 >_{[\![\tau]\!]} \ldots$ could be projected to a descending chain $f_1(x) >_{[\![\beta]\!]} f_2(x) >_{[\![\beta]\!]} \ldots$, which is well-founded by induction hypothesis.*
- *$\tau = \mathcal{F}_E(\gamma)$ Let $E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I}$. The order $>_{[\![\tau]\!]}$ is the smallest transitive relation satisfying the following:*
  - $\forall i \in I,\ \forall p \in [\![\alpha_i]\!],\ \forall c \in [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta_i]\!]},\ \forall x \in [\![\beta_i]\!].\ \mathtt{op}_i(p, c) >_{[\![\tau]\!]} c(x)$

  The proof of the well-foundedness of this relation was given in the definition of the interpretation of a handler (Definition 5.6). It relies on the fact that $[\![\mathcal{F}_E(\gamma)]\!]$ is defined as a union of an increasing sequence of sets where $c(x)$ always belongs to a set preceding the one in which $\mathtt{op}_i(p, c)$ appears for the first time.

As our labels, we will use denotations of (possibly open) $(\!|\,\lambda\,|\!)$ terms. These objects are functions from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ for some typing context Γ and type τ. We will need to compare denotations of two objects having the same type but not necessarily occurring in the same typing context. We introduce some notation to deal with context and valuation extensions.

Notation 7.31. *Let Γ and Δ be typing contexts. The typing context Γ, Δ, the* ***extension of*** *Γ* ***with*** *Δ, is defined by:*

$$(\Gamma, \Delta)(x) = \begin{cases} \Delta(x), & \text{if } \Delta(x) \text{ is defined} \\ \Gamma(x), & \text{otherwise} \end{cases}$$

Notation 7.32. *Let e and d be valuations[41] for the typing contexts Γ and Δ, respectively. The valuation $e + d$ for the context Γ, Δ, called the* ***extension of*** *e* ***with*** *d, is defined by:*

$$(e + d)(x) = \begin{cases} d(x), & \text{if } x \in \mathrm{dom}(d) \\ e(x), & \text{otherwise} \end{cases}$$

[41] Not IDTS valuations, but the valuations used by the denotational semantics in 5.1.

Notation 7.33. *We will use the term $D(\tau)$ for the set $\bigcup_\Gamma [\![\tau]\!]^{[\![\Gamma]\!]}$, the set of* ***possible denotations*** *of τ-typed $(\!|\lambda|\!)$ terms.*

Definition 7.34. *Let τ be a $(\!|\lambda|\!)$ type. The well-founded strict partial order $>_{D(\tau)}$ on the set $D(\tau)$ is defined by:*

- *$f >_{D(\tau)} g$ if and only if:*
  - $f : [\![\Gamma]\!] \to [\![\tau]\!]$
  - $g : [\![\Gamma, \Delta]\!] \to [\![\tau]\!]$
  - $\forall e \in [\![\Gamma]\!],\ \forall d \in [\![\Delta]\!].\ f(e) >_{[\![\tau]\!]} g(e + d)$

We will use the notation $\geq_{D(\tau)}$ for the reflexive closure of $>_{D(\tau)}$.

For every symbol f to label, we will now choose a non-empty well-founded poset $(S_f, \geq_{S_f})$, called the *semantic label set*. In our application of the technique, we will always choose the set of possible denotations of the argument that is being recursively decreased by the function. For the symbols that we do not care to label, we will assume that their semantic label set is the singleton set 1.

- For $(\!|\ |\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1 \to (\beta_1 \to \mathcal{F}_{E'}(\delta)),\ldots,\alpha_n \to (\beta_n \to \mathcal{F}_{E'}(\delta)),\gamma \to \mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$, we take as the semantic label set the poset $D(\mathcal{F}_E(\gamma))$ ordered by $\geq_{D(\mathcal{F}_E(\gamma))}$.
- For $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha \to \mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}$, we take as the semantic label set the poset $D(\alpha \to \mathcal{F}_E(\beta))$ ordered by $\geq_{D(\alpha\to\mathcal{F}_E(\beta))}$.

Having fixed the semantic label sets, we will now choose the *semantic label maps*. For each symbol $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\gamma}$ to be labelled, we define a weakly monotonic arrow $\langle\langle - \rangle\rangle^f$ in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$:

$$\langle\langle - \rangle\rangle^f : \delta_{\vec{\alpha_1}} T_{\beta_1}^{/\rightleftarrows} \times \cdots \times \delta_{\vec{\alpha_n}} T_{\beta_n}^{/\rightleftarrows} \longrightarrow K_{S_f}$$

where K_A is the constant presheaf $K_A(\Gamma) = A$. This semantic label map has access to the interpretations of all of the function symbol's arguments and needs to map them to an element of the semantic label set. In our model, the carrier containing the interpretations is the presheaf $T^{/\rightleftarrows}$ of interreducibility classes of $(\!|\lambda|\!)$ terms. However, the interreducibility relation $\rightleftarrows$ will be a congruence for all of the semantic label maps that we will define and so we will define them directly on terms instead of interreducibility classes. This means that for every $(\!|\ |\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1 \to (\beta_1 \to \mathcal{F}_{E'}(\delta)),\ldots,\alpha_n \to (\beta_n \to \mathcal{F}_{E'}(\delta)),\gamma \to \mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$, we need to give:

$$\langle\langle - \rangle\rangle^{(\!|\ |\!)} : T_{\alpha_1 \to (\beta_1 \to \mathcal{F}_{E'}(\delta))} \times \cdots \times T_{\alpha_n \to (\beta_n \to \mathcal{F}_{E'}(\delta))} \times T_{\gamma \to \mathcal{F}_{E'}(\delta)} \times T_{\mathcal{F}_E(\gamma)} \longrightarrow K_{D(\mathcal{F}_E(\gamma))}$$

We do so by projecting the last argument, which is a $(\!|\lambda|\!)$ term of type $\mathcal{F}_E(\gamma)$ in the context Γ, and finding its denotation using $[\![-]\!]$.

$$\langle\langle M_1, \cdots, M_n, M_\eta, N \rangle\rangle_\Gamma^{(\!|\ |\!)} = [\![N]\!]$$

We will do the same for the $\mathcal{C}$ symbols. For every $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}$, we give a:

$$\langle\langle - \rangle\rangle^{\mathcal{C}} : T_{\alpha \to \mathcal{F}_E(\beta)} \to K_{D(\alpha \to \mathcal{F}_E(\beta))}$$

by:

$$\langle\langle M \rangle\rangle_\Gamma^{\mathcal{C}} = [\![M]\!]$$

We can check that interreducibility is indeed a congruence for these semantic label maps: denotations are preserved under reduction (Property 5.8), and therefore all the terms in an interreducibility class have the same denotation. This means that we can extend these semantic label maps to $T^{/\rightleftarrows}$, the carrier of our quasi-model $(\mathcal{T}^{/\rightleftarrows}, \twoheadrightarrow)$. The semantic label maps must also be weakly monotonic. That is a condition that our maps satisfy: whenever we have $\mathcal{M} \twoheadrightarrow \mathcal{N}$, then by Property 5.8, the denotations $[\![M]\!]$ and $[\![N]\!]$ for $M \in \mathcal{M}$ and $N \in \mathcal{N}$ will be equal and therefore so will be the labels $\langle\langle \mathcal{M} \rangle\rangle$ and $\langle\langle \mathcal{N} \rangle\rangle$. Since, $\langle\langle \mathcal{M} \rangle\rangle = \langle\langle \mathcal{N} \rangle\rangle$, we have $\langle\langle \mathcal{M} \rangle\rangle \geq \langle\langle \mathcal{N} \rangle\rangle$. We have now built up enough structure to correctly label our IDTS with denotations. Let us start with the alphabet.

Definition 7.35. *Let $\Sigma = (\mathcal{B}, \mathcal{X}, \mathcal{F}, \mathcal{Z})$ be the alphabet of an IDTS $(\Sigma, \mathcal{R})$ and S_f the chosen semantic label sets. The* ***alphabet of the labelled IDTS*** *$(\overline{\Sigma}, \overline{\mathcal{R}})$ is the IDTS alphabet $\overline{\Sigma} = (\mathcal{B}, \mathcal{X}, \overline{\mathcal{F}}, \mathcal{Z})$ where:*

- *For every symbol $f \in \mathcal{F}_{\alpha_1,\ldots,\alpha_n,\beta}$ and for every label p in S_f, we will have $f^p \in \overline{\mathcal{F}}_{\alpha_1,\ldots,\alpha_n,\beta}$.*

To complete our new IDTS, we will also have to transform the rules, so we will need a way to label metaterms.

Definition 7.36. *Let $\phi : Z \to T^{/\rightleftarrows}$ be a term-generated assignment with $\phi = \,!\circ\theta$. The **labelling map** $\phi^{\mathrm{L}} : M_\Sigma Z \to M_{\overline{\Sigma}} Z$ is the arrow in $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ defined by:*

$$\begin{aligned}\phi^{\mathrm{L}}_{\tau,\Gamma}(x) &= x\\ \phi^{\mathrm{L}}_{\tau,\Gamma}(Z_{\alpha_1,\ldots,\alpha_n,\beta}(t_1,\ldots,t_n)) &= Z(\phi^{\mathrm{L}}_{\alpha_1,\Gamma}(t_1),\ldots,\phi^{\mathrm{L}}_{\alpha_n,\Gamma}(t_n))\\ \phi^{\mathrm{L}}_{\tau,\Gamma}(f([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n)) &= f^{\langle\langle !\,\theta^*(t_1),\ldots,!\,\theta^*(t_n)\rangle\rangle^f_\Gamma}([\vec{x_1}]\phi^{\mathrm{L}}_{\beta_i,(\Gamma,\vec{x_1}:\vec{\alpha_1})}(t_1),\ldots,[\vec{x_n}]\phi^{\mathrm{L}}_{\beta_n,(\Gamma,\vec{x_n}:\vec{\alpha_n})}(t_n))\end{aligned}$$

where $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\tau}$.

The labelling map traverses an IDTS metaterm and replaces unlabelled function symbols from $\mathcal{F}$ with labelled ones from $\overline{\mathcal{F}}$. Note that the term-generated assignment is not used to rewrite the metavariables: the assignment has values in the carrier presheaf of our $(V+\Sigma)$-algebra and it can therefore be something completely different than an IDTS term. The term-generated assignment $\phi = \,!\circ\theta$ is only used when labelling a function symbol. The IDTS valuation θ is used to replace the metavariables in all of the arguments with some specific terms and the resulting IDTS terms are then interpreted in our algebra $\mathcal{T}^{/\rightleftarrows}$ using ! (which turns them into interreducibility classes of $(\!|\lambda|\!)$ terms). These interpretations are then given as arguments to the semantic label map $\langle\langle-\rangle\rangle^f$, which chooses a label from the label set. Note also that there is no case for bare abstraction ($[x]t$). In the theory of higher-order semantic labelling presented in [78], the IDTS is assumed to not contain any bare abstractions: abstractions should always be arguments to function symbols. This is the case in our IDTS $(\!|\lambda|\!)_\tau$. Knowing how to label metaterms, we can now label the rules of an IDTS.

Definition 7.37. *Given an IDTS $(\Sigma,\mathcal{R})$, a $(V+\Sigma)$-algebra M and a choice of semantic label sets S_f and maps $\langle\langle-\rangle\rangle^f$, we define the **rules of the labelled IDTS** $(\overline{\Sigma},\overline{\mathcal{R}})$ with:*

- $\overline{\mathcal{R}} = \{\phi^{\mathrm{L}}_{\tau,\emptyset}(l) \to \phi^{\mathrm{L}}_{\tau,\emptyset} \mid l \to r : \tau \in \mathcal{R}, \textit{term-generated assignment } \phi : Z \to M\}$

The labelled IDTS will multiply the number of rules. For every possible IDTS valuation of the free metavariables of a rule, there will be a new rule in which the function symbols have been labelled using the interpretations of their arguments. As we have done in 7.2, we will have to show that termination of this new labelled system gives us termination of the unlabelled one. This is the object of the principal result in [78] (Theorem 3.7):

Theorem 7.38. ***Higher-order semantic labelling*** *Let M be a quasi-model for an IDTS $(\Sigma,\mathcal{R})$ and $(\overline{\Sigma},\overline{\mathcal{R}})$ the labelled IDTS with respect to M. Then $(\Sigma,\mathcal{R})$ is terminating if and only if $(\overline{\Sigma},\overline{\mathcal{R}}\cup\mathrm{Decr})$ is terminating.*

Definition 7.39. *Given a labelled IDTS alphabet $\overline{\Sigma}$ with semantic label sets S_f, the rules of the IDTS $(\overline{\Sigma},\mathrm{Decr})$ (called **decreasing rules**) consist of:*

$$f^p([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n) \longrightarrow f^q([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n)$$

where $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\gamma}$ and $p >_{S_f} q$.

The decreasing rules allow us to freely adjust the labels on function symbols to fit rewrite rules as long as we do not increase them.

7.4.4 Verifying the General Schema

Now we will retrace the steps we have carried out in 7.3, this time with our semantically labelled system $\overline{(\!|\lambda|\!)_\tau}$.

1. every constructor is positive

2. no left-hand side of rule is headed by a constructor

3. both $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ are well-founded

4. $stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$

First we have to check off the assumptions (A.1) through (A.4), repeated above. The constructors of $\overline{(\!|\lambda|\!)_\tau}$ are the same as the ones in $(\!|\lambda|\!)_\tau$ and so we validate assumption (A.2). It also means that the induced ordering $>_{\mathcal{B}}$ is the same as before and it is therefore still well-founded, so we have the first half of assumption (A.3). Since $>_{\mathcal{B}}$ is still the same, then so is $=_{\mathcal{B}}$, which is used in the definition of positive constructors (Definition 7.18). The constructors are therefore still positive as well and we get assumption (A.1). To verify the second half of assumption (A.3) and assumption (A.4), we will need to investigate the ordering on function symbols $>_{\mathcal{F}}$ and it is here that we will reap the benefits of our labelling. We need to give a well-founded partial order $\geq_{\mathcal{F}}$ on the function symbols such that whenever we have a rule $f(l_1, \ldots, l_n) \to r$, then $f \geq_{\mathcal{F}} g$ for all function symbols g occurring in r. We propose the following relation:[42]

- $(\!|\,|\!)^p_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} >_{\mathcal{F}} (\!|\,|\!)^q_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'}$ if $p >_{S_{(\!|\,|\!)}} q$
- $(\!|\,|\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n} >_{\mathcal{F}} \mathtt{op}_i$
- $(\!|\,|\!) >_{\mathcal{F}} \mathtt{ap}$
- $(\!|\,|\!) >_{\mathcal{F}} \lambda$
- $\mathcal{C}^p_{\alpha,\beta,E} >_{\mathcal{F}} \mathcal{C}^q_{\alpha,\beta,E}$ if $p >_{S_{\mathcal{C}}} q$
- $\mathcal{C} >_{\mathcal{F}} \mathtt{op}$
- $\mathcal{C} >_{\mathcal{F}} \eta$
- $\mathcal{C} >_{\mathcal{F}} \lambda$

Whenever we elide indices in the above (for labels, types or the operations in a handler), we assume that they are universally quantified over. This relation is indeed a well-founded strict partial order: $\mathtt{ap}$, $\mathtt{op}$, η and λ are minimal elements and decreasing chains of $(\!|\,|\!)$ or $\mathcal{C}$ symbols are all finite since the underlying semantic label set orderings $>_{S_f}$ are well-founded. This means that our $>_{\mathcal{F}}$ ordering validates the second half of assumption (A.3). We also let $\geq_{\mathcal{F}}$ be the reflexive closure of $>_{\mathcal{F}}$ and then we validate assumption (A.4) because $f =_{\mathcal{F}} g$ only if $f = g$. We have checked off all of the assumptions and so now we need to check whether the rewrite rules of our labelled IDTS $\overline{(\!|\lambda|\!)_\tau}$ all follow the General Schema. This boils down to checking whether the $\geq_{\mathcal{F}}$ order correctly describes the recursive behavior of our function definitions. Whenever we use a function symbol g in the right-hand side r of a rule $f(l_1, \ldots, l_n) \to r$, we need to show that $f \geq_{\mathcal{F}} g$. Furthermore, if $f =_{\mathcal{F}} g$, we need to show that the arguments passed to g are smaller than the arguments $l_1, \ldots, l_n$ passed to f. However, thanks to the semantic labelling, we will be able to show that for every rule $f(l_1, \ldots, l_n) \to r$, $f >_{\mathcal{F}} g$ for any function symbol g occurring in r. We first check the rules in Decr. These work out because $>_{\mathcal{F}}$ contains the label ordering for both labelled function symbols, $(\!|\,|\!)$ and $\mathcal{C}$ (i.e. $(\!|\,|\!)^p >_{\mathcal{F}} (\!|\,|\!)^q$ and $\mathcal{C}^p >_{\mathcal{F}} \mathcal{C}^q$ whenever $p > q$).

$$(\!|\,|\!)^p(M_i \ldots, M_\eta, N) \to (\!|\,|\!)^q(M_i \ldots, M_\eta, N)$$
$$\mathcal{C}^p(M) \to \mathcal{C}^q(M) \qquad \text{whenever } p > q$$

Then we check the rules that correspond to reductions in $(\!|\lambda|\!)$, looking at either the original formulation on Figure 4 or the CRS/IDTS versions on Figure 5. For most of the rules, it is just a matter of checking that only certain symbols appear in the right-hand sides of certain rules. However, in rules $(\!|\mathtt{op}|\!)$, $(\!|\mathtt{op}'|\!)$ and $C_{\mathtt{op}}$, we have the same (unlabelled) symbol on both the left-hand side and the right-hand side of the rule. In these cases, we will need to prove that the label on the right-hand side occurrence is strictly smaller than the label on the left-hand side occurrence. We will start with the rules $(\!|\mathtt{op}|\!)$ and $(\!|\mathtt{op}'|\!)$.

$$(\!|(\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta|\!)^p\,(\mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c})) \to M_j\, N_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta|\!)^q\, N_\mathrm{c}) \qquad \text{where } j \in I$$
$$(\!|(\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta|\!)^p\,(\mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c})) \to \mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i{:}\, M_i)_{i\in I},\ \eta{:}\, M_\eta|\!)^q\, N_\mathrm{c}) \qquad \text{where } j \notin I$$

[42] In Subsection 7.3, we said that the ordering $\geq_{\mathcal{F}}$ is induced by the form of the rewrite rules. Actually, we are free to define $\geq_{\mathcal{F}}$ ourselves as long as it validates the assumptions and the General Schema.

In both cases, the $(\!|\ |\!)$ on the left-hand side is applied to $\Gamma \vdash \mathsf{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c}) : \mathcal{F}_E(\gamma)$ whereas the $(\!|\ |\!)$ on the right-hand side is applied to $\Gamma, x : \beta_j \vdash N_\mathrm{c} : \mathcal{F}_E(\gamma)$ where $\mathsf{op}_j : \alpha_j \rightarrowtail \beta_j \in E$. The label p of the left $(\!|\ |\!)$ will be the denotation $[\![\mathsf{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})]\!]$ whereas the label q of the right $(\!|\ |\!)$ will be the denotation $[\![N_\mathrm{c}]\!]$. The ordering on these labels is the $>_{D(\mathcal{F}_E(\gamma))}$ ordering. For the first to be greater than the second, we will need to prove for all $e \in [\![\Gamma]\!]$ and all $d \in [\![\beta_j]\!]$ that $[\![\mathsf{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})]\!](e) >_{[\![\mathcal{F}_E(\gamma)]\!]} [\![N_\mathrm{c}]\!](e[x := d])$.

$$[\![\mathsf{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})]\!](e) = \mathsf{op}_j([\![N_\mathrm{p}]\!](e), \lambda X.\, ([\![N_\mathrm{c}]\!](e[x := X])))$$

From the definition of $>_{[\![\mathcal{F}_E(\gamma)]\!]}$ (Definition 7.30), we know that for all $d \in [\![\beta_j]\!]$, $\mathsf{op}_j([\![N_\mathrm{p}]\!](e), (\lambda X.\, [\![N_\mathrm{c}]\!](e[x := X]))) >_{[\![\mathcal{F}_E(\gamma)]\!]} [\![N_\mathrm{c}]\!](e[x := d])$ which is exactly what we wanted to show. Now we look at the C_{op} rule.

$$\mathcal{C}^p\, (\lambda x.\, \mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})) \to \mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, \mathcal{C}^q\, (\lambda x.\, M_\mathrm{c}))$$

On the left-hand side, $\mathcal{C}$ is applied to $\Gamma \vdash \lambda x.\, \mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c}) : \gamma \to \mathcal{F}_E(\delta)$, and on the right-hand side, it is applied to $\Gamma, y : \beta \vdash \lambda x.\, M_\mathrm{c} : \gamma \to \mathcal{F}_E(\delta)$ where $\mathsf{op} : \alpha \rightarrowtail \beta \in E$. The label p of the left $\mathcal{C}$ is the denotation $[\![\lambda x.\, \mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})]\!]$ while the label q of the right-hand side $\mathcal{C}$ is $[\![\lambda x.\, M_\mathrm{c}]\!]$. These labels are ordered by the $>_{D(\gamma \to \mathcal{F}_E(\delta))}$ ordering under which $p > q$ if for all $e \in [\![\Gamma]\!]$ and all $d \in [\![\beta]\!]$, we have $p(e) >_{[\![\gamma \to \mathcal{F}_E(\delta)]\!]} q(e[y := d])$. Then to show that $p(e) >_{[\![\gamma \to \mathcal{F}_E(\delta)]\!]} q(e[y := d])$, we will need to show that they are both functions and that for all $c \in [\![\gamma]\!]$, we have $p(e)(c) >_{\mathcal{F}_E(\delta)} q(e[y := d])(c)$.

$$\begin{aligned}
[\![\lambda x.\, \mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})]\!](e)(c) &= (\lambda X.\, ([\![\mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})]\!](e[x := X])))(c) \\
&= [\![\mathsf{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})]\!](e[x := c]) \\
&= \mathsf{op}([\![M_\mathrm{p}]\!](e[x := c]), \lambda Y.\, ([\![M_\mathrm{c}]\!](e[x := c, y := Y]))) \\
[\![\lambda x.\, M_\mathrm{c}]\!](e[y := d])(c) &= (\lambda X.\, ([\![M_\mathrm{c}]\!](e[y := d, x := X])))(c) \\
&= [\![M_\mathrm{c}]\!](e[y := d, x := c]) \\
&= [\![M_\mathrm{c}]\!](e[x := c, y := d])
\end{aligned}$$

We elaborate both of the expressions. The last step in rewriting $[\![\lambda x.\, M_\mathrm{c}]\!](e[y := d])(c)$ is due to $e[x := X, y := Y] = e[y := Y, x := X]$ for distinct variables x and y. From the definition of $>_{\mathcal{F}_E(\delta)}$ (Definition 7.30), we get that for all $d \in [\![\beta]\!]$, $\mathsf{op}([\![M_\mathrm{p}]\!](e[x := c]), \lambda Y.\, ([\![M_\mathrm{c}]\!](e[x := c, y := Y]))) > ([\![M_\mathrm{c}]\!])(e[x := c, y := d])$, which is exactly what we need. Having shown that the function symbols that head the left-hand sides of rules are strictly larger (in a well-founded poset) than the function symbols that occur in the right-hand sides gives us termination for the labelled IDTS $\overline{(\!|\lambda|\!)_\tau}$ via Theorem 7.12.

Theorem 7.40. ***(Termination of** $\overline{(\!|\lambda|\!)_\tau}$**)** *The reduction relation induced by the labelled IDTS* $\overline{(\!|\lambda|\!)_\tau}$ *is terminating.*[43]

Proof. Proof given above by the application of the General Schema presented in [77]. □

Corollary 7.41. ***(Termination of** $(\!|\lambda|\!)_\tau$**)** *The reduction relation induced by the IDTS* $(\!|\lambda|\!)_\tau$ *is terminating.*

Proof. By Theorem 7.40 and Theorem 7.38. □

Corollary 7.42. ***(Termination of** $(\!|\lambda|\!)_{-\eta}$**)**
The reduction relation of $(\!|\lambda|\!)$ *without* η*-reduction is terminating.*

Proof. By Corollary 7.41 and Lemma 7.11. □

7.5 Putting η *Back in* $(\!|\lambda|\!)$

We have shown termination for $(\!|\lambda|\!)_{-\eta}$. We know that the η-reduction on $(\!|\lambda|\!)$ is terminating: it decreases the number of λ-abstractions in the term by one in each step. We would now like to show that the combination of $(\!|\lambda|\!)_{-\eta}$ and η is terminating as well. Note that we could not have used the General Schema to prove that $(\!|\lambda|\!)$ with η is terminating. The General Schema does not admit η-reduction. The left-hand side of every rule needs to be headed by a function symbol which is not a contructor. If we tried declaring that λ is not a

[43] This result can be extended to $(\!|\lambda|\!)$ with sums and products. The pair construction $\langle -, - \rangle$ and the injections inl and inr will be the constructors for $\alpha \times \beta$ and $\alpha + \beta$, respectively, with $\alpha <_\mathcal{B} \alpha \times \beta$, $\beta <_\mathcal{B} \alpha \times \beta$, $\alpha <_\mathcal{B} \alpha + \beta$ and $\beta <_\mathcal{B} \alpha + \beta$. All of the rules defining case analysis and projections π_1 and π_2 satisfy the General Schema.

constructor, we would run into problems with the notion of accessibility. When accessing the metavariables of the left-hand side of a rule, we can access all of the arguments of a constructor but we can only access the arguments of a non-constructor symbol that has a basic type. The type of the argument of $\lambda_{\alpha,\beta}$ is $\alpha \Rightarrow \beta$ and so we could not access the arguments of λ in our rules (which would break the β rule, η rule and the $\mathcal{C}$ rules). Termination is generally not a modular property of higher-order rewriting systems [82]. Our plan will be to show that η-reduction does not interfere with the rewrite rules of $(\!|\lambda|\!)_{-\eta}$. Then we will be able to take any reduction chain in $(\!|\lambda|\!)$ and pull out from it a chain which only uses rules from $(\!|\lambda|\!)_{-\eta}$. Since this chain must be finite due to the termination of $(\!|\lambda|\!)_{-\eta}$, we will have a proof of finiteness for the reduction chain in $(\!|\lambda|\!)$.

Definition 7.43. *An* ***n-ary evaluation context*** C^n *is a* $(\!|\lambda|\!)$ *term in which* n *disjoint subterms have been replaced with the symbol* $[]$*. We write* $C^n[M]$ *for the term in which all of the occurrences of the symbol* $[]$ *have been replaced with* M*.*

Lemma 7.44. ***Exchanging*** η ***with*** $(\!|\lambda|\!)_{-\eta}$ *For every well-typed reduction chain* $s \to_\eta t \to_{(\!|\lambda|\!)_{-\eta}} u$*, there exists a well-typed reduction chain* $s \to^+_{(\!|\lambda|\!)_{-\eta}} t' \to^*_\eta u$*.*

Proof. We will consider all the possible relative positions of the contractum of the first reduction and the redex for the second reduction within t.

- Assume the two are disjoint, i.e. $s = C[M,N]$, $t = C[M',N]$ with $M \to_\eta M'$ and $u = C[M',N']$ with $N \to_{(\!|\lambda|\!)_{-\eta}} N'$. Then we can easily reorder the two reductions, producing the chain $C[M,N] \to_{(\!|\lambda|\!)_{-\eta}} C[M,N'] \to_\eta C[M',N']$.
- Assume that the contractum of the first reduction contains the redex for the second reduction, i.e. $s = C[M]$, $t = C[D[N]]$ with $M \to_\eta D[N]$ and $u = C[D[N']]$ with $N \to_{(\!|\lambda|\!)_{-\eta}} N'$. Since M is an η-redex, $M = \lambda x.\, D[N]\, x$. We can now build the chain $C[\lambda x.\, D[N]\, x] \to_{(\!|\lambda|\!)_{-\eta}} C[\lambda x.\, D[N']\, x] \to_\eta C[D[N']]$.
- Assume that the redex for the second reduction contains the contractum of the first reduction, i.e. $s = C[D[M]]$, $t = C[D[M']]$ with $M \to_\eta M'$ and $u = C[N']$ with $D[M'] \to_{(\!|\lambda|\!)_{-\eta}} N'$. Let R be the rule used in $D[M'] \to_{(\!|\lambda|\!)_{-\eta}} N$. We will now distinguish two scenarios:
  - The occurrence of M' in $D[M']$ is matched by a metavariable in the left-hand side of rule R.
    The R-redex N' of $D[M']$ will be a term $E^n[M',\dots,M']$ where E^n is an n-ary context for some n which depends on the rule R and the metavariable that was matched.[44] Furthermore, we can replace M' with any other term of the same type and the reduction will still go through, e.g. notably $D[M] \to_R E^n[M,\dots,M]$. We can now build our chain $C[D[M]] \to_{(\!|\lambda|\!)_{-\eta}} C[E^n[M,\dots,M]] \to^*_\eta C[E^n[M',\dots,M']] = C[N']$.
  - The occurrence of M' in $D[M']$ is not matched by a metavariable.
    M' is an η-contractum and must therefore have a function type. If we investigate the left-hand sides of all the rewriting rules in $(\!|\lambda|\!)_{-\eta}$ and search for terms that have a function type, we end up with:[45]
    * $D = []\, N$ and $R = \beta$
    * $D = \mathcal{C}\, []$ and $R = \mathcal{C}_{\mathsf{op}}$ or $R = \mathcal{C}_\eta$

    We note that in all of these rules, the symbol which replaces $[]$ must be a λ-abstraction. Therefore, if $D[M'] \to_R N$, then $M' = \lambda x.\, M''$. From $M \to_\eta M'$, we also know that $M = \lambda x.\, M'\, x$. We can replace this step by a β-reduction: $M = \lambda x.\, (\lambda x.\, M'')\, x \to_\beta \lambda x.\, M'' = M'$. The β rule is a part of $(\!|\lambda|\!)_{-\eta}$ and so we can now build the chain $C[D[M]] \to_{(\!|\lambda|\!)_{-\eta}} C[D[M']] \to_{(\!|\lambda|\!)_{-\eta}} C[N']$.

□

Lemma 7.45. ***Pulling a*** $(\!|\lambda|\!)_{-\eta}$ ***link from a*** $(\!|\lambda|\!)$ ***chain***
Let $t_1 \to t_2 \to \dots$ *be an infinite reduction chain in* $(\!|\lambda|\!)$*. Then there exists another infinite reduction chain* $u_1 \to u_2 \to \dots$ *in* $(\!|\lambda|\!)$ *and* $t_1 \to_{(\!|\lambda|\!)_{-\eta}} u_1$*.*

---

[44] Rules like $(\!|\eta|\!)$ can delete metavariables ($n = 0$ for the metavariable M_η), while others, like $(\!|\,|\!)$, can copy them ($n = 2$ for the variable M_j)

[45] The case of $D = []$ is not considered, because it is covered by the case where the contractum of the first reduction containts the redex of the second reduction.

Proof. The goal of this lemma is to show that we can find an $(\!|\lambda|\!)_{-\eta}$ link in every infinite $(\!|\lambda|\!)$ and move it to the beginning of the chain. An infinite chain in $(\!|\lambda|\!)$ must use a rule from $(\!|\lambda|\!)_{-\eta}$, otherwise it would be an η chain and those cannot be infinite since η is terminating.

Let $t_k \to t_{k+1}$ be the first link in the chain that uses a rule from $(\!|\lambda|\!)_{-\eta}$. We will prove this lemma by induction on k.

If $k = 1$, then we can use the chain $t_2 \to t_3 \to \ldots$ which also uses rules from $(\!|\lambda|\!)_{-\eta}$ infinitely often and which satisfies $t_1 \to_{(\!|\lambda|\!)_{-\eta}} t_2$.

If $k > 1$, then we replace the segment $t_{k-1} \to_\eta t_k \to_{(\!|\lambda|\!)_{-\eta}} t_{k+1}$ with the segment $t_{k-1} \to^+_{(\!|\lambda|\!)_{-\eta}} t_k \to^*_\eta t_{k+1}$ using Lemma 7.44. By induction hypothesis, the chain $t_1 \to \ldots \to t_{k-1} \to^+_{(\!|\lambda|\!)_{-\eta}} t_k \to^*_\eta t_{k+1} \to t_{k+2} \to \ldots$ gives us the necessary chain $u_1 \to u_2 \to \ldots$ with $t_1 \to_{(\!|\lambda|\!)_{-\eta}} u_1$. □

Theorem 7.46. ***Termination of*** $(\!|\lambda|\!)$

The reduction relation $\to$ on $(\!|\lambda|\!)$ terms given by the rules in Figure 4 is terminating.

Proof. We will prove this theorem by contradiction. Let $t_1 \to t_2 \to \ldots$ be an infinite reduction chain in $(\!|\lambda|\!)$. Since we have an infinite chain in $(\!|\lambda|\!)$, we can iterate Lemma 7.45 to get an infinite sequence of chains such that the first element of every chain reduces via $(\!|\lambda|\!)_{-\eta}$ to the first element of the next chain in the sequence. The first elements of these chains form an infinite reduction chain $(\!|\lambda|\!)_{-\eta}$, which is in contradiction with the termination of $(\!|\lambda|\!)_{-\eta}$. □

Theorem 7.47. ***Strong normalization of*** $(\!|\lambda|\!)$

There are no infinite reduction chains in $(\!|\lambda|\!)$ and all maximal reduction chains originating in a $(\!|\lambda|\!)$ term M terminate in the same term, the normal form of M.

Proof. The lack of infinite reduction chains is due to termination of $(\!|\lambda|\!)$ (Theorem 7.46) and the fact that all maximal reduction chains lead to the same term is entailed by confluence of $(\!|\lambda|\!)$ (Theorem 6.18). □

8 Comparison with Existing Work and Conclusion

8.1 Calculus

$(\!|\lambda|\!)$ can be compared to several existing calculi and implementations of effects and handlers:

- System F (i.e. the polymorphic λ-calculus or the second-order λ-calculus)

  $(\!|\lambda|\!)$ extends the simply-typed λ-calculus with computation types $\mathcal{F}_E(\alpha)$. Computations are algebraic expressions and as such can be expressed as inductive data types.[46]

  In $(\!|\lambda|\!)$, a computation of type $\mathcal{F}_E(\alpha)$ can also be given the type $\mathcal{F}_{E \uplus E'}(\alpha)$, where $E \uplus E'$ is an extension of E. However, in the direct encoding of $(\!|\lambda|\!)$ into System F, for every effect signature $E \uplus E'$ that we would like to ascribe to a computation, we would end up with a different term. On the other hand, in $(\!|\lambda|\!)$ we can keep using the same term. This lets us give a semantics to lexical items that does not have to change when new effects are introduced into the theory.

- *Eff*

  The *Eff* language [25] is an ML-like programming language with effects and handlers. Like in ML, effects can be freely used within any expression, without any term encoding (we say that the calculus is *direct-style*). For this to work correctly, the calculus has a fixed evaluation order, which, following ML, is call-by-value.

  We have used *Eff* in our first explorations of effects and handlers in natural language semantics [83], benefiting from the existing implementation. However, we have found that besides call-by-value, call-by-name evaluation is also common, notably on the boundaries of lexical items. Call-by-name can be simulated in call-by-value by passing around thunks (functions of type $1 \to \alpha$ for some α). However, in the presence of both call-by-name and call-by-value, we have opted for an indirect presentation of effects using monads which favors neither call-by-value nor call-by-name and that lets us manipulate the order of execution using $\gg\!=$.

[46] An inductive type is a recursive type with positive constructors. In 7.3, we have seen that a computation type $\mathcal{F}_E(\alpha)$ has positive constructors η and $\mathtt{op}$ for every $\mathtt{op} \in E$.

Finally, we note that *Eff* is a general-purpose programming language which includes general recursion (`let rec`) and therefore it is not terminating, contrary to $(\!|\lambda|\!)$.

- λ_{eff}

  The λ_{eff} calculus [23] is a call-by-push-value λ-calculus [58] with operations and handlers. Call-by-push-value is special in introducing two kinds of terms: computations and values. The intuition behind the two is that computations *do*, whereas values *are*. Two of the crucial things that computations do are to pop values from a stack (that is what abstractions do) and to push values to the stack (that is what applications do). Therefore, applications and abstractions are considered as computations. Furthermore, the function in an application term must be a computation term (which is expected to, among other things, pop a value from the stack), whereas the argument, which is the value to be pushed to the stack, must be a value term.

  This might make it look like that call-by-push-value is like call-by-value since all the arguments passed to functions are values. However, in true call-by-value, we can use complex expressions as arguments and we expect that the reduction system will evaluate the arguments down to values before passing them to the function. To do this in call-by-push-value, we have to implement this manually by evaluating the argument computation down to a value x and then passing the value x to the function in question (i.e. in λ_{eff} syntax $\textbf{let}\ x \leftarrow M\ \textbf{in}\ F\,x$). In $(\!|\lambda|\!)$, this amounts to the term $M \gg\!= F$, where $M : \mathcal{F}_E(\alpha)$ and $F : \alpha \to \mathcal{F}_E(\beta)$. To implement call-by-name, computations can be mapped to values by wrapping them in thunks, which are primitive constructs in call-by-push-value (in λ_{eff} syntax $F\,\{M\}$, where M is a computation and the thunk $\{M\}$ is a value). In $(\!|\lambda|\!)$, the corresponding term is $F\,M$, where $M : \mathcal{F}_E(\alpha)$ and $F : \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)$.

  λ_{eff} presents an intriguing alternative to $(\!|\lambda|\!)$. The call-by-push-value calculus is flexible enough to accommodate both call-by-name and call-by-value. λ-abstractions and operations are both treated as effects, which might make the definition of the $\mathcal{C}$ operator, which permutes λ with operations, more intuitive.[47] λ_{eff} also has a well-developed metatheory, developed in [23]: it is both confluent (due to its reduction relation being deterministic) and terminating (thanks to its effect type system).

  λ_{eff} served as an inspiration to the design of $(\!|\lambda|\!)$; notably, $(\!|\lambda|\!)$'s effect system is based on that of λ_{eff}. However, $(\!|\lambda|\!)$ diverges from λ_{eff} in that it is a proper extension of the simply-typed λ-calculus (STLC): every term, type, typing judgment or reduction in STLC is also a term, type, typing judgment or reduction in $(\!|\lambda|\!)$. For example, the STLC term $\lambda x.\, x$ is not a λ_{eff} term. Its closest counterparts in λ_{eff} would be either $\vdash_\emptyset \lambda x.\, \textbf{return}\ x : A \to F(A)$, where A is a value type, or $\vdash_E \lambda x.\, x! : U_E(C) \to C$, where C is a computation type (a function or an effectful computation). On the other hand, in $(\!|\lambda|\!)$, $\vdash \lambda x.\, x : \alpha \to \alpha$ is a valid term for any α, be it an atomic type such as o, a function type such as $\iota \to o$ or a computation type such as $\mathcal{F}_E(o)$.

  The fact that $(\!|\lambda|\!)$ is an extension of STLC motivates its use for two reasons. First, STLC is the lingua franca of formal semantics. $(\!|\lambda|\!)$ already introduces a lot of new notation and the use of effects in natural language semantics is not yet ubiquitous. By basing $(\!|\lambda|\!)$ on STLC, we narrow the gap between the common practice of formal semantics and our use of effects and monads, hopefully making the technique more approachable to researchers in the field. Second, the purpose of the calculus is to write down computations that produce logical representations. By having STLC as a subpart of $(\!|\lambda|\!)$, terms of Church's simple theory of types (i.e. formulas of higher-order logic) are already included in our calculus and we can reuse the same notions of λ-abstraction and variables. In λ_{eff}, we would either need to add constructors for logic formulas (i.e. having some logic as an object language over the terms of which the meta language λ_{eff} would calculate) or use call-by-push-value computations in our logical representations.

- Extensible Effects of Kiselyov et al [21] and other implementations of effect systems in pure functional programming languages (Haskell, Idris ...)

[47] The extra typing rule for the $\mathcal{C}$ construction in λ_{eff} would look like this:

$$\frac{\Gamma \vdash_E M : A \to C}{\Gamma \vdash_E \mathcal{C}\,M : F(U_\emptyset(A \to C))}$$

> Our adoption of a free monad and effect handlers was motivated by the paper of Kiselyov, Sabry and Swords on *extensible effects* [21]. The paper presented a Haskell library for encoding effectful computations, combining computations with diverse effects and interpreting them by composing a series of modular interpreters. The library used a free monad (in the style of [8]): a computation is either a pure value (η in $(\!|\lambda|\!)$) or a request to perform some kind of effect (an operation in $(\!|\lambda|\!)$). These requests are then handled by interpreters which behave similarly to effect handlers (the authors of [21] also relate handlers to the technique of "extensible denotational language specifications" published in 1994 by Cartwright and Felleisen [12]). The paper demonstrated that the approach is more flexible when it comes to combining interacting effects than the existing state-of-the-art technique of using monad transformers. A more refined version of the approach was published in [22] and similar implementations of effects and handlers exist also in other pure functional programming languages such as Idris [24].
>
> The extensible effects discipline provides the tools that we would like to use to build a modular semantics of natural language. However, we do not want our formal semantics to depend on the semantics of a large programming language such as Haskell[48] or Idris. We created $(\!|\lambda|\!)$ to reap the benefits of extensible effects without incurring the complexity of using a language like Haskell as our meta language. $(\!|\lambda|\!)$ extends STLC only with computation types, two constructors (η and operations), two destructors (handlers and $\downarrow$) and the $\mathcal{C}$ operator. Unlike Haskell, our extension of STLC preserves strong normalization.

We have introduced $(\!|\lambda|\!)$, a formal calculus that extends the simply-typed λ-calculus (STLC) with effects and handlers. The definition of $(\!|\lambda|\!)$ is given in Section 2. $(\!|\lambda|\!)$ introduces a new family of types into STLC, the computation types, and new terms, which are built out of computation constructors and destructors. We gave a type system to the calculus which extends that of STLC and a reduction semantics which combines the STLC β and η reductions with definitions of the new function symbols. In our exposition of the calculus, we have given two perspectives on the intended meaning of the terms: computations can be seen as programs that interact with a system through a selected set of "system calls" (operations) or they can be seen as algebraic expressions built upon an infinitary algebraic signature.

We then devoted most of the article to the development of the metatheory of $(\!|\lambda|\!)$. In Section 3, concepts which are primitive in some other languages (closed handlers and the $\gg=$ operator) were defined within $(\!|\lambda|\!)$ and their typing rules and reduction rules were derived from those of $(\!|\lambda|\!)$. In Section 5, we connected the calculus to the theory of monads by identifying a monad in the category in which we interpret $(\!|\lambda|\!)$ with our *denotational semantics*. In Section 4, we proved *subject reduction* of $(\!|\lambda|\!)$. This result gives a basic coherence between the type system of $(\!|\lambda|\!)$ and the reduction semantics, guaranteeing that types are preserved under reduction. This is complemented by a proof of *progress*, which states that terms which do not use any of the partial operators and which can no longer be reduced must have a very specific shape.

We followed this with another fundamental property: *strong normalization*. Its proof was split into two parts: *confluence* (proved in Section 6) and *termination* (proved in Section 7). The proofs of both confluence and termination proceed by similar strategies: prove the property for the calculus without η-reduction by applying a general result and then extend the property to the complete calculus. In the case of confluence, the general result is the confluence of orthogonal Combinatory Reduction Systems [74]. In the case of termination, we rely on two techniques: the termination of the reduction relation of Inductive Data Type Systems that validate the General Schema [77] and Higher-Order Semantic Labelling [78], which lets us use our denotational semantics to label the terms of our calculus so that it validates the General Schema.

Finally we briefly discuss the comparison of $(\!|\lambda|\!)$ with other existing frameworks.

Andrej Bauer made the analogy that effects and handlers are to delimited continuations what while loops or if-then-else statements are to gotos [84], continuations themselves having proven to be a useful tool in natural language semantics [85, 86, 3, 87, 88, 89].

References

[1] B. H. Partee, The development of formal semantics in linguistic theory.

[2] C. Shan, Monads for natural language semantics, arXiv preprint cs/0205026.
URL `http://arxiv.org/pdf/cs/0205026`

[48] The implementations of extensible effects in Haskell make use of a wealth of language extensions which are not even part of the Haskell standard.

[3] C. Shan, Linguistic side effects, in: In Proceedings of the Eighteenth Annual IEEE Symposium on Logic and Computer Science (LICS 2003) Workshop on Logic and Computational, University Press, 2005, pp. 132–163.

[4] C. Shan, Linguistic side effects, Ph.D. thesis, Harvard University Cambridge, Massachusetts (2005).

[5] S. Charlow, On the semantics of exceptional scope, Ph.D. thesis, New York University (2014).

[6] H. Kamp, U. Reyle, From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory, no. 42, Kluwer Academic Pub, 1993.

[7] J. Groenendijk, M. Stokhof, Dynamic predicate logic, Linguistics and philosophy 14 (1) (1991) 39–100.

[8] W. Swierstra, Data types à la carte, Journal of functional programming 18 (04) (2008) 423–436.

[9] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1995, pp. 333–343.

[10] N. Wu, Transformers, handlers in disguise, haskell eXchange 2015, recording at `https://skillsmatter.com/skillscasts/6733-transformers-handlers-in-disguise` (2015).
URL `https://skillsmatter.com/skillscasts/6733-transformers-handlers-in-disguise`

[11] D. Lewis, General semantics, Synthese 22 (1) (1970) 18–67.

[12] R. Cartwright, M. Felleisen, Extensible denotational language specifications, in: Theoretical Aspects of Computer Software, Springer, 1994, pp. 244–272.
URL `http://dx.doi.org/10.1007/3-540-57887-0_99`

[13] M. Hyland, G. Plotkin, J. Power, Combining effects: Sum and tensor, Theoretical Computer Science 357 (1) (2006) 70–99.

[14] G. Plotkin, M. Pretnar, Handlers of algebraic effects, in: Programming Languages and Systems, Springer, 2009, pp. 80–94.
URL `http://dx.doi.org/10.1007/978-3-642-00590-9_7`

[15] M. Pretnar, Logic and handling of algebraic effects, Ph.D. thesis, The University of Edinburgh (2010).
URL `http://matija.pretnar.info/pdf/the-logic-and-handling-of-algebraic-effects.pdf`

[16] G. D. Plotkin, M. Pretnar, Handling algebraic effects, arXiv preprint arXiv:1312.1399.

[17] G. Plotkin, J. Power, Computational effects and operations: An overview, Electronic Notes in Theoretical Computer Science 73 (2004) 149–163, proceedings of the Workshop on Domains VI. `doi:https://doi.org/10.1016/j.entcs.2004.08.008`.
URL `https://www.sciencedirect.com/science/article/pii/S1571066104050893`

[18] M. Pretnar, An introduction to algebraic effects and handlers. invited tutorial paper, Electronic notes in theoretical computer science 319 (2015) 19–35.

[19] D. Hillerström, S. Lindley, Shallow effect handlers, in: Asian Symposium on Programming Languages and Systems, Springer, 2018, pp. 415–435.

[20] D. Biernacki, M. Piróg, P. Polesiuk, F. Sieczkowski, Abstracting algebraic effects, Proceedings of the ACM on Programming Languages 3 (POPL) (2019) 1–28.

[21] O. Kiselyov, A. Sabry, C. Swords, Extensible effects: an alternative to monad transformers, in: Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, ACM, 2013, pp. 59–70.
URL `http://dx.doi.org/10.1145/2578854.2503791`

[22] O. Kiselyov, H. Ishii, Freer monads, more extensible effects, in: ACM SIGPLAN Notices, Vol. 50, ACM, 2015, pp. 94–105.

[23] O. Kammar, S. Lindley, N. Oury, Handlers in action, in: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, ACM, 2013, pp. 145–158.
URL `http://dx.doi.org/10.1145/2544174.2500590`

[24] E. Brady, Programming and reasoning with algebraic effects and dependent types, in: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, ACM, 2013, pp. 133–144.

[25] A. Bauer, M. Pretnar, Programming with algebraic effects and handlers, J. Log. Algebr. Meth. Program. 84 (1) (2015) 108–123. `doi:10.1016/j.jlamp.2014.02.001`.
URL `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`

[26] S. Lindley, C. McBride, C. McLaughlin, Do be do be do, draft available at `http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-july2016.pdf` (2016).
URL `http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-july2016.pdf`

[27] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, A. Madhavapeddy, Effective concurrency through algebraic effects, in: OCaml Workshop, 2015, p. 13.

[28] O. Kiselyov, K. Sivaramakrishnan, Eff directly in ocaml, in: ML Workshop 2016, 2018, p. 23–58. `doi:10.4204/EPTCS.285.2`.

[29] D. Hillerström, S. Lindley, K. Sivaramakrishnan, Compiling links effect handlers to the ocaml backend, in: ML Workshop, 2016, pp. 1–2.

[30] J. I. Brachthäuser, P. Schuster, K. Ostermann, Effect handlers for the masses, Proceedings of the ACM on Programming Languages 2 (OOPSLA) (2018) 1–27.

[31] D. Leijen, Implementing algebraic effects in c, in: Asian Symposium on Programming Languages and Systems, Springer, 2017, pp. 339–363.

[32] M. Hyland, J. Power, The category theoretic understanding of universal algebra: Lawvere theories and monads, Electronic Notes in Theoretical Computer Science 172 (2007) 437–458.
URL `http://dx.doi.org/10.1016/j.entcs.2007.02.019`

[33] F. W. Lawvere, Algebraic theories, algebraic categories, and algebraic functors, in: The theory of models, Elsevier, 2014, pp. 413–418.

[34] F. Bonchi, D. Pavlovic, P. Sobocinski, Functorial semantics for relational theories, arXiv preprint arXiv:1711.08699.

[35] J. C. Baez, C. Williams, Enriched lawvere theories for operational semantics, arXiv preprint arXiv:1905.05636.

[36] T. Letan, Y. Régis-Gianas, P. Chifflier, G. Hiet, Modular verification of programs with effects and effects handlers, Formal Aspects of Computing (2020) 1–24.

[37] P. E. de Vilhena, F. Pottier, A separation logic for effect handlers, Proceedings of the ACM on Programming Languages 5 (POPL) (2021) 1–28.

[38] O. Kiselyov, S.-C. MU, A. Sabry, Not by equations alone: Reasoning with extensible effects, Journal of Functional Programming 31.

[39] Z. LUKSIC, M. PRETNAR, Local algebraic effect theories, Journal of Functional Programming 30. `doi:10.1017/s0956796819000212`.
URL `http://dx.doi.org/10.1017/S0956796819000212`

[40] W. Swierstra, T. Baanen, A predicate transformer semantics for effects (functional pearl), Proceedings of the ACM on Programming Languages 3 (ICFP) (2019) 1–26.

[41] D. Biernacki, M. Piróg, P. Polesiuk, F. Sieczkowski, Handle with care: relational interpretation of algebraic effects and handlers, Proceedings of the ACM on Programming Languages 2 (POPL) (2017) 1–30.

[42] C. Matache, S. Staton, A sound and complete logic for algebraic effects, in: Foundations of Software Science and Computation Structures: 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 22, Springer, 2019, pp. 382–399.

[43] D. Ahman, Handling fibred algebraic effects, Proc. ACM Program. Lang. 2 (POPL). doi:10.1145/3158095.
URL https://doi.org/10.1145/3158095

[44] R. P. Pieters, E. Rivas, T. Schrijvers, Generalized monoidal effects and handlers, Journal of Functional Programming 30.

[45] D. Biernacki, M. Piróg, P. Polesiuk, F. Sieczkowski, Abstracting algebraic effects, Proc. ACM Program. Lang. 3 (POPL). doi:10.1145/3290319.
URL https://doi.org/10.1145/3290319

[46] D. Hillerström, S. Lindley, R. Atkey, Effect handlers via generalised continuations, Journal of Functional Programming 30. doi:10.1017/S0956796820000040.

[47] S. Kawahara, Y. Kameyama, One-shot algebraic effects as coroutines, in: A. Byrski, J. Hughes (Eds.), Trends in Functional Programming, Springer International Publishing, Cham, 2020, pp. 159–179.

[48] S. L. P. Jones, Haskell 98 language and libraries: the revised report, Cambridge University Press, 2003.

[49] M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth, A metalanguage for interactive proof in lcf, in: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1978, pp. 119–130.

[50] R. Milner, Implementation and applications of scott's logic for computable functions, ACM sigplan notices 7 (1) (1972) 1–6.

[51] D. Hillerström, S. Lindley, R. Atkey, K. Sivaramakrishnan, Continuation passing style for effect handlers.

[52] E. Moggi, Notions of computation and monads, Information and computation 93 (1) (1991) 55–92.
URL http://dx.doi.org/10.1016/0890-5401(91)90052-4

[53] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1992, pp. 1–14.

[54] M. Felleisen, M. Wand, D. Friedman, B. Duba, Abstract continuations: a mathematical semantics for handling full jumps, in: Proceedings of the 1988 ACM conference on LISP and functional programming, ACM, 1988, pp. 52–62.

[55] M. P. Jones, Functional programming with overloading and higher-order polymorphism, in: International School on Advanced Functional Programming, Springer, 1995, pp. 97–136.

[56] mtl: Monad classes, using functional dependencies, http://hackage.haskell.org/package/mtl, accessed: 2016-07-22.

[57] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: Functional Programming Languages and Computer Architecture, Springer, 1991, pp. 124–144.

[58] P. B. Levy, Call-by-push-value: A subsuming paradigm, in: Typed Lambda Calculi and Applications, Springer, 1999, pp. 228–243.

[59] P. de Groote, On logical relations and conservativity.

[60] S. Lindley, Algebraic effects and effect handlers for idioms and arrows, in: Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, ACM, 2014, pp. 47–58.

[61] C. McBride, R. Paterson, Applicative programming with effects, Journal of functional programming 18 (01) (2008) 1–13.

[62] B. Knaster, A. Tarski, Un théoreme sur les fonctions d'ensembles, Ann. Soc. Polon. Math 6 (133) (1928) 2013134.

[63] A. Tarski, et al., A lattice-theoretical fixpoint theorem and its applications, Pacific journal of Mathematics 5 (2) (1955) 285–309.

[64] S. C. Kleene, Introduction to metamathematics.

[65] E. Moggi, An abstract view of programming languages, University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[66] V. Balat, R. Di Cosmo, M. Fiore, Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums, ACM SIGPLAN Notices 39 (1) (2004) 64–76.

[67] C. Unger, Dynamic semantics as monadic computation, in: JSAI International Symposium on Artificial Intelligence, Springer, 2011, pp. 68–81.

[68] L. Champollion, Back to events: More on the logic of verbal modification, University of Pennsylvania Working Papers in Linguistics 21 (1) (2015) 7.

[69] G. Giorgolo, A. Asudeh, M. Butt, T. H. King, Multidimensional semantics with unidimensional glue logic, Proceedings of LFG11 (2011) 236–256.

[70] G. Giorgolo, A. Asudeh, Monads for conventional implicatures, in: Proceedings of sinn und bedeutung, Vol. 16, 2012, pp. 265–278.

[71] G. Giorgolo, A. Asudeh, Monads as a solution for generalized opacity, EACL 2014 (2014) 19.

[72] G. Giorgolo, A. Asudeh, Natural language semantics with enriched meanings (2015).

[73] C. Barker, D. Bumford, Monads for natural language (2015).

[74] J. W. Klop, V. Van Oostrom, F. Van Raamsdonk, Combinatory reduction systems: introduction and survey, Theoretical computer science 121 (1) (1993) 279–308.

[75] J. W. Klop, et al., Term rewriting systems, Handbook of logic in computer science 2 (1992) 1–116.

[76] F. Blanqui, J.-P. Jouannaud, M. Okada, Inductive-data-type systems, Theoretical Computer Science 272 (1) (2002) 41–68.

[77] F. Blanqui, Termination and confluence of higher-order rewrite systems, in: Rewriting Techniques and Applications, Springer, 2000, pp. 47–61.

[78] M. Hamana, Higher-order semantic labelling for inductive datatype systems, in: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM, 2007, pp. 97–108.

[79] W. W. Tait, Intensional interpretations of functionals of finite type i, The journal of symbolic logic 32 (02) (1967) 198–212.

[80] M. P. Fiore, G. Plotkin, D. Turi, Abstract syntax and variable binding.

[81] N. G. 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), Vol. 75, Elsevier, 1972, pp. 381–392.

[82] C. Appel, V. v. Oostrom, J. Grue Simonsen, Higher-order (non-) modularity, Logic Group Preprint Series 284 (2010) 1–26.

[83] J. Maršík, M. Amblard, Algebraic effects and handlers in natural language interpretation, in: Natural Language and Computer Science, 2014.

[84] A. Bauer, Lambda the ultimate — programming with algebraic effects and handlers, personal Communication (2012).
URL http://lambda-the-ultimate.org/node/4481#comment-69863

[85] P. de Groote, Type raising, continuations, and classical logic, in: Proceedings of the thirteenth Amsterdam Colloquium, 2001.

[86] C. Barker, Continuations and the nature of quantification, Natural language semantics 10 (3) (2002) 211–242.
URL http://dx.doi.org/10.1023/A:1022183511876

[87] P. de Groote, Towards a montagovian account of dynamics, in: Proceedings of SALT, Vol. 16, 2006.
URL http://elanguage.net/journals/salt/article/download/16.1/1791

[88] C. Barker, Continuations in natural language.

[89] C. Barker, C. Shan, Continuations and natural language, Vol. 53, Oxford University Press, USA, 2014.

[90] D. Sitaram, Handling control, in: ACM SIGPLAN Notices, Vol. 28, ACM, 1993, pp. 147–155.