Thèse de doctorat de l'université de Toulouse III - Paul Sabatier
Institut de Recherche en Informatique de Toulouse
École Doctorale Mathématiques, Informatique et Télécommunications de Toulouse.
Spécialité informatique
Dirigée en cotutelle / Ko-Betreuung
Dissertation an der Fakultät für Mathematik, Informatik und Statistik der
Ludwig-Maximilian-Universität München.

Decidability for Non-Standard Conversions in Typed Lambda-Calculi

Freiric Barral

Thèse pour obtenir le grade de docteur d'université.
Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften.

Supervisors:
Prof. Dr. Martin Hofmann
Prof. Dr. Sergeï Soloviev

First external reviewer:	Prof. Dr. Olivier Danvy
Second external reviewer:	Prof. Dr. Pierre Damphousse
First examiner:	Prof. Dr. Rolf Hennicker
Second examiner:	Prof. Dr. Hans Jürgen Ohlbach

Date of the oral examination: 6 June 2008

Munich, April 2008

Decidability for Non-Standard Conversions in Typed Lambda-Calculi

Thèse de doctorat de l'université de Toulouse III - Paul Sabatier
Institut de Recherche en Informatique de Toulouse
École Doctorale Mathématiques, Informatique et Télécommunications de Toulouse.
Spécialité informatique
Dirigée en cotutelle / Ko-Betreuung
Dissertation an der Fakultät für Mathematik, Informatik und Statistik der
Ludwig-Maximilian-Universität München.

Decidability for Non-Standard Conversions in Typed Lambda-Calculi

Freiric Barral

Thèse pour obtenir le grade de docteur d'université.
Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften.

Supervisors:
Prof. Dr. Martin Hofmann
Prof. Dr. Sergeï Soloviev

First external reviewer:	Prof. Dr. Olivier Danvy
Second external reviewer:	Prof. Dr. Pierre Damphousse
First examiner:	Prof. Dr. Rolf Hennicker
Second examiner:	Prof. Dr. Hans Jürgen Ohlbach

Date of the oral examination: 6 June 2008

Munich, April 2008

Contents

My first thanks go to my supervisors Martin Hofmann and Sergeï Soloviev.

The faculty of Martin Hofmann to quickly understand my ideas as well as his ability to always abstract the essential concepts, has been impressive.

Sergeï Soloviev, in spite of the distance, has always been behind me, providing me with an inestimable guidance in my intellectual formation, and continually supporting me in this sometimes morally exhausting enterprise.

Helmut Schwichtenberg, is the man without whom this works would never have been possible, he supplied me both with the necessary financial support and a stimulating research environment.

I gratefully acknowledge the funding of the *Graduierten Kolleg in der Informatik* which allowed me to concentrate on my research and to participate to numerous interesting conferences.

Thanks to Andreas Abel who has proof read significant parts of this thesis, since their foetusial versions. His careful readings is the cause of many technical corrections and improvements. I want to thank Olivier Danvy and Pierre Damphousse to have agreed to be my reviewers. The review of Olivier Danvy has led to many style improvements and bibliographical precisions. Of course, the remaining errors and imprecisions are mine and only mine.

My thanks go as well to David Chemouil who introduced me in the field of modular rewriting, and to Ralph Matthes to have presented me the Λ_J Calculus as the "simplest calculus with permutative conversions".

My colleagues and friends of the underground seminar, beside actively undergoing my talks have proved me that their is a life beyond the Ph.D: Diana Ratiu, Luca Chiarabini, Basil Karadais, Bogomil Kovachev, Markus Sauerman, Stefan Schimanski and Trifon Trifonov.

Special thanks to my wife Oksana.

Introduction

This thesis is concerned with the decidability of the theory of typed λ-calculi. Typed λ-calculi are formal systems to explore the notion of computation, and as such, are fundamental tools in computer science. However, the study of the theory of the λ-calculus does not reduce solely to computer science. Its deep connections with logic and its mathematical clarity, have settled the basis of a fertile interaction between computer science, logic and mathematics. Understanding the motivations of the theory of λ-calculi leads one naturally to a detour by the study of its connections with logic and mathematics. We begin with a brief history of the genesis of the λ-calculus, aimed at underlining its strong interaction with the development of logic.

Origins

The λ-calculus comes from the attempt in the 1930s by Alonzo Church first in [25], and in a corrected version in [25] to build an alternative formalisation of mathematics whose fundamental building blocks are functions. Soon, however his two students, Kleene and Rosser, prove his system in [59] to be inconsistent, suffering from the same paradox, discovered in 1902 by Russell, which affects the foundations of mathematics in terms of set theory.

Church, Kleene and Rosser then give up this fundamental attempt and turn to the study of computability with a tool at the basis of Church's system of logic; this tool is nowadays known as the untyped λ-calculus. This minimalist language with three symbols (two parentheses and a Greek letter) and a denumerable stock of variables will prove to have a high expressive power, it will even be proven to be Turing complete, which is to say, according to the Church-Turing thesis, that the untyped λ-calculus is the most expressive programming language, it can implement any computable functions.

Among the numerous results obtained by Church for the λ-calculus, probably the most famous is the undecidability of convertibility of terms [26] in the untyped λ-calculus, i.e.,, whether two programs compute the same function. The subject of this thesis tackles the opposite consideration, to find effectively computable procedures to decide equality between terms. We will have therefore to restrain this equality for certain classes of programs. The restriction used is a partial labelling of the

λ-terms following certain rules, the terms which can not be labelled are ruled out of the system, this technique is known as *typing*.

The theory of (ramified) types appears originally in the work of Russell and Whitehead [93] to cure the set-theoretical foundations crisis of mathematics of the beginning of the century. This theory is roughly a way to classify propositional sentences into a hierarchy to avoid self-references which are at the heart of many paradoxes (among them the paradox of Russell) undermining the naïve set theory of Cantor, considered then to be the only possible foundation of mathematics. This theory has inspired Church and Curry to develop typed versions of λ-calculi and combinatory logic (an analog to the λ-calculus where the explicit handling of abstraction and variable is abandoned and replaced by the use of primitive functional combinators).

The work of Church and Curry, one working on the λ-calculus, the other on combinatory logic, are tightly interleaved. They give in the same period their first formulations of a functional theory of types, Curry in [36], Church in [27]. For a nice exposition, the reader is invited to consult [83].

Logical Motivation

In 1934 in [44], Gentzen invented two formal systems to write deductions, natural deduction and sequent calculus, and proves his *Hauptsatz* for the sequent calculus. The *Hauptsatz* shows how it is possible to bring deductions into a normal form, which are deductions of a simpler structure. Although sequent calculus is important in its own right (in particular in proof theory), we will focus on natural deduction, which has a much clearer relationship with programming.

Prawitz, in 1965 in [77], rehabilitated natural deduction by proving an equivalent of the *Hauptsatz* for this system, the normalization of detour elimination (which corresponds to the β-reduction of the lambda-calculus).

After Curry had noted a correspondence between his combinatory logic and the logic as formulated by Hilbert, Howard, in 1969, extended this correspondence between natural deduction and λ-calculus. After having circulated informally for ten years and deeply influenced the community the manuscript of Howard [53] is eventually published. The correspondence pointed out by Howard which relates formulas and types, and proofs and terms is even an isomorphism, because the detour elimination of proof in natural deduction corresponds to reductions in the λ-calculus.

This isomorphism, known as the Curry-Howard isomorphism, provides a formalisation of the Brower-Heyting-Kolmogorov interpretation of proofs as constructions and marks the beginning of a fertile period of interplay between (mainly intuitionistic) logic and computer science.

It has given rise in turn to a bundle of logical systems based on the λ-calculus, interpreting the logical connectives as types, and proofs as terms; for example, the

type theory of Martin-Löf [67], Girard/Reynolds System F [47; 79], the Calculus of Constructions of Coquand and Huet [33; 32], or the extension of the latter with inductive types, the Calculus of Inductive Constructions [34].

All these systems, formulated using natural deduction, were proved strongly normalizing following a variation of the method of reducibility predicate originally introduced by Tait in [85] to prove a weak normalization for a calculus with combinators and recursion operators.

This change of perspective suggests to consider new conversions. Already in the seminal works of Prawitz, other conversions than the traditional η or β-reduction are present, the so called permutative reduction or π-conversion for sum type and existential type. They are needed in natural deduction in the presence of disjunction or existential quantifiers in order to obtain normal forms of deductions which satisfy the subformula property.

Categorical Motivation

The real shift from a set-theoretical foundation to a functional foundation of mathematics has been successfully achieved by category theory. In this light it is not surprising that category theory had an important impact on both logic and λ-calculus. The works of Lambek and Scott [60; 61; 62; 63] are particularly illuminating, where it is shown that there is an isomorphism between Cartesian Closed Categories and the theory of λ-calculi, and that categories are deductive systems with an equivalence relation on proofs. The categorical analysis of proofs and their equality has since then been pursued (in particular in the works [66] of Mann, and [72] of Mints).

The π-conversions of Prawitz, mentioned above, have a very natural meaning in category theory, they hold in fact automatically, if type formers are interpreted using universal properties, e.g., $\times$ is interpreted as categorical product, $\Longrightarrow$ as exponential object, etc. But other non-standard conversions hold as well. The categorical interpretation gives us in fact extensional models of λ-calculi, and hence all the possible conversions we might want to add.

We may not want to have such a powerful conversion relation (which corresponds to an extensional equality). For example in the case of inductive types, extensional equality is known to be undecidable. Still, the categorical perspective can be a guideline, if we want to strengthen the conversion relation, as it implies categorical properties to hold directly in our calculus. The question could now be what properties do we want to have in our calculus?

Programming Motivation and Applications

As already mentioned, the λ-calculus turns out to be a powerful tool to study computability. Its high level of abstraction as opposed to other model of computation such as Von Neumann models make it a convenient paradigmatic programming lan-

guage to concentrate on the program independently of the machine the program is to be implemented on. Its influence had spread over the whole computer science community (see [12]). In particular, functional languages are implementations of the λ-calculus with some primitives added.

Although far from the fundamental considerations of Russell, the advent of type systems in programming languages has analogously structured values and programs in order to forbid incoherences.

Experience shows that a great number of bugs can be detected automatically by a pass of type inference before actually running the program. However the gain of typing is not limited to bug elimination. It allows for example for optimization: if we know by typing that a program can be transformed in another one respecting the same specification and that this later is more efficient (although maybe less natural for human eyes), why not take the later? Two examples are partial typed directed evaluation and deforestation. In fact these two are examples of extended conversions as we will consider throughout this thesis.

Another nice application of strengthened conversion relation is the ability to decide provable isomorphisms between types. Two types are isomorphic if they carry the same information organized differently, i.e.,, if there exists function forth and back between this two types such that their composition is the identity. A characterisation of provable isomorphisms in a Cartesian closed category is first given by Soloviev in [84] and independently by Bruce, Di Cosmo and Longo in [22]. Rittri in [82] uses isomorphisms for retrieval methods in a library of programs (see [35] for a detailed exposition). Another application is the design of algorithms to be applied generically to a class of isomorphic datatypes, an example of an implementation is described in the paper of Atanassow and Jeuring [6].

Maybe the greatest outcome of the advent of types in programming and logic is the possibility to express in a same uniform framework, thanks to the Curry-Howard isomorphism, programs, properties thereof and checkable proofs. In such environment, on the one hand, proofs of properties of programs can be checked automatically and correctness of programs reaches a higher level of confidence, and on the other hand parts of proofs can be seen as programs which allow for more automation of formal proofs. For such a proof assistant to be convenient, it should obviously do as much as possible of these automations to assist the user in the obvious part and let him concentrate on the tedious part. A possible solution to achieve this goal is once again to strengthen the decidable conversion relation of such proof assistants.

State of the Art

The minimal theory of every λ-calculus is a computational equality for functions, the β-equality, two terms are equal if they reduce through β-reductions to a same third, where β-reduction is an evaluation step for a program applied to an argument, which

substitute this argument to the first formal parameter of the program. In particular, this equality identifies programs at different states of their evaluation.

Another conversion, traditionally not considered as a computational one, is η-conversion, this equality identifies terms with "dummy" abstractions: for each functional program p_1 of one argument, one can construct another program p_2 taking one argument and applying the program p_1 to this argument. In λ-calculus notations, one writes that p_2 is defined by $\lambda x.p_1 x$. So p_1 and p_2 are essentially the same program, and η-equality identifies them.

The β- together with η-equality lead to an extensional equality for the simply typed λ-calculus. Two terms of the simply typed λ-calculus are equal with respect to $\beta\eta$-equality if and only if their interpretations are the same in an extensional model. Roughly, programs which for the same argument return the same result are identified. Moreover, this equality is decidable for the simply typed λ-calculus.

This extensional equality is much harder to decide as soon as one wants to extend or generalise the system. There is no problem in adding a product type, or unit type (see for example the book *Proof and Types* of Girard, Lafont, Taylor [48]). Adding sum types is already much more problematic, and designing a deciding algorithm is a non trivial task, if obtained by reduction based normalization as by Ghani (see [45]), or reduction free normalization ; an algorithm is implicitly present in a constructive proof of Altenkirch et al. [4], and a type directed Normalization algorithm is obtained in [9] by Balat, using a call-by-value interpretation and control operators (where although some strong hints are given to justify the algorithm, the correctness is not rigorously proven). For inductive types, it is even known that extensional equality is undecidable ([76], or [51]). This problem of undecidability leads one to consider a conversion relation stronger than merely β, but still weaker than the extensional one.

Permutative reductions coming directly from logic in the tradition of Prawitz are such an example. A proof for strong normalization has been given initially by Prawitz in [78] for second-order natural deduction, but needed some supplementary details to be complete. This completion of the proof has be given only recently by Tatsuta and Mints in [87] and [86]. Other proofs of strong normalization for systems with permutative conversion have been studied by Joachimski and Matthes in [55] for a generalisation of the simply typed λ-calculus called Λ_J and the sum type and by David and Nour for classical natural deduction with disjunction in [75] and by Matthes for second-order natural deduction [70].

These permutative conversions, although being a real improvement on the mere β-(or $\beta\eta$-)conversions seem to be in some cases too weak (why taking permutative conversion for sum type when one can have extensional equality?) or maybe too strong (in case of inductive types).

Another possibility, inspired by category theory, is to design the conversion relation in order to obtain certain properties to be decidable, for example classes of isomorphisms, see Barthes and Pons [15], or Chemouil [23].

The λ-calculi with the reductions we propose to add are an instance of higher-order rewrite systems whose general theory is therefore relevant for us. Significant results for extending typed λ-calculus by higher order rewrite systems have been obtained by Blanqui using the General Schema [21], and Walukiewicz-Chrząszcz [92] using a higher order version of the recursive path ordering. Nonetheless, the conversions (oriented as rewrite rules) we will discuss for inductive type are not captured by these two frameworks.

The work of Matthes [68] extends system F to allow primitive recursion on monotone inductive types, by replacing the syntactical strict positivity condition by a monotonicity witness obligation packed into the inductive definition. Abel, Uustalu and Matthes then extend iteration to nested inductive types in [1]. Although not considered, adding conversions to such systems would be the natural continuation of these works.

Related Work

Disregarding the question of decidability, there is a too large field of study devoted to equality of functional programs to be able to give an exhaustive account. Maybe an influential paper is the one of Backus [8], where some equational laws are stated for an algebra of programs. It was further developed by Bird [20] and Meertens [71], who developed a computational approach, for program transformation, now known as the Bird-Meertens formalism. Malcolm developing the work of Hagino [49] generalises the result of Bird and Meertens for arbitrary datatypes by categorical considerations [64]. The recent developments and applications of transformational method abound ; Wadler, for example, presents certain proofs of equality as a result of the abstraction theorem of Reynolds [91].

Overview

An equality is decidable when one has an algorithm to decide it. This statement of the obvious leads us not only to study proofs of decidability but to study the deciding algorithms as well. We focus essentially on normalization algorithms. The principle is to first select a set of normal forms for which we have a decidable equality (syntactical equality for example) and to design an algorithm which map every terms to a convertible normal form.

The traditional way of normalizing a term system is to consider a conversion relation as generated by a rewrite system and to take as normal forms the irreducible terms: normalization then boils down to reduction.

Algorithms following these reduction strategy for normalization, also called *reduction-based* algorithms, are however not the only possibility, we will see in particular examples of a class of reduction-free algorithm, called *normalization by evaluation.*

The principle is to evaluate the interpretation of the term we want to normalize in a suitable model or language, and to define a function which gives back a normal form from a value.

After some preliminaries in the first chapter, the second chapter describes the normalization by evaluation algorithm for the simplest typed system, the simply typed calculus. While normalization by evaluation for the simply typed λ-calculus is well known, special care is taken here to handle variables properly but still in an intuitive way.

The third chapter is concerned with the study of the simplest system, admitting additional conversions, the Λ_J-calculus, which generalises the λ-calculus with a notion of double substitution.

Even for simple type systems where extensional equality is decidable such as the simply typed λ-calculus enriched by sum type, the algorithms to decide the equality are quite intricate. The fourth chapter presents a simple algorithm to decide conversion of a calculus with strong sums. The algorithm in these three chapters have all been implemented in the functional programming language Haskell.

The fifth chapter studies some possible conversion relations for inductive types.

Chapter 1

Preliminaries

1.1 Notations

We introduce here notations used throughout this thesis.

Notation 1 (binder)**.** *The use a dot to separate a binder of a variable from its scope means that the scope has to be extended as much as it is syntactically possible to the left. For example in $\lambda x.rs$, x is bound in r and s.*

Notation 2 (set-theoretic notations)**.** *We will write $A + B$ for the disjoint union of the sets A and B, the injection of an element $a \in A$ (resp. $b \in B$) into $A + B$ will be written $\iota_A\, a$ (resp. $\iota_B\, b$).*

The function space between the sets A and B will be written $A \to B$. To distinguish with the λ-abstraction of syntactic term which is just written with a single lambda λ, the abstraction at the semantic level (in the pseudo-programming language used to describe the algorithms) is written with a bold lambda $\mathbb{\lambda}$. The function application of a function f to an element a will sometimes be written as a syntactic application, i.e., we will use the notation fa, instead of the more traditional notation $f(a)$.

The cartesian product of the sets A and B is written $A \times B$.

Moreover we allow pattern matching in semantic abstraction. For example if we are abstracting over a cartesian product, we will write: $\mathbb{\lambda}(v1, v2).p$ instead of writing $\mathbb{\lambda}v.p$ and using projections in the body p of the algorithm. Similarly, if we are abstracting over the element of a one element set $\{\bot\}$, we will write $\mathbb{\lambda}\bot.p$.

Notation 3 (list of syntactic expressions)**.** *We will use the appropriate vector notation $\overrightarrow{e} = e_1, \ldots, e_n$ for finite lists of syntactic expressions. The empty list will be written ε. Within more complex type and term expressions it is to be unfolded as follows:*

types

$$(\rho, \overrightarrow{\rho}) \to \sigma ::= \rho \to \overrightarrow{\rho} \to \sigma$$
$$\varepsilon \to \sigma ::= \sigma$$

abstractions

$$\lambda(x, \overrightarrow{x}).r ::= \lambda x.\lambda \overrightarrow{x}.r$$
$$\lambda\varepsilon.r ::= r$$

applications

$$r(s, \overrightarrow{s}) ::= (rs)\overrightarrow{s}$$
$$r\varepsilon ::= r$$

composition

$$r^{\sigma\to\tau} \circ s^{\overrightarrow{\rho}\to\sigma} ::= \lambda\overrightarrow{x^{\rho}}.r(s\,\overrightarrow{x})$$

1.2 Monads

We will describe algorithms which depend in an essential way on the *order of evaluation*. The use of monads allows to both fix the order of evaluation and provide atomic operations to structure these algorithms in a pure functional setting. Another advantage is that they have a direct mathematical meaning, and avoid to rely on a particular programming language. We present here the monads we will use in a set-theoretic version.

There exists several equivalent definition of monads (see [65]), we give here a definition convenient to describe computations.

Definition 1.1 (Monad)**.** *A* monad $\mathcal{M}$ *is a triple* $\langle M, \nu, \star\rangle$ *where* M *maps every set* A *to a set* MA*,* ν *is a family of functions* $\nu^A : A \to MA$*, and* $\star$ *is a family of functions* $\star^{A,B} : MA \times (A \to MB) \to MB$ *(application of* $\star$ *is written in infix notation), such that:*

$$\nu^A(a) \star^{A,B} f = fa \qquad \text{(beta)}$$
$$m \star^{A,A} \nu^A = m \qquad \text{(eta)}$$
$$(m \star^{A,B} f) \star^{B,C} g = m \star^{A,C} (\lambda\!\!\lambda a.f(a) \star^{B,C} g) \qquad \text{(assoc)}$$

The family of functions ν *is called the* unit *of the monad, and the family of functions* $\star$ *is called the* multiplication *of the monad.*

Notation 4. *We will in the following omit to write the indices of the family* $\star$ *and* ν*.*

Example 1 (Identity Monad)**.** *The triple* $\langle I, \mathsf{id}, ; \rangle$ *where* I *maps every set* A *to itself* $IA = A$*,* id *is the identity function, and* $;$ *is defined by* $a; f = f(a)$ *is a monad and is called the* identity monad*.*

To express our algorithms we will need two kind of monads ; a state reader monad, and an exception monad. The state reader monad will be used to express computations sharing a same environment.

Definition 1.2 (State reader Monad)**.** *Given a set* E*, a* state reader monad *is a set operator* $\mathbf{St}_E(-)$ *defined by:*

$$\mathbf{St}_E(A) ::= E \to A$$

together with the family of functions $\nu : A \to \mathbf{St}_E(A)$ *and* $\star : \mathbf{St}_E(A) \times (A \to \mathbf{St}(B)) \to \mathbf{St}_E(B)$ *defined by:*

$$\begin{aligned} \nu(a)(e) &::= a \\ (m \star f)(e) &::= f(m(e))e \end{aligned}$$

One checks easily (one needs functional extensionality) that for a given set E, M is a monad in the sense of definition 1.1.

Remark 1. *The* state monad $\mathbf{ST}_E = E \to A \times E$ *is a more general monad. We do not need here the full generality of the state monad because our computations will not need to modify the environment (so that the computation does not need to return the environment* $e \in E$ *in addition to a value* $a \in A$*, but merely a value in* A*).*

The exception monad is used to express computations which may fail. In this case, special values are returned which detail the cause of the failure.

Definition 1.3 (Exception Monad)**.** *Given a set* E*, an* exception *monad is a set operator* $-_{\bot(E)}$ *defined by:*

$$A_{\bot(E)} ::= E + A$$

where $E + A$ *is the disjoint union of* E *and* A*. together with the family of functions* $\nu : A \to A_{\bot(E)}$ *and* $\star : A_{\bot(E)} \times (A \to B_{\bot(E)}) \to B_{\bot(E)}$ *defined by:*

$$\begin{aligned} \nu(a) &::= \iota_A\, a \\ \iota_A\, a \star f &::= f(a) \\ \iota_E\, e \star f &::= \iota_E\, e \end{aligned}$$

where ι_A *(resp.* ι_E*) is the canonical injection from* A *(resp.* E*) in* $E + A$*.*

In the definition above the value $e \in E$ stands for the exceptional value and the value $a \in A$ as the normal value (as a mnemonic, the right value is on the right side).

When the set E involved in the definition of the exception monad above is a one element set $\{*\}$, there is no information associated with the exception, we will just write $A_\perp$ for $A_{\perp(\{*\})}$. This monad is called the *partiality* monad.

The two basic operations associated to handle exception are throwing an exception and catching/handling an exception. For commodity we give the definition here:

Definition 1.4 (throw and catch)**.** *The functions* $\mathsf{throw} : E \to A_{\perp(E)}$ *and* $\mathsf{catch} : A_{\perp(E)} \to (E \to A_{\perp(E)}) \to A_{\perp(E)}$ *associated with an exception monad* $-_{\perp(E)}$ *are defined as follows:*

$$\mathsf{throw}\ e ::= \iota_E\, e$$

$$\mathsf{catch}\ mh ::= \begin{cases} m & \text{if } m = \iota_A\, a \\ he & \text{if } m = \iota_E\, e \end{cases}$$

for $m \in A_{\perp(E)}$*, and* $h \in E \to A_{\perp(E)}$*.*

Apart of the following chapter (2) where we use only one state reader monad, we will use mostly a combinations of the monads above. Although combining monads does not present any difficulty in simple cases, there is no canonical way to combine two monads into a new one (Some authors propose abstract methods [57; 56; 46], but these are ineluctably partial solutions). We will refrain to describe here the (simple) combinations we use, and postpone the definitions to the concerned chapters.

Chapter 2

Simply Typed λ-Calculus

In this chapter we handle the simply typed calculus and devise a normalization by evaluation algorithm for it.

This system is certainly the simplest, the most studied and the best-known of the typed λ-calculi. This will give us the opportunity to explain informally the algorithm. Then, the notion of freshness for variables is formalised via monads and from these considerations, a rigorous and original treatment follows in the last section.

2.1 System

Definition 2.1 (types)**.** *Given a ground type o, the set of types* Ty *of the simply typed λ-calculus is defined inductively by:*

$$\mathsf{Ty} \ni \rho, \sigma ::= o \mid \rho \to \sigma$$

We will work throughout this thesis with typed λ-calculi à la Church, i.e., we require a binder to assign a type to the variable it binds. While our algorithms extend without difficulty to λ-calculi à la Curry (i.e., binders do not assign type to variable), proofs are generally simpler in typed λ-calculi à la Church ; another advantage to work with λ-calculi à la Church is their direct relationship to natural deduction by the so-called Curry-Howard isomorphism (see the logical motivation section of the introduction, p.4).

Definition 2.2 (terms)**.** *Given a countable infinite set of term variables* Var*, the set of terms* Tm *of the simply typed λ-calculus is defined inductively by:*

$$\mathsf{Tm} \ni r, s ::= x \mid \lambda x^{\rho}.r \mid rs$$

where x is a variable ($x \in \mathsf{Var}$), and ρ is a type ($\rho \in \mathsf{Ty}$).

A term r, can contain a same subterm s at different places, in this cases we will speak about the different occurrences of the subterm s in r. An occurrence of a variable x is free in a term r if it does not appear in a subterm of r of the form $\lambda x^\rho.t$, otherwise the occurrence of x is said to be bound (it is bound by the binder λ of the smallest subterm $\lambda x^\rho.t$ containing x). A variable x is said to be free in a term r, if there is free occurrence of x in r.

Notation 5. *For readability, we will sometimes omit to write the type of the bound variable (i.e., we will write $\lambda x.t$ instead of $\lambda x^\rho.t$) if it is clear from the context or irrelevant.*

A term is closed if the set of its free variables is empty. The set of free variables of a term can be simply computed:

Definition 2.3 (Set of free variables of a term)**.** *We define the set of* free variables $\mathsf{FV}(r)$ *of a term r inductively as follows:*

$$\begin{aligned} \mathsf{FV}(x) &::= x \\ \mathsf{FV}(\lambda x.r) &::= \mathsf{FV}(r) \setminus \{x\} \\ \mathsf{FV}(rs) &::= \mathsf{FV}(r) \cup \mathsf{FV}(s) \end{aligned}$$

Typing contexts associate types to variables, they will be used in the definition of typing below to associate a type to each free variable of a term,

Definition 2.4 (typing context)**.** *We define a* typing context *as a finite set of pairs $(x : \rho)$ of a variable and a type such that for two pairs $(x : \rho)$ and $(y : \sigma)$ in a same typing context, $x \neq y$.*

The typing relation associates types to terms, under the precondition that free variables in the term have already been assigned some types.

Definition 2.5 (typing)**.** *The* typing relation *is a ternary relation between contexts, terms and types and is defined inductively by*

$$\frac{(x : \rho) \in \Gamma}{\Gamma \vdash x : \rho}\ (\textsc{Var}) \qquad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^\rho.r : \rho \to \sigma}\ (\to\text{I}) \qquad \frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}\ (\to\text{E})$$

Notation 6. *We will write Tm^ρ_Γ for the set of terms r typable with type ρ in context Gamma (i.e., $\Gamma \vdash r : \rho$) and Tm^ρ for the set of terms r of type ρ such that there exists a context Γ with $r \in \mathsf{Tm}^\rho_\Gamma$.*

Before giving the theory, i.e., the conversions of the simply typed λ-calculus, we need to define substitution. To avoid a phenomena known as capture of variable (a free variable in a term become bound when the term is substituted), this substitution has to be done modulo renaming of bound variables. In a first step we define a

contextual substitution which does not avoid capture of variables, this will allow us to define in a second step both renaming of bound variables (or α-conversion) and a correct notion of substitution.

Definition 2.6 (contextual substitution)**.** *Given a list of distinct variables* $\overrightarrow{x} = x_1, \ldots, x_n$ *and of terms* $\overrightarrow{r} = r_1, \ldots, r_n$ *of same length, the effect of the* contextual substitution $[\overrightarrow{x}/\overrightarrow{r}]$ *is defined by induction on* Tm *as follows:*

$$x[\overrightarrow{x}/\overrightarrow{r}] = \begin{cases} r_i & \text{if } x = x_i \wedge x_i \in \overrightarrow{x}, \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.r)[\overrightarrow{x}/\overrightarrow{r}] = \begin{cases} \lambda x.r[x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n/r_1,\ldots,r_{i-1},r_{i+1},\ldots,r_n] & \text{if } x = x_i \wedge x_i \in \overrightarrow{x} \\ \lambda x.r[\overrightarrow{x}/\overrightarrow{r}] & \text{otherwise} \end{cases}$$

$$(rs)[\overrightarrow{x}/\overrightarrow{r}] = (r[\overrightarrow{x}/\overrightarrow{r}])(s[\overrightarrow{x}/\overrightarrow{r}])$$

In the above definition, a capture of variable can happen in the second clause, if the bound variable x of the term $\lambda x.r$ occurs free in one of the term $r_j \in \overrightarrow{r}$ being actually substituted in $\lambda x.r$, (if $x \in \mathsf{FV}(r_j)$ and $x_j \in \mathsf{FV}(\lambda x.r)$ for $r_j \in \overrightarrow{r}$).

In the following we will define conversions on λ-terms expressed as axioms. These conversions are the smallest congruence relations containing these axioms. A congruence relation is an equivalence relation which is contextually closed (terms differing by convertible subterms will be convertible). We make this notion precise with the following definition.

Definition 2.7 (Contextual closure and equivalence)**.** *A relation* $=_R$ *is contextually closed if the following rules hold*

Structural rules:

$$\frac{r =_R s}{rt =_R st}\ (R\text{-AppL}) \qquad \frac{r =_R s}{tr =_R ts}\ (R\text{-AppR}) \qquad \frac{r =_R s}{\lambda x.r =_R \lambda x.s}\ (R\text{-}\xi)$$

The α-conversion expresses that the choice of the precise names of formal parameters of a function has no influence on the actual meaning or behaviour of this function.

Definition 2.8 (α-conversion)**.** *The axiom of* α*-conversion is given by*

$$\lambda x.r =_\alpha \lambda y.r[x/y] \qquad y \notin \mathsf{FV}(r) \qquad (\alpha)$$

We can now define a capture avoiding substitution.

Definition 2.9 (correct substitution)**.** *Given a list of distinct variables* $\overrightarrow{x} = x_1, \ldots, x_n$ *and of terms* $\overrightarrow{s} = s_1, \ldots, s_n$ *of same length, the effect of the* correct substitution $\{\overrightarrow{x}/\overrightarrow{s}\}$ *is defined as follows:*

$$r\{\overrightarrow{x}/\overrightarrow{s}\} = r'[\overrightarrow{x}/\overrightarrow{s}]$$

where $r =_\alpha r'$ *and if a variable* y *occurs free in* s_i*, then no occurrence of* x_i *appears within a subterm of* r' *of the form* $\lambda y.t$*.*

The correct substitution is thus a well-defined function only if λ-terms are considered up to α-conversion.

Remark 2. *A well established practice in the literature is to just ignore the difference between terms and terms modulo α-conversion and not introduce new notations to distinguish these two classes. It is safe in general, because most notions, properties or definitions are equivalently defined on a particular term or on a class of terms modulo α-conversion (as typing), or make only sense modulo α-conversion (as substitution). We will follow this practice here because it improves readability and the main concern of this work is the decidability of conversion and not the renaming of bound variables. However, we will study algorithms which produce genuine syntactic terms, not class of terms, and will underline abuses of notations where the difference is sensitive.*

The theory $=_{\beta\eta}$ of the simply typed λ-calculus Λ is defined as the union of the conversion $=_\beta$ and $=_\eta$ below. These conversions are understood between terms, typable with the same type in a same context.

Notation 7. *We will write* $\Gamma \vdash r =_R s : \rho$*, if the terms* r *and* s *are* R*-convertible and typable in the context* Γ *with type* ρ *and just* $r =_R s$ *if the context* Γ *and the type* ρ *are already clear from the context or if their mentions is irrelevant.*

The computational part of the conversion is given by the β-conversion.

Definition 2.10 (β-conversion)**.** *The axiom of β-conversion is given by*

$$(\lambda x.r)s =_\beta r\{^x/_s\} \qquad (\beta)$$

The axiom of η-conversion is the axiom of extensionality for the simply typed λ-calculus.

Definition 2.11 (η-conversion)**.** *The axiom of η-conversion is given by*

$$r =_\eta \lambda x.rx \qquad (x \notin \mathsf{FV}(r)) \qquad (\eta)$$

Because the definition of β-conversion uses substitution, the β and $\beta\eta$ conversion are only defined on classes of terms convertible by α-conversion. These conversions are extended to terms by stating that terms in convertible classes are convertible.

2.2 Normalization by Evaluation

2.2.1 Informal Description

Given a term language Tm, i.e., a term algebra generated from variables and (possibly binding) symbols, with an equality or conversion relation $=_E$ between terms, generated by contextual closure from a set of equations E, normal forms are considered in an abstract way as representatives of each equivalence class modulo $=_E$ with certain desirable property.

Because here we are interested in the decidability of the conversion relation, this property will be the decidability of conversion between terms considered as normal forms.

Focusing on a normalization function rather than on normal forms, an abstract definition of normalization function could be formulated as follows

Definition 2.12 (Normalization Function)**.** *Given a decidable equality* $\equiv \subseteq =_E$ *a* normalization function *is a function* $\mathsf{nf} : \mathsf{Tm} \to \mathsf{Tm}$ *with the following property:*

$$r =_E \mathsf{nf}(r),$$
$$r =_E s \Leftrightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s).$$

If the function nf terminates then it provides a decision algorithm of the conversion relation $=_E$, because $\equiv$ is decidable.

Remark that $\equiv \subseteq =_E$ trivially implies $\mathsf{nf}(r) \equiv \mathsf{nf}(s) \Rightarrow r =_E s$. This allows us to use a modified formulation of this definition, provided by the following (easy) lemma:

Lemma 1. *Given a decidable equality* $\equiv \subseteq =_E$*, a function* $\mathsf{nf} : \mathsf{Tm} \to \mathsf{Tm}$ *is a normalization function if and only if:*

$$r =_E \mathsf{nf}(r), \tag{i}$$
$$r =_E s \Rightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s). \tag{ii}$$

The essential idea of Normalization by Evaluation is to define a semantics of our language Tm containing enough information to be able to extract a term $\mathsf{nf}(r)$ from the interpretation of a term r, such that nf is a normalization function.

For this, one needs a function $\downarrow$, called *reify*, from the semantics to the term language.

For the $\downarrow$ function to be able to produce terms, the semantics should already contain some piece of syntax; this is called a residualizing semantics.

In this introductory section, we expose informally the Normalization by Evaluation algorithm for the simply typed λ-calculus with $\beta\eta$-equality. It has been first designed by Ulrich Berger and Helmut Schwichtenberg [19].

The semantics is given by the standard set-theoretic interpretation with the ground type interpreted as the set of terms of ground type.

Definition 2.13 (Type Interpretation)**.** *We define the* interpretation $[\![\rho]\!]$ of a type ρ, *by induction on* $\rho \in \mathsf{Ty}$*:*

$$\begin{aligned} [\![o]\!] &::= \mathsf{Tm}^o \\ [\![\rho \to \sigma]\!] &::= [\![\rho]\!] \to [\![\sigma]\!] \end{aligned}$$

where Tm^o *is the set of all (closed and open) terms of ground type* o*, and* $[\![\rho]\!] \to [\![\sigma]\!]$ *is the full function space between the set* $[\![\rho]\!]$ *and* $[\![\sigma]\!]$*.*

The interpretation of types $[\![\mathsf{Ty}]\!]$ of the simply typed lambda calculus Λ *is then:*

$$[\![\mathsf{Ty}]\!] = \bigcup_{\rho \in \mathsf{Ty}} [\![\rho]\!]$$

To define the interpretation of terms we need first to define the auxiliary notion of valuation.

Definition 2.14 (Valuation)**.** *Given a context* Γ*, we define a* valuation *on* Γ *(*$\eta \vDash \Gamma$*) to be a partial function* $\eta : \mathsf{Var} \to [\![\mathsf{Ty}]\!]_\perp$ *such that for* $x : \rho \in \Gamma$*, we have* $\eta(x)$ *is defined and* $\eta(x) \in [\![\rho]\!]$*. Given a context* Γ*, a valuation* η *on* Γ*, a variable* $y \notin \Gamma$ *and an element* $a \in [\![\sigma]\!]$*, we define a valuation* $(\eta, y \mapsto a)$ *on* $\Gamma \cup \{(y : \sigma)\}$*, called the* extension *of* η *by* $y \mapsto a$ *by,*

$$(\eta, y \mapsto a)(x) ::= \begin{cases} a & \text{if } x = y, \\ \eta(x) & \text{otherwise.} \end{cases}$$

Remark 3. *Valuation functions are partial functions represented as total functions from the set of variables* Var *into the set* $[\![\mathsf{Ty}]\!]_\perp = [\![\mathsf{Ty}]\!] + \{\star\}$*, i.e., the interpretation of types extended with an element* $\star$ *playing the rôle of an undefined value. In the following, whenever we will use a valuation applied to some variable, this variable will belong to the domain of definition of the valuation and the result will therefore be defined. And although, strictly speaking we should do a case distinction on the result to know wheither it is defined (an alternative would be to use an exception monad as presented in definition 1.3), we will consider it to be an element of* $[\![\mathsf{Ty}]\!]$*.*

After having defined the interpretation of a type as a set, we now define the interpretation of a term simply as an element of the interpretation of its type:

Definition 2.15 (Term Interpretation)**.** *We define the* interpretation $[\![r]\!]_\eta$ of a term r*, whenever there is a context* Γ*, such that* $\Gamma \vdash r : \rho$ *and* η *is a valuation on* Γ*, to be an element of* $[\![\rho]\!]$*, by the following inductive definition:*

$$\begin{aligned} [\![x]\!]_\eta &::= \eta(x) \\ [\![\lambda x^\rho . r]\!]_\eta(v) &::= [\![r]\!]_{\eta, x \mapsto v} \\ [\![rs]\!]_\eta &::= [\![r]\!]_\eta([\![s]\!]_\eta) \end{aligned}$$

The *reify* function $\downarrow$ from the interpretation of the types $[\![\mathsf{Ty}]\!]$ to the term language Tm will be defined by simultaneous induction on the types as a function $\downarrow_\rho$ from the interpretation $[\![\rho]\!]$ of a type ρ to the set of terms Tm^ρ of type ρ together with a function $\uparrow^\rho$ called *reflect*[1] from Tm^ρ to $[\![\rho]\!]$.

Definition 2.16 (Reify $\downarrow$ and Reflect $\uparrow$)**.** *The functions* $\downarrow_\rho \colon [\![\rho]\!] \to \mathsf{Tm}^\rho$ *and* $\uparrow^\rho \colon \mathsf{Tm}^\rho \to [\![\rho]\!]$ *are defined by simultaneous induction on* $\rho \in \mathsf{Ty}$ *by:*

$$\begin{array}{lcll} \uparrow^o e & ::= & e & \\ (\uparrow^{\rho\to\sigma} e)(a) & ::= & \uparrow^\sigma e(\downarrow_\rho a) & \\ \downarrow_o e & ::= & e & \\ \downarrow_{\rho\to\sigma} f & ::= & \lambda x^\rho.\, \downarrow_\sigma (f(\uparrow^\rho x)) & (x \text{ new}) \end{array}$$

The function $\downarrow$ at arrow type creates a variable x and returns an abstraction with respect to this variable. Informally, the condition "x new" ensures that this abstraction binds only occurrences corresponding to this created variable x. The mathematical formalisation is somewhat technical and we will deal with it later on.

Remark and notation 1. *The interpretations of the types are disjoint, hence the function* $\downarrow \colon [\![\mathsf{Ty}]\!] \to \mathsf{Tm}$ *defined by* $\downarrow = \bigcup_{\rho\in\mathsf{Ty}} \downarrow_\rho$ *is well defined.*

On the contrary, sets of terms typable by different types are not disjoint (a variable, for example, can be typed with all types). Hence erasing the type ρ *in* $\uparrow^\rho$ *does not formally make sense.*

However, to improve readability, we will in both cases, sometimes write $\downarrow$ *and* $\uparrow$ *instead of* $\downarrow_\rho$ *and* $\uparrow^\rho$ *when the type* ρ *is clear from the context .*

Notation 8. *For a given type* ρ*, we have* $\uparrow^\rho x \in [\![\rho]\!]$*, hence* $\uparrow$ *can be considered as a valuation function on any context* Γ*. We will not introduce a new notation in this case.*

For example for a typed term $\Gamma \vdash r : \sigma$*, in* $[\![r]\!]_\uparrow$*,* $\uparrow$ *shall denote the valuation defined on all variables* x *such that* $x : \rho \in \Gamma$ *by* $\uparrow^\sigma x$*.*

We are now in position to define the normalization function nf.

Definition 2.17 (The nf function)**.** *The function* $\mathsf{nf} : \mathsf{Tm} \to \mathsf{Tm}$ *is defined by:*

$$\mathsf{nf}(r) ::= \downarrow [\![r]\!]_\uparrow$$

For now, we want to give a justification that nf is indeed a normalization function, we want to verify (i) $r =_{\beta\eta} \mathsf{nf}(r)$ and (ii) $r =_{\beta\eta} s \Rightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s)$ where $\equiv \subseteq \beta\eta$ is syntactical equality.

[1] One find also the names *quote* and *unquote* for $\downarrow$ and $\uparrow$ in the literature.

Justification of $r =_{\beta\eta} \mathsf{nf}(r)$

To justify $r =_{\beta\eta} \mathsf{nf}(r)$ we will present informally a proof due to Martin Hofmann [52], which uses a logical relation. The idea is to relate a term r of type τ and a value a of the interpretation of τ by a logical relation $\mathcal{R}^\tau \subseteq \mathsf{Tm}_\tau \times [\![\tau]\!]$, such that

$$x\ \mathcal{R}^\tau\ \uparrow x$$
$$r\ \mathcal{R}^\tau\ a \Rightarrow r =_{\beta\eta} \downarrow a$$

Because $\mathcal{R}$ is a logical relation (logical relations are presented in more detail in the next section), the so-called basic lemma holds:

Lemma 2 (Basic Lemma)**.** *Given a term r typed by $\Gamma \vdash r : \rho$ with free variables $\overrightarrow{x} : \overrightarrow{\sigma} \in \Gamma$, we have:*

$$\forall\, \overrightarrow{s}\ \mathcal{R}^{\overrightarrow{\sigma}}\ \overrightarrow{b} \Rightarrow r_{\overrightarrow{x}}[\overrightarrow{s}]\ \mathcal{R}\ [\![r]\!]_{\overrightarrow{x} \mapsto \overrightarrow{b}}$$

In particular for $\overrightarrow{b} = \overrightarrow{\uparrow x}$, we get the desired result: $r =\downarrow [\![r]\!]_\uparrow$.

In more detail, the logical relation $\mathcal{R}$ can be defined by

Definition 2.18 (Logical Relation $\mathcal{R}$)**.** *We define a logical relation $\mathcal{R} = \bigcup \mathcal{R}^\rho$ with $\mathcal{R}^\rho \subseteq \mathsf{Tm}_\rho \times [\![\rho]\!]$ defined by induction on the type ρ by:*

$$\begin{aligned} r\ \mathcal{R}^o\ a &::= r =_{\beta\eta} a \\ r\ \mathcal{R}^{\rho\to\sigma}\ a &::= \forall\, s\ \mathcal{R}^\rho\ b, rs\ \mathcal{R}^\sigma\ ab \end{aligned}$$

Lemma 3. *The following implications hold:*

$$r\ \mathcal{R}^\tau\ a \Rightarrow r =_{\beta\eta} \downarrow a \qquad (1)$$
$$r =_{\beta\eta} s \Rightarrow r\ \mathcal{R}^\tau\ \uparrow s \qquad (2)$$

Proof. By induction on the type τ

- Case ι, obvious
- Case $\rho \to \sigma$ (1),
  We want to show $r =_{\beta\eta} \downarrow^{\rho\to\sigma} a$
  By definition:

  $$\downarrow^{\rho\to\sigma} a = \lambda x^\rho.\, \downarrow a \uparrow x \qquad (x \text{ new})$$

  By induction hypothesis of (2) on ρ, $x\ \mathcal{R}^\rho\ \uparrow x$, so by definition of $r\ \mathcal{R}^{\rho\to\sigma}\ a$, we have $rx\ \mathcal{R}^\sigma\ a \uparrow x$ and by induction hypothesis of (1) on σ, $rx =_{\beta\eta} \downarrow a \uparrow x$. Finally $r =_{\beta\eta} \lambda x^\rho.rx = \lambda x.\, \downarrow a \uparrow x =\downarrow a$.

- case $\rho \to \sigma$ (2),
  Given $t\ \mathcal{R}^{\rho}\ a$ we want to show $rt\ \mathcal{R}^{\sigma}\ (\uparrow^{\rho\to\sigma} s)(a)$
  By definition:

  $$(\uparrow^{\rho\to\sigma} s)(a) = \uparrow^{\sigma} (s \downarrow^{\rho} a)$$

  Now by induction hypothesis of (1) on ρ, $t =_{\beta\eta} \downarrow^{\rho} a$, so $rt =_{\beta\eta} s \downarrow a$, and by induction hypothesis of (2) on σ, $rt\ \mathcal{R}^{\sigma}\ \uparrow^{\sigma} s \downarrow a$.

□

Justification of $r =_{\beta\eta} s \Rightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s)$

To justify $r =_{\beta\eta} s \Rightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s)$ with $\equiv \subseteq \beta\eta$, the traditional argument uses the soundness of the interpretation with respect to the equality $=_{\beta\eta}$, i.e., that for a given valuation function η (on a context typing r and s):

$$r =_{\beta\eta} s \Rightarrow [\![r]\!]_{\eta} = [\![s]\!]_{\eta}$$

As $\downarrow$ is a function from the interpretation to the term language, it follows that $\downarrow ([\![r]\!]_{\eta}) \equiv \downarrow ([\![s]\!]_{\eta})$ where $\equiv$ is the syntactical equality, which is decidable and contained in the conversion relation $=_{\beta\eta}$. By taking the valuation η to be $\uparrow$, we obtain the result:

$$r =_{\beta\eta} s \Rightarrow \downarrow ([\![r]\!]_{\uparrow}) \equiv \downarrow ([\![s]\!]_{\uparrow})$$

However, if the interpretation is not a model of our term language in the sense that two $\beta\eta$-equal terms are not interpreted by the same element, this argument does not apply directly and needs to be adapted.

It will be in particular the case of our interpretation when dealing with the *new variable* problem.

Remark 4. *The normalization by evaluation algorithm is often referred to be reduction free because, as in the argument above one can avoid any reference to rewriting theory.*

However, for the function nf *to be terminating one need the evaluation of the interpretation itself to terminate.*

More generally one can see normalization by evaluation as a way to focus on the study of the rewriting theory of the interpretation of a language instead of the rewriting theory of the language itself. The work of Klaus Aehlig and Felix Joachimski (see [2]) use a two level lambda calculus, where the interpretation itself is a syntactic lambda calculus, and this allows them to carry out a fine-grained rewriting analysis of the rewriting involved in the NbE algorithm.

2.2.2 Normal Forms

As we are only interested in the correctness of our algorithm this introduction could end here but there is another interesting question, namely:

How does the terms produced by the NbE algorithm look like?

Let us call NF (for set of normal form) the subset of terms which are results of the function nf. Hence nf is a function from Tm to NF. Because $\mathsf{nf}(r)$ is defined as $\downarrow(\llbracket r \rrbracket_{\uparrow})$, the function $\downarrow$ applied to a value $\llbracket r \rrbracket_{\uparrow}$ should only produce terms in NF.

Then by analysing the algorithm we can as well restrict the domain of $\uparrow$. We note that in the first step of the evaluation of nf, we apply $\uparrow$ only to variables, then if the variable is of arrow type, we apply $\uparrow$ to an application of a variable to a term resulting from $\downarrow$, i.e., a normal form $N \in \mathsf{NF}$. It is easy to see that in fact the domain of $\uparrow$ is a set of neutral term Ne given by

$$\mathsf{Ne} \ni n ::= x \mid n\, N$$

where $N \in \mathsf{NF}$ is a normal form.

Let us write NF^{ρ} (resp. Ne^{ρ}) the set of normal terms (resp. neutral terms) typable with type ρ. The function $\downarrow$ at base type is the identity, and at arrow type involves the term abstraction over a recursively obtained result of $\downarrow$, i.e., a term in NF, and it is easy to see that NF^{τ}, the set of normal forms of type τ, verifies:

$$r \in \mathsf{NF}^{\tau} \Leftrightarrow \begin{cases} r \in \llbracket o \rrbracket\ , & \text{if } \tau = o \\ r = \lambda x.s \wedge s \in \mathsf{NF}^{\sigma}, & \text{if } \tau = \rho \to \sigma \end{cases}$$

Now we want to restrict the interpretation. As the function $\uparrow$ is the identity at base type, the interpretation at base type $\llbracket o \rrbracket$ should at least contain the domain of $\uparrow$, i.e., neutral terms of base type Ne^{o}. It is in fact sufficient to take:

$$\llbracket o \rrbracket = \mathsf{Ne}^{o}$$

To summarise we have informally shown that the set NF of normal forms given by the NbE algorithm is inductively defined by:

Definition 2.19 (Λ-normal form)**.**

$$\frac{(x:\rho) \in \Gamma}{\Gamma \vdash_{\mathsf{Ne}} x : \rho} \qquad \frac{\Gamma \vdash_{\mathsf{Ne}} n : \rho \to \sigma \qquad \Gamma \vdash_{\mathsf{NF}} N : \rho}{\Gamma \vdash_{\mathsf{Ne}} nN : \sigma} \qquad \frac{\Gamma \vdash_{\mathsf{Ne}} n : o}{\Gamma \vdash_{\mathsf{NF}} n : o} \qquad \frac{\Gamma, x:\rho \vdash_{\mathsf{NF}} N : \sigma}{\Gamma \vdash_{\mathsf{NF}} \lambda x.N : \sigma}$$

A more precise specification for the function $\uparrow$, $\downarrow$ and nf can now be given with the following domain and codomain:

$$\uparrow^{\rho} : \mathsf{Ne}^{\rho} \to \llbracket o \rrbracket$$
$$\downarrow_{\rho} : \llbracket o \rrbracket \to \mathsf{NF}^{\rho}$$
$$\mathsf{nf} : \mathsf{Tm} \to \mathsf{NF}$$

The normal forms in NF are known under the name of long normal forms. These normal forms are obtained as the irreducible terms if one orients the conversions as reductions as follows,

$$\begin{aligned}(\lambda x.r)s &\longrightarrow_\beta r\{^x/_s\}\\ r &\longrightarrow_\eta \lambda x.rx \qquad (x \notin \mathsf{FV}(r))\end{aligned}$$

with the further requirement that in the η-reduction the term r is not an abstraction and is not in applicative position.

This definition of long normal forms first appeared in the works of Gerard Huet [54].

This agreement of normal forms obtained by reductions and by the *NbE* algorithm is indeed no accidental coïncidence. In [17], Ulrich Berger extracts a normalization algorithm from a reduction based normalization proof; the extracted algorithm is the *NbE* algorithm presented above. This extraction has been recently formalized in different proof assistant (Coq, Isabelle, minlog), the interested reader can consult [18]. In [30] and [31], Thierry Coquand and Peter Dybjer show that the proof of correctness of their *NbE* normalization function is in fact an optimized version of a standard proof of normalization of intuitionistic type theory.

2.2.3 Name Generation Environment

Let us return to the problem of the "new" variable.

It appears in the evaluation of $\mathsf{nf}(r) =\downarrow [\![r]\!]_\uparrow$, in each recursive call of $\downarrow$ at arrow type:

$$\downarrow_{\rho\to\sigma} f = \lambda x.\ \downarrow_\sigma f(\uparrow^\rho x) \qquad\qquad x \text{ new}$$

The term $\downarrow_\sigma f(\uparrow^\rho x)$ can contain free variables which are either free variables of r or which have been created by other recursive calls of $\downarrow$. The side condition "x new" means that x should be different from these variables.

These two situations are exposed in the example below:

Example 2. *A newly created variable should be different from one occurring free in r. Let r be the term y with the typing $y : o \to o \vdash y : o \to o$*

$$\begin{aligned}\mathsf{nf}(r) &=\downarrow [\![y]\!]_\uparrow\\ &=\downarrow (\uparrow^{o\to o} y)\\ &= \lambda x.\ \downarrow_o (\uparrow^{o\to o} y)(\uparrow^o x) && x \textit{ new}\\ &= \lambda x.\ \downarrow_o (\uparrow^{o\to o} y)(x) && x \textit{ new}\\ &= \lambda x.(\uparrow^{o\to o} y)(x)\\ &= \lambda x.\ \uparrow^o y \downarrow_o x\\ &= \lambda x.\ \uparrow^o yx\\ &= \lambda x.yx\end{aligned}$$

A newly created variable should be different from already created variables. Let r be the term $\lambda y.y$ with the typing $\vdash \lambda y.y : (o \to o) \to o \to o$

$$\begin{aligned}
\mathsf{nf}(r) =& \downarrow [\![\lambda y.y]\!]_{\uparrow} \\
&= \lambda x.\, \downarrow_{o\to o} ([\![\lambda y.y]\!]_{\uparrow}(\uparrow^{o\to o} x)) && x \text{ new} \\
&= \lambda x.\, \downarrow_{o\to o} (\uparrow^{o\to o} x) \\
&= \lambda x.\lambda y.\, \downarrow_{o} ((\uparrow^{o\to o} x)\uparrow^{o} y) && y \text{ new} \\
&= \lambda x.\lambda y.(\uparrow^{o\to o} x)y \\
&= \lambda x.\lambda y.\uparrow^{o} x(\downarrow^{o} y) \\
&= \lambda x.\lambda y.xy
\end{aligned}$$

Said informally, in the expression

$$\lambda x.\, \downarrow_{\sigma} f(\uparrow^{\rho} x) \qquad x \text{ new}$$

the function f already "contains" the necessary information (i.e., the variables already used) to compute the new variable x. However f is a function, and to extract this information, one has to apply f to an argument, but we are precisely looking for an appropriate argument of f.

To solve this dilemma, a possible solution is to record along the evaluation of $\downarrow [\![r]\!]_{\uparrow}$ which variables have been used (those free in r and those already created by a call of $\downarrow$ at arrow type).

In fact, all we need to create new variables, is to have at hand a set of variables, which does not contain the already used variables. Hence, we do not even need to record all the used variables, but merely a set of unused ones. This weaker solution can read informally as follows:

- We begin the evaluation of $\downarrow [\![r]\!]_{\uparrow}$ with a denumerable set $e = e_r$ of variables not containing those free in r,
- when evaluating $\downarrow$ at arrow type, we first pick a new variable x from e, and continue the computation with the set $e \setminus \{x\}$.

We can see in this informal exposition that the set e acts exactly like an environment. The computation for the function $\downarrow$ at arrow type needs to read a value, a fresh variable x, from e and run some subcomputations in an updated environment e^x (without this fresh variable).

After this informal description it is now time to give a formal specification:

Definition 2.20 (Name Generation Environment)**.** *We define a set of* name generation environment *or set of environment for short, as a set E together with an update function $(-)^- : E \to \mathsf{Var} \to E$ and an access function $\mathsf{new} : E \to \mathsf{Var}$.*

The extension of the update function $e^- : V \to E$ to a function from a list of variables $e : \mathbf{L}(V) \to E$ is defined in a canonical way by

$$e^{x,\overrightarrow{x}} ::= (e^x)^{\overrightarrow{x}}$$
$$e^{\varepsilon} ::= e.$$

The function $(-)^-$ and new *have moreover to satisfy the following property for all $e \in E$:*

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{x,\overrightarrow{x}}) \qquad (\dagger)$$

An environment $e \in E$ is meant to be a denumerable set of variables, the update function $-^x$ applied to an environment e is meant to remove a variable x from e, and $\mathsf{new}(e)$ to pick a variable from e.

An explanation for the condition (†) is given after having introduced the following notation.

Notation 9. *The condition that the* new *function applied to an environment $e \in E$ should never return a given variable x can be expressed by:*

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{\overrightarrow{x}})$$

We will abbreviate this condition by $x \notin e$. In the same way we will abbreviate for a given set of variable X, $\forall\, x \in X, x \notin e$ by $X \notin e$.

With this notation, the condition (†) reads:

$$x \notin e^x$$

This means that once a variable x has been removed from an environment e with the function $(-)^-$, x can not be picked out anymore.

Returning to the *NbE* algorithm, the initialisation step for a term r consists in finding an environment e_r such that the new function will never give back a variable among those free in r.

$$\forall\, x \in \mathsf{FV}(r), x \notin e_r$$

In fact for an arbitrary given e, $e^{\mathsf{FV}(r)}$ does the job.

Example 3. *This name generation environment has been implemented by Ulrich Berger in [17] with indexed variables, i.e., of the form x_k where $k \in \mathbb{N}$, and is essentially as follows.*

The set of environment E is taken to be $\mathbb{N}$, and a natural number k is a code for the sequence $x_k, x_{k+1}, \ldots$. The initialisation part consists to look for the higher index k of the variables of the form x_k occurring in r. Because for all $k' \geqslant k+1$, $x_{k'}$ is not free in r, $k_r = k+1$ is a code for a sequence of fresh variables for r and we only need to propagate a natural number instead of a set of variables.

The implementation is

$$\begin{aligned} k^{x_j} &::= \max(k, j+1) \\ \mathsf{new}(k) &::= x_k \\ k_r &::= 0^{(\mathsf{FV}(r))} \end{aligned}$$

In particular, the use of indexed variables by Berger corresponds to de Bruijn levels where an index of a variable corresponds to the number of lambda abstraction in the syntax tree of the term, from the occurrence of this variable to the root of the tree.

Remark 5. *In an impure functional programming language, the concept of environment, sequence of instructions and assignment are primitive. The implementation is then easy, it suffices to define such an e in the global environment and update it with an assignment instruction $e := e^x$, which will update the environment before further computations.*

However this would take us a step further away from a mathematical formalisation. It is why we prefer to stick to a pure functional programming language setting, where the primitive notion of function has a direct counterpart in a mathematical setting. We will then implement impure functional concept such as environments (in the formalization section of chapter 2, 3 and 4) or exceptions (in the formalization section of chapter 3 and 4) within our functional settings with the help of monads.

Rest now to redesign the NbE algorithm to propagate in an adequate way this environment e through the computation.

2.3 Formalization

2.3.1 Name Generation Interpretation

We will be guided by the fact that this notion of computation in environment is naturally captured by a state reader monad.

The basic idea is that if a computation which produces a value in some set A needs to access an environment then we will have to pass this environment as a supplementary argument to the function corresponding to this computation, hence one can replace these set A in the specification of the function by a function space $\mathbf{St}_E(A) = E \to A$ from the set of environments E to A. The set A has been replaced by the state reader monad $\mathbf{St}_E$ over A.

In particular, in the computation of the function $\downarrow_\tau$, an environment e should be available in order to be able to pick a fresh variable x from it. The specification of $\downarrow_\tau$ hence becomes

$$[\![\tau]\!] \to \mathbf{St}_E(\mathsf{NF}^\tau) = [\![\tau]\!] \to E \to \mathsf{NF}^\tau.$$

Notice also that the specification of the operation $\mathsf{new} : E \to \mathsf{Var}$ now reads:

$$\mathsf{new} : \mathbf{St}_E(\mathsf{Var})$$

The update operation $(-)^- : E \to \mathsf{Var} \to E$ for a name generation environment E extends to an update function on elements of the monad $\mathbf{St}_E(A)$.

Definition 2.21 (update for the Name Generation Monad)**.** *We define an update function* $(-)^- : \mathbf{St}_E(A) \to \mathsf{Var} \to \mathbf{St}_E(A)$ *operation by extending the update operation for the name generation monad by:*

$$m^v(e) ::= m(e^v)$$

A First Try

To define correctly the function $\downarrow_{\rho\to\sigma}$ we have to first take a fresh variable from the environment and then pass an updated environment to the subcomputation in the body of the function.

Maybe the first solution coming to mind is to define the function $\downarrow_{\rho\to\sigma}$ as follows:

$$\downarrow_{\rho\to\sigma} f = \mathbb{\lambda} e.(\mathbb{\lambda} v.\lambda v(\downarrow f \uparrow v)e^v)(\mathsf{new}(e))$$

or in monadic notation

$$\downarrow_{\rho\to\sigma} f = \mathsf{new} \star \mathbb{\lambda} v.(\downarrow f \uparrow v)^v \star \mathbb{\lambda} r.\nu(\lambda v.r)$$

There is still a problem with this definition: the function $\uparrow^\tau$ has here the same specification as before, i.e., $\mathsf{Tm} \to [\![\tau]\!]$, and at arrow type a computation of $\uparrow^{\rho\to\sigma}$ contains a subcomputation of $\downarrow_\rho$, which require an argument $e \in E$, but we do not have this argument at this point. What would be the second argument of the function $\downarrow_\rho$ in this case?

$$\uparrow^{\rho\to\sigma} r(a) = \uparrow^\sigma r(\downarrow_\rho (a)??)$$

An arbitrary environment $e \in E$ does not fit because we do still need to know which are the variables used in a. One could then think of simply adding an environment as argument to the function $\uparrow$. Alas, this simple solution alone does not work; this notion of environment for names has to be incorporated into the interpretation.

Modifying the Interpretation

By analogy with programming we can think of the application of a monad to a set as the interpretation of a type together with computational content. We will develop two different ways to integrate such a computational type into the interpretation of the simply typed λ-calculus, differing in the intended strategy of evaluation of the interpreted terms.

Following the work of Nick Benton, John Hughes and Eugenio Moggi in [16], we will call our first modified interpretation the *Algol interpretation* and the second the *call-by-value interpretation*.

The idea to use a monadic interpretation in conjunction with NbE is due to the works of Andrzej Filinski and Peter Dybjer, [40] and [43]. There, they show that, in calculi possibly extended with constants, NbE actually provides normalization functions for different evaluation strategies, the normalization is proved correct and complete with respect to an equivalence defined via the interpretation (what is called Algol interpretation here is called call-by-name interpretation there). Here we are concerned with $\beta\eta$-conversions, and these evaluation strategies does not in general correspond exactly with $\beta\eta$-conversions. As we will see, in the case of the Algol interpretation the NbE algorithm provides already a normalization function with respect to $\beta\eta$-conversions. But it is even true in the case of the call-by-value interpretation, it is a somewhat surprising result because, in this case, the interpretation is unsound for $\beta\eta$-conversions (two $\beta\eta$-convertible terms may have a different interpretation) and the last sections of this chapter are devoted to prove this.

In the following definitions of interpretations, a valuation on a context Γ is defined as before as a partial function from the set of variables to the interpretation of types, such that a typed variable $x : \rho$ in Γ is mapped to an element of the interpretation $[\![\rho]\!]$ of the type ρ. As remarked in section (2.2.2), one only needs neutral terms of ground type in the interpretations of ground type.

One can allow computational effects only at base type as in the programming language idealised Algol (see [80] or [81]). For us it will mean that the monad only appear in the interpretation of ground type:

Definition 2.22 (Algol interpretation)**.** *The Algol interpretation* $[\![]\!]^{alg}$ *is given on the type by*

$$\begin{aligned} [\![o]\!]^{alg} &::= \mathbf{St}_E(\mathsf{Ne}_o) \\ [\![\rho \to \sigma]\!]^{alg} &::= [\![\rho]\!]^{alg} \to [\![\sigma]\!]^{alg} \end{aligned}$$

Given a valuation η *on* Γ*, a typed term* $\Gamma \vdash r : \rho$ *is interpreted as an element of* $[\![\rho]\!]^{alg}$ *by*

$$\begin{aligned} [\![x]\!]^{alg}_\eta &::= \eta(x) \\ [\![rs]\!]^{alg}_\eta &::= [\![r]\!]^{alg}_\eta [\![s]\!]^{alg}_\eta \\ [\![\lambda x.r]\!]^{alg}_\eta &::= \lambda\!\!\lambda a.[\![r]\!]^{alg}_{\eta, x \mapsto a} \end{aligned}$$

One can think of the interpretation of a λ-term as a program in a call-by-value setting. In this case, a program takes a value as argument and produces a computation (the monad appears in the codomain of the interpretation of function spaces). Hence the interpretation is:

Definition 2.23 (call-by-value interpretation)**.** *The call-by-value interpretation* $[\![]\!]^{val}$ *is defined on the types by*

$$\begin{aligned} [\![o]\!]^{val} &::= \mathsf{Ne}_o \\ [\![\rho \to \sigma]\!]^{val} &::= [\![\rho]\!]^{val} \to \mathbf{St}_E([\![\sigma]\!]^{val}) \end{aligned}$$

Given a valuation η *on* Γ*, a typed term* $\Gamma \vdash r : \rho$ *is interpreted as an element of* $\mathbf{St}_E([\![\rho]\!]^{val})$ *by*

$$\begin{aligned} [\![x]\!]^{val}_\eta &::= \nu(\eta(x)) \\ [\![\lambda x.r]\!]^{val}_\eta &::= \nu(\lambda\!\!\lambda a.[\![r]\!]^{val}_{\eta, x \mapsto a}) \\ [\![rs]\!]^{val}_\eta &::= [\![r]\!]^{val}_\eta \star \lambda\!\!\lambda f.[\![s]\!]^{val}_\eta \star \lambda\!\!\lambda a.fa \end{aligned}$$

We present the call-by-name interpretation although we will not further analyse it. In a call by name setting a program is expected to take as argument a computation and to produce another computation (the monad appears both in the domain and codomain of the interpretation of function spaces), hence this last interpretation is:

Definition 2.24 (call-by-name interpretation)**.** *The* call-by-name interpretation $[\![]\!]^{name}$ *is defined on type by*

$$\begin{aligned} [\![o]\!]^{name} &::= \mathsf{Ne}_o \\ [\![\rho \to \sigma]\!]^{name} &::= \mathbf{St}_E([\![\rho]\!]^{name}) \to \mathbf{St}_E([\![\sigma]\!]^{name}) \end{aligned}$$

Given a valuation η on Γ, a typed term $\Gamma \vdash r : \rho$ is interpreted as an element of $\mathbf{St}_E([\![\rho]\!]^{name})$ by

$$\begin{aligned} [\![x]\!]^{name}_\eta &::= \eta(x) \\ [\![\lambda x.r]\!]^{name}_\eta(m) &::= m \star \mathbb{\lambda} a.[\![r]\!]^{name}_{\eta, x \mapsto a} \\ [\![rs]\!]^{name}_\eta &::= [\![r]\!]^{name}_\eta \star \mathbb{\lambda} f. f [\![s]\!]^{name}_\eta \end{aligned}$$

Several Solutions

In the work using *NbE*, for the most part, the usual solution is to chose what we have called the Algol interpretation. The immediate advantage is that this interpretation provides a Henkin Model. By well known results (see for example [73]), we know that this interpretation is sound for the $\beta\eta$-conversion, which means that for an arbitrary valuation η on Γ, terms r and s $\beta\eta$-convertible, their interpretations are equal $[\![r]\!]^{alg}_\eta = [\![s]\!]^{alg}_\eta$. Hence the argument exposed in the informal presentation to prove $r =_{\beta\eta} s \Rightarrow \mathsf{nf}(r) \equiv \mathsf{nf}(s)$ is directly applicable.

Choosing this interpretation, the *NbE* algorithm reads in a monadic style as follows:

Code 1 (Algol *NbE*)**.** *The function $\downarrow_\tau : [\![\tau]\!] \to \mathbf{St}_E(\mathsf{NF}_\tau)$ and $\uparrow_\tau : \mathbf{St}_E(\mathsf{Ne}_\tau) \to [\![\tau]\!]$ are defined simultaneously.*

$$\begin{array}{lcl} \downarrow_o r & = & r \\ \downarrow_{\rho\to\sigma} f & = & \mathsf{new} \star \mathbb{\lambda} v.(\downarrow f \uparrow \nu(v))^v \star \mathbb{\lambda} t.\nu(\lambda v.t) \\ \uparrow^o F & = & F \\ \uparrow^{\rho\to\sigma} F(a) & = & \uparrow^\sigma (F \star \mathbb{\lambda} r.\downarrow_\rho a \star \mathbb{\lambda} s.\nu(rs)) \end{array}$$

Code 2. *Given an environment e_r such that $\mathsf{FV}(r) \notin e_r$ the normalization function $\mathsf{nf} : \mathsf{Tm} \to \mathsf{NF}$ is defined by*

$$\mathsf{nf}(r) = \downarrow [\![r]\!]_{\uparrow \circ \nu} e_r$$

However when dealing with sum type, we will need to define the function $\uparrow$ at sum type. But to define the result of $\uparrow^{\rho_0+\rho_1} m$ for m of type $M(\mathsf{Tm}_{\rho_0+\rho_1})$, we need first to know if the term "contained" in m corresponds to a left injection, a right injection, or if we don't know yet. In this setting, the only possibility to extract a term from an element $m \in M(\mathsf{Tm}_{\rho_0+\rho_1})$ is to apply the function $\uparrow$ and this should be done at

a structurally smaller type, i.e., either at ρ_0 or ρ_1. But to know whether it is ρ_0 or ρ_1, we are once again faced to the problem of analysing the term contained in $m \in M(\mathsf{Tm}_{\rho_0+\rho_1})$. We didn't find any natural solution to this problem, which leads us to the second solution proposal, the call-by-value interpretation.

Remark 6. *A refined analysis that monadic values $F \in M\mathsf{Tm}$ occurring in the Algol NbE algorithm are term families which when applied to environments only differ by their bound variables.*

Hence we can extract the fresh variables from such a monadic value $F \in \mathsf{Tm}$ by applying it to an arbitrary environment $\mathsf{FV}(Fe)$. Now that we know the fresh variables, we can extract a term from F, by applying it to an environment e' updated with these fresh variables: $Fe'^{\mathsf{FV}(Fe)}$.

Disregarding that this solution does not seem very natural, we will have to thread exceptions for the treatment of sum types into the NbE algorithm and this is not evident with the Algol interpretation.

As already mentioned, the call-by-value interpretation already appeared in the paper [43] of Andrzej Filinski and [40] with Peter Dybjer. In these papers, they based the name generation interpretation on a state monad. They are then able to define an extension of NbE for sum type in the setting of the call-by-value interpretation. In this sense the works of Andrzej Filinski and Peter Dybjer can be seen as the closest to ours. The first difference is that they used a state passing monad $\mathbf{ST}_E A = E \to E \times A$ whereas we use simply a state reader monad $\mathbf{St}_E A = E \to A$. At first, this seems to be a minor difference, but it will simplify the proof of correctness as we will not have to deal with administrative product types in the interpretation. The second and more important difference is that they proved correctness of NbE for terms which have the same call-by-value interpretation. We will prove correctness for terms which are $\beta\eta$ equal.

The main advantage of this interpretation is to simplify the type of the argument of the reflect function $\uparrow$. In the Algol interpretation an argument of $\uparrow^\tau$ has type $M(\mathsf{Ne}^\tau)$, and in the call-by-value interpretation it has the type Ne^τ. This direct access to the term argument of the reflect function $\uparrow$ will allow us to extend quite naturally the NbE algorithm to sum type in chapter 4.

The algorithm for the call-by-value interpretation reads:

Code 3 (call-by-value NbE)**.** *The function $\downarrow_\tau \colon [\![\tau]\!]^{val} \to \mathbf{St}_E(\mathsf{Ne}_\tau)$ and $\uparrow_\tau \colon \mathsf{Ne}_\tau \to [\![\tau]\!]^{val}$ are defined simultaneously by:*

$$
\begin{aligned}
\downarrow_o r &= \nu(r) \\
\downarrow_{\rho\to\sigma} f &= \mathsf{new} \star \mathbb{\lambda} v.(f \uparrow v\star \downarrow)^v \star \mathbb{\lambda} t.\nu(\lambda v.t) \\
\uparrow^o r &= r \\
\uparrow^{\rho\to\sigma} r(a) &= (\downarrow_\rho a) \star \mathbb{\lambda} s.\nu(\uparrow^\sigma rs)
\end{aligned}
$$

Code 4. *Given an environment e_r such that $\mathsf{FV}(r) \notin e_r$ the normalization function* nf *is defined by*

$$\mathsf{nf}(r) = (\llbracket r \rrbracket^{val}_{\uparrow} \star \downarrow) e_r$$

A problem of the call-by-value interpretation is that it does not generally provide a sound interpretation, i.e., we can have $r =_{\beta\eta} s$ but $\llbracket r \rrbracket^{val}_{\eta} \neq \llbracket s \rrbracket^{val}_{\eta}$ for a given valuation η. It is in particular the case of our set theoretical interpretation with the state reader monad as shown in the following example:

Example 4 (unsoundness of call-by-value interpretation)**.** *Let η be a valuation with $\eta(z) = f \in \llbracket o \to o \rrbracket^{val}, \eta(z') = a \in \llbracket o \rrbracket^{val}$,*

$$z : o \to o, z' : o \vdash (\lambda xy.x)(zz') =_{\beta} \lambda y.zz' : o \to o$$

the interpretation of these terms are

$$\begin{aligned}
\llbracket (\lambda xy.x)(zz') \rrbracket^{val}_{\eta} &= \llbracket \lambda xy.x \rrbracket^{val}_{\eta} \star \mathbb{\lambda} g.\llbracket zz' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} a.ga && \text{by definition 2.23} \\
&= \nu(\mathbb{\lambda} b.\llbracket \lambda y.x \rrbracket^{val}_{\eta, x \mapsto b}) \star \mathbb{\lambda} g.\llbracket zz' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} a.ga && \text{by definition 2.23} \\
&= (\mathbb{\lambda} g.\llbracket zz' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} a.ga)(\mathbb{\lambda} b.\llbracket \lambda y.x \rrbracket^{val}_{\eta, x \mapsto b}) && \text{by definition 1.1} \\
&= \llbracket zz' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} a.(\mathbb{\lambda} b.\llbracket \lambda y.x \rrbracket^{val}_{\eta, x \mapsto b})a) \\
&= \llbracket zz' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} a.\llbracket \lambda y.x \rrbracket^{val}_{\eta, x \mapsto a} \\
&= (\llbracket z \rrbracket^{val}_{val} \star (\mathbb{\lambda} g.\llbracket z' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} b.gb)) \star \mathbb{\lambda} a.\llbracket \lambda y.x \rrbracket^{val}_{\eta, x \mapsto a} && \text{by definition 2.23} \\
&= (\llbracket z \rrbracket^{val}_{val} \star (\mathbb{\lambda} g.\llbracket z' \rrbracket^{val}_{\eta} \star \mathbb{\lambda} b.gb)) \star \mathbb{\lambda} a.\mathbb{\lambda} c.\llbracket x \rrbracket^{val}_{\eta, x \mapsto a, y \mapsto c} && \text{by definition 2.23} \\
&= (\nu(f) \star (\mathbb{\lambda} g.\nu(a) \star \mathbb{\lambda} b.gb)) \star \mathbb{\lambda} a.\nu(\mathbb{\lambda} c.\nu(a)) && \text{by definition 2.23} \\
&= (\mathbb{\lambda} g.\nu(a) \star \mathbb{\lambda} b.gb)f) \star \mathbb{\lambda} a.\nu(\mathbb{\lambda} c.\nu(a)) && \text{by definition 1.1} \\
&= (\mathbb{\lambda} g.(\mathbb{\lambda} b.gb)a)f) \star \mathbb{\lambda} a.\nu(\mathbb{\lambda} c.\nu(a)) && \text{by definition 1.1} \\
&= fa \star \mathbb{\lambda} a.\nu(\mathbb{\lambda} c.\nu(a)) \\
\llbracket \lambda y.zz' \rrbracket^{val}_{\eta} &= \nu(\mathbb{\lambda} c.\llbracket zz' \rrbracket^{val}_{\eta}) ece' \\
&= \nu(\mathbb{\lambda} c.fa) && \text{see computations above}
\end{aligned}$$

And for two environments $e, e' \in E$, such that $e \neq e'$ and an element $c \in \llbracket o \rrbracket^{val}$, we have

$$\begin{aligned}
\llbracket (\lambda xy.x)(zz') \rrbracket^{val}_{\eta} ece' &= (fa \star \mathbb{\lambda} a.\nu(\mathbb{\lambda} c.\nu(a)))ece' \\
&= (\mathbb{\lambda} c.\nu(fae))ce' && \text{by definition 1.2} \\
&= fae && \text{by definition 1.2} \\
\llbracket \lambda y.zz' \rrbracket^{val}_{\eta} ece' &= \nu(\mathbb{\lambda} c.fa)ece' \\
&= (\mathbb{\lambda} c.fa)ce' && \text{by definition 1.2} \\
&= fae'
\end{aligned}$$

As soon as E as more than one element, it is obviously possible to find a function f in $[\![o \to o]\!]^{val} = \mathsf{Tm} \to (E \to \mathsf{Tm})$ such that

$$f\,ae \neq f\,ae'$$

For the completeness property, which relied on the soundness of the interpretation in the informal presentation, we have seen in the example above that the set theoretical equality is unsuitable. We will have to provide a coarser equality relation in section 2.3.4 which we will prove to be sound with respect to $\beta\eta$.

In contrast the soundness property does not need much adaptation. We will have to take into account the name generation monad and adapt the correctness lemma to this setting in section 2.3.3.

In these two proofs we will make use of the concept of logical relation, a class of relations which relate the meanings of terms in a structured way. We have already mentioned this concept in the introduction, the next section is devoted to it and its extension to our monadic settings.

2.3.2 Monadic logical relation

The work of Andrzej Filinski was again a great source of inspiration. In [42], Kripke relations are used to show the completeness and correctness of a normalization by evaluation algorithm in a call-by-name setting. In [41], Filinski defines a monadic logical relation in a call-by-value setting, notably extended for sum types, to implement rigorously and efficiently effects layered in a functional programming language. This monadic logical relation is similar to our definition 2.29 (ours without sum types). One difference is that we use general typed applicative structures as we want to handle syntactical applications and not only set-theoretical applications. Another difference is that in these syntactical application structures we need to consider open terms, and therefore use Kripke applicative structures. An example of a formalized proof of correctness of an NbE algorithm using a Kripke applicative structure can be found in the work of Catarina Coquand [28; 29]. The book of Mitchell [73] is a nice introduction of the concept of Kripke relations that we have reworked in the context of monadic interpretation.

As already mentioned, a logical relation relates the "meanings" of terms in a structured way. Here meaning has to be understood in an extended sense. Logical relations do not only allow to relate two different values of an interpretation, but as well to relate a term and a value in the interpretation (so that a term can be seen as the "meaning" of itself).

Accordingly, in a first step, we have to capture the similarity shared by the term and semantical structures. This is the rôle of (typed) applicative structures.

Definition 2.25 (Typed applicative structure)**.** *A* typed applicative structure $\mathcal{A}$ *is a pair* $\langle \{A^\rho\}, \{\mathsf{App}^{\rho,\sigma}\} \rangle$*, where* $\{A^\rho\}$ *is a family of sets* A^ρ *indexed by types* $\rho \in \mathsf{Ty}$ *and* $\{\mathsf{App}^{\rho,\sigma}\}$ *is a family of application functions* $\mathsf{App}^{\rho,\sigma} : A^{\rho\to\sigma} \to A^\rho \to A^\sigma$ *indexed by pair of types* $\rho, \sigma \in \mathsf{Ty}$*. We will write* A *for* $\bigcup_{\rho\in\mathsf{Ty}} A^\rho$

The two following examples show how the term and semantical structures can be both captured by typed applicative structures.

Example 5 (Set of terms typable in a given context)**.** *Given a context* Γ*, the pair* $\langle \{A^\rho\}, \{\mathsf{App}^{\rho,\sigma}\} \rangle$ *where* $A^\sigma = \{r \mid \Gamma \vdash r : \sigma\}$*, and* $\mathsf{App}^{\rho,\sigma}\ rs = rs$ *is a typed applicative structure.*

Example 6 (Type interpretation)**.** *The pair* $\langle \{A^\rho\}, \{\mathsf{App}^{\rho,\sigma}\} \rangle$ *where* $\{A^\rho\} = [\![\rho]\!]$ *is the type interpretation as given in definition* (2.22) *and* $\mathsf{App}^{\rho,\sigma}\ fa = f(a)$ *is the set-theoretic function application, is an applicative structure.*

An applicative structure can be seen as a special case of typed applicative structures where the indexed family of sets $\{A^\rho\}$ has been shrunk down to one unique set.

Example 7 (Untyped λ-calculus)**.** *The pair* $\langle \{A^\rho\}, \{\mathsf{App}^{\rho,\sigma}\} \rangle$ *where* $A^\sigma = \mathsf{Tm}$*, and* $\mathsf{App}^{\rho,\sigma}\ rs = rs$ *is a typed applicative structure.*

One can define a generalisation of applicative structure to monads by requiring that the application operator has a monadic result.

Definition 2.26 (Monadic applicative structure)**.** *A* monadic applicative structure *over a monad* M *is a tuple* $\langle \{A^\rho\}, \{\mathsf{MApp}^{\rho,\sigma}\} \rangle$ *where* $\{A^\rho\}$ *is a family of sets* A^ρ *indexed by types* $\rho \in \mathsf{Ty}$ *and* $\{\mathsf{MApp}^{\rho,\sigma}\}$ *is a family of application functions* $\mathsf{MApp}^{\rho,\sigma} : A^{\rho\to\sigma} \to A^\rho \to MA^\sigma$ *indexed by pair of types* $\rho, \sigma \in \mathsf{Ty}$*.*

Remark 7. *Monadic applicative structures are a generalisation of typed applicative structures, as a typed applicative structure is just a monadic applicative relation over the identity monad.*

Notation 10. *We will write* $a \cdot^{\rho,\sigma} b$ *(or* $a \cdot b$ *when the types are clear from the context) for* $\mathsf{MApp}^{\rho,\sigma}\ a\ b$*.*

Example 8 (call-by-value type interpretation)**.** *The pair* $\langle \{A^\rho\}, \{\cdot^{\rho,\sigma}\} \rangle$ *where* $\{A^\rho\} = [\![\rho]\!]^{val}$ *is the call-by-value interpretation of types as given in definition* (2.23) *and* $\cdot^{\rho,\sigma} fa = f(a)$ *is the set-theoretic function application, is a monadic applicative structure.*

Just like for interpretation we need to define the auxiliary notion of valuation to define the meaning of terms in an applicative structures.

Definition 2.27 (Valuation)**.** *A* valuation η *in an applicative structure* $\langle\{A^\rho\},\{\cdot^{\rho,\sigma}\}\rangle$ *is a partial function from* Var *to* $A = \bigcup_{\rho\in\mathsf{Ty}} A^\rho$*.*

Given a context Γ*, a* valuation η on Γ*, written* $\eta \vDash \Gamma$*, is a valuation such that for* $x : \rho \in \Gamma$*,* $\eta(x)$ *is defined and* $\eta(x) \in A^\rho$*.*

Given a context Γ*, a valuation* η *on* Γ*, a variable* $y \notin \Gamma$ *and an element* $a \in A^\sigma$*, we define a valuation* $(\eta, y \mapsto a)$ *on* $\Gamma \cup \{(y : \sigma)\}$*, called* the extension of η *by* $y \mapsto a$ *by,*

$$(\eta, y \mapsto a)(x) ::= \begin{cases} a & \text{if } x = y, \text{ and} \\ \eta(x) & \text{otherwise.} \end{cases}$$

A meaning function is a compatible function from the term structure to the monadic applicative structure in the following sense.

Definition 2.28 (call-by-value acceptable meaning functions)**.** *Given a monad* M *and an applicative structure* $\mathcal{A} = \langle\{A^\rho\},\{\cdot^{\rho,\sigma}\}\rangle$*, a partial function* $[\![]\!]^{val} : \mathsf{Tm} \times (\mathsf{Var} \rightharpoonup A) \rightharpoonup MA$ *from terms and valuation to* A *is an* acceptable call-by-value meaning function *in* $\mathcal{A}$ *if*

$$\forall\, \Gamma,\ \Gamma \vdash r : \rho \ \wedge\ \eta \vDash \Gamma \Rightarrow [\![r]\!]^{val}_\eta \in MA^\rho$$

and

$$[\![x]\!]^{val}_\eta = \nu(\eta(x))$$
$$[\![rs]\!]^{val}_\eta = [\![r]\!]^{val}_\eta \star \lambda\!\!\lambda f.[\![s]\!]^{val}_\eta \star \lambda\!\!\lambda a.f\cdot a$$

Terms can be seen as the meaning of themselves because substitution is an acceptable meaning function in syntactical applicative structure. Substitution is a well defined function only for classes of terms modulo α-conversions, but this is not a problem because quotients of syntactic applicative structures by the α-conversion are again applicative structures.

Example 9 (substitution)**.** *In the applicative structure of terms modulo* α*-conversion* $\langle \mathsf{Tm}/_{=_\alpha}, \cdot\rangle$*, the function* $[\![]\!] : \mathsf{Tm} \times (\mathsf{Var} \to \mathsf{Tm}_\perp) \to \mathsf{Tm}/_{=_\alpha}$ *defined by* $[\![r]\!]_\eta = r\{^{\mathsf{FV}(r)}/_{\eta(\mathsf{FV}(r))}\}$ *is an acceptable meaning function.*

We define now a generalisation of logical relation for monadic applicative structures, as a pair of family relations; one for the normal meaning and one for the monadic meaning.

Definition 2.29 (monadic logical relation)**.** *Given monadic applicative structures* $\mathcal{A} = \langle \{A^\rho\}, \{\cdot^{\rho,\sigma}\}\rangle$ *and* $\mathcal{B} = \langle \{B^\rho\}, \{\cdot^{\rho,\sigma}\}\rangle$ *over monads* $M_\mathcal{A}$ *and* $M_\mathcal{B}$*, a* monadic logical relation *is a pair* $\langle \mathcal{R}, \mathcal{S}\rangle$*, where* $\mathcal{R}$ *and* $\mathcal{S}$ *are family of relations* $\mathcal{R}^\tau \subseteq A^\tau \times B^\tau$ *and* $\mathcal{S}^\tau \subseteq MA^\tau \times MB^\tau$ *indexed by a type* τ *such that the following property holds:*

$$f\ \mathcal{R}^{\rho\to\sigma}\ g \Leftrightarrow \forall\, a\ \mathcal{R}^\rho\ b,\ f{\cdot}a\ \mathcal{S}^\sigma\ g{\cdot}b \tag{2.1}$$

$$a\ \mathcal{R}\ b \Rightarrow \nu(a)\ \mathcal{S}\ \nu(b) \tag{unit}$$

$$m\ \mathcal{S}^\rho\ m' \wedge f\ \mathcal{R}^{\rho\to\sigma}\ f' \Rightarrow m \star \mathbb{\lambda} a.f{\cdot}a\ \mathcal{S}^\sigma\ m' \star \mathbb{\lambda} a.f'{\cdot}a \tag{mult}$$

Notation 11. *For two valuations* η *and* δ *on a context* Γ*,* $\eta \vDash \Gamma$ *and* $\delta \vDash \Gamma$*, we will write* $\eta\ \mathcal{R}\ \delta$ *if their values are related for all variables in* Γ*, i.e.,* $\forall\,(x,\rho) \in \Gamma,\ \eta(x)\ \mathcal{R}^\rho\ \delta(x)$*.*

Admissibility is a technical requirement to be able to relate meaning of abstraction terms.

Definition 2.30 (admissibility)**.** *Let* $\mathcal{A}[\![]\!]^{val}$ *in* $\mathcal{A}$ *and* $\mathcal{B}[\![]\!]^{val}$ *in* $\mathcal{B}$ *be call-by-value acceptable meaning functions, a monadic logical relation* $\mathcal{R}$ *is* admissible *for* $\mathcal{A}[\![]\!]^{val}$ *and* $\mathcal{B}[\![]\!]^{val}$ *if given elements* $\overrightarrow{a}$ *and* $\overrightarrow{b}$ *of* $\mathcal{A}$ *and* $\mathcal{B}$ *with* $\overrightarrow{a}\ \mathcal{R}\ \overrightarrow{b}$*, the following property holds:*

$$\forall\, a\ \mathcal{R}^\rho\ b,\ \mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{a},a}\ \mathcal{S}^\sigma\ \mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{b},b} \Rightarrow \mathcal{A}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{a}}\ \mathcal{S}^{\rho\to\sigma}\ \mathcal{B}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{b}} \tag{adm}$$

The basic lemma is the fundamental tool provided by the approach with logical relations. It allows one to relate meaning of terms as soon as the meaning of their free variables are related. The following lemma is its extension to our monadic settings.

Lemma 4 (basic lemma)**.** *Given monadic applicative structure* $\mathcal{A}$ *and* $\mathcal{B}$ *over monads* $M_\mathcal{A}$ *and* $M_\mathcal{B}$*, call-by-value acceptable meaning function* $\mathcal{A}[\![]\!]^{val}$ *in* $\mathcal{A}$ *and* $\mathcal{B}[\![]\!]^{val}$ *in* $\mathcal{B}$*, an admissible monadic logical relation* $\langle \mathcal{R}, \mathcal{S}\rangle$*, and a typed term* $\Gamma \vdash r : \rho$*, with valuation* η *and* δ *on* Γ *(*$\eta \vDash \Gamma$ *and* $\delta \vDash \Gamma$*), the following holds:*

$$\eta\ \mathcal{R}\ \delta \Rightarrow \mathcal{A}[\![r]\!]^{val}_\eta\ \mathcal{S}^\rho\ \mathcal{B}[\![r]\!]^{val}_\delta$$

Proof. By induction on $\Gamma \vdash r : \rho$,

case x**,** by definition of meaning functions $\mathcal{A}[\![x]\!]^{val}_\eta = \nu^\mathcal{A}(\eta(x))$ and $\mathcal{B}[\![x]\!]^{val}_\delta = \nu^\mathcal{B}(\delta(x))$, by hypothesis, $\eta(x)\ \mathcal{R}\ \delta(x)$, and by (unit), $\nu^\mathcal{A}(\eta(x))\ \mathcal{S}\ \nu^\mathcal{B}(\delta(x))$,

case $\lambda x.r$**,** by induction hypothesis, we have:

$$\forall\, \eta, x \mapsto a\ \mathcal{R}\ \delta,\ x \mapsto b, \mathcal{A}[\![r]\!]^{val}_{\eta,x\mapsto a}\ \mathcal{S}\ \mathcal{B}[\![r]\!]^{val}_{\delta,x\mapsto b},$$

and by admissibility (adm) of $\mathcal{R}$, we obtain:

$$\mathcal{A}[\![\lambda x.r]\!]^{val}_\eta\ \mathcal{S}\ \mathcal{B}[\![\lambda x.r]\!]^{val}_\delta$$

case rs**,** given $\eta\ \mathcal{R}\ \delta$, by induction hypothesis on s, we have:

$$\mathcal{A}[\![s]\!]^{val}_{\eta}\ \mathcal{S}^{\rho}\ \mathcal{B}[\![s]\!]^{val}_{\delta},$$

by (mult), given $f\ \mathcal{R}^{\rho\to\sigma}\ f'$, we have:

$$\mathcal{A}[\![s]\!]^{val}_{\eta} \star^{\mathcal{A}} \mathbb{\lambda} a.f{\cdot}a\ \mathcal{S}^{\sigma}\ \mathcal{B}[\![s]\!]^{val}_{\delta} \star^{\mathcal{B}} \mathbb{\lambda} a.f'{\cdot}a$$

hence by definition,

$$\mathbb{\lambda} f.\mathcal{A}[\![s]\!]^{val}_{\eta} \star^{\mathcal{A}} \mathbb{\lambda} a.f{\cdot}a\ \mathcal{R}^{(\rho\to\sigma)\to\sigma}\ \mathbb{\lambda} f'.\mathcal{B}[\![s]\!]^{val}_{\delta} \star^{\mathcal{B}} \mathbb{\lambda} a.f'{\cdot}a,$$

the xinduction hypothesis on r gives:

$$\mathcal{A}[\![r]\!]^{val}_{\eta}\ \mathcal{S}^{\rho\to\sigma}\ \mathcal{B}[\![r]\!]^{val}_{\delta}$$

and by (mult) again,

$$\mathcal{A}[\![r]\!]^{val}_{\eta} \star^{\mathcal{A}} \mathbb{\lambda} f.\mathcal{A}[\![s]\!]^{val}_{\eta} \star^{\mathcal{A}} \mathbb{\lambda} a.f{\cdot}a\ \mathcal{S}^{\sigma}\ \mathcal{B}[\![r]\!]^{val}_{\delta} \star^{\mathcal{B}} \mathbb{\lambda} f'.\mathcal{B}[\![s]\!]^{val}_{\delta} \star^{\mathcal{B}} \mathbb{\lambda} a.f'{\cdot}a$$

which is equivalent by the definition of $[\![rs]\!]^{val}$ to:

$$\mathcal{A}[\![rs]\!]^{val}_{\eta}\ \mathcal{S}^{\sigma}\ \mathcal{B}[\![rs]\!]^{val}_{\delta}$$

□

In the next sections, in the proofs of the main lemmata, i.e., correctness (10) and completeness (19), we will need to extend a context with a new variable whose type will only be known in the induction step.

In particular we would like to be able to see the whole set of terms typable in arbitrary contexts as an applicative structure. In an ordinary applicative structures this is not possible: there is no way to forbid the application between two terms typed in incompatible contexts (i.e., containing a same variable with different types), which could lead to an untypable term.

Therefore we define a weak[2] variant of Kripke applicative structures.

Definition 2.31 (Kripke applicative structure)**.** *A* Kripke applicative structure *is a tuple* $\langle \mathcal{W}, \leqslant, \langle \{A^{\rho}_{w}\}, \{\cdot^{\rho,\sigma}_{w}\}\rangle\rangle$ *where:*

- $\mathcal{W}$ *is a set of "possible worlds" partially ordered by* $\leqslant$*.*
- $\langle \{A^{\rho}_{w}\}, \{\cdot^{\rho,\sigma}_{w}\}\rangle$ *is a family of typed applicative structure indexed by worlds* $w \in \mathcal{W}$

such that for two worlds $w \leqslant w'$ *the following properties hold:*

$$A^{\rho}_{w} \subseteq A^{\rho}_{w'}$$
$$\forall\, f \in A^{\rho\to\sigma}_{w} a \in A^{\rho}_{w},\ \cdot^{\rho,\sigma}_{w} fa = \cdot^{\rho,\sigma}_{w'} fa$$

[2] In a general definition the relation between sets A^{ρ}_{w} and $A^{\rho}_{w'}$ with $w \leqslant w'$ does not need to be an inclusion, but merely the existence of an injective transition functions.

We are now able to encompass typable terms in arbitrary contexts in an applicative structure.

Example 10 (Set of typable terms)**.** *The tuple* $\langle \mathcal{W}, \leqslant, \langle \{A^\rho_w\}, \{\cdot^{\rho,\sigma}_w\}\rangle\rangle$ *where* $\mathcal{W}$ *is the set* Con *of typing contexts* Γ*,* $\leqslant$ *is the inclusion relation between contexts,* A^ρ_Γ *is the set* Tm^ρ_Γ *of well typed terms of type* ρ *in context* Γ*, and* $\cdot^{\rho,\sigma}_\Gamma$ *is syntactic application, is a Kripke applicative structure.*

In the same way that Kripke applicative structures are families of applicative structures indexed by a partial order, we define a Kripke monadic applicative structure to be a family of monadic applicative structures.

Definition 2.32 (Kripke monadic applicative structure)**.** *A* Kripke monadic applicative structure *over a monad* M *is a tuple* $\langle \mathcal{W}, \leqslant, \langle \{A^\rho_w\}, \{\cdot^{\rho,\sigma}_w\}\rangle\rangle$ *where:*

- $\mathcal{W}$ *is a set of "possible worlds" partially ordered by* $\leqslant$*.*
- $\langle \{A^\rho_w\}, \{\cdot^{\rho,\sigma}_w\}\rangle$ *is a family of monadic typed applicative structures indexed by worlds* $w \in \mathcal{W}$ *such that for two worlds* $w \leqslant w'$ *the following properties hold:*

$$A^\rho_w \subseteq A^\rho_{w'}$$
$$\forall\, f \in A^{\rho\to\sigma}_w, a \in A^\rho_w, \cdot^{\rho,\sigma}_w f a = \cdot^{\rho,\sigma}_{w'} f a$$

One can define a notion of Kripke relation enriched over a monad.

Definition 2.33 (Kripke monadic logical relation)**.** *Given Kripke monadic applicative structures*

$$\mathcal{A} = \langle \mathcal{W}, \leqslant, \langle \{A^\rho_w\}, \{\cdot^{\rho,\sigma}_w\}\rangle\rangle$$

and

$$\mathcal{B} = \langle \mathcal{W}, \leqslant, \langle \{B^\rho_w\}, \{\cdot^{\rho,\sigma}_w\}\rangle\rangle$$

over monads $M_\mathcal{A}$ *and* $M_\mathcal{B}$*, a* Kripke monadic logical relation *is a pair* $\langle \mathcal{I}, \mathcal{J}\rangle$*, where* $\mathcal{I}$ *and* $\mathcal{J}$ *are families of relations* $\mathcal{I}^\tau_w \subseteq A^\tau_w \times B^\tau_w$ *and* $\mathcal{J}^\tau_w \subseteq MA^\tau_w \times MB^\tau_w$ *indexed by type* $\tau \in \mathsf{Ty}$ *and world* $w \in W$*, such that the following property holds:*

$$f\ \mathcal{I}^{\rho\to\sigma}_w\ g \Leftrightarrow \forall\, w' \geqslant w,\ \forall\, a\ \mathcal{I}^\rho_{w'}\ b,\ fa\ \mathcal{J}^\sigma_{w'}\ gb \qquad \text{(comp)}$$
$$a\ \mathcal{I}^\rho_w\ b \Rightarrow \forall\, w' \geqslant w,\ a\ \mathcal{I}^\rho_{w'}\ b \qquad \text{(mono)}$$
$$a\ \mathcal{I}_w\ b \Rightarrow \nu(a)\ \mathcal{J}_w\ \nu(b) \qquad \text{(unit)}$$
$$m\ \mathcal{J}^\rho_w\ m' \wedge f\mathcal{I}^{\rho\to\sigma}_w f' \Rightarrow m \star \mathbb{\lambda} a.f\cdot a\ \mathcal{J}^\sigma_w\ m' \star \mathbb{\lambda} a.f'\cdot a \qquad \text{(mult)}$$

We extend the definition of admissibility for call-by-value acceptable meaning functions.

Definition 2.34 (admissibility)**.** *Let $\mathcal{A}[\![]\!]^{val}$ in $\mathcal{A}$ and $\mathcal{B}[\![]\!]^{val}$ in $\mathcal{B}$ be call-by-value acceptable meaning functions, a monadic logical relation $\langle \mathcal{I}, \mathcal{J} \rangle$ is admissible for $\mathcal{A}[\![]\!]^{val}$ and $\mathcal{B}[\![]\!]^{val}$, if given elements $\overrightarrow{a}$ and $\overrightarrow{b}$ of $\mathcal{A}$ and $\mathcal{B}$ with $\overrightarrow{a}\ \mathcal{I}_w\ \overrightarrow{b}$, the following property holds:*

$$\forall\, w' \geqslant w, \forall\, a\ \mathcal{I}^{\rho}_{w'}\ b,\ \mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x}, x \mapsto \overrightarrow{a}, a}\ \mathcal{J}^{\sigma}_{w'}\ \mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x}, x \mapsto \overrightarrow{b}, b} \Rightarrow \mathcal{A}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x} \mapsto \overrightarrow{a}}\ \mathcal{J}^{\rho \to \sigma}_{w}\ \mathcal{B}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x} \mapsto \overrightarrow{b}} \quad \text{(adm)}$$

The basic lemma valid for Kripke logical monadic relation reads as follows.

Lemma 5 (basic lemma for Kripke monadic logical relation)**.** *Given Kripke monadic applicative structures $\mathcal{A}$ and $\mathcal{B}$ over $M_{\mathcal{A}}$ and $M_{\mathcal{B}}$, call-by-value acceptable meaning function $\mathcal{A}[\![]\!]^{val}$ in $\mathcal{A}$ and $\mathcal{B}[\![]\!]^{val}$ in $\mathcal{B}$, a Kripke admissible monadic logical relation $\langle \mathcal{I}, \mathcal{J} \rangle$, and a typed term $\Gamma \vdash r : \rho$, with valuation η and δ on Γ ($\eta \vDash \Gamma$ and $\delta \vDash \Gamma$), the following property holds, for an arbitrary $w \in \mathcal{W}$:*

$$\eta\ \mathcal{I}_w\ \delta \Rightarrow \forall\, w' \geqslant w,\ \mathcal{A}[\![r]\!]^{val}_{\eta}\ \mathcal{J}_{w'}\ \mathcal{B}[\![r]\!]^{val}_{\delta}$$

Proof. By induction on $r \in \mathsf{Tm}_{\rho}$,

case x**,** by hypothesis, $\eta(x)\ \mathcal{I}_w\ \delta(x)$, by monotonicity this holds for $w' \geqslant w$, $\eta(x)\ \mathcal{I}_{w'}\ \delta(x)$, and by (unit) we have:

$$\forall\, w' \geqslant w, \mathcal{A}[\![x]\!]^{val}_{\eta}\ \mathcal{J}_{w'}\ \mathcal{B}[\![x]\!]^{val}_{\delta},$$

case rs**,** by induction hypothesis on s, $\forall\, w' \geqslant w$, $\mathcal{A}[\![s]\!]^{val}_{\eta}\ \mathcal{J}^{\rho}_{w'}\ \mathcal{B}[\![s]\!]^{val}_{\delta}$, by (mult), given $w' \geqslant w$ and $f\ \mathcal{I}^{\rho \to \sigma}_{w'}\ f'$ we have

$$\mathcal{A}[\![s]\!]^{val}_{\eta} \star \mathbb{\lambda} a.f \cdot a\ \mathcal{J}^{\sigma}_{w'}\ \mathcal{B}[\![s]\!]^{val}_{\delta} \star \mathbb{\lambda} a.f' \cdot a, \tag{2.2}$$

and hence by the comprehension property (comp):

$$\mathbb{\lambda} f.\mathcal{A}[\![s]\!]^{val}_{\eta} \star \mathbb{\lambda} a.f \cdot a\ \mathcal{I}^{(\rho \to \sigma) \to \sigma}_{w}\ \mathbb{\lambda} f'.\mathcal{B}[\![s]\!]^{val}_{\delta} \star \mathbb{\lambda} a.f' \cdot a,$$

hence by monotonicity (mono), this holds for $w' \geqslant w$,

$$\mathbb{\lambda} f.\mathcal{A}[\![s]\!]^{val}_{\eta} \star \mathbb{\lambda} a.f \cdot a\ \mathcal{I}^{(\rho \to \sigma) \to \sigma}_{w'}\ \mathbb{\lambda} f'.\mathcal{B}[\![s]\!]^{val}_{\delta} \star \mathbb{\lambda} a.f' \cdot a, \tag{$*$}$$

by IH on r, $\forall\, w' \geqslant w$, $\mathcal{A}[\![r]\!]^{val}_{\eta}\ \mathcal{J}^{\rho \to \sigma}_{w'}\ \mathcal{B}[\![r]\!]^{val}_{\delta}$, by the multiplication property (mult), given $w' \geqslant w$ and $g\ \mathcal{I}^{(\rho \to \sigma) \to \sigma}_{w}\ g'$, we have

$$\mathcal{A}[\![r]\!]^{val}_{\eta} \star \mathbb{\lambda} f.g \cdot f\ \mathcal{J}^{\sigma}_{w'}\ \mathcal{B}[\![r]\!]^{val}_{\delta} \star \mathbb{\lambda} f.g' \cdot f, \tag{$**$}$$

taking g and g' as given in $(*)$ we obtain:

$$\mathcal{A}[\![r]\!]^{val}_{\eta} \star \mathbb{\lambda} f.\mathcal{A}[\![s]\!]^{val}_{\eta} \star \mathbb{\lambda} a.f{\cdot}a \ \mathcal{J}^{\sigma}_{w'} \ \mathcal{B}[\![r]\!]^{val}_{\delta} \star \mathbb{\lambda} f.\mathcal{B}[\![s]\!]^{val}_{\delta} \star \mathbb{\lambda} a f{\cdot}a,$$

which by definition of acceptable meaning functions is equivalent to:

$$\mathcal{A}[\![rs]\!]^{val}_{\eta} \ \mathcal{J}^{\sigma}_{w'} \ \mathcal{B}[\![rs]\!]^{val}_{\delta},$$

case $\lambda x.r$**,** let us assume $w'' \geqslant w' \geqslant w$ and $a \ \mathcal{I}_{w''} \ b$, by hypothesis $\eta \ \mathcal{I}_{w} \ \delta$, and by monotonicity (mono) we have also, $\eta \ \mathcal{I}_{w''} \ \delta$, by induction hypothesis on r, we have:

$$\mathcal{A}[\![r]\!]^{val}_{\eta,x\mapsto a} \ \mathcal{J}_{w''} \ \mathcal{B}[\![r]\!]^{val}_{\delta,x\mapsto b},$$

by admissibility (adm),

$$\mathcal{A}[\![\lambda x.r]\!]^{val}_{\eta} \ \mathcal{J}_{w'} \ \mathcal{B}[\![\lambda x.r]\!]^{val}_{\delta}.$$

□

As in the non-monadic case, the basic lemma holds automatically between interpretations (as opposed to acceptable meaning functions) related by a logical relation, because the admissibility property is immediately verified.

Lemma 6 (admissibility of interpretation)**.** *Let $\mathcal{A}[\![]\!]^{val}$ in $\mathcal{A}$ and $\mathcal{B}[\![]\!]^{val}$ in $\mathcal{B}$ be call-by-value interpretations, a monadic logical relation $\langle \mathcal{I}, \mathcal{J} \rangle$ between $\mathcal{A}[\![]\!]^{val}$ and $\mathcal{B}[\![]\!]^{val}$ is admissible.*

Proof. Assume $\mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{a},a} \ \mathcal{J}^{\sigma}_{w'} \ \mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{b},b}$ for all $a \ \mathcal{I}^{\rho}_{w'} \ b$ and $w' \geqslant w$, we have to show:

$$\mathcal{A}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{a}} \ \mathcal{J}^{\rho\to\sigma}_{w} \ \mathcal{B}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{b}}.$$

The hypothesis $\mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{a},a} \ \mathcal{J}^{\sigma}_{w} \ \mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{b},b}$ valid for all $a \ \mathcal{I}^{\rho}_{w} \ b$ and $w' \geqslant w$, implies that by comprehension (comp),

$$\mathbb{\lambda} a.\mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{a},a} \ \mathcal{I}^{\rho\to\sigma}_{w} \ \mathbb{\lambda} b.\mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{b},b},$$

by the unit property (unit), we have then:

$$\nu(\mathbb{\lambda} a.\mathcal{A}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{a},a}) \ \mathcal{J}^{\sigma}_{w} \ \nu(\mathbb{\lambda} b.\mathcal{B}[\![r]\!]^{val}_{\overrightarrow{x},x\mapsto\overrightarrow{b},b}),$$

which by definition of $[\![\lambda x.r]\!]^{val}$ is equivalent to:

$$\mathcal{A}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{a}} \ \mathcal{J}^{\rho\to\sigma}_{w} \ \mathcal{B}[\![\lambda x.r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{b}}.$$

□

2.3.3 Correctness

In order to prove the correctness we need to modify the logical relation presented in the informal exposition to take into account the reader monad $\mathbf{St}_E$ used to generate fresh names for variables. The following relation between the term and the call-by-value type interpretation applicative structures, will turn out to be a Kripke monadic logical relation between substitution and term interpretation.

Definition 2.35 (Correctness relation $\mathcal{R}$)**.** *We define a family of relation* $\mathcal{R}^\rho_\Gamma \subseteq \mathsf{Tm}^\rho_\Gamma/_{=_\alpha} \times [\![\rho]\!]^{val}$ *indexed by type* ρ *and context* Γ *by induction on* $\rho \in \mathsf{Ty}$ *as follows:*

$$r\ \mathcal{R}^o_\Gamma\ a ::= \Gamma \vdash r =_{\beta\eta} a : o$$
$$r\ \mathcal{R}^{\rho\to\sigma}_\Gamma\ a ::= \forall\, \Gamma' \geqslant \Gamma, \forall\, s\ \mathcal{R}^\rho_{\Gamma'}\ b,\ rs\ \mathcal{S}^\sigma_{\Gamma'}\ ab$$

where $\mathcal{S}^\sigma_\Gamma$ *is defined by*

$$r\ \mathcal{S}^\sigma_\Gamma\ m ::= \forall\, e, \mathsf{FV}(r) \notin e,\ r\ \mathcal{R}^\sigma_\Gamma\ me$$

Remark 8. *Notice that in the clause* $\Gamma \vdash r =_{\beta\eta} a : o$*, the* $\beta\eta$*-conversion has been implicitly extended between classes of terms modulo* α*-conversion and genuine syntactical terms. This can be made explicit as follows: in case* r *is a class of terms modulo* α*-conversion and* a *is a term,* $r =_{\beta\eta} a$ *holds if and only if the class* r *is* $\beta\eta$*-convertible to a class* r' *for which* a *is a representative.*

Lemma 7. $\langle \mathcal{R}, \mathcal{S} \rangle$ *is a Kripke monadic logical relation between the Kripke applicative structure* $\langle \mathsf{Con}, \leqslant, \langle \{\mathsf{Tm}^\rho_\Gamma/_{=_\alpha}\}, \{\cdot^{\rho,\sigma}_\Gamma\}\rangle\rangle$ *whose set of worlds are contexts and application is just syntactic application and the typed applicative structure* $\langle [\![\rho]\!]^{val}, \cdot^{\rho,\sigma}\rangle$ *whose application is set-theoretic functional application.*

Proof. The comprehension property (comp) is verified by definition. The monotonicity property (mono) can be shown to hold by an easy induction on the type ρ of $\mathcal{R}^\rho$. The unit property (unit) obviously holds. We show the multiplication property (mult), i.e.,:

$$r\ \mathcal{R}^{\rho\to\sigma}_\Gamma\ f \wedge s\ \mathcal{S}^\rho_\Gamma\ m \Rightarrow rs\ \mathcal{S}^\sigma_\Gamma\ m \star f.$$

On the one hand by unfolding the definition of $r\mathcal{R}^{\rho\to\sigma}_\Gamma f$, we obtain, given s, a and e such that $\mathsf{FV}(rs) \notin e$:

$$\forall\, \Gamma' \geqslant \Gamma,\ s\ \mathcal{R}^\rho_{\Gamma'}\ a \Rightarrow rs\ \mathcal{R}^\sigma_{\Gamma'}\ fae, \qquad (\star)$$

on the other hand, by unfolding the definition of $s\ \mathcal{S}^\rho_\Gamma\ m$, we obtain, given an environment e' such that $\mathsf{FV}(s) \notin e'$, that

$$s\ \mathcal{R}^\rho_\Gamma\ me',$$

in particular for an e such that $\mathsf{FV}(rs) \notin e$, $\mathsf{FV}(s) \notin e$ holds, and hence we have

$$s\ \mathcal{R}^{\rho}_{\Gamma}\ me,$$

by using $(\star)$, with $\Gamma' := \Gamma$ and $a := me$, we obtain:

$$\forall\, e, \mathsf{FV}(rs) \notin e, rs\ \mathcal{R}^{\sigma}_{\Gamma}\ f(me)e.$$

which is the definition of:

$$rs\ \mathcal{S}^{\sigma}_{\Gamma}\ m \star f.$$

□

By its definition, the call-by-value interpretation verifies the property to be an acceptable meaning function, moreover from example (9), we have seen that substitution is an acceptable meaning function.

However substitution is not an interpretation, so that lemma (6), only valid when the logical relation relates two interpretations, does not apply here and we have to show that the relation $\mathcal{R}$ is an admissible logical relation.

We need first the following technical lemma:

Lemma 8. *Given a typed term $\Gamma \vdash s : \sigma$ the following propositions hold:*

$$r\{^{s}/_{x}\}\overrightarrow{s}\ \mathcal{S}^{\tau}_{\Gamma}\ m \Rightarrow (\lambda x^{\sigma}.r)s\overrightarrow{s}\ \mathcal{S}^{\tau}_{\Gamma}\ m \tag{2.3}$$

$$r\{^{s}/_{x}\}\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ a \Rightarrow (\lambda x^{\sigma}.r)s\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ a \tag{2.4}$$

Proof. We prove proposition (2.3) and (2.4) simultaneously by induction on $\mathcal{S}_{\Gamma}$ and $\mathcal{R}_{\Gamma}$,

proposition (2.3)**,** by hypothesis, given e such that $\mathsf{FV}(r\{^{s}/_{x}\}\overrightarrow{s}) \notin e$, we have:

$$r\{^{s}/_{x}\}\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ me,$$

hence by I.H. on $\mathcal{R}$ for (2.4), $(\lambda xr)s\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ me$, and we have proved

$$\forall\, e, \mathsf{FV}(r\{^{s}/_{x}\}\overrightarrow{s}) \notin e \Rightarrow (\lambda xr)s\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ me,$$

but $\mathsf{FV}(r\{^{s}/_{x}\}\overrightarrow{s}) \subseteq \mathsf{FV}((\lambda x.r_x)s\overrightarrow{s})$, hence

$$\mathsf{FV}((\lambda x.r)s\overrightarrow{s}) \notin e \Rightarrow \mathsf{FV}(r\{^{s}/_{x}\}\overrightarrow{s}) \notin e$$

and finally

$$\forall\, e, \mathsf{FV}((\lambda x.r)s\overrightarrow{s}) \notin e \Rightarrow (\lambda xr)s\overrightarrow{s}\ \mathcal{R}^{\tau}_{\Gamma}\ me$$

which is equivalent to

$$(\lambda xr)s\overrightarrow{s}\ \mathcal{S}^{\tau}_{\Gamma}\ m$$

proposition (2.4)**,**

case o**,** clearly $r\{^{s}/_{x}\}\overrightarrow{s} =_{\beta\eta} (\lambda x.r)s\overrightarrow{s}$, and we have to check that Γ is a valid typing context for $(\lambda xr)s\overrightarrow{s}$ but this follows easily because as s is typable in Γ', the free variables of s (possibly not occurring in $r\{^{s}/_{x}\}\overrightarrow{s}$) are in Γ,

case $\rho \to \sigma$, by hypothesis for $\Gamma' \geqslant \Gamma$ and $t\ \mathcal{R}^{\rho}_{\Gamma'}\ b$ we have:

$$r\{^{s}/_{x}\}\overrightarrow{s}t\ \mathcal{S}^{\sigma}_{\Gamma'}\ ab,$$

and by induction hypothesis on σ, we have

$$(\lambda xr)s\overrightarrow{s}t\ \mathcal{S}^{\sigma}_{\Gamma'}\ ab.$$

□

Corollary 1 (admissibility)**.** *Given a typed term r of type σ with free variables $\overrightarrow{x} : \overrightarrow{\rho}, x : \rho$, and terms s of type $\overrightarrow{\rho}$ and values $\overrightarrow{a} \in \overrightarrow{[\![\rho]\!]}$ such that $\overrightarrow{s}\ \mathcal{R}_{\Gamma}\ \overrightarrow{a}$, then:*

$$\forall\, s\ \mathcal{R}^{\rho}_{\Gamma}\ a,\ r\{^{\overrightarrow{s},s}/_{\overrightarrow{x},x}\}\ \mathcal{S}^{\sigma}_{\Gamma}\ [\![r]\!]_{\overrightarrow{x},x\mapsto\overrightarrow{a},a} \Rightarrow \lambda x.r\{^{\overrightarrow{s}}/_{\overrightarrow{x}}\}\ \mathcal{S}^{\rho\to\sigma}_{\Gamma}\ [\![\lambda x.r]\!]_{\overrightarrow{x}\mapsto\overrightarrow{a}}$$

Proof. Given a term s of type ρ and a value $a \in [\![\rho]\!]$, such that $s\ \mathcal{R}^{\rho}_{\Gamma}\ a$ and $r\{^{\overrightarrow{s},s}/_{\overrightarrow{x},x}\}\ \mathcal{S}^{\sigma}_{\Gamma}\ [\![r]\!]_{\overrightarrow{x},x\mapsto\overrightarrow{a},a}$, and an environment e with $\mathsf{FV}(\lambda x.r\{^{\overrightarrow{s}}/_{\overrightarrow{x}}\}) \notin e$, we have to show that:

$$\lambda x.r\{^{\overrightarrow{s}}/_{\overrightarrow{x}}\}\ \mathcal{R}^{\rho\to\sigma}_{\Gamma}\ [\![\lambda x.r]\!]_{\overrightarrow{x}\mapsto\overrightarrow{a}}e,$$

i.e.,, by unfolding the definition of $\mathcal{R}^{\rho\to\sigma}_{\Gamma}$ and $[\![\lambda x.r]\!]$ we have to show:

$$\forall\,\Gamma' \geqslant \Gamma, \forall\, s\ \mathcal{R}^{\rho}_{\Gamma'}\ a, (\lambda x.r\{^{\overrightarrow{s}}/_{\overrightarrow{x}}\})s\ \mathcal{S}^{\sigma}_{\Gamma'}\ [\![r]\!]_{\overrightarrow{x},x\mapsto\overrightarrow{a},a}.$$

But this last proposition is obtained by induction on $\overrightarrow{s}$ using the previous lemma (8) from the hypothesis $r\{^{\overrightarrow{s},s}/_{\overrightarrow{x},x}\}\ \mathcal{S}^{\sigma}_{\Gamma'}\ [\![r]\!]_{\overrightarrow{x},x\mapsto\overrightarrow{a},a}$ as s is typable in Γ follows from $s\ \mathcal{R}^{\rho}_{\Gamma'}\ a$. □

Lemma 9 (Basic lemma)**.** *Given a typed term $\{x_1 : \rho_1, \ldots, x_n : \rho_n\} \vdash r : \rho$, typed terms $\overrightarrow{s} = s_1, \ldots, s_n$ with $\Gamma \vdash s_i : \rho_i$, and values $\overrightarrow{a} = a_1, \ldots, a_n$ with $a_i \in [\![\rho_i]\!]$, the following proposition hold:*

$$\overrightarrow{s}\ \mathcal{R}_{\Gamma}\ \overrightarrow{a} \Rightarrow r\{^{\overrightarrow{s}}/_{\overrightarrow{x}}\}\ \mathcal{S}_{\Gamma}\ [\![r]\!]^{val}_{\overrightarrow{x}\mapsto\overrightarrow{a}}$$

Proof. As $\mathcal{R}$ is an admissible monadic logical relation, we can apply lemma (13) □

After this preliminary work, we arrive at the core of the proof with the following lemma:

Lemma 10 (correctness)**.** *For terms r, s, value a, and an environment e such that $\mathsf{FV}(r) \notin e$, the following properties hold:*

$$\Gamma \vdash r =_{\beta\eta} s : \rho \Rightarrow r\ \mathcal{R}^{\rho}_{\Gamma}\ \uparrow^{\rho} s \tag{1}$$

$$r\ \mathcal{R}^{\rho}_{\Gamma}\ a \Rightarrow \Gamma \vdash r =_{\beta\eta} (\downarrow_{\rho} a)e : \rho \tag{2}$$

Proof. We show (1) and (2) by simultaneous induction on the type $\rho \in \mathsf{Ty}$ of r.

case o**,** by definition,

case $\rho \to \sigma$**,**

(1)**,** given $\Gamma \vdash r = s : \rho \to \sigma$, we have to show:

$$\forall\, \Gamma' \geqslant \Gamma,\ \forall\, t\ \mathcal{R}^{\rho}_{\Gamma'}\ a, rt\ \mathcal{S}^{\sigma}_{\Gamma'}\ (\uparrow^{\rho\to\sigma} s)(a),$$

that is, given $t\ \mathcal{R}^{\rho}_{\Gamma'}\ a$ and e such that $\mathsf{FV}(rt) \notin e$,

$$rt\ \mathcal{R}^{\sigma}_{\Gamma'}\ (\uparrow^{\rho\to\sigma} s)(a)e.$$

By induction hypothesis (2) on ρ, given an e such that $\mathsf{FV}(t) \notin e$, we have $t =_{\beta\eta} (\downarrow a)e$, and hence $rt =_{\beta\eta} s((\downarrow a)e)$.
By induction hypothesis (1) on σ, given an e such that $\mathsf{FV}(rt) \notin e$ we have, $rt\ \mathcal{R}^{\sigma}_{\Gamma}\ \uparrow s((\downarrow a)e)$.
But $\uparrow s((\downarrow a)e)$ is equal to

$$((\downarrow_{\rho} a) \star \mathbb{\lambda} t.\eta(\uparrow^{\sigma} st))e,$$

and by definition of $\uparrow$ at arrow type, this last term is equal to:

$$(\uparrow^{\rho\to\sigma} s)(a)e,$$

and $\mathsf{FV}(rt) \notin e$ implies $\mathsf{FV}(t) \notin e$.

(2) We have to show for e with $\mathsf{FV}(r) \notin e$:

$$r\ \mathcal{R}^{\rho}_{\Gamma}\ f \Rightarrow r =_{\beta\eta\downarrow} (f)e.$$

By definition of $\mathcal{R}$ at arrow type:

$$r\ \mathcal{R}^{\rho\to\sigma}\ f \equiv \forall\, \Gamma' \geqslant \Gamma,\ \forall\, s\ \mathcal{R}^{\rho}_{\Gamma'}\ a, rs\ \mathcal{S}^{\sigma}_{\Gamma'}\ fa.$$

Let us take $x = \mathsf{new}(e)$, by induction hypothesis (1) on ρ, $x\ \mathcal{R}^{\rho}_{\Gamma,x:\rho}\ \uparrow x$, and hence

$$rx\ \mathcal{S}^{\sigma}_{\Gamma,x:\rho}\ f \uparrow x.$$

By definition, $\forall e', \mathsf{FV}(rx) \notin e' \Rightarrow rx\ \mathcal{R}^{\sigma}_{\Gamma,x:\rho}\ f(\uparrow x)e'$, and this holds in particular for $e' = e^x$.
By induction hypothesis (2) on σ, we have $rx =_{\beta\eta} (\downarrow f(\uparrow x)e^x)e^x$. So we have:

$$\lambda x.rx =_{\beta\eta} \lambda x.(\downarrow f(\uparrow x)e^x)e^x = \lambda x.(f \uparrow x\star\downarrow)e^x.$$

This last term typable in Γ is equal to

$$\mathsf{new} \star \mathbb{\lambda} v.(f \uparrow v\star\downarrow)^v \star \mathbb{\lambda} t.\nu(\lambda v.t)e =\downarrow (f)e.$$

□

Lemma 11. *For a term r, and an environment e such that $\mathsf{FV}(r) \notin e$*

$$r =_{\beta\eta} ([\![r]\!]_{\uparrow}\star\downarrow)e$$

Proof. By part (1) of lemma (10) we have

$$x\ \mathcal{R}\ \uparrow x.$$

Because $\mathcal{R}$ is a logical relation, we have by the fundamental lemma:

$$r\ \mathcal{S}\ [\![r]\!]_{\uparrow}.$$

By definition, $\forall e, \mathsf{FV}(r) \notin e, r\ \mathcal{R}\ [\![r]\!]_{\uparrow}e$. Now by the second part of the previous lemma given a e' such that $\mathsf{FV}(r) \notin e'$ we have:

$$r = (\downarrow [\![r]\!]_{\uparrow}e)e',$$

and so it holds in particular for $e = e'$, i.e.,:

$$r =_{\beta\eta} (\downarrow ([\![r]\!]_{\uparrow}e))e = ([\![r]\!]_{\uparrow}\star\downarrow)e.$$

□

2.3.4 Completeness

In this section we prove the completeness of the *NbE* algorithm, that normal forms of terms $\beta\eta$-convertible are themselves α-convertible, that is:

$$\Gamma \vdash r =_{\beta\eta} s : \rho \Rightarrow \forall e \not\ni \Gamma,\ \Gamma \vdash ([\![r]\!]_{\uparrow}\star\downarrow)e =_{\alpha} ([\![s]\!]_{\uparrow}\star\downarrow)e : \rho.$$

The idea is to define a relation $\mathcal{J}_{\Gamma}$ such that the two following conditions are provable

$$a\ \mathcal{J}^{\rho}_{\Gamma}\ b \Rightarrow \forall e \not\ni \Gamma,\ \Gamma \vdash (a\star\downarrow)e =_{\alpha} (b\star\downarrow)e : \rho \quad (1)$$

$$\Gamma \vdash r =_{\beta\eta} s : \rho \Rightarrow [\![r]\!]_{\uparrow}\ \mathcal{J}^{\rho}_{\Gamma}\ [\![s]\!]_{\uparrow} \quad (2)$$

In fact, to prove the first condition above involving α-conversion, we will use induction on the structure of terms. A problem here is that α-conversion is about bound variables only, if it says that the terms $\lambda x.x$ and $\lambda y.y$ are equivalent, it says nothing about their immediate subterms x and y (see footnote 3, in the proof of theorem (19), p. 56). As a consequence the needed induction principle will have necessarily to be a generalisation of α-conversion, it should enable us to compare terms up to renaming of their free variables. Although, we would never argue that "α-conversion is easy", the approach here is related to the paper of Thorsten Altenkirch [3].

Definition 2.36 (Variables renaming, context renaming)**.** *A renaming of variables is a substitution where the terms substituted for are variables. Given lists of variables* $\overrightarrow{x} = x_1, \ldots, x_n$, *and* $\overrightarrow{y} = y_1, \ldots, y_n$, *the effect of a renaming* $\{\overrightarrow{y}/\overrightarrow{x}\}$ *on a context* Γ *is defined componentwise:*

$$\emptyset\{\overrightarrow{y}/\overrightarrow{x}\} ::= \emptyset$$

$$(\{x:\rho\} \cup \Gamma)\{\overrightarrow{y}/\overrightarrow{x}\} ::= \begin{cases} \{y_i:\rho\} \cup \Gamma\{\overrightarrow{y}/\overrightarrow{x}\} & \text{if } x = x_i \text{ for } x_i \in \overrightarrow{x} \\ \{x:\rho\} \cup \Gamma\{\overrightarrow{y}/\overrightarrow{x}\} & \text{otherwise} \end{cases}$$

We introduce now a notation to compare term up to renaming of their free variables, this is a generalisation of α-conversion to renaming.

Notation 12. *The renaming* $\{\overrightarrow{y}/\overrightarrow{x}\}$ *from a context* $\Gamma = \{x_1 : \rho_1, \ldots, x_n : \rho_n\}$ *into the context* $\Gamma' = \{y_1 : \rho_1, \ldots, y_n : \rho_n\}$, *will be simply written* $\Gamma; \Gamma'$ *although formally* Γ *and* Γ' *are unordered sets of pairs.*

We will write $\Gamma; \Gamma' \vdash r =_\alpha r' : \rho$ *where* $\Gamma; \Gamma'$ *is the renaming given above if* $\Gamma \vdash r : \rho$ *and* $\Gamma' \vdash r : \rho$ *if* $\Gamma'\{\overrightarrow{y}/\overrightarrow{x}\} = \Gamma$ *and* $r = r'\{\overrightarrow{y}/\overrightarrow{x}\}$

With this notation the α-conversion between well typed terms r and r' of type ρ in a context Γ can be expressed as $\Gamma; \Gamma \vdash r =_\alpha r' : \rho$ (i.e., the renaming is the identity).

The generalisation of condition (1) for which we can use induction on the structure of terms reads:

$$a \; \mathcal{J}^{\rho}_{\Gamma,\Gamma'} \; b \Rightarrow \forall\, (e, e') \not\ni (\Gamma, \Gamma'),\ \Gamma; \Gamma' \vdash (a\star\downarrow)e =_\alpha ((b\star\downarrow)e') : \rho \qquad (1')$$

.

The inclusion relation on contexts induces a relation on context renaming:

Definition 2.37 (order on contexts renamings)**.** *The partial order* $\leqslant$ *on context renaming is the least symmetric transitive relation such that:*

$$\Gamma; \Gamma' \leqslant \Gamma, (x:\rho); \Gamma', (y:\rho)$$

We define below a pair of relations $\mathcal{I}$ and $\mathcal{J}$, relating values of the interpretations, $\mathcal{I}$ for normal values and $\mathcal{J}$ for monadic values. We will show that these relations form a logical relation and are partial equivalence relations (i.e., symmetric and transitive).

Definition 2.38 (completeness relation)**.** *We define a family of logical relations* $\mathcal{I}^{\rho}_{\Gamma,\Gamma'} \subseteq [\![\rho]\!] \times [\![\rho]\!]$ *and* $\mathcal{J}^{\rho}_{\Gamma,\Gamma'} \subseteq \mathbf{St}_E[\![\rho]\!] \times \mathbf{St}_E[\![\rho]\!]$ *indexed by a type* ρ *and a context* $\Gamma \in \mathsf{Con}$ *inductively by:*

$$
\begin{aligned}
r\ \mathcal{I}^{o}_{\Gamma,\Gamma'}\ s &::= \Gamma;\Gamma' \vdash r =_{\alpha} r' : o \\
f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ g &::= \forall\, \Delta;\Delta' \geqslant \Gamma;\Gamma',\ \forall a\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ b,\ fa\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ fb \\
m\ \mathcal{J}^{\rho}_{\Gamma,\Gamma'}\ m' &::= \forall\, e, e', (\Gamma,\Gamma') \notin (e,e'),\ me\ \mathcal{I}^{\rho}_{\Gamma,\Gamma'}\ m'e'
\end{aligned}
$$

Lemma 12. *The pair of relations* $\langle \mathcal{I}, \mathcal{J} \rangle$ *forms a Kripke monadic logical relation on (two copies of) the typed applicative structure* $\langle [\![\rho]\!], \cdot \rangle$ *where* $\cdot$ *is syntactic application.*

Proof. The comprehension property (comp) is verified by definition. The monotonicity properties (mono) and the unit property (unit) obviously hold.

We show the multiplication property (mult), i.e.,:

$$f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ f' \wedge m\ \mathcal{J}^{\rho}_{\Gamma,\Gamma'}\ m' \Rightarrow m \star f\ \mathcal{J}^{\sigma}_{\Gamma,\Gamma'}\ m' \star f'$$

On the one hand by definition of $f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ f'$, we obtain in particular for $a\ \mathcal{I}^{\rho}_{\Gamma,\Gamma'}\ a'$:

$$fa\ \mathcal{J}^{\sigma}_{\Gamma,\Gamma'}\ f'a'.$$

which implies by the definition of $\mathcal{J}$ for e, e' such that $\Gamma, \Gamma' \notin e, e'$:

$$fae\ \mathcal{I}^{\sigma}_{\Gamma,\Gamma'}\ f'a'e'.$$

On the other hand, by unfolding the definition of $m\ \mathcal{J}^{\rho}_{\Gamma,\Gamma'}\ m'$, for e, e' and $\Gamma;\Gamma'$ as above, we obtain:

$$me\ \mathcal{I}^{\rho}_{\Gamma,\Gamma'}\ m'e'.$$

And hence we have

$$f(me)e\ \mathcal{I}^{\rho}_{\Gamma,\Gamma'}\ f'(m'e')e'.$$

which is the definition of:

$$(m \star f)\ \mathcal{J}^{\rho}_{\Gamma,\Gamma'}\ (m' \star f').$$

□

Lemma 13 (basic lemma)**.**

$$\mu\ \mathcal{I}_{\Gamma,\Gamma'}\ \nu \Rightarrow [\![r]\!]_\mu\ \mathcal{J}_{\Gamma,\Gamma'}\ [\![r]\!]_\nu$$

Proof. $(\mathcal{I}, \mathcal{J})$ forms a logical relation as shown in lemma (12), and the admissibility property holds immediately for interpretation as shown in lemma (6). □

Lemma 14 (symmetry)**.** *$\mathcal{I}$ is a symmetric relation:*

$$\forall\, ab,\ a\ \mathcal{I}_{\Gamma,\Gamma'}\ b \Rightarrow b\ \mathcal{I}_{\Gamma,\Gamma'}\ a$$

Proof. Remark first that symmetry of $\mathcal{I}^\rho_{\Gamma,\Gamma'}$ implies symmetry of $\mathcal{J}^\rho_{\Gamma,\Gamma'}$:

$$\forall\, m\, m',\ m\ \mathcal{J}_{\Gamma,\Gamma'}\ m' \Rightarrow m'\ \mathcal{J}_{\Gamma,\Gamma'}\ m$$

The proof is by induction on the type $\rho \in \mathsf{Ty}$:

case $\rho = o$**,** follows from symmetry of $=_\alpha$,

case $\rho \to \sigma$**,** we want to show:

$$f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ g \Rightarrow g\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ f$$

By definition,

$$f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ g \Leftrightarrow \forall\, \Delta, \Delta' \geqslant \Gamma, \Gamma',\ \forall\, a\ \mathcal{I}^\rho_{\Delta,\Delta'}\ b,\ fa\mathcal{J}^\sigma_{\Delta,\Delta'}\ gb$$
$$g\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ f \Leftrightarrow \forall\, \Delta, \Delta' \geqslant \Gamma, \Gamma',\ \forall\, b\ \mathcal{I}^\rho_{\Delta,\Delta'}\ a,\ gb\mathcal{J}^\sigma_{\Delta,\Delta'}\ fa.$$

By induction hypothesis on ρ we have:

$$b\ \mathcal{I}^\rho_{\Delta,\Delta'}\ a \Rightarrow a\ \mathcal{I}^\rho_{\Delta,\Delta'}\ b$$

by hypothesis this implies $fa\ \mathcal{J}^\sigma_{\Delta,\Delta'}\ gb$ and by induction hypothesis on σ, we have

$$fa\ \mathcal{J}^\sigma_{\Delta,\Delta'}\ gb \Rightarrow gb\ \mathcal{J}^\sigma_{\Delta,\Delta'}\ fa.$$

□

Lemma 15 (transitivity)**.** *$\mathcal{I}$ is a transitive relation:*

$$\forall\, a\, b\, c,\ a\ \mathcal{I}_{\Gamma,\Gamma'}\ b \wedge b\ \mathcal{I}_{\Gamma,\Gamma'}\ c \Rightarrow a\ \mathcal{I}_{\Gamma,\Gamma'}\ c$$

Proof. Remark first that transitivity of $\mathcal{I}^\rho_{\Gamma,\Gamma'}$ implies transitivity of $\mathcal{J}^\rho_{\Gamma,\Gamma'}$:

$$\forall\, m\, m'\, m'',\ m\ \mathcal{J}_{\Gamma,\Gamma'}\ m' \wedge m'\ \mathcal{J}_{\Gamma,\Gamma'}\ m'' \Rightarrow m\ \mathcal{J}_{\Gamma,\Gamma'}\ m''$$

The proof is by induction on the type $\rho \in \mathsf{Ty}$:

case $\rho = o$**,** follows from transitivity of $=_\alpha$,

case $\rho \to \sigma$**,** we want to show

$$f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ g \wedge g\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ h \Rightarrow f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ h$$

By definition,

$$f\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ g \Leftrightarrow \forall\, \Delta, \Delta' \geqslant \Gamma, \Gamma',\ \forall\, a\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ b,\ fa\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ gb, \text{ and}$$
$$g\ \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ h \Leftrightarrow \forall\, \Delta, \Delta' \geqslant \Gamma, \Gamma',\ \forall\, a\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ b,\ ga\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ hb$$

By symmetry, $a\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ b$, implies $b\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ a$, by induction hypothesis on ρ, $a\ \mathcal{I}^{\rho}_{\Delta,\Delta'}\ a$, and hence $fa\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ ga$ and $ga\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ hb$, and by induction hypothesis on σ, $fa\ \mathcal{J}^{\sigma}_{\Delta,\Delta'}\ hb$. This concludes the proof.

□

Conversions $\beta\eta$

On the one hand the fact that $\langle \mathcal{I}, \mathcal{J} \rangle$ forms a logical relation ensures that the interpretation of a term is related with itself, i.e., the restriction of the relation $\mathcal{J}$ to the interpretation of terms is reflexive. On the other hand, the restriction of a partial equivalence relation R on those elements where R is reflexive ($\{x \mid xRx\}$) is an equivalence relation. In particular, the restriction of the interpretation to the subset where $\mathcal{J}$ is reflexive, is already enough to interpret terms. On this restriction the relation $\mathcal{J}$ is an equivalence relation and can be seen as coarser equality than the set-theoretic equality.

We will now prove that, if we consider this latter equality as the actual equality on the interpretation, the interpretation of terms is sound with respect to the $\beta\eta$-conversion, (i.e., $r =_{\beta\eta} s \Rightarrow [\![r]\!]\mathcal{J}[\![s]\!]$).

The following technical lemma relates semantical valuation and syntactical substitution and will be needed to relate the interpretation of β-convertible terms.

Lemma 16 (substitution)**.** *Given a typed term $\Gamma_s \vdash s : \sigma$, two valuations η and δ on Γ_s, (i.e., $\eta \vDash \Gamma_s$, and $\delta \vDash \Gamma_s$), a typed term $\Gamma_s, x : \rho, \Gamma_r \vdash r : \sigma$, and two valuations η' and δ' on Γ_r (i.e., $\eta' \vDash \Gamma_r$, and $\delta' \vDash \Gamma_r$). Moreover, given contexts $\Gamma, \Gamma', \Delta, \Delta'$ with $\eta\ \mathcal{I}_{\Gamma,\Gamma'}\ \delta$, an environment e such that $\Gamma \notin e$, $\Delta; \Delta' \geqslant \Gamma; \Gamma'$ and $\eta'\ \mathcal{I}_{\Delta,\Delta'}\ \delta'$ we have:*

$$[\![r]\!]_{\eta, x\mapsto [\![s]\!]_\eta e, \eta'}\ \mathcal{J}_{\Delta,\Delta'}\ [\![r\{^s/_x\}]\!]_{\delta,\delta'}$$

Remark that an environment e is present in the left hand side of the relation above but not in the right hand side. This will allow to relate the terms of the algorithm which are not syntactically equal but only α-convertible.

Proof. By induction on the typing derivation of r, we have to show for $\Delta, \Delta' \notin d, d'$

$$[\![r]\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d \; \mathcal{I}_{\Delta,\Delta'} \; [\![r\{^s/_x\}]\!]_{\delta,\delta'} d'$$

case $r = x$**,** As the variables of the domain of the valuation δ' do not occur in s, we have immediately that $[\![s]\!]_\delta = [\![s]\!]_{\delta,\delta'}$ and we have to prove:

$$[\![s]\!]_\eta e \; \mathcal{I}_{\Delta,\Delta'} \; [\![s]\!]_\delta d'$$

By the basic lemma, we already have $[\![s]\!]_\eta \; \mathcal{J}_{\Gamma,\Gamma'} \; [\![s]\!]_\delta$, which means by definition that for e and e' with $\Gamma, \Gamma' \notin e, e'$ we have:

$$[\![s]\!]_\eta e \; \mathcal{I}_{\Gamma,\Gamma'} \; [\![s]\!]_\delta e',$$

In particular for $\Delta' \geqslant \Gamma'$, and d' such that $\Delta' \notin d'$ we have $\Gamma' \notin d'$, so that we have $[\![s]\!]_\eta e \; \mathcal{I}_{\Gamma,\Gamma'} \; [\![s]\!]_\delta d'$. And by monotonicity, we obtain finally:

$$[\![s]\!]_\eta e \; \mathcal{I}_{\Delta,\Delta'} \; [\![s]\!]_\delta d',$$

case $r = r's'$**,** We have to show:

$$[\![r']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d([\![s']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d) d \; \mathcal{I}_{\Delta,\Delta'} \; [\![r'\{^s/_x\}]\!]_{\delta,\delta'} d'([\![s'\{^s/_x\}]\!]_{\delta,\delta'} d') d'$$

By induction hypothesis on s', for d, d' such that $\Delta, \Delta' \notin d, d'$ we have

$$[\![s']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d \; \mathcal{I}_{\Delta,\Delta'} \; [\![s'\{^s/_x\}]\!]_{\delta,\delta'} d'$$

By induction hypothesis on r', we have

$$[\![r']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} \; \mathcal{I}_{\Delta,\Delta'} \; [\![r'\{^s/_x\}]\!]_{\delta,\delta'}$$

But because r is in applicative position it is of arrow type and by unfolding the definition we have for $E, E \geqslant \Delta, \Delta'$, $a \; \mathcal{I}_{E,E'} \; b$ and f, f' such that $E, E' \notin f, f'$:

$$[\![r']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} a f \; \mathcal{I}_{E,E'} \; [\![r'\{^s/_x\}]\!]_{\delta,\delta'} b f'$$

Taking Δ, Δ' for E, E', and $[\![s']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d$ and $[\![s'\{^s/_x\}]\!]_{\delta,\delta'} d'$ for a and b, and d, d' for f, f' yields the result.

case $r = \lambda y.r'$**,** First remark that $\lambda y.r$ is of arrow type, say $\rho \to \sigma$, so that we have to prove for $E, E' \geqslant \Delta, \Delta'$ and $a \; \mathcal{I}^\rho_{E,E'} \; b$:

$$[\![\lambda y.r']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta'} d a \; \mathcal{J}^{\rho \to \sigma}_{E,E'} \; [\![\lambda y.r'\{^s/_x\}]\!]_{\delta,\delta'} d' b$$

which simplifies into

$$[\![r']\!]_{\eta, x \mapsto [\![s]\!]_\eta e, \eta', y \mapsto a} \; \mathcal{J}^{\rho \to \sigma}_{E,E'} \; [\![\lambda y.r'\{^s/_x\}]\!]_{\delta,\delta', y \mapsto b}$$

By monotonicity (mono), $\eta' \; \mathcal{I}_{\Delta,\Delta'} \; \delta' \Rightarrow \eta' \; \mathcal{I}_{E,E'} \; \delta'$ and hence $\eta', y \mapsto a \; \mathcal{I}_{E,E'} \; \delta', y \mapsto b$. One can then apply the induction hypothesis on r', which yields the result.

□

Corollary 2 (β-conversion)**.** *Given a typed term $\Gamma_s \vdash s : \sigma$, two valuations η and δ on Γ_s, (i.e., $\eta \vDash \Gamma_s$, and $\delta \vDash \Gamma_s$), a typed term $\Gamma_s, x : \rho \vdash r : \sigma$. Moreover, given contexts Γ, Γ' with $\eta\ \mathcal{I}_{\Gamma,\Gamma'}\ \delta$, we have:*

$$[\![(\lambda x.r)s]\!]_\eta\ \mathcal{J}_{\Gamma,\Gamma'}\ [\![r\{^s/_x\}]\!]_\delta$$

Proof. Given e, e' such that $\Gamma, \Gamma' \notin e, e'$, we have to prove:

$$[\![(\lambda x.r)s]\!]_\eta e\ \mathcal{I}_{\Gamma,\Gamma'}\ [\![r\{^s/_x\}]\!]_\delta e'$$

which is equivalent to

$$[\![r]\!]_{\eta, x \mapsto [\![s]\!]_\eta e}\ e\ \mathcal{I}_{\Gamma,\Gamma'}\ [\![r\{^s/_x\}]\!]_\delta e'$$

By instantiating the previous lemma (16) with $\Delta, \Delta' = \Gamma, \Gamma'$, $\eta = \eta$ and $\delta = \delta$, we have:

$$[\![r]\!]_{\eta, x \mapsto [\![s]\!]_\eta e}\ \mathcal{J}_{\Gamma,\Gamma'}\ [\![r\{^s/_x\}]\!]_\delta$$

and in particular, we get the result:

$$[\![r]\!]_{\eta, x \mapsto [\![s]\!]_\eta e}\ e\ \mathcal{I}_{\Gamma,\Gamma'}\ [\![r\{^s/_x\}]\!]_\delta e'$$

□

Lemma 17 (η-conversion)**.** *Given a typed term $\Gamma_r \vdash r : \sigma$, two valuations η and δ on Γ_r, (i.e., $\eta \vDash \Gamma_r$ and $\delta \vDash \Gamma_r$). Moreover, given contexts Γ, Γ' with $\eta\ \mathcal{I}_{\Gamma,\Gamma'}\ \delta$, we have:*

$$[\![r]\!]_\eta\ \mathcal{J}^{\rho \to \sigma}_{\Gamma,\Gamma'}\ [\![\lambda x.rx]\!]_\delta$$

Proof. We have to show, given e, e' such that $\Gamma, \Gamma' \notin e, e'$, and $\Delta, \Delta' \geqslant \Gamma, \Gamma'$ and $a\mathcal{I}_{\Delta,\Delta'}b$, d, d' such that $\Delta, \Delta' \notin d, d'$, that:

$$[\![r]\!]_\eta ead\ \mathcal{J}_{\Delta,\Delta'}\ [\![\lambda x.rx]\!]_\delta e'bd'$$

which simplifies into $[\![r]\!]_\eta ead\ \mathcal{J}_{\Delta,\Delta'}\ [\![r]\!]_\delta d'bd'$

By the basic lemma (13),

$$[\![r]\!]_\eta\ \mathcal{J}^{\rho \to \sigma}_{\Gamma,\Gamma'}\ [\![r]\!]_\delta$$

By definition, for e, e' such that $\Gamma, \Gamma' \notin e, e'$, we have

$$[\![r]\!]_\eta e \; \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'} \; [\![r]\!]_\delta e'$$

But for $\Delta \geqslant \Gamma$, $\Delta \notin d'$ implies $\Gamma \notin d'$, so that we have $[\![r]\!]_\eta e \; \mathcal{I}^{\rho\to\sigma}_{\Gamma,\Gamma'} \; [\![r]\!]_\delta d'$. And in turn we have, for $\Delta, \Delta' \geqslant \Gamma, \Gamma'$, and $a \; \mathcal{I}_{\Delta,\Delta'} \; b$:

$$[\![r]\!]_\eta e a \; \mathcal{J}^{\sigma}_{\Delta,\Delta'} \; [\![r]\!]_\delta d' b$$

And finally,

$$[\![r]\!]_\eta e a d \; \mathcal{I}^{\sigma}_{\Delta,\Delta'} \; [\![r]\!]_\delta d' b d'$$

□

Lemma 18. *Given two typed terms $\Gamma \vdash r : \tau$ and $\Gamma \vdash s : \tau$, two valuation η and δ on Γ (i.e., $\eta \vDash \Gamma$ and $\delta \vDash \Gamma$), and two contexts Δ, Δ', we have:*

$$\Gamma \vdash r =_{\beta\eta} s : \tau \Rightarrow \forall\, \eta \; \mathcal{I}_{\Delta,\Delta'} \; \delta, \; [\![r]\!]_\eta \; \mathcal{J}^{\tau}_{\Delta,\Delta'} \; [\![s]\!]_\delta$$

Proof. By induction on the relation $\Gamma \vdash r =_{\beta\eta} s : \tau$,
The case of reflexivity is the basic lemma. (13) The cases of β-conversion, η-conversion, reflexivity, symmetry and transitivity have already been handled in lemma (14)(15)(17) and corollary (2).
Rest the structural rules:

$$\Gamma \vdash tr =_{\beta\eta} ts : \rho \Rightarrow \forall\, \eta \; \mathcal{I}_{\Delta,\Delta'} \; \delta, \; [\![tr]\!]_\eta \; \mathcal{J}^{\rho}_{\Delta,\Delta'} \; [\![ts]\!]_\delta$$
$$\Gamma \vdash rt =_{\beta\eta} st : \rho \Rightarrow \forall\, \eta \; \mathcal{I}_{\Delta,\Delta'} \; \delta, \; [\![rt]\!]_\eta \; \mathcal{J}^{\rho}_{\Delta,\Delta'} \; [\![st]\!]_\delta$$
$$\Gamma \vdash \lambda x^\rho.r =_{\beta\eta} \lambda x^\rho.s : \sigma \Rightarrow \forall\, \eta \; \mathcal{I}_{\Delta,\Delta'} \; \delta, \; [\![\lambda x^\rho.r]\!]_\eta \; \mathcal{J}^{\rho\to\sigma}_{\Delta,\Delta'} \; [\![\lambda x^\rho.s]\!]_\delta$$

They all follow from the induction hypothesis on r and s and use of the basic lemma for t in the two first cases. □

Completeness Lemma

In the informal presentation the reify function $\downarrow$ is a function from the interpretation to the set of terms and the function reflect is a function from the set of terms to the interpretation. The following lemma states a similar result for the call-by-value interpretation when the set of terms is quotiented by α-conversion, and the (restriction of the) interpretation by the partial equivalence relation $\mathcal{I}$ or $\mathcal{J}$.

Lemma 19 (Completeness)**.** *Given $a, b \in [\![\rho]\!]$, $r \in \mathsf{Tm}_\rho$, and $e, e' \in E$ the following propositions hold:*

$$m \; \mathcal{J}^{\rho}_{\Gamma,\Gamma'} \; m' \Rightarrow \forall\, e, e', (\Gamma, \Gamma') \notin (e, e'), \; \Gamma; \Gamma' \vdash (m\star\downarrow)e =_\alpha (m'\star\downarrow)e' : \rho \qquad (1')$$
$$\Gamma; \Gamma' \vdash r =_\alpha r' : \rho \Rightarrow \uparrow^\rho r \; \mathcal{I}_{\Gamma,\Gamma'} \; \uparrow^\rho r' \qquad (2')$$

Proof. By simultaneous induction on the type $\rho \in \mathsf{Ty}$,

case $\rho = o$ (1'), by definition of $\downarrow^o$, we have:

$$(m\star \downarrow^o) = (m \star \nu) = m.$$

By definition of $\mathcal{J}^o_{\Gamma,\Gamma'}$, for e, e' such that $\Gamma, \Gamma' \notin e, e'$ we have $me\ \mathcal{I}^o_{\Gamma,\Gamma'}\ me'$, and by definition again:

$$\Gamma, \Gamma' \vdash r =_\alpha r' : o$$

case $\rho = o$ (2'), follows from the definition as $\uparrow^o$ is the identity.

case $\rho \to \sigma$ (1'), given $m\ \mathcal{J}^{\rho\to\sigma}_{\Gamma,\Gamma'}\ m'$, and e, e' such that $\Gamma, \Gamma' \notin e, e'$ we want to show that:

$$\Gamma; \Gamma' \vdash (m\star \downarrow)e =_\alpha (m'\star \downarrow)e' : \rho$$

By induction hypothesis (2') on ρ, for $x^\rho, y^\rho \notin \Gamma, \Gamma'$ we have:

$$\uparrow^\rho x\ \mathcal{I}_{\Gamma,x^\rho;\Gamma',y^\rho}\ \uparrow^\rho y$$

In particular, for e, e' such that $\Gamma, \Gamma' \notin e, e'$ one can take $\mathsf{new}(e)$ and $\mathsf{new}(e')$ for x and y.

By hypothesis, this implies

$$me \uparrow^\rho x\ \mathcal{J}^\sigma_{\Gamma,x^\rho;\Gamma',y^\rho}\ m'e' \uparrow^\rho y$$

As $\Gamma, \Gamma' \notin e, e'$ implies $(\Gamma, x : \rho; \Gamma', y : \rho) \notin e^x, e'^y$, we obtain by induction hypothesis (1') on σ,

$$\Gamma, x : \rho; \Gamma', y : \rho \vdash (me \uparrow^\rho x\star \downarrow)e^x =_\alpha (m'e' \uparrow^\rho y\star \downarrow)e'^y : \rho$$

We can abstract on both sides to obtain:

$$\Gamma; \Gamma' \vdash \lambda x^\rho.(me \uparrow^\rho x\star \downarrow)e^x =_\alpha \lambda y^\rho((m'e' \uparrow^\rho y\star \downarrow)e'^y) : \rho$$

With a bit of computation, one can see that the left hand side is equal to:

$$\begin{aligned}\lambda x.(me(\uparrow x)\star \downarrow^\sigma)e^x &= \lambda x.(me(\uparrow x)\star \downarrow^\sigma)^x e \\ &= ((me(\uparrow x)\star \downarrow^\sigma)^x \star \mathbb{\lambda} t.\nu(\lambda x.t))e \\ &= (m \star \mathbb{\lambda} f.(f(\uparrow x)\star \downarrow^\sigma)^x \star \mathbb{\lambda} t.\nu(\lambda x.t))e\end{aligned}$$

and by choosing $\mathsf{new}(e)$ for x,

$$\lambda x.(me(\uparrow x)\star \downarrow^\sigma)e^x =_\alpha (m\star \downarrow)e$$

By an analogous computation (and choosing $\mathsf{new}(e')$ for y) the right hand side is α-equal to $(m'\star \downarrow)e'$ [3].

case $\rho \to \sigma$, (2') Under the hypothesis

$$\Gamma; \Gamma' \vdash r =_\alpha r' : \rho \to \sigma$$

we want to show, given $\Delta, \Delta' \geqslant \Gamma, \Gamma'$ and $a\ \mathcal{I}^\rho_{\Delta,\Delta'}\ b$ that:

$$(\uparrow^{\rho\to\sigma} r)a\ \mathcal{J}^\sigma_{\Delta,\Delta'}\ (\uparrow^{\rho\to\sigma} s)b.$$

By (unit), we have $\nu(a)\ \mathcal{J}^\rho_{\Delta,\Delta'}\ \nu(b)$, and by induction hypothesis (1') on ρ we obtain, for d, d' such that $\Delta, \Delta' \notin d, d'$:

$$\Delta; \Delta' \vdash \nu(a)\star \downarrow_\rho d =_\alpha (\nu(b)\star \downarrow_\rho d') : \rho$$

which can be simplified into

$$\Delta; \Delta' \vdash (\downarrow_\rho a)d =_\alpha (\downarrow_\rho b)d' : \rho$$

and hence

$$\Delta; \Delta' \vdash r(\downarrow_\rho a)d =_\alpha r(\downarrow_\rho b)d' : \sigma$$

By induction hypothesis (2') on σ,

$$\uparrow r(\downarrow_\rho a)d\ \mathcal{I}_{\Delta,\Delta'}\ \uparrow r(\downarrow_\rho b)d'$$

but $\uparrow^\sigma r(\downarrow_\rho a)d = \downarrow_\rho a \star \mathbb{\lambda} s.\nu(\uparrow^\sigma rs)d = (\uparrow^{\rho\to\sigma} r)ad$, and analogously $\uparrow^\sigma r(\downarrow_\rho b)d' = (\uparrow^{\rho\to\sigma} r)bd'$, and the last relation is equivalent to:

$$(\uparrow^{\rho\to\sigma} r)ad\ \mathcal{I}_{\Delta,\Delta'}\ (\uparrow^{\rho\to\sigma} r)bd'$$

And we have proved

$$(\uparrow^{\rho\to\sigma} r)a\ \mathcal{J}^\sigma_{\Delta,\Delta'}\ (\uparrow^{\rho\to\sigma} s)b.$$

□

Corollary 3. *Given two typed terms $\Gamma \vdash r : \tau$ and $\Gamma \vdash s : \tau$, we have:*

$$\Gamma \vdash r =_{\beta\eta} s : \tau \Rightarrow [\![r]\!]_\uparrow\ \mathcal{J}^\tau_{\Gamma,\Gamma}\ [\![s]\!]_\uparrow$$

Proof. Follows from the lemma (18) with $\eta ::= \delta ::= \uparrow$, as the previous lemma (3) states that $\uparrow \mathcal{I}_{\Gamma,\Gamma} \uparrow$. □

Theorem 1. *Given two typed terms $\Gamma \vdash r : \tau$ and $\Gamma \vdash s : \tau$, we have:*

$$\Gamma \vdash r =_{\beta\eta} s : \tau \Rightarrow \forall e,\ \Gamma \notin e \Rightarrow \Gamma \vdash ([\![r]\!]_\uparrow \star \downarrow)e = ([\![s]\!]_\uparrow \star \downarrow)e : \tau$$

Proof. Follows from the previous corollary (3) and the completeness lemma (19) □

[3] Notice that we needed a more general induction principle than the first proposal (1), because we had to prove a conclusion of the form $\lambda x.r =_\alpha \lambda y.s$ where x and y may be different, and needed to apply the induction hypothesis on r and s.

Chapter 3

Generalized Applications

By the Curry-Howard isomorphism, the system of natural deduction NJ for minimal logic is in one-to-one correspondence with the simply typed λ-calculus Λ. The extensional conversion $\beta\eta$ of the λ-calculus corresponds to equivalence of proofs in natural deduction (in particular β-reduction corresponds to detour elimination).

It is well known however that the correspondence between sequent calculus and natural deduction is not one-to-one: for one proof in natural deduction there is in general several proofs in the sequent calculus.

Although informally cut elimination corresponds to detour elimination, and hence to β-reduction, even the correspondence between cut-free derivation and β-normal terms is not one-to-one, just as above: for one detour-free proof in natural deduction there is in general several cut-free proofs in the sequent calculus.

Because η-conversion equates more derivations in natural deduction, the situation is even more complicated with this conversion.

Nevertheless, Jan Von Plato designed in [90] a system of natural deduction where derivations are in one-to-one correspondence with cut-free sequent proofs.

In [55], Felix Joachimski and Ralph Matthes designed a λ-calculus Λ_J whose normal forms with respect to β-reduction and certain permutative reductions (or π-reductions) are exactly the derivations of the system of Von Plato, and they show its strong normalization.

Beside detour elimination and permutative reduction, one can consider other conversions in this calculus. The class of terms modulo these extra conversions is in one-to-one correspondence with the normal forms of the simply typed λ-calculus. Ralph Matthes used these generalised conversion relation in [69] to prove an interpolation result and show some of its applications.

Here we will discuss a definition of normal forms for the calculus Λ_J with these extended conversions. These normal forms are not unique representatives of the classes of terms modulo the extended conversions. But normal terms belonging to the same class are mutually convertible thanks to a decidable conversion (which we have called here circular conversion). We will then describe a normalization function

using normalization by evaluation. This provides a decision procedure for the whole theory of the Λ_J-calculus, which is, to the best of our knowledge, a new result.

This provides in turn a notion of normal proofs for the sequent calculus which correspond modulo circular conversions to normal proofs in natural derivation.

3.1 From Sequent Calculus to Λ_J

The formula we consider are the types of the simply typed calculus as defined in chapter 2. We restate the definition of the set Ty of type here for completeness:

$$\mathsf{Ty} \ni \rho, \sigma ::= o \mid \rho \to \sigma.$$

A sequent is a pair $\Gamma \vdash \rho$ of the context Γ, which is a multiset of formulae, and the formula ρ. Just as in natural deduction, contexts could have been presented as lists, but in this case we would have needed the structural rule of exchange to identify sequents differing in the order of their contexts.

For convenience we will use Kleene's version of the sequent calculus LJ [58].

Definition 3.1 (LJ). *The axioms and inference rules of LJ are:*

$$\frac{}{\Gamma, \rho \vdash \rho}\ \textit{(Var)} \qquad \frac{\Gamma, \rho \to \sigma \vdash \rho \qquad \Gamma, \rho \to \sigma, \sigma \vdash \tau}{\Gamma, \rho \to \sigma \vdash \tau}\ (\to\vdash)$$

$$\frac{\Gamma, \rho \vdash \sigma}{\Gamma \vdash \rho \to \sigma}\ (\vdash\to) \qquad \frac{\Gamma \vdash \rho \qquad \Gamma, \rho \vdash \sigma}{\Gamma \vdash \sigma}\ \textit{(Cut)}$$

Kleene's version has the advantage, thanks to the formulation of the rule $(\to\vdash)$ above, to be strictly cumulative, which means that if the sequent $\Gamma \vdash \rho$ appears in the conclusion of a rule, then Γ is already a sub-multiset of the antecedents of the premises of the rule (for details, the reader may wish to consult the comprehensive textbook of Helmut Schwichtenberg and Anne S. Troelstra, Basic Proof Theory [88]).

To a statement $\Gamma \vdash \rho$ we can assign a λ-term t so that t is well-typed in context Γ with type ρ, i.e., $\Gamma \vdash t : \rho$ (and hence t is a proof in natural deduction), in the following way:

$$\frac{}{\Gamma, x : \rho \vdash x : \rho}\ (\text{Var}) \qquad \frac{\Gamma, y : \rho \to \sigma \vdash r : \rho \qquad \Gamma, y : \rho \to \sigma, x : \sigma \vdash s : \tau}{\Gamma, y : \rho \to \sigma \vdash s\{^{yr}/_{x}\} : \tau}\ (\to\vdash)$$

$$\frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^{\rho}.r : \rho \to \sigma}\ (\vdash\to) \qquad \frac{\Gamma \vdash r : \rho \qquad \Gamma, x : \rho \vdash s : \sigma}{\Gamma \vdash s\{^{r}/_{x}\} : \sigma}\ (\text{Cut})$$

The notations $s\{^{r}/_{x}\}$ and $s\{^{yr}/_{x}\}$ in the rule $(\to\vdash)$ and (Cut) are not term formation rules but notations on the meta-level for substitution. It is a reason why the simply typed λ-calculus cannot be used to faithfully model proofs of the sequent calculus.

Example 11. *Obviously, after having decorated a sequent proof with λ-terms, taking the last decorated statement in the proof provides a derivation in natural deduction. But this translation from sequent calculus to natural deduction is not one-to-one. The two following decorated sequent calculus derivations provide the same derivation (where $z : \rho \to \sigma \in \Gamma$):*

$$\dfrac{\dfrac{\Gamma, x : \rho \vdash r : \sigma \qquad \Gamma, x : \rho, y : \tau \vdash s : \upsilon}{\Gamma, x : \rho \vdash s\{^{zr}/_{y}\} : \upsilon}\,(\to\vdash)}{\Gamma \vdash \lambda x^{\rho}.s\{^{zr}/_{y}\} : \rho \to \upsilon}\,(\vdash\to) \qquad \dfrac{\Gamma \vdash r : \sigma \qquad \dfrac{\Gamma, x : \rho, y : \tau \vdash s : \upsilon}{\Gamma, y : \tau \vdash \lambda x^{\rho}.s : \upsilon}\,(\vdash\to)}{\Gamma \vdash \lambda x^{\rho}.s\{^{zr}/_{y}\} : \rho \to \upsilon}\,(\to\vdash)$$

To have a faithful notation system for the sequent calculus, it is possible to work with explicit substitution. The notation $-\{^{-}/_{x}\}$ would then be a term constructor. This approach has been studied by Herbelin in [50].

The idea in [55] is that if one excludes the cut-rule, one does not need to consider general substitution but only the one occurring in the rule $(\to\vdash)$: $-\{^{--}/_{x}\}$. The term constructor corresponding to an explicit substitution of this form is called a generalised application and is noted $r(s, x^{\sigma}.t)$ instead of $t\{^{rs}/_{x}\}$.

Now, in the assignment of λ-terms to sequent derivations, one can replace the standard application of the λ-calculus by this generalised application in the rule $(\to\vdash)$:

$$\dfrac{\Gamma, y : \rho \to \sigma \vdash s : \rho \qquad \Gamma, y : \rho \to \sigma, x : \sigma \vdash t : \tau}{\Gamma, y : \rho \to \sigma \vdash y(s, x^{\sigma}.t) : \tau}\,(\to\vdash)$$

The cut-rule allows us to substitute the variable y in the above rule by an arbitrary term, giving rise to the Λ_J calculus:

$$\Lambda_J \ni r, s, t ::= x \;\mid\; \lambda x^{\rho}.r \;\mid\; r(s, x^{\sigma}.t)$$

Hence one can annotate sequents of derivations in the sequent calculus with terms of the Λ_J-calculus, just in the same way as with terms of Λ. The cut-free derivations correspond to terms whose subterms in applicative position are assumptions (a variable y):

$$\mathsf{NF}_{\Lambda_J} \ni r, s, t ::= x \;\mid\; \lambda x^{\rho}.r \;\mid\; y(s, x^{\sigma}.t)$$

This latter terms are normal forms for the β and permutative reductions[1]:

$$\begin{aligned} (\lambda x^{\rho}.r)(s, y^{\sigma}.t) &\longrightarrow_{\beta} t\{^{r\{^{s}/_{x}\}}/_{y}\} \\ r(s, x^{\sigma}.t)(s', x'^{\tau}.t') &\longrightarrow_{\pi} r(s, x^{\sigma}.t(s', x'^{\tau}.t')) \end{aligned}$$

[1]The permutative reductions allows to recover the subformula property, i.e., the type of any subterm of a term r of type ρ is either the type of a free variable or a syntactic subtype of ρ.

Replacing inductively the explicit substitutions $r(s, x^\sigma.t)$ by actual substitutions $t\{^{rs}/_x\}$, provides a translation $-^\Lambda$ from Λ_J to Λ. This translation justifies the above rules as a standard β-conversion and a substitution lemma:

$$\begin{aligned} t\{^{(\lambda x^\rho.r)s}/_y\} &=_\beta t\{^{r\{^s/_x\}}/_y\} \\ t'\{^{(t\{^{rs}/_x\})s'}/_{x'}\} &=_\pi t'\{^{(ts')\{^{rs}/_x\}}/_{x'}\} \end{aligned}$$

The terms of Λ_J can as well be used to annotate propositions of a natural deduction system which provides a syntax-directed typing system as follows:

Definition 3.2 (Typing)**.** *The* typing relation *is a ternary relation between contexts, terms and types and is defined inductively by the following rules:*

$$\frac{x : \rho \in \Gamma}{\Gamma \vdash x : \rho}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho \qquad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash r(s, x^\sigma.t) : \tau}\ (\to\text{E}) \qquad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^\rho.r : \rho \to \sigma}\ (\to\text{I})$$

In the terminology used for natural deduction, the rule ($\to$E) is called an elimination rule, and the rule ($\to$I) an introduction rule. The premiss carrying the symbol $\to$ in the elimination rule is called the major premiss, the other premises are called minor premises. By analogy, we will say that in the term $r(s, x.t)$, r is a major premiss and s and t are minor premises.

Typed terms of Λ_J are not in a one-to-one correspondence with derivations of sequent calculus, for one derivation in Λ_J there exists several sequent derivations with cut. In the decoration of the sequent calculus proofs, each rule of the sequent calculus (except (Cut)) is associated to a different term formation rule of the Λ_J-calculus. Hence for a normal Λ_J derivation with respect to $\longrightarrow_\beta$ and $\longrightarrow_\pi$ reductions there exists one and only one cut-free sequent derivation.

As in the previous chapter we will often omit the type of the bound variables to improve readability.

3.2 Extended Conversions and Normal Forms

There are more conversions to be considered than only $=_\beta$ and $=_\pi$. We have at hand a translation $-^\Lambda$ from Λ_J to the simply typed λ-calculus Λ. Because $\beta\pi$-normal forms of Λ_J correspond to only one cut-free sequent derivation whereas β-normal forms of Λ may correspond to several cut-free sequents derivations, one can define the missing conversions between terms r and s of Λ_J such that the translations of r and s in Λ are the same (modulo β-equality).

We will give the conversions in form of axioms; the corresponding conversions are then obtained as the smallest congruence, a contextually closed equivalence relation, containing the given axiom(s).

Definition 3.3 (Contextual closure)**.** *A relation $=_R$ is contextually closed if the following rules hold*

$$\frac{r =_R r'}{r(s,x.t) =_R r'(s,x.t)}\ (R\text{-AppL1}) \qquad \frac{s =_R s'}{r(s,x.t) =_R r(s',x.t)}\ (R\text{-AppL2})$$

$$\frac{t =_R t'}{r(s,x.t) =_R r(s,x.t')}\ (R\text{-AppR}) \qquad \frac{r =_R s}{\lambda x.r =_R \lambda x.s}\ (R\text{-}\xi)$$

We first give the axiom for the β-conversion; read from left to right this conversion corresponds to a computation step of the calculus.

Definition 3.4 (β-conversion)**.** *The axiom of β-conversion is defined by:*

$$(\lambda x.r)(s,y.t) \ =_\beta \ t\{^{r\{^{s}/_{x}\}}/_{y}\} \qquad (\beta)$$

Remark 9. *We gloss over the notions of free, bound variables, set of free variable $\mathsf{FV}(r)$ of a term r, α-conversion and substitution which are similarly defined as they were in the simply typed λ-calculus in chapter 2 (the only difference being that in the term $r(s,x.t)$ the variable x is bound in t) and just remark that the substitution used here is a variable capture avoiding one where α-conversion may be needed.*

The π-conversion permutes general applications if they form a particular pattern (the major premiss of a general application is itself a general application). But if the π-conversion is read as a substitution lemma for the corresponding explicit substitution, then in the general case one has to pull out a general application in an arbitrary context. Hence we define our general notion of permutative conversion as follows:

Definition 3.5 (π^e-conversion)**.** *The axiom of* extensional permutative conversion *is defined by:*

$$r\{^{s(t,y.u)}/_{x}\} =_{\pi^e} s(t,y.r\{^{u}/_{x}\}) \qquad y \notin \mathsf{FV}(r)$$

Moreover one can define a conversion between terms of Λ_J such that their translations in the simply typed λ-calculus Λ are not only β- but η-equal as well, this leads to the following conversion[2].

[2]we remind the reader that the dot bind as far as syntactically possible to the left, so that the axiom η has to be read as $r =_\eta \lambda z^\rho.(r(z,y.y))$

Definition 3.6. *The axiom of η-conversion is defined by*

$$r =_\eta \lambda z^\rho.r(z, y.y) \qquad z \notin \mathsf{FV}(r) \qquad (\eta)$$

Remark 10. *A characteristic of the Λ_J-calculus is that we only need to define η-conversion on variables. An η-conversion of a λ-abstraction can already be achieved with a β-conversion just as for the simply typed λ-calculus. And an η-conversion of a generalised application $r(s, x.t)$ boils down modulo permutative conversion* (3.5) *to an (η)-conversion of its minor premiss s.*

We can give a local version of the π^e-conversion. This will help us to justify the structure of the term outputted by the NbE-algorithm, the normal forms.

Lemma 20 (π^e-conversion (local version))**.** *The π^e-conversion can be defined by the following axioms:*

$$\begin{array}{rcll}
r(s, x.r')(s', x'.t') & =_{\pi_1^e} & r(s, x.r'(s', x'.t')) & (x \notin \mathsf{FV}(s'), x \notin \mathsf{FV}(t')) \quad (\pi_1^e) \\
r(r'(s', x'.t'), x.t) & =_{\pi_2^e} & r'(s', x'.r(t', x.t)) & (x' \notin \mathsf{FV}(r), x' \notin \mathsf{FV}(t)) \quad (\pi_2^e) \\
\lambda x.r(s, y.t) & =_{\pi_3^e} & r(s, y.\lambda x.t) & (x \notin \mathsf{FV}(r), x \notin \mathsf{FV}(s)) \quad (\pi_3^e) \\
r(s, x.t) & =_{\mathsf{is}} & t & (x \notin \mathsf{FV}(t)) \quad (\mathsf{is}) \\
r(s, x.r'(s', x'.t')) & =_{\pi_4^e} & r'(s', x'.r(s, x.t')) & (x \notin \mathsf{FV}(r'), x \notin \mathsf{FV}(s'), \\
 & & & \quad x' \notin \mathsf{FV}(r), x' \notin \mathsf{FV}(s)) \quad (\pi_4^e) \\
r(s, x.r(s, x'.t')) & =_{\pi_5^e} & r(s, x'.t'\{^{x'}/_x\}) & (x \notin \mathsf{FV}(r), x \notin \mathsf{FV}(s)) \quad (\pi_5^e)
\end{array}$$

The first four conversions (π_1^e to π_4^e) are just instantiations of the general π^e-conversion. Their contextual closure is a special case of the general π^e-conversion where the substitution involved only concerns one unique variable. The axiom (π_5^e) allows to take into account the case when in the π^e-conversion the substitution concerns multiple occurrences of the same variable. The last conversion (is) handles the case where the variable to be substituted for in the general π^e-conversion does not actually occur in the term.

We will describe the normal forms of this system by giving some sensible criteria. First we want our normal forms to correspond to cut-free sequent derivations, so that they should at least be some restrictions of the π-normal forms as given in section 2, i.e., normal forms for the β-, π_1^e-reduction (oriented from left to right):

$$\mathsf{NF}_{\Lambda_J} \ni N, N_0, N_1 ::= x \;\mid\; \lambda x.N \;\mid\; y(N_0, x.N_1)$$

The conversion (π_2^e), read from left to right, pulls out the application on the right hand side of the term. This is already the effect of the π-reduction, so that we keep this direction. The normal forms will not be allowed to have an application as the

minor premiss s of another application $r(s, x.t)$. To achieve this, we have to redefine our normal forms as follows:

$$\begin{aligned} \mathsf{PNF}_{\Lambda_J} \ni P &::= x \mid \lambda x.N \\ \mathsf{NF}_{\Lambda_J} \ni N &::= P \mid y(P, x.N) \end{aligned}$$

The rule (π_3^e) says that a λ-abstraction can be moved inside a generalised application. It seems reasonable to move it as deep as possible, in order to introduce variables just before their uses. To that effect, we define the deepest λ condition, $\mathsf{DL}(\mathbb{X}, r)$ for a set of variables $\mathbb{X}$ and a term r, to be true if r is a variable or a λ-abstraction and if r is a generalised application $(r(s, y.t))$ by the following:

$$\mathsf{DL}(\mathbb{X}, (r(s, y.t))) :\Leftrightarrow \mathbb{X} \cap (\mathsf{FV}(r) \cup \mathsf{FV}(s)) \neq \emptyset \wedge \mathsf{DL}(\mathbb{X} \cup \{y\}, t)$$

We impose the condition $\mathsf{DL}(\{x\}, N)$, in case of the abstraction $\lambda x.N$.

Immediate Simplification

For the conversion (is), known as *immediate simplification*, we follow Prawitz [78] by simplifying unused arguments and impose the condition $x \in \mathsf{FV}(N_1)$ in case of the application $y(N_0, x.N_1)$.

The rule (π_4^e) is somewhat problematic because by itself it creates a loop, whatever orientation one chooses for it, and we will come back to it later, together with the last rule (π_5^e).

Conversion η and Normal Forms

In contrast with the conversions β and π^e, the η-conversion requires us to consider the types of the term involved.

The rule of η-conversion oriented as an expansion (i.e., from left to right in the definition above) expands a term of arrow type into a generalised application. Considered together with the β conversion oriented in a reduction from left to right, one has to impose restrictions on η-expansion to prevent infinite sequence of reductions. Namely an η-expansion should not expand a term in abstraction form nor the major premiss r of a generalised application $r(s, x.t)$.

To obtain normal forms for the η-conversion, it suffices to restrict the formation of pure normal forms for variables to those of ground type. Hence a final step is to incorporate typing in our normal forms, they will have to verify the following definition.

Definition 3.7 (Λ_J-normal form)**.** *The set* NF_{Λ_J} *of* Λ_J*-normal forms is defined inductively together with the set* PNF_{Λ_J} *of pure normal forms as follows:*

$$\frac{(x:o)\in\Gamma}{\Gamma\vdash_{\mathsf{PNF}} x:o} \qquad \frac{\Gamma,x:\rho\vdash_{\mathsf{NF}} N:\sigma}{\Gamma\vdash_{\mathsf{PNF}}\lambda x.N:\rho\to\sigma}$$

$$\frac{\Gamma\vdash_{\mathsf{PNF}} P:\rho}{\Gamma\vdash_{\mathsf{NF}} P:\rho} \qquad \frac{\Gamma,y:\rho\to\sigma\vdash_{\mathsf{PNF}} P:\rho \qquad \Gamma,y:\rho\to\sigma,x:\sigma\vdash_{\mathsf{NF}} N:\tau}{\Gamma\vdash_{\mathsf{NF}} y(P,x.N):\tau}$$

in the clause with conclusion $\Gamma\vdash_{\mathsf{PNF}}\lambda x.P : \rho\to\sigma$*, we impose the condition* $DL(\{x\},P)$ *which is true if* P *is not an generalised elimination and defined otherwise by:*

$$\mathsf{DL}(\mathbb{X},(r(s,y.t))) :\Leftrightarrow \mathbb{X}\cap(\mathsf{FV}(r)\cup\mathsf{FV}(s))\neq\emptyset\wedge\mathsf{DL}(\mathbb{X}\cup\{y\},t)$$

and in the clause with conclusion $\Gamma\vdash_{\mathsf{NF}} y(P,x.N):\tau$*, we have the condition* $x\in\mathsf{FV}(N)$*.*

Circular Conversions

Let us come back to the conversions (π_4^e) and (π_5^e).

In normal terms as given above, these *circular conversions* concern occurrences of lists of generalised applications of the form:

$$x_0(P_0,y_0.x_1(P_1,y_1.x_2(P_2,y_2\ldots))$$

(where it may be the case that $x_i=x_j$ and $P_i=P_j$). The conversion (π_4^e) identifies those such lists where some generalised applications have been permuted, whereas (π_5^e) identifies those where some generalised applications have been duplicated.

A (decidable) test for the conversion (π_4^e) and (π_5^e) can be done a posteriori on these normal forms.

Notation 13. *We will call the conversions* (π_4^e) *and* (π_5^e) circular permutative conversion *and write* π_c^e *for their union.*

Remark 11 (variations on normal forms)**.** *One could further impose conditions on normal forms: If one does not want to allow for repetition, i.e., one forbids* $x_i=x_j$ *and* $r_i=_{\pi_c^e} r_j$ *in a list of generalised applications, then one has only to check for permutation of these generalised applications and the class of convertible normal forms even becomes finite. But then a test of inequality modulo* $=_{\pi_c^e}$ *has to be incorporated in the formulation of normal forms.*

3.3 Informal Description

We use normalization by evaluation as a normalizing procedure. First we have to give a suitable interpretation. In this informal description our interpretation of the Λ_J calculus is a standard set-theoretical one with the ground type interpreted as the set of variables:

Definition 3.8 (Type Interpretation)**.** *We define the* interpretation $[\![\rho]\!]$ of a type ρ, *by induction on* $\rho \in \mathsf{Ty}$

$$\begin{aligned} [\![o]\!] &::= \mathsf{Var} \\ [\![\rho \to \sigma]\!] &::= [\![\rho]\!] \to [\![\sigma]\!], \end{aligned}$$

where $[\![\rho]\!] \to [\![\sigma]\!]$ *is the full function space between the set* $[\![\rho]\!]$ *and* $[\![\sigma]\!]$.

The interpretation of types $[\![\mathsf{Ty}]\!]$ of the typed lambda calculus with generalised application Λ_J *is then:*

$$[\![\mathsf{Ty}]\!] ::= \bigcup_{\rho \in \mathsf{Ty}} [\![\rho]\!].$$

To define the interpretation of terms we need first to define the auxiliary notion of valuation.

Definition 3.9 (Valuation)**.** *Given a context* Γ, *we define a* valuation *on* Γ *to be a partial function* $\eta : \mathsf{Var} \to [\![\mathsf{Ty}]\!]_\perp$ *such that for* $x : \rho \in \Gamma$, $\eta(x)$ *is defined and* $\eta(x) \in [\![\rho]\!]$. *Given a context* Γ, *a valuation* η *on* Γ, *a variable* $y \notin \Gamma$ *and an element* $a \in [\![\sigma]\!]$, *we define a valuation* $(\eta, y \mapsto a)$ *on* $\Gamma \cup \{(y : \sigma)\}$, *called the extension of* η *by* $y \mapsto a$ *by,*

$$(\eta, y \mapsto a)(x) ::= \begin{cases} a & \text{if } x = y, \\ \eta(x) & \text{otherwise.} \end{cases}$$

Remark 12. *Valuation functions are partial functions represented as total functions from the set of variables* Var *into the set* $[\![\mathsf{Ty}]\!]_\perp = [\![\mathsf{Ty}]\!] + \{\star\}$, *i.e., the interpretation of types extended with an element* $\star$ *playing the rôle of an undefined value. In the following, whenever we will use a valuation applied to some variable, this variable will belong to the domain of definition of the valuation and the result will therefore be defined. And although, strictly speaking we should do a case distinction on the result to know wheither it is defined (an alternative would be to use an exception monad as presented in definition 1.3), we will consider it to be an element of* $[\![\mathsf{Ty}]\!]$.

The interpretation of a term is an element of the interpretation of its type.

Definition 3.10 (Term Interpretation)**.** *We define the* interpretation $[\![r]\!]_\eta$ of a term r*, whenever there is a context* Γ*, such that* $\Gamma \vdash r : \rho$ *and* η *is a valuation on* Γ*, to be an element of* $[\![\rho]\!]$*, by the following inductive definition:*

$$\begin{aligned}[\![x]\!]_\eta &= \eta(x)\\ [\![\lambda x.r]\!]_\eta(a) &= [\![r]\!]_{\eta,x\mapsto a}\\ [\![r(s,x.t)]\!]_\eta &= [\![t]\!]_{\eta,x\mapsto [\![r]\!]_\eta([\![s]\!]_\eta))}\end{aligned}$$

In the normalization algorithm exposed below we will regularly use exceptions. While a purely functional semantics of exceptions is used in the next section, we prefer to explain them informally in this section. Let be given a value e of type E, an exception carrying the exceptional value e can be thrown with the $\mathsf{throw}(e)$ operator. Given a handling function h of type $E \to T$, if an exception is thrown with $\mathsf{throw}(e)$ inside a program p of type T, such an exception e can be caught with the operator $\mathsf{catch}(p)(h)$ and evaluated to $h(e)$.

3.3.1 Several Attempts

Let us write nf for the normalizing function we will describe. In the chapter on simply typed λ-calculus, this function was given by applying the reify function $\downarrow$ to the interpretation of a term t with the function reflect $\uparrow$ as valuation, $\mathsf{nf}(t) = \downarrow [\![t]\!]_\uparrow$.

The principle here is still the same and we will start from the algorithm of the last chapter and modify it to produce normal terms of Λ_J. The only syntactical difference between Λ_J and Λ is in applications, generalised applications for the former, simple applications for the latter. The only place where syntactical applications were created in the NbE algorithm of the last chapter was in the definition of the function $\uparrow$ at arrow type:

$$(\uparrow^{\rho\to\sigma} r)(a) ::= \uparrow^\sigma r(\downarrow_\rho a)$$

Because a generalised application of the form $r(s, z.z)$ translates to a simple application $r^\Lambda s^\Lambda$, to get a generalised application instead of a simple application, we could just replace this definition with the following:

$$(\uparrow^{\rho\to\sigma} r)(a) ::= \uparrow^\sigma r((\downarrow_\rho a), z.z)$$

However there is a problem with this definition. Let us call the pair of term (r, s) in a generalised application $r(s, x.t)$, a *pseudo-application*. If we identify λ-terms with programs, the term r can be seen as a subprogram of t with argument s.

The system Λ_J can be considered as a simple programming language allowing to factorise out subprograms occurring with the same argument, i.e., to move out several occurrences of a generalised application (r, s) to a same more external position.

Example 12. *Consider the term* $T ::= t\{^{y}/_{r(s,x.t')}\}\{^{y'}/_{r(s,x.t')}\}$ *with two subterm* $r(s, x.t')$ *and* $r(s, x.t'')$*, where all variables of* r *and* s *are free in* T*.* T *can be converted to* $T' = r(s, x.t\{^{y}/_{t'}\}\{^{y'}/_{t''}\})$*.*

In this latter term the pseudo-application (r, s) *has been factorized out to the exterior of the term, and the variable* x *bound to this pseudo-application is then used twice.*

The normalization function should perform this factorisation of pseudo-applications, but it is not the case using the algorithm of the last chapter modified as above. To allow the algorithm to deliver factorized terms, we use the following idea.

When normalizing a term t, if we know that a pseudo-application (r, s) will actually occur in the normalized term, then one can begin by introducing a generalized application, the normalized function nf would then read:

$$\mathsf{nf}(t) = r(s, z. \downarrow [\![t]\!]_{\uparrow})$$

Of course we have now to modify the function $\uparrow$ so that it does not introduce a second time the pseudo application (r, s). This can be achieved in this particular case by defining $\uparrow$ to be

$$(\uparrow^{\rho\to\sigma} t)(a) ::= \begin{cases} z & \text{if } t = r \text{ and } \downarrow_\rho a = s \\ \uparrow^{\sigma} t((\downarrow_\rho a), z.z) & \text{otherwise} \end{cases}$$

To know if a pseudo-application will actually occur in a normalized term, one can use the NbE algorithm itself. We use environments ϵ associating a pseudo-applications to a variable. This environment ϵ is a partial function. In the environment ϵ, a pseudo application (r, s) can be associated with no variable, in this case the result of the application $\epsilon(r, s)$ of the environment to the pseudo-application is undefined, and is set to the element $\star$. Thus an environment is implemented as a total function $\epsilon : \mathsf{Tm} \times \mathsf{Tm} \to \mathsf{Var}_\perp$ where $\mathsf{Var}_\perp = \mathsf{Var} \cup \{\star\}$ is the extension of the set Var by the element $\star$.

Finally, we modify once again the function $\uparrow^{\rho\to\sigma} t$ so that applied to a value a, it continues the computation if the pseudo-application $(t, \downarrow a)$ is in the environment, or else interrupt the computation and return this pseudo-application as exception. Thus a first run of the NbE-algorithm in an empty environment may return a pseudo application as exception, in this case we run the NbE-algorithm anew in an updated environment where the pseudo-application got in the first run is associated to a new variable. This has to be repeated until no more exceptions are thrown, i.e., until a normal form is produced.

Hence the final definition of the function $\uparrow$ at arrow type is as follows:

$$(\uparrow^{\rho\to\sigma} t)(a) ::= \begin{cases} z & \text{if } \epsilon(t, \downarrow_\rho a) = z \\ \mathsf{throw}(t, \downarrow_\rho a) & \text{if } \epsilon(t, \downarrow_\rho a) = \perp \end{cases}$$

Remark 13. *As one needs to run the NbE algorithm for each creation of a pseudo-application, the implementation is particularly inefficient. One could easily optimize it, for example one could use continuations which would retain the computation done so far and avoid to do the same computation several times. The advantage of this use of exceptions is that it provides a simpler framework to reason about the program. Transposing a reasoning done for a simple program to its optimized version should be simpler than carrying it out directly for the optimized program.*

3.3.2 The Binding Variable Problem

The problem is just a bit more complicated than explained above because the *NbE* algorithm can generate subterms with new bound variables as in the definition of $\downarrow$ at arrow type:

$$\downarrow_{\rho\to\sigma} f ::= \lambda x.\downarrow f \uparrow x \qquad (x \text{ new})$$

If we only catch exceptions at the top-level, we would then possibly catch a pseudo-application with a variable which is not yet bound.

To prevent this situation, we will catch exceptions as soon as new variables are introduced, i.e., just after a λ. This will allow us to compare the free variables of the pseudo-application thrown as exception and the variable bound at this λ. If the bound variable does not occur free in the pseudo-application, then this pseudo-application is thrown further, otherwise a generalized application is created at this point.

We are now in position to write a first version of our algorithm. Similarly to the case of the simply typed λ-calculus, it is possible to restrict both the domain of the function $\uparrow$ and the codomain of the function $\uparrow$ to respectively variables and pure normal forms.

We shall introduce the function doAGA? which produces normal forms from pure normal forms, and finally observe that the domain of the environments can be restricted to those pseudo-applications whose first component is a variables and second component a pure normal form.

Code 5 (↑ and ↓). *Let* $\epsilon_0 \in \mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp$ *be an environment. The* reify *function* $\downarrow^\rho$: $(\mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp) \to [\![\rho]\!] \to \mathsf{PNF}$ *and* reflect *function* $\uparrow^\rho$: $(\mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp) \to \mathsf{Var} \to [\![\rho]\!]$ *are defined simultaneously by induction on the type* ρ*:*

$$\begin{aligned}
\downarrow^o_{\epsilon_0} x &::= x \\
\downarrow^{\rho\to\sigma}_{\epsilon_0} f &::= \lambda x.\mathsf{doAGA?}\ ((\mathbb{\lambda}\epsilon.\downarrow^\sigma_\epsilon f(\uparrow^\rho_\epsilon x)), \{x\})\ \epsilon_0 \qquad x \text{ new} \\
\uparrow^o_{\epsilon_0} x &::= x \\
\uparrow^{\rho\to\sigma}_{\epsilon_0} x(a) &::= \mathsf{let}\ P ::= \downarrow^\rho_{\epsilon_0} a\ \mathsf{in} \\
&\qquad \mathsf{catch}(\uparrow^\sigma_{\epsilon_0} \epsilon_0(x,P))(\mathbb{\lambda}\perp.\mathsf{throw}(x,P))
\end{aligned}$$

together with the function $\mathsf{doAGA?} : ((\mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp) \to \mathsf{PNF}) \times \mathcal{P}(\mathsf{Var}) \to (\mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp) \to \mathsf{NF}$ *which inserts generalized application when needed:*

$$\mathsf{doAGA?}\ (f,\ X)\epsilon_0 ::= \mathsf{catch}(f\epsilon_0)$$

$$\mathbb{\lambda}(x,P).\begin{cases} \mathsf{throw}(x,P) & \text{if } x \notin X \text{ and } X \cap \mathsf{FV}(P) = \emptyset \\ x(P, y.\mathsf{doAGA?}(f,\ (\{y\} \cup X))(\epsilon_0{}^{(x,P)\mapsto y})) & \text{otherwise} \end{cases}$$

The auxiliary function doAGA? (for "do A Generalized Application") introduces generalized applications as soon as the bound variables of a pseudo-application are bound.

The normalization function for a term r is obtained by applying the function doAGA? to $\mathbb{\lambda}\epsilon \downarrow^\epsilon_\rho [\![r]\!]_{\uparrow_\epsilon}$ and all free variables of r in the empty environment (i.e., the environment everywhere undefined $\mathbb{\lambda}t.\perp$).

Definition 3.11 (normalization function). *We define the normalization function* $\mathsf{nf} : \mathsf{Tm} \to \mathsf{Tm}$ *by :*

$$\mathsf{nf}(t) = \mathsf{doAGA?}(\mathbb{\lambda}\epsilon \downarrow^\epsilon_\rho [\![r]\!]_{\uparrow_\epsilon}, \mathsf{FV}(t))(\mathbb{\lambda}t.\perp)$$

3.4 Formalization

We now precisely define the notions we have let informal in the previous sections: the environment needed to generate new names, the environment associating pseudo-applications to variables and the handling of exceptions containing pseudo-applications. To be correctly threaded through the *NbE*-algorithm both of these environments and the exceptions have to be incorporated in the interpretation. To capture this notion in pure functional settings, we will use a combination of monads (a reader monad for each environment, and an exception or error monad for exceptions).

Name Generation Environment

The name generation is dealt with as in the first chapter by defining a name generation environment. We restate the definition here.

Definition 3.12 (Name Generation Environment)**.** *We define a set of* name generation environment *or set of environment for short, as a set E together with an update function $(-)^- : E \to \mathsf{Var} \to E$ and an access function $\mathsf{new} : E \to \mathsf{Var}$. The extension of $e^- : V \to E$ to a function from a list of variables $e^- : \mathbf{L}(V) \to E$ is defined in a canonical way by*

$$e^{x,\overrightarrow{x}} ::= (e^x)^{\overrightarrow{x}}$$
$$e^{\varepsilon} ::= e.$$

The function $(-)^-$ and new have moreover to satisfy the following property for all $e \in E$:

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{x,\overrightarrow{x}}) \qquad (\dagger)$$

Notation 14. *The condition that the new function applied to an environment $e \in E$ should never return a given variable x can be expressed by:*

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{\overrightarrow{x}})$$

We will abbreviate this condition by $x \notin e$. In the same way we will abbreviate for a given set of variable X, $\forall\, x \in X, x \notin e$ by $X \notin e$.

Postdiction Environment

We call the environment associating a pseudo-application to a variable a postdiction environment because on the one hand, this environment can be seen as an oracle predicting in a run of the NbE algorithm with which variables a pseudo-application has to be replaced, but on the other hand these predictions are obtained from previous runs of the NbE-algorithms.

Definition 3.13 (Postdiction Environment)**.** *We define the* postdiction environment $\mathbf{P}$ *as the set $\mathsf{Var} \times \mathsf{PNF} \to \mathsf{Var}_\perp$ together with an update function $(-)^- : \mathbf{P} \to ((\mathsf{Var} \times \mathsf{PNF}) \times \mathsf{Var}) \to \mathbf{P}$ defined by:*

$$\epsilon^{(x,P)\mapsto y}(x', P') ::= \begin{cases} y & \text{if } x = x' \text{ and } P =_{\pi_c^e} P' \\ \epsilon(x'P'), & \text{otherwise} \end{cases}$$

Interpretation Monad

We will now define the monad used in the interpretation, a computation should take place in a name environment E and a postdiction environment $\mathbf{P}(= (\mathsf{Var} \times \mathsf{PNF}) \to \mathsf{Var}_\perp)$, and it can be aborted by raising an exception containing a pair of a variable and a pure normal form. Hence the specification of a computation with result within a set A is:

$$\mathbf{T}A ::= \mathbf{St}_E(\mathbf{St}_\mathbf{P}(A_{\perp(\mathsf{Var}\times\mathsf{PNF})})) = E \to ((\mathsf{Var} \times \mathsf{PNF}) \to \mathsf{Var}_\perp) \to (\mathsf{Var} \times \mathsf{PNF} + A)$$

The unit and multiplication of the monad of the interpretation are defined quite naturally from the definition of the unit and multiplication of its monad components.

Definition 3.14 (Interpretation monad $\mathbf{T} ::= \mathbf{St}_E \circ \mathbf{St}_\mathbf{P} \circ -_{\perp(\mathsf{Ne})}$)**.** *The (set operator of) the interpretation monad* $\mathbf{T}$ *is defined by*

$$\mathbf{T}A ::= \mathbf{St}_E(\mathbf{St}_\mathbf{P}(A_{\perp(\mathsf{Ne})})) \ (= E \to (\mathsf{Ne} \to \mathsf{Var}_\perp) \to \mathsf{Ne} + A)$$

Given $m \in \mathbf{T}A$ *and* $f \in A \to \mathbf{T}B$*, we define the unit* $\nu : A \to \mathbf{T}A$ *and multiplication* $\star : \mathbf{T}A \to (A \to \mathbf{T}B) \to \mathbf{T}B$ *of the interpretation monad by:*

$$\nu^{\mathbf{T}A} ::= \nu^{\mathbf{St}_E} \circ \nu^{\mathbf{St}_\mathbf{P}} \circ \nu^{A_{\perp(\mathsf{Ne})}}$$
$$(m \star^\mathbf{T} f)(e)(\epsilon) ::= m(e)(\epsilon) \star^{-\perp(E)} \lambda a.f(a)(e)(\epsilon)$$

The update operations for the name environment monad and the postdiction monad extend component-wise to the interpretation monad.

Definition 3.15 (update for the Name Generation Monad)**.** *We define an update function* $(-)^- : \mathbf{T}(A) \to \mathsf{Var} \to \mathbf{T}(A)$ *operation by extending the update operation for the name generation monad by:*

$$m^v(e) ::= m(e^v)$$

Definition 3.16 (update for the Postdiction Monad)**.** *We define an update function* $(-)^- : \mathbf{St}_{\mathbf{P}(A)} \to (\mathsf{Ne} \times \mathsf{Var}) \to \mathbf{St}_{\mathbf{P}(A)}$ *operation by extending the update operation for postdiction environment by:*

$$m^{(x,P)\mapsto y}(\epsilon) ::= m(\epsilon^{(x,P)\mapsto y})$$

This update operation extends in turn into an operation $(-)^- : \mathbf{T}A \to (\mathsf{Ne} \times \mathsf{Var}) \to \mathbf{T}A$ *to the interpretation monad just in the same way by:*

$$m^{(x,P)\mapsto y}(e)(\epsilon) ::= m(e)(\epsilon^{(x,P)\mapsto y})$$

The interpretation is a call by value interpretation for computations in the monad $\mathbf{T}$:

Definition 3.17 (interpretation)**.** *The* call-by-value interpretation *of the* Λ_J*-calculus is given on types by*

$$\begin{aligned} [\![o]\!] &::= \mathsf{Var} \\ [\![\tau_1 \to \tau_2]\!] &::= [\![\rho]\!] \to \mathbf{T}([\![\sigma]\!]) \end{aligned}$$

Given a valuation η *on a context* Γ *(*$\eta \vDash \Gamma$*), the* interpretation of a typed term $\Gamma \vdash t : \tau$ *is the monadic value* $[\![t]\!]_\eta \in \mathbf{T}([\![\tau]\!])$ *defined by:*

$$\begin{aligned} [\![x]\!]_\eta &::= \nu(\eta(x)) \\ [\![r(s, x.t)]\!]_\eta &::= [\![r]\!]_\eta \star \mathbb{\lambda} f.[\![s]\!]_\eta \star \mathbb{\lambda} a.fa \star \mathbb{\lambda} v.[\![t]\!]_{\eta, x \mapsto v} \\ [\![\lambda x.r]\!]_\eta &::= \nu(\mathbb{\lambda} a.[\![r]\!]_{\eta, x \mapsto a}) \end{aligned}$$

Code 6 (call-by-value *NbE*)**.** *The function* $\downarrow_\tau : [\![\tau]\!] \to \mathbf{T}(\mathsf{PNF}_\tau)$ *and* $\uparrow^\tau : \mathsf{Var}_\tau \to [\![\tau]\!]$ *are defined simultaneously by:*

$$\begin{aligned} \downarrow_o x &::= \nu(x) \\ \downarrow_{\rho\to\sigma} f &::= \mathsf{new} \star \mathbb{\lambda} x.\mathsf{doAGA?}(f(\uparrow^\rho x) \star \downarrow_\sigma, \{x\})^x \star \mathbb{\lambda} e.\nu(\lambda x.e) \\ \uparrow^o x &::= x \\ \uparrow^{\rho\to\sigma} x(a) &::= \mathbb{\lambda} k\ \epsilon.(\downarrow_\rho a)k\epsilon \star \mathbb{\lambda} P. \begin{cases} \mathsf{throw}(x, P) & \text{if } \epsilon(x, P) = \bot \\ (\uparrow^\sigma y)\epsilon & \text{if } \epsilon(x, P) = y \end{cases} \end{aligned}$$

where the auxiliary function $\mathsf{doAGA?} : \mathbf{T}(\mathsf{PNF}) \to \mathcal{P}(\mathsf{Var}) \to \mathbf{T}(\mathsf{NF})$ *is defined by*

$$\begin{aligned} &\mathsf{doAGA?}(f, X) ::= \mathsf{catch}\, f \\ &\qquad \mathbb{\lambda}(x, P).\ \mathsf{if} \quad (\{x\} \cup \mathsf{FV}(P)) \cap X = \emptyset\ \mathsf{then}\ \mathsf{throw}(x, P) \\ &\qquad\qquad \mathsf{else} \quad \mathsf{new} \star \mathbb{\lambda} y. \\ &\qquad\qquad\quad \mathsf{doAGA?}(f^{x, P \mapsto y}, X \cup \{y\})^y \star \mathbb{\lambda} t. \\ &\qquad\qquad\qquad \mathsf{if} \quad y \notin \mathsf{FV}(t) \\ &\qquad\qquad\qquad \mathsf{then} \quad \nu(t) \\ &\qquad\qquad\qquad \mathsf{else} \quad \nu(x(p, y.t)) \end{aligned}$$

The normalization function is obtained by applying the function doAGA? to $[\![r]\!]_\uparrow \star \downarrow_\tau$ in an initial name environment e_r containing the variables free in r, and an empty postdiction environment (i.e., the function everywhere undefined $\mathbb{\lambda} n.\bot$)

Code 7. *Given a typed term* $\Gamma \vdash r : \tau$*, the normalization of the term* r *is given by:*

$$\mathsf{nf}^\tau(r) ::= \mathsf{doAGA?}([\![r]\!]_\uparrow \star \downarrow_\tau, e_r)(\mathbb{\lambda} n.\bot)$$

The claims of this section state the correctness and completeness of the algorithm (6) above.

Claim 1 (Correctness)**.**

$$r =_{\beta\eta} \mathsf{nf}(r)$$

Claim 2 (Completeness)**.**

$$r =_{\beta\eta} s \Longrightarrow \mathsf{nf}(r) =_{\mathsf{cc}} \mathsf{nf}(s)$$

The results in the previous chapter (2) were developed to be a basis to prove these results. A first exploration lead us to use logical monadic relations whose monadic part itself is defined inductively. This monadic part has to encompass the notion of fresh variables, as in the previous chapter, but also the notion of attempts needed to run the *NbE* until no exceptions are raised (the base case corresponding to a monadic value returning directly a value, and the step case corresponding to a monadic value returning an exception.) For time reason, we have let a formal study for further research. The conceptual simplicity of the approach using exceptions allowed us to implement the *NbE* algorithm in the pure functional language Haskell without using any primitive operators and we are confident that proofs of the claims (3) and (4) above will for the same reason be feasible.

Chapter 4

Sums

This section is dedicated to the study of the extension of the simply typed λ-calculus by sum types and to the design of a decision algorithm for its theory.

Although the extension itself is seemingly simple (after all we are only adding the possibility to do case analysis into the system), the study of its theory is extremely complex. For a general introduction of the difficulties encountered, the reader may consult [39]. These difficulties depend on which kind of rules one wants to add besides β-reduction. For the extension by permutative conversion, the system is proved strongly normalizing in [38] by a CPS-translation, in [75] by a variant of the reducibility candidate method, and in [55] by an original syntax directed characterisation of the strong normalizable terms. For the extension to a full extensional system, the permutation of independent `case` expressions prevents the design of a reasonable system with the confluence and strong normalization property. Nevertheless, Neil Ghani proposes a proof of decidability entirely based on rewriting theory [45].

Another way, is to use normalization by evaluation. The first algorithm of normalization by evaluation for a system with sum types is given by Olivier Danvy in [37], using continuation and control operators. Andrzej Filinski presents an algorithm in [43], as well based on continuations and control operators, but in a monadic setting, which is of particular interest to us.

In [4], Thorsten Altenkirch, Peter Dybjer, Martin Hofmann and Phil Scott prove constructively the existence of an NbE algorithm, able to decide the whole extensional theory of a simply typed lambda extended with sum types, and hence a direct program extraction from a formalisation of the proof should provide the algorithm. An algorithm for an extension by a Boolean type together with the proof of its correctness and completeness is given in [5]. The challenge to give a practical NbE algorithm for a system with general sum types, and for a conversion stronger than merely beta was first taken on by Vincent Balat and Olivier Danvy in [11]. This approach using continuations and control operators is then further pursued by Vincent Balat in [9] and his coauthors Roberto Di Cosmo, and Marcelo Fiore in [10]. In

these latter works, the algorithm is optimized and the set of normal forms resulting from the algorithm is further constrained, and this allows to design an algorithm for the whole extensional conversion of the calculus.

Our work is based both on the approach developed by Altenkrich and his coauthors and Balat and his coauthors, we will present an algorithm based on normalization by evaluation for the normalization of terms in a system with sum types and extensional conversions. However, our algorithm is based on the notion of exceptions, which we consider as a much simpler framework than continuations (exceptions can be seen as trivial continuations).

Moreover we do not use any control operators or imperative effects such as assignment. Although for the commodity of a first exposition we will use a pseudo programming language where exceptions are a built-in feature, we will eventually give the algorithm in a pure functional style, i.e., in a monadic setting in the continuation of Filinski [43], but for a stronger conversion relation.

4.1 System

In this first section we present the system Λ_+, which is an extension of the simply typed λ-calculus, Λ, by sum type. A sum type of two types ρ and σ will be interpreted as a disjoint union of the interpretations of ρ and σ. We do not consider product type, which could be interpreted as a cartesian product, because on the one hand, the essential difficulties are related to sum types and product types would not introduce much more, and on the other hand a presentation including products would be heavier.

Types and terms

The sum type constructor collates two types in a new one.

Definition 4.1 (Types)**.** *Given a base type o*

$$\mathsf{Ty} \ni \rho, \sigma ::= o \mid \rho \to \sigma \mid \rho + \sigma$$

The sum type comes equipped with two distinct injection constants and a case analysis operator. A simple example is the type of Boolean, which can be seen as a sum type of two copies of the unit type. In this case, an injection from the unit type can be seen as a constant, either true or false. The case analysis operator would then be simply an if operator. In general, the sum type is built from two types with more than one inhabitant, the particular inhabitant is then relevant and the case analysis operator introduces a binding in each branch which allows to use it.

Definition 4.2 (Terms)**.** *Given a countable infinite set of terms variables* Var*, the set of terms* Tm *of the simply typed* λ*-calculus with sum type is defined inductively by:*

$$\mathsf{Tm} \ni r, s, t ::= x \mid \lambda x^{\rho}.r \mid rs \mid \mathsf{in}_i\, r \mid \mathsf{case}(r, x^{\rho}.s, y^{\sigma}.t)$$

where $x \in \mathsf{Var}$

Remark and notation 2. *We have chosen to express our system in Church style, which means that types of bound variables are explicitly given at the binding symbol. The dot after a variable in a branch of a* case *term binds the variable in the branch, hence this variable is written with its type. However, for readability, we will allow us to omit these types when they are clear from the context.*

Notation 15. *Because a term containing case symbol* case *can become very large, we will use sometimes an alternative vertical notation* $\mathsf{case}\begin{pmatrix} r, & x \cdot s, \\ & y \cdot t \end{pmatrix}$ *for* $\mathsf{case}(r, x.s, y.t)$.

Typing

The extension of the typing system from the Λ-calculus to the Λ_+ is defined as follows:

Definition 4.3 (Typing)**.** *The* typing relation *is a ternary relation between contexts, terms and types and is defined inductively by*

$$\frac{(x : \rho) \in \Gamma}{\Gamma \vdash x : \rho}\ (\textsc{Var})$$

$$\frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^{\rho}.r : \rho \to \sigma}\ (\to\mathrm{I}) \qquad \frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}\ (\to\mathrm{E})$$

$$\frac{\Gamma \vdash r : \rho_i}{\Gamma \vdash \mathsf{in}_i\, r : \rho_0 + \rho_1}\ (+\mathrm{I})$$

$$\frac{\Gamma \vdash r : \rho_0 + \rho_1 \qquad \Gamma, x_0 : \rho_0 \vdash s_0 : \sigma \qquad \Gamma, x_1 : \rho_1 \vdash s_1 : \sigma}{\Gamma \vdash \mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) : \sigma}\ (+\mathrm{E})$$

In the terminology of natural deduction, the typing rules are separated into two classes: those where a type constructor (here $\to$ or $+$) appears in a premise are called elimination rules, whereas the ones where the type constructor appears in the conclusion are called introduction rules. The premiss of an elimination rule carrying the eliminated type constructor is called a major premiss, and the others are called minor premises.

Conversions

We will define the theory of the Λ_+-calculus, as a list of axioms. The actual conversions are obtained from these axioms as the smallest congruence relation on the term structure containing these axioms, i.e., a relation which is an equivalence relation and closed under term formation rules.

We begin by defining what is the contextual closure of an arbitrary relation in the Λ_+-calculus. The name of the conversion R in the definition below can be for example β, η, π or a combination of those.

Definition 4.4 (Contextual closure)**.** *A relation $=_R$ is contextually closed if the following rules hold.*

$$\frac{r =_R s}{rt =_R st}\ (R\text{-AppL}) \qquad \frac{r =_R s}{tr =_R ts}\ (R\text{-AppR}) \qquad \frac{r =_R s}{\lambda x.r =_R \lambda x.s}\ (R\text{-}\xi)$$

$$\frac{r =_R s}{\mathsf{in}_0\, r =_R \mathsf{in}_0\, s}\ (R\text{-Inj0}) \qquad \frac{r =_R s}{\mathsf{in}_1\, r =_R \mathsf{in}_1\, s}\ (R\text{-Inj1})$$

$$\frac{r =_R s}{\mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) =_R \mathsf{case}(s, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1)}\ (R\text{-Case1})$$

$$\frac{s_0 =_R s_0'}{\mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) =_R \mathsf{case}(r, x_0^{\rho_0}.s_0', x_1^{\rho_1}.s_1)}\ (R\text{-Case2})$$

$$\frac{s_1 =_R s_1'}{\mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) =_R \mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1')}\ (R\text{-Case3})$$

The axioms for β-conversions are defined as usual.

Definition 4.5 (β-conversion)**.** *The axioms of the β-conversion are given by:*

$$(\lambda x.r)s =_\beta r\{^s/_x\} \qquad (\beta_\to)$$
$$\mathsf{case}(\mathsf{in}_i\, r, x_0.s_0, x_1.s_1) =_\beta s_i\{^r/_{x_i}\} \qquad (\beta_+)$$

As mentioned in the introduction, the theory augmented by permutative conversion has already been studied by several authors. This theory allows to identify more terms than the β-conversion alone and is useful in a natural deduction formulation of logic with disjunction because they allow to recover a variant of the subformula property (see [88] or [48] for details).

Definition 4.6 (π-conversion)**.** *The axioms of the* permutative conversion *are given by*

$$\mathsf{case}(\mathsf{case}\begin{pmatrix} r, & x_0 \cdot s_0, \\ & x_1 \cdot s_1 \end{pmatrix}, y_0.t_0, y_1.t_1) =_\pi \mathsf{case}\begin{pmatrix} r, & x_0 \cdot \mathsf{case}(s_0, y_0.t_0, y_1.t_1), \\ & x_1 \cdot \mathsf{case}(s_1, y_0.t_0, y_1.t_1) \end{pmatrix} \tag{π_1}$$

$$\mathsf{case}(r, x_0.s_0, x_1.s_1)t =_\pi \mathsf{case}(r, x_0.s_0t, x_1.s_1t) \tag{π_2}$$

where in the axiom π_2, neither x_0 nor x_1 occurs free in t.

Considering the type derivations as natural deduction proofs, a permutative conversion permutes an elimination rule in a major premiss of an elimination rule upward over the minor premiss. For example, the second rule above reads with type derivation as follows (with neither x_0 nor x_1 occurring free in t):

$$\dfrac{\dfrac{\Gamma \vdash r : \rho_0 + \rho_1 \qquad \Gamma, x_0 : \rho_0 \vdash s_0 : \sigma \to \tau \qquad \Gamma, x_1 : \rho_1 \vdash s_1 : \sigma \to \tau}{\Gamma \vdash \mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1) : \sigma \to \tau}\,(+\text{-E}) \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash \mathsf{case}(r, x_0^{\rho_0}.s_0, x_1^{\rho_1}.s_1)t : \tau}\,(\to\text{-E})$$

$$=_\pi$$

$$\dfrac{\Gamma \vdash r : \rho_0 + \rho_1 \qquad \dfrac{\Gamma, x_0 : \rho_0 \vdash s_0 : \sigma \to \tau \quad \Gamma, x_0 : \rho_0 \vdash t : \sigma}{\Gamma, x_0 : \rho_0 \vdash s_0t : \tau} \qquad \dfrac{\Gamma, x_1 : \rho_1 \vdash s_1 : \sigma \to \tau \quad \Gamma, x_1 : \rho_1 \vdash t : \sigma}{\Gamma, x_1 : \rho_1 \vdash s_1t : \tau}}{\Gamma \vdash \mathsf{case}(r, x_0^{\rho_0}.s_0t, x_1^{\rho_1}.s_1t) : \tau}$$

To sum up, given a term r, the π-conversion pulls out to the exterior of the term a test term s of a case-subterm $\mathsf{case}(s, x.t, y.u)$ only if it occurs in certain position of the term r. This occurrence conditions of the case-subterm can be relaxed, leading to the generalisation π^e below of the π conversion. We will call the π^e the extensional permutative conversion, because together with the η-conversion, it provides an extensional conversion relation.

Definition 4.7 (π^e-conversion)**.** *The axiom of the extensional* permutative conversion *is given by:*

$$r\{^{\mathsf{case}(s, y_0.t_0, y_1.t_1)}/_x\} =_{\pi^e} \mathsf{case}(s, y_0.r\{^{t_0}/_x\}, y_1.r\{^{t_1}/_x\}) \tag{π}$$

where neither y_0 nor y_1 occurs free in r.

Remark 14. *We use a capture avoiding substitution in the definition of the π^e-conversion for sum type above so that the free variables of s can not be bound in r.*

We define now the η-conversions.

Definition 4.8 (η-conversion)**.** *The axioms of η-conversion, the axioms $\eta_\to$ and η_+ are given by*

$$r =_{\eta_\to} \lambda x.rx \qquad (\eta_\to)$$
$$r =_{\eta_+} \mathsf{case}(r, x_0.\mathsf{in}_0\, x_0, x_1.\mathsf{in}_1\, x_1) \qquad (\eta_+)$$

where in the axiom $\eta_\to$, x does not occur free in r.

The η_+-conversion is the analogue of the $\eta_\to$-conversion for sum types. It takes an arbitrary terms of sum types and converts it to a term involving the term constructor for sum type, i.e., injections.

Remark 15. *A difference with the Λ_J-system is that it is possible to present the η_+- and π-conversion in a unique rule:*

$$r\{^s/_x\} =_{\eta\pi} \mathsf{case}(s, x.r\{^{\mathsf{in}_0\, x}/_x\}, x.r\{^{\mathsf{in}_1\, x}/_x\})$$

In this case, we have to make sure that in the typing derivation of $r\{^s/_x\}$, the term s is typed with a sum type for the right hand side to be well-typed. To the contrary, checking the applicability of the π^e-conversion is syntactical[1]*.*

The existence of a strongly normalizing and confluent rewriting system generating this theory is in fact very unlikely (the only work based on rewriting theory to decide this theory [45] does not devise a strongly normalizing system). It is probably for this reason that until very recently, the scientific community has only focused on some sub-conversions of the whole theory.

4.2 Extended Conversion and Normal Forms

In the definition of the π^e-conversion above, because of the substitution involved there, we have to consider the term at an arbitrary depth. We give here a local decomposition of this conversion. Apart from the fact that it is always interesting to give a local characterisation of a global phenomena, this decomposition will help us to explain difficulties inherent to the extensional calculus with sum types and to justify the normal form of the term as computed by the *NbE* algorithm.

[1] Only in the case the variable x does not actually occurs in the term r of the left hand side of the π^e-conversion rule above, we have to check that the term $\mathsf{case}(s, y_0.t_0, y_1.t_1)$ to be substituted for is typable.

Lemma 21 (π^e*-conversion* (local version))**.** *The extensional* permutative conversion *can be defined by the following axioms*

$$\mathsf{case}\left(\mathsf{case}\left(r,\ \begin{array}{l} x_0\cdot s_0,\\ x_1\cdot s_1\end{array}\right),\ \begin{array}{l} y_0\cdot t_0,\\ y_1\cdot t_1\end{array}\right) =_{\pi_1^e} \mathsf{case}\left(r,\ \begin{array}{l} x_0\cdot \mathsf{case}(s_0, y_0.t_0, y_1.t_1),\\ x_1\cdot \mathsf{case}(s_1, y_0.t_0, y_1.t_1)\end{array}\right)$$
$$\textit{where } x_0, x_1 \notin \mathsf{FV}(t_0, t_1)$$

$$\mathsf{case}(r, x_0.s_0, x_1.s_1)t =_{\pi_2^e} \mathsf{case}(r, x_0.s_0 t, x_1.s_1 t)$$
$$\textit{where } (x_0, x_1 \notin \mathsf{FV}(t))$$

$$t\,\mathsf{case}(r, x_0.s_0, x_1.s_1) =_{\pi_3^e} \mathsf{case}(r, x_0.t\,s_0, x_1.t\,s_1)$$

$$\mathsf{in}_i\,\mathsf{case}(r, x_0.s_0, x_1.s_1) =_{\pi_4^e} \mathsf{case}(r, x_0.\mathsf{in}_i\,s_0, x_1.\mathsf{in}_i\,s_1)$$

$$\lambda x.\mathsf{case}(r, x_0.s_0, x_1.s_1) =_{\pi_5^e} \mathsf{case}(r, x_0.\lambda x.s_0, x_1.\lambda x.s_1)$$
$$\textit{where } x \notin \mathsf{FV}(r)$$

$$\mathsf{case}(r, x_0.\mathsf{case}(s, y_0.t_0, y_1.t_1), x_1.u) =_{\pi_6^e} \mathsf{case}\left(s,\ \begin{array}{l} y_0\cdot \mathsf{case}(r, x_0.t_0, x_1.u),\\ y_1\cdot \mathsf{case}(r, x_0.t_1, x_1.u)\end{array}\right)$$
$$\textit{where } y_0, y_1 \notin \mathsf{FV}(u), x_0 \notin \mathsf{FV}(s)$$

$$\mathsf{case}(r, x_0.u, x_1.\mathsf{case}(s, y_0.t_0, y_1.t_1)) =_{\pi_7^e} \mathsf{case}\left(s,\ \begin{array}{l} y_0\cdot \mathsf{case}(r, x_0.u, x_1.t_0),\\ y_1\cdot \mathsf{case}(r, x_0.u, x_1.t_1)\end{array}\right)$$
$$\textit{where } y_0, y_1 \notin \mathsf{FV}(u), x_1 \notin \mathsf{FV}(t)$$

$$\mathsf{case}(r, x_0.\mathsf{case}(r, x_0.s_0, x_1.s_1), x_1.t) =_{\pi_8^e} \mathsf{case}(r, x_0.s_0, x_1.t)$$

$$\mathsf{case}(r, x_0.s, x_1.\mathsf{case}(r, x_0.t_0, x_1.t_1)) =_{\pi_9^e} \mathsf{case}(r, x_0.s, x_1.t_1)$$

$$r =_{\mathsf{is}} \mathsf{case}(s, x_0.r, x_1.r)$$
$$\textit{where } x_i \notin \mathsf{FV}(r)$$

The seven first conversions (π_1^e to π_7^e) above are just instantiation of the general π-conversion. The structural closure of these rule generates the general π^e-conversion where substitution is restricted to one occurrence of a variable. The conversion π_8^e and π_9^e handles the case of multiple occurrences whereas the conversion is handles the case of zero occurrence.

We will follow the scheme of the last chapter and describe progressively the normal forms of our system, i.e., we will give some sensible criteria based on the local definition of the π^e-conversion. Whereas the NbE algorithm is not based on rewriting theory, these normal forms can be understood, up to a certain stage, as the irreducible terms of a rewriting system obtained by orienting the conversions above.

The normal forms that we will eventually present are essentially the same as in the works [9] of Balat and [4] of Altenkirch, Dybjer, Hofmann and Scott. In [4] a normal form is not merely a term but a set of terms where terms that are convertible with the conversions that create circularity are identified. Although, this shortens the presentation we prefer to stick to a more syntactical presentation of normal forms

as in [9].

First we want our normal forms to be a restriction of the existing normal forms for the traditional permutative conversion (given as π in the system section or π_1^e and π_2^e in the alternative definition of π^e-conversion above) in order to retain the subformula property. The set of normal forms NF_{Λ_+} for permutative conversions π and β-conversions are defined inductively together with a set of neutral terms Ne_{Λ_+} as follows:

$$\begin{aligned}\mathsf{Ne}_{\Lambda_+} \ni n &::= x \mid nN\\ \mathsf{NF}_{\Lambda_+} \ni N &::= n \mid \lambda x.N \mid \mathsf{in}_i\, N \mid \mathsf{case}(n, x_1.N_1, x_0.N_0)\end{aligned}$$

The rule π_3^e, read from left to right, transforms an application term $t\,\mathsf{case}(r, x_0.s_0, x_1.s_1)$ in a case term $\mathsf{case}(r, x_0.t\,s_0, x_1.t\,s_1)$. This is already the effect of the traditional permutative conversion π, so that we keep this direction. Similarly, the rule π_4^e move injections $\mathsf{in}_i\,\mathsf{case}(r, x_0.s_0, x_1.s_1)$ inside a case term $\mathsf{case}(r, x_0.\mathsf{in}_i\,s_0, x_1.\mathsf{in}_i\,s_1)$. The normal forms NF need once more to be split and are defined simultaneously with neutral forms Ne and PNF.

$$\begin{aligned}\mathsf{Ne}_{\Lambda_+} \ni n &::= x \mid nP\\ \mathsf{PNF}_{\Lambda_+} \ni P &::= n \mid \lambda x.N \mid \mathsf{in}_i\, P\\ \mathsf{NF}_{\Lambda_+} \ni N, N_0, N_1 &::= P \mid \mathsf{case}(n, x_1.N_1, x_0.N_0)\end{aligned}$$

The conversion π_5^e says that a λ-abstraction can be moved inside a case term. We choose just as in the last chapter to move the binders λ as deep as possible in order to introduce variables just before their uses. For that we need to define a condition which express that a λ is already at the deepest possible position.

For example for the term $\lambda z.\mathsf{case}(r, x.s, y.t)$ this condition should state that the variable z should occur in r, otherwise λz could be moved in front of s and t. However, to check whether the variable z occurs in r is not enough, for if the term s is itself a *case* term, say $s = \mathsf{case}(r', x'.s', y'.t')$, and neither z nor x occurs in r' then we could move λz in a more internal position as follows:

1. first move the term r' to the outside by converting

   $$\lambda z.\mathsf{case}(r, x.\mathsf{case}(r', x'.s', y'.t'), y.t)$$

   into

   $$\lambda z.\mathsf{case}(r', x'.\mathsf{case}(r, x.s', y.t), y'.\mathsf{case}(r, y'.t', y.t))$$

2. and now we can move λz inside by converting

   $$\lambda z.\mathsf{case}(r', x'.\mathsf{case}(r, x.s', y.t), y'.\mathsf{case}(r, y'.t', y.t))$$

   into

   $$\mathsf{case}(r', x'.\lambda z.\mathsf{case}(r, x.s', y.t), y'.\lambda z.\mathsf{case}(r, y'.t', y.t))$$

Therefore we define the deepest λ condition $\mathsf{DL}(\mathbb{X}, r)$, for a set of variables $\mathbb{X}$ and a term r, this condition is true if r is a variable or a λ-abstraction and if r is a case term $\mathsf{case}(n, x_0.N_0, x_1.N_1)$, it is defined as follows:

$$\mathsf{DL}(\mathbb{X}, \mathsf{case}(n, x_0.N_0, x_1.N_1)) ::= \mathbb{X} \cap \mathsf{FV}(n) \neq \emptyset \wedge \mathsf{DL}(\mathbb{X} \cup x_0, N_0) \wedge \mathsf{DL}(\mathbb{X} \cup x_1, N_1)$$

Conversion η and prenormal forms

As already noticed, η-conversions require to consider the types of the term involved. The rule of η-conversions, oriented as expansions (i.e., from left to right in the definition above), expand a term r into an abstraction or a case term according whether the type of r is an arrow type or a sum type.

Considered together with the β-conversion oriented in a reduction from left to right, one has to impose restrictions on the $\eta_{\to}$-expansion to prevent infinite sequence of reductions. Namely an $\eta_{\to}$-expansion should not expand a λ-abstraction nor the major premiss r of an application rs. Similarly an η_{+}-expansion should not expand an injection term $\mathsf{in}_i\, r$ nor the major premiss r of a case term $\mathsf{case}(r, x.s, y.t)$.

To obtain normal forms for the η-conversion, it suffices to restrict the formation of pure normal forms for neutral terms to those of ground type. We will need to further restrict these normal forms in the next section and call prenormal forms the normal forms obtained at this stage.

Definition 4.9 (prenormal forms)**.** *We define by simultaneous induction three relations $-\vdash_{\mathsf{Ne}} - : -$, $-\vdash_{\mathsf{PNF}} - : -$, and $-\vdash_{\mathsf{NF}} - : -$ between context, term and type,*

$$\frac{}{\Gamma \vdash_{\mathsf{Ne}} x : \rho} \qquad \frac{\Gamma \vdash_{\mathsf{Ne}} n : \rho \to \sigma \qquad \Gamma \vdash_{\mathsf{PNF}} P : \rho}{\Gamma \vdash_{\mathsf{Ne}} nP : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{Ne}} n : o}{\Gamma \vdash_{\mathsf{PNF}} n : o} \qquad \frac{\Gamma, x : \rho \vdash_{\mathsf{NF}} N : \sigma}{\Gamma \vdash_{\mathsf{PNF}} \lambda x.N : \rho \to \sigma} \qquad \frac{\Gamma \vdash_{\mathsf{PNF}} P : \rho_i}{\Gamma \vdash_{\mathsf{PNF}} \mathsf{in}_i\, P : \rho_0 + \rho_1}$$

$$\frac{\Gamma \vdash_{\mathsf{PNF}} P : \rho}{\Gamma \vdash_{\mathsf{NF}} P : \rho} \qquad \frac{\Gamma \vdash_{\mathsf{Ne}} n : \rho_0 + \rho_1 \qquad \Gamma, x : \rho_i \vdash_{\mathsf{NF}} N_i : \sigma}{\Gamma \vdash_{\mathsf{NF}} \mathsf{case}(n, x_0.N_0, x_1.N_1) : \sigma}$$

in the rule with conclusion $\Gamma \vdash_{\mathsf{PNF}} \lambda x.N : \rho \to \sigma$ we impose the condition

$$\mathsf{DL}(\{x\}, N) \qquad \text{(deepest } \lambda\text{)}$$

which is true if N is not a case term and otherwise defined by:

$$\mathsf{DL}(\mathbb{X}, \mathsf{case}(n, x_0.N_0, x_1.N_1)) ::= \mathbb{X} \cap \mathsf{FV}(n) \neq \emptyset \wedge \mathsf{DL}(\mathbb{X} \cup x_0, N_0) \wedge \mathsf{DL}(\mathbb{X} \cup x_1, N_1)$$

The condition (deepest λ) ensures that the case are introduced as soon as possible, i.e., as soon as variables occurring in the term to be tested are available, which means just after a binding symbols or at the top-level.

Notation 16. *We will write* $\mathsf{Ne}^{\rho}_{\Gamma}$ *(resp.* $\mathsf{NF}^{\rho}_{\Gamma}$ *and* $\mathsf{PNF}^{\rho}_{\Gamma}$*) for the set of terms* r *such that* $\Gamma \vdash_{\mathsf{Ne}} r : \rho$ *(resp.* $\Gamma \vdash_{\mathsf{NF}} r : \rho$ *and* $\Gamma \vdash_{\mathsf{PNF}} r : \rho$*) and* Ne^{ρ} *(resp.* NF^{ρ} *and* PNF^{ρ}*) for the set of terms* r *such that there exists a context* Γ *and* $r \in \mathsf{Ne}^{\rho}_{\Gamma}$ *(resp.* $r \in \mathsf{NF}^{\rho}_{\Gamma}$ *and* $r \in \mathsf{PNF}^{\rho}_{\Gamma}$*).*

Circular Conversions and Immediate Simplification

To design a decision algorithm for the whole conversion relation, one has to face a first problem, namely the inherent circularity of the π^e-conversion.

As one can see in the following example, this circularity problem cannot be solved by the orientation of the rewrite relation alone.

Example 13. *In the example below the term on the right hand side is obtained by pulling out the two occurrences of the subterm* r' *thanks to the conversion* π^e_6 *and* π^e_7*, then the term is simplified by using* π^e_8 *or* π^e_9*, and by reducing the generated* β*-redex:*

$$\mathsf{case}\begin{pmatrix} r, & x \cdot \mathsf{case}(r', x'.s', y'.t'), \\ & y \cdot \mathsf{case}(r', x'.s'', y'.t'') \end{pmatrix} = \mathsf{case}\begin{pmatrix} r', & x' \cdot \mathsf{case}(r, x.s', y.s''), \\ & y' \cdot \mathsf{case}(r, x.t', y.t'') \end{pmatrix}$$

with $x', y' \notin \mathsf{FV}(r)$ *and* $x, y \notin \mathsf{FV}(r')$*.*

The two terms of the example above have exactly the same structure, and although it may be possible to break this circularity, e.g. by introducing an order on terms (for example induced by the variables they contain), it seems artificial to select one of these terms as more normal than the other. In terms of programming if we have to do two tests which are independent from each other, there is no reason to choose to do a particular one first.

Hence, the normal forms can no longer be unique, their equality is no longer syntactic but has to be tested with respect to this conversions (i.e., π^e_6 to π^e_9).

Notation 17. *Due to this circularity, we will call the conversions* π^e_6 *to* π^e_9 circular permutative conversion *and write* π^e_c *for their union as relations.*

$$r =_{\pi^e_c} s ::= r =_{\pi^e_6} s \vee r =_{\pi^e_7} s \vee r =_{\pi^e_8} s \vee r =_{\pi^e_9} s$$

A second problem is that the *NbE* algorithm tends to produce terms which are in expanded form, i.e., the π^e-conversion is oriented from left to right. In particular this means that if x does not actually occur in r, the term $r\{^{\mathsf{case}(s,x_0.t_0,x_1.t_1)}/_{x}\}$ is converted into $\mathsf{case}(s, x_0.r, x_1.r)$ (is-*conversion*) .

Seen as a reduction from left to right this would lead immediately to an infinite loop. Besides, it does not make any sense to introduce superfluous test term as s

above.Hence the normal forms will have to be irreducible terms for the inverse reduction (oriented from right to left) known from Prawitz [78] as *immediate simplification reduction.*

$$\mathsf{case}(s, x_0.r, x_1.r) \longrightarrow_{\mathsf{is}} r \qquad (x_0, x_1 \notin \mathsf{FV}(r))$$

We will deal with these conversions in more detail after having defined a first version of the normalization algorithm for the prenormal forms defined above.

4.3 Informal Description

To explain the algorithm, we will proceed informally as in the chapter on the simply typed λ-calculus (2.2.1).

First we need an interpretation, we take the standard set theoretical interpretation where a sum type is interpreted as a disjoint union of its components and the ground type is interpreted as the set of neutral term Ne^o:

Definition 4.10 (Type Interpretation)**.** *We define the* interpretation $[\![\rho]\!]$ of a type ρ*, by induction on* $\rho \in \mathsf{Ty}$*:*

$$\begin{aligned} [\![o]\!] &::= \mathsf{Ne}^o \\ [\![\rho \to \sigma]\!] &::= [\![\rho]\!] \to [\![\sigma]\!] \\ [\![\rho + \sigma]\!] &::= [\![\rho]\!] + [\![\sigma]\!] \end{aligned}$$

where $[\![\rho]\!] \to [\![\sigma]\!]$ *is the full function space between the set* $[\![\rho]\!]$ *and* $[\![\sigma]\!]$*, and* $[\![\rho]\!] + [\![\sigma]\!]$ *is the disjoint union of the set* $[\![\rho]\!]$ *and* $[\![\sigma]\!]$*.*

The interpretation of types $[\![\mathsf{Ty}]\!]$ of the typed lambda calculus with sum types Λ_+ *is then:*

$$[\![\mathsf{Ty}]\!] = \bigcup_{\rho \in \mathsf{Ty}} [\![\rho]\!]$$

To define the interpretation of terms we need first to define the auxiliary notion of valuation.

Definition 4.11 (Valuation)**.** *Given a context* Γ*, we define a* valuation *on* Γ *to be a partial function* $\eta : \mathsf{Var} \to [\![\mathsf{Ty}]\!]_\perp$ *such that for* $x : \rho \in \Gamma$*, we have* $\eta(x)$ *is defined and* $\eta(x) \in [\![\rho]\!]$*. Given a context* Γ*, a valuation* η *on* Γ*, a variable* $y \notin \Gamma$ *and an element* $a \in [\![\sigma]\!]$*, we define a valuation* $(\eta, y \mapsto a)$ *on* $\Gamma \cup \{(y : \sigma)\}$*, called the extension of* η *by* $y \mapsto a$ *by,*

$$(\eta, y \mapsto a)(x) ::= \begin{cases} a & \text{if } x = y, \\ \eta(x) & \text{otherwise.} \end{cases}$$

Remark 16. *Valuation functions are partial functions represented as total functions from the set of variables* Var *into the set* $[\![\mathsf{Ty}]\!]_\perp = [\![\mathsf{Ty}]\!] + \{\star\}$*, i.e., the interpretation of types extended with an element* $\star$ *playing the rôle of an undefined value. In the following, whenever we will use a valuation applied to some variable, this variable will belong to the domain of definition of the valuation and the result will therefore be defined. And although, strictly speaking we should do a case distinction on the result to know wheither it is defined (an alternative would be to use an exception monad as presented in definition 1.3), we will consider it to be an element of* $[\![\mathsf{Ty}]\!]$*.*

The interpretation of a typed term is an element of the interpretation of its type.

Definition 4.12 (Term Interpretation)**.** *Finally, we define the* interpretation $[\![r]\!]_\eta$ of a term r*, whenever there is a context* Γ*, such that* $\Gamma \vdash r : \rho$ *and* η *is a valuation on* Γ*, to be an element of* $[\![\rho]\!]$*, by the following inductive definition:*

$$
\begin{aligned}
[\![x]\!]_\eta &::= \eta(x) \\
[\![\lambda x^\rho.r]\!]_\eta(v) &::= [\![r]\!]_{\eta, x\mapsto v} \\
[\![rs]\!]_\eta &::= [\![r]\!]_\eta([\![s]\!]_\eta) \\
[\![\mathsf{in}_i\, r]\!]_\eta &::= \iota_i [\![r]\!]_\eta \\
[\![\mathsf{case}(r, x^\rho.s, y^\sigma.t)]\!]_\eta &::= \begin{cases} [\![s]\!]_{\eta, x\mapsto a} & \text{if } [\![r]\!]_\eta = \iota_0\, a, \\ [\![t]\!]_{\eta, y\mapsto a} & \text{if } [\![r]\!]_\eta = \iota_1\, a \end{cases}
\end{aligned}
$$

4.3.1 Several Attempts

We want now to define a reify function $\downarrow$, from the interpretation of types to the set of terms. As previously, we will define it inductively on the type simultaneously with a function $\uparrow$ from the terms to the interpretation.

Let us try to extend to sum types the definition which was given in chapter 2 for ground and arrow types. An element of the interpretation of a sum type is either a left injection or a right injection, and hence the function $\downarrow$ can be directly extended by:

$$\downarrow_{\rho_0+\rho_1} \iota_i a ::= \mathsf{in}_i \downarrow_{\rho_i} a$$

The first difficulty comes when trying to extend the function $\uparrow$ at sum type. Let us look at the problem with an example:

Example 14 (the unsolvable problem)**.** *Let us assume we want to normalize the term* xy *of sum type with the following typing:*

$$x : o \to o + o, y : o \vdash xy : o + o$$

by $\downarrow [\![xy]\!]_\uparrow$*.*

We are faced with the problem to define the result of the function $\uparrow^{o+o}$ *applied to* xy*:*

$$\uparrow^{o+o} xy$$

According to the interpretation the result should lay in a disjoint union, but should it be a left or a right injection? The core idea is that a sensible answer is "I don't know".

Example 15 (a solution)**.** *The very fact that we failed to choose a value for the reflection of the term* xy*, will guide us in a second run of the normalization algorithm. This time we begin by a case distinction on* xy*, and in each branch of the alternative, we memorise whether* xy *corresponds to a left or a right injection.*

$$\mathsf{case}(xy, z.\downarrow^{xy\mapsto in_0 z} [\![xy]\!]_\uparrow, z.\downarrow^{xy\mapsto in_1 z} [\![xy]\!]_\uparrow)$$

The function $\downarrow^{xy\mapsto \mathsf{in}_0\, z}$ *follows the same definition as* $\downarrow$ *except that in the computation of* $\downarrow^{xy\mapsto \mathsf{in}_0\, z} [\![xy]\!]_\uparrow$*, a call to the function* $\uparrow xy$ *will always return* $\mathsf{in}_0 \uparrow z$*.*

What we just described can be quite naturally implemented with exceptions and environments.

Every function is evaluated within an environment which is a finite map from terms to the injection of a variable. In the first computation of the normalization function, we begin with the empty environment; this environment corresponds to our knowledge that at a certain point in the computation we are in a certain branch of a case and that the test term of the corresponding branch is the left or right injection of the bound variable introduced by the case.

If in a subcomputation, the function $\uparrow$ at sum type is applied to a term for which we do not know whether the result should be a left or right injection (i.e., the term is not in the environment), then we abort the computation by raising an exception containing this term.

In a second computation we use this term for a case distinction and in each branch we update the environment with the binding of the term to the left or the right injection.

4.3.2 The Binding Variable Problem

The problem is just a bit more complicated than explained above because the *NbE* algorithm can generate subterms with new bound variables as in the definition of $\downarrow$ at arrow type:

$$\downarrow_{\rho\to\sigma} f ::= \lambda x.\downarrow f \uparrow x$$

If we only catch exceptions at the top-level, we would then possibly catch a term with a variable which is not yet bound.

What we can do however is to catch exceptions as soon as the new variables are introduced, i.e., just after the binding symbols, the λ or the binding dots of a case expression.

We are now in position to write a first version of our algorithm:

The environment is implemented as a partial function $\epsilon : \mathsf{Ne} \to \mathsf{Var} + \mathsf{Var}$ taking a (neutral) term as argument and returning an injection of a variable x, $\mathsf{in}_0\, x$ or $\mathsf{in}_1\, x$. The update operation $-^{n \mapsto \mathsf{in}_i\, x}$ for the environment is defined by

$$\epsilon^{n \mapsto \mathsf{in}_i\, x} n' ::= \begin{cases} \mathsf{in}_i\, x & \text{if } n' = n, \\ \epsilon(n') & \text{otherwise.} \end{cases}$$

The function catch and throw have standard behaviours: In an expression $\mathsf{throw}(n)$, the function throw packs the term n into an exception and throw this exception. This exception is then caught in the first upper expression of the form $\mathsf{catch}\ E\ h$ (where $\mathsf{throw}(n)$ occurs in E), i.e., the expression $\mathsf{catch}\ E\ h$ evaluate to $h\, n$, the application of the handler h to the content n of the exception.

Code 8 (*NbE* for sum types)**.** *Let* $\epsilon_0 \in \mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp$ *be an environment. The function* $\downarrow^\rho$: $(\mathsf{Ne} \to (\mathsf{Var}+\mathsf{Var})_\perp) \to [\![\rho]\!] \to \mathsf{PNF}$ *and* $\uparrow^\rho$: $(\mathsf{Ne} \to (\mathsf{Var}+\mathsf{Var})_\perp) \to \mathsf{Ne} \to [\![\rho]\!]$ *are defined simultaneously by induction on the type* ρ*:*

$$\begin{aligned}
\downarrow^{o}_{\epsilon_0} n &::= n \\
\downarrow^{\rho\to\sigma}_{\epsilon_0} f &::= \lambda x.\mathsf{doACase?}(\mathbb{\lambda}\epsilon.\ \downarrow^{\sigma}_{\epsilon} f(\uparrow^{\rho}_{\epsilon} x), \{x\})\epsilon_0 && x \text{ new} \\
\downarrow^{\rho_0+\rho_1}_{\epsilon_0} \iota_i c &::= \mathsf{in}_i\ \downarrow^{\rho_i}_{\epsilon_0} c \\
\uparrow^{o}_{\epsilon_0} n &::= n \\
\uparrow^{\rho\to\sigma}_{\epsilon_0} n &::= \mathbb{\lambda} v.\ \uparrow^{\sigma}_{\epsilon_0} (n\ \downarrow^{\rho}_{\epsilon_0} v) \\
\uparrow^{\rho+\sigma}_{\epsilon_0} n &::= \begin{cases} \mathsf{let}\ \mathsf{in}_i\, x = (\mathsf{catch}\ \epsilon_0(n)\ \mathsf{throw}(n))\ \mathsf{in} \\ \iota_i\ \uparrow^{\rho_i}_{\epsilon_0} x \end{cases}
\end{aligned}$$

where doAcase? : $((\mathsf{Ne} \to (\mathsf{Var}+\mathsf{Var})_\perp) \to \mathsf{PNF}) \times \mathcal{P}(\mathsf{Var}) \to (\mathsf{Ne} \to \mathsf{Var}+\mathsf{Var}) \to \mathsf{NF}$ *is defined by*

$$\begin{aligned}
&\mathsf{doAcase?}(f, X)\epsilon_0 ::= \mathsf{catch}\ (f\epsilon_0) \\
&\quad \mathbb{\lambda} n.\mathsf{if}\ \mathsf{FV}(n) \cap X = \emptyset\ \mathsf{then}\ \mathsf{throw}(n) \\
&\quad\quad \mathsf{else\ case} \begin{pmatrix} n, & x \cdot \mathsf{doAcase?}(f, (X \cup \{x\}))\epsilon_0{}^{n \mapsto \mathsf{in}_0\, x}, \\ & x \cdot \mathsf{doAcase?}(f, (X \cup \{x\}))\epsilon_0{}^{n \mapsto \mathsf{in}_1\, x} \end{pmatrix}, && x \text{ new}
\end{aligned}$$

The auxiliary function doAcase? takes three arguments. Its first argument is a function taking itself as first argument an environment, its second argument is a set of variables, and its third argument is an environment (a function from neutral terms to an injection of a variable $\mathsf{in}_i\, x$). In the expression $\mathsf{doAcase?}(f, \mathbb{X})\epsilon_0$, the function

f is evaluated in the environment ϵ_0, the result is either a term or an exception $\bot(n)$ containing a neutral term n. In case of an exception $\bot(n)$, the set of free variables of n is compared to $\mathbb{X}$, and if there is no variable in common then the exception is thrown further up, else a case term is created whose test term is the neutral term n and each branch is the result of the evaluation of doAcase? for the same function f, but in an updated environment (in each branch the neutral term n is now associated either with $\mathsf{in}_0\, x$ or $\mathsf{in}_1\, x$ for a new variable x).

Remark 17. *In the algorithm above, the line in the function* doAcase?*:*

$$\mathsf{if}\ \mathsf{FV}(n) \cap X = \emptyset\ \mathsf{then}\ \mathsf{throw}(n)$$

ensures that if a neutral term n thrown as an exception and containing a variable x that has not been bound either by a λ-abstraction λx directly before the doAcase? *function or by a binding in a case expression $x\cdot$ after this function, will be thrown upper in the computation tree. This ensures that neutral terms of sum type are captured at the uppermost binding possible, just after their creation by a binding or at the top-level. This is not a trivial idea, as we could just have caught every exceptional neutral terms after the function* doAcase?*, and create a new case distinction operator at this level.*

This latter approach would still yield a correct algorithm in the sense that the result is convertible with the term given as input in the normalization function.

The advantage to create distinction operator as soon as possible is that it allows to further constrain the set of normal forms, and this is essential if we are after a decision algorithm. This idea to create distinction operator as soon as possible first appears explicitly in the works of Vincent Balat [9], and his coauthors, Roberto Di Cosmo and Marcelo Fiore [10], with an implementation with controls operators and an election of a best prompt (uppermost binding).

The normalization function for a term r is obtained by applying the function doAcase? to $\mathbb{\lambda}\epsilon \downarrow_\rho^\epsilon [\![r]\!]_{\uparrow_\epsilon}$ and all free variables of r in an empty environment, i.e., the function undefined everywhere $\mathbb{\lambda}t.\bot$.

Code 9 (Normalization function)**.** *The normalization function* $\mathsf{nf} : \mathsf{Tm} \to \mathsf{NF}$ *is defined by:*

$$\mathsf{nf}(\Gamma \vdash r : \rho) ::= \mathsf{doAcase?}(\mathbb{\lambda}\epsilon \downarrow_\epsilon^\rho [\![r]\!]_{\uparrow_\epsilon}, \mathsf{FV}(t))(\mathbb{\lambda}t.\bot)$$

4.3.3 Toward Completeness

Although the algorithm we have just presented is sound in the sense that the result of our algorithm is $\beta\eta$-equal to the initial term, it is not yet complete. In particular two terms which are $\beta\eta$-equal can be normalized to syntactically different terms. As explained in the introduction, the syntactical equality is not suitable to identify

terms which only differ in the order of independent case eliminations. What we need is an equality test based on the circular conversion (conversion π_6^e to π_9^e) and the immediate simplification (is).

First, let us consider the immediate simplification conversion:

$$\mathsf{case}(r, x_0.s, x_1.s) \longrightarrow_{\mathsf{is}} s \qquad (x_0, x_1 \notin \mathsf{FV}(r)) \tag{is}$$

Whether the test term r in the rule above is a left or a right injection is irrelevant, because the two branches of the alternative are anyway the same. We call such terms redundant.

The reduction (is) above erases the whole term r, and hence the variables it contains. A consequence is that this reduction does not preserve prenormal forms because it can break the deepest λ condition (deepest λ).

Example 16. *Suppose the following term is a prenormal form,*

$$\lambda x.\mathsf{case}(t\{^{\mathsf{case}(r,x_0.s,x_1.s)}/_z\}, y_0.t_0, y_1.t_1)$$

with $x \in \mathsf{FV}(r)$ *but* $x \notin \mathsf{FV}(s)$ *and* $x \notin \mathsf{FV}(t)$*. An immediate simplification reduction leads to the following term:*

$$\lambda x.\mathsf{case}(t\{^{s}/_z\}, y_0.t_0, y_1.t_1)$$

but this term does no longer verify the deepest λ *condition* (deepest λ) *because* $x \notin t\{^r/_z\}$.

To avoid these redundancies we want to define the set of normal forms as a restriction of the set of prenormal forms. But as the example above shows, this can not be achieved by merely firing the immediate simplification reductions (is) from a term in prenormal form. To define the set of normal forms, we will integrate a test of redundancy directly in the inductive definition of prenormal forms.

Among the circular conversions, let us consider the conversion π_8^e and π_9^e.

$$\begin{aligned} \mathsf{case}(r, x_0.\mathsf{case}(r, x_0.s_0, x_1.s_1), x_1.t) &=_{\pi_8^e} \mathsf{case}(r, x_0.s_0, x_1.t) \\ \mathsf{case}(r, x_0.s, x_1.\mathsf{case}(r, x_0.t_0, x_1.t_1)) &=_{\pi_9^e} \mathsf{case}(r, x_0.s, x_1.t_1) \end{aligned}$$

If we consider λ-terms as programs, the term s_1 in the conversion π_8^e and the term t_0 in the conversion π_9^e are superfluous, they will never be used whatever value r is evaluated to. We call such terms junk terms. And although we do not want to select a normal form between two terms which differ only in the order of independent test, we do want to eliminate such junk terms from normal forms.

We will define the normal forms as a restriction of the prenormal forms by introducing two conditions, one to avoid redundancy (redundancy freeness) and one to avoid junk (junk freeness).

These two conditions require an equality test between two occurrences of a subterm, but using syntactical equality for this equality test is not enough. For example, if the terms r and r' only differ by the order of independent case elimination (i.e., $r =_{\pi_c^e} r'$), we still want to eliminate the junk term s_1 in the term

$$\mathsf{case}(r, x_0.\mathsf{case}(r', x_0.s_0, x_1.s_1), x_1.t),$$

or remove the redundancy in the term

$$\mathsf{case}(r, x_0.r', x_1.t) \qquad (\text{if } x_0 \notin \mathsf{FV}(s) \text{ and } x_1 \notin \mathsf{FV}(t)).$$

This equality has indeed to be tested modulo circular conversions.

The work of Thorsten Altenkirch and his coauthors [4] avoids this test of modulo circular conversions altogether by defining a normal form, which identifies terms differing in the order of independent case eliminations (independent case eliminations are grouped together into a binary function from a set of neutral terms). To keep the presentation simple, we prefer a more syntactical presentation to this elegant solution, and therefore follow the way of Vincent Balat [9] and his coauthors [10] and use a test modulo circular permutative conversions.

4.3.4 Normal Forms

We now define the normal forms of our system. Eventually, the goal of designing normal forms is to obtain a set of representatives for each class of terms modulo $\beta\eta$. The considerations in the last sections lead to the following definition of normal forms.

Definition 4.13 (normal forms)**.** *We define by simultaneous induction three relations* $-\vdash_{\mathsf{Ne}} - : -$, $-\vdash_{\mathsf{PNF}} - : -$, *and* $-\vdash_{\mathsf{NF}} - : -$ *between context, term and type,*

$$\frac{(x,\rho)\in\Gamma}{\Gamma\vdash_{\mathsf{Ne}} x:\rho} \qquad \frac{\Gamma\vdash_{\mathsf{Ne}} n:\rho\to\sigma \qquad \Gamma\vdash_{\mathsf{PNF}} P:\rho}{\Gamma\vdash_{\mathsf{Ne}} nP:\sigma}$$

$$\frac{\Gamma\vdash_{\mathsf{Ne}} n:o}{\Gamma\vdash_{\mathsf{PNF}} n:o} \qquad \frac{\Gamma,x:\rho\vdash_{\mathsf{NF}} N:\sigma}{\Gamma\vdash_{\mathsf{PNF}} \lambda x.N:\rho\to\sigma} \qquad \frac{\Gamma\vdash_{\mathsf{PNF}} P:\rho_i}{\Gamma\vdash_{\mathsf{PNF}} \mathsf{in}_i\, P:\rho_0+\rho_1}$$

$$\frac{\Gamma\vdash_{\mathsf{PNF}} P:\rho}{\Gamma\vdash_{\mathsf{NF}} P:\rho} \qquad \frac{\Gamma\vdash_{\mathsf{Ne}} n:\rho_0+\rho_1 \qquad \Gamma,x_i:\rho_i\vdash_{\mathsf{NF}} N_i:\sigma}{\Gamma\vdash_{\mathsf{NF}} \mathsf{case}(n,x_0.N_0,x_1.N_1):\sigma}$$

in the rule with conclusion $\Gamma\vdash_{\mathsf{PNF}} \lambda x.N:\rho\to\sigma$ *we impose the condition*

$$\mathsf{DL}(\{x\},N), \qquad \text{(deepest } \lambda\text{)}$$

where for a set of variables $\mathbb{X}$ *and normal form* N, $\mathsf{DL}(\mathbb{X},N)$ *is true if* N *is not a* case *term and otherwise defined by:*

$$\mathsf{DL}(\mathbb{X},\mathsf{case}(n,x_0.N_0,x_1.N_1)) ::= \mathbb{X}\cap\mathsf{FV}(n)\neq\emptyset\wedge\mathsf{DL}(\mathbb{X}\cup x_0,N_0)\wedge\mathsf{DL}(\mathbb{X}\cup x_1,N_1)$$

in the rule with conclusion $\Gamma\vdash_{\mathsf{NF}} \mathsf{case}(n,x_0.N_0,x_1.N_1):\sigma$*, we impose the condition*

$$\mathsf{RF}(\mathsf{case}(n,x_0.N_0,x_1.N_1)) ::= x_0\notin\mathsf{FV}(N_0)\wedge x_1\notin\mathsf{FV}(N_1)\Rightarrow\Gamma\vdash N_0\neq_{\pi_c^e} N_1:\sigma \qquad \text{(redundancy freeness)}$$

and the condition

$$\mathsf{JF}_\Gamma(\emptyset,\mathsf{case}(n,x_0.N_0,x_1.N_1)) \qquad \text{(junk freeness)}$$

where $\mathsf{JF}_\Gamma(\mathbb{Ne},x,N)$ *is true if* N *is not a* case *term and otherwise defined by:*

$$\mathsf{JF}_\Gamma(\mathbb{Ne},\mathsf{case}(n,x_0.N_0,x_1.N_1)) ::= \forall n'\in\mathbb{Ne},\Gamma\vdash_{\mathsf{Ne}} n\neq_{\pi_c^e} n':\sigma\wedge\mathsf{JF}_{\Gamma,x_i}(\mathbb{Ne},N_i) \qquad (4.1)$$

To produce such normal forms, the first change in the algorithm is that the new condition of redundancy freeness has to be checked where the algorithm produces a case term $\mathsf{case}(n,x.N_0,x.N_1)$ by testing if x occurs in N_0 or N_1 and if $N_0=_{\pi_c^e} N_1$. The auxiliary function doAcase? is changed accordingly.

The second change concerns the environment ϵ associating a neutral term to an injection of a variable x, $\mathsf{in}_0\,x$ or $\mathsf{in}_1\,x$. These environment should now be defined on the set of neutral terms up to circular permutative conversions, and hence the

update operation has to be changed to:

$$\epsilon^{n \mapsto \mathsf{in}_i\, x} s = \begin{cases} \mathsf{in}_i\, x & \text{if } s =_{\pi_c^e} n \\ \epsilon(s) & \text{otherwise} \end{cases}$$

Code 10 (*NbE* for sum types)**.** *Let* $\epsilon_0 \in \mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp$ *be an environment. The* reify *function* $\downarrow^\rho$: $(\mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp) \to [\![\rho]\!] \to \mathsf{PNF}$ *and* reflect *function* $\uparrow^\rho$: $(\mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp) \to \mathsf{Ne} \to [\![\rho]\!]$ *are defined simultaneously by induction on the type* ρ*:*

$$\begin{aligned}
\downarrow^o_{\epsilon_0} n &::= n \\
\downarrow^{\rho\to\sigma}_{\epsilon_0} f &::= \lambda x.\mathsf{doAcase?}(\mathbb{\lambda}\epsilon\, \downarrow^\sigma_\epsilon f(\uparrow^\rho_\epsilon x), \{x\})\epsilon_0 \\
\downarrow^{\rho_0+\rho_1}_{\epsilon_0} \iota_i c &::= \mathsf{in}_i\ \downarrow^{\rho_i}_{\epsilon_0} c \\
\uparrow^o_{\epsilon_0} n &::= n \\
\uparrow^{\rho\to\sigma}_{\epsilon_0} n &::= \mathbb{\lambda} v.\ \uparrow^\sigma_{\epsilon_0} (n \downarrow^\rho_{\epsilon_0} v) \\
\uparrow^{\rho+\sigma}_{\epsilon_0} n &::= \begin{cases} \mathsf{let}\ \mathsf{in}_i\, x = (\mathsf{catch}\ \epsilon_0(n)\ \mathsf{throw}(n))\ \mathsf{in} \\ \iota_i\ \uparrow^{\rho_i}_{\epsilon_0} x \end{cases}
\end{aligned}$$

where doAcase? *is defined by*

$$\begin{aligned}
&\mathsf{doAcase?}(f, X)\epsilon_0 ::= \mathsf{catch}\ (f\epsilon_0) \\
&\quad \mathbb{\lambda} n.\mathsf{if}\ \mathsf{FV}(n) \cap X = \emptyset\ \mathsf{then}\ \mathsf{throw}(n) \\
&\quad\quad \mathsf{else\ let}\ N_0 = \mathsf{doAcase?}(f, X \cup \{x\})\epsilon_0{}^{n \mapsto \mathsf{in}_0\, x}\ \mathsf{in} \\
&\quad\quad\quad \mathsf{let}\ N_1 = \mathsf{doAcase?}(f, X \cup \{x\})\epsilon_0{}^{n \mapsto \mathsf{in}_1\, x}\ \mathsf{in} \\
&\quad\quad\quad\quad \mathsf{if}\ x \notin \mathsf{FV}(N_0) \wedge N_0 =_{\pi_c^e} N_1\ \mathsf{then}\ N_0 \\
&\quad\quad\quad\quad \mathsf{else}\ \mathsf{case}(n, x.N_0, x.N_1)
\end{aligned}$$

The auxiliary function doAcase? has to be applied at the top-level to catch exception containing neutral terms with only free variables.

Code 11 (Normalization function)**.**

$$\mathsf{nf}(\Gamma \vdash r : \rho) = \mathsf{doACase?}(\mathbb{\lambda}\epsilon \downarrow^\epsilon_\rho [\![r]\!]_{\uparrow_\epsilon}, \mathbb{\lambda} t.\bot, \mathsf{FV}(t))$$

4.4 Formalization

We return now to the notions we have let informal in the previous sections: the name generation environment needed to generate new names, the environment associating variables to neutral terms and the handling of exceptions. This section is very similar to the corresponding one of the last chapter. We use a reader monad for each environment and an exception monad to handle the exceptions.

Name Generation Environment

The definition of the name generation environment here is the same as in the chapter handling the simply typed λ-calculus (2), we restate it here.

Definition 4.14 (Name Generation Environment)**.** *We define a set of* name generation environment *or set of environment for short, as a set E together with an update function $(-)^- : E \to \mathsf{Var} \to E$ and an access function $\mathsf{new} : E \to \mathsf{Var}$.*

The extension of $e^- : V \to E$ to a function from a list of variables $e : \mathbf{L}(V) \to E$ is defined in a canonical way by

$$e^{x,\overrightarrow{x}} ::= (e^x)^{\overrightarrow{x}}$$
$$e^{\varepsilon} ::= e.$$

The function $(-)^-$ and new have moreover to satisfy the following property for all $e \in E$:

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{x,\overrightarrow{x}}) \qquad (\dagger)$$

Notation 18. *The condition that the new function applied to an environment $e \in E$ should never return a certain variable x can be expressed by:*

$$\forall\, \overrightarrow{x}, x \neq \mathsf{new}(e^{\overrightarrow{x}})$$

We will abbreviate this condition by $x \notin e$. In the same way we will abbreviate for a given set of variables X, $\forall\, x \in X, x \notin e$ by $X \notin e$.

Postdiction Environment

The second environment we will use, associates injections of a variable to neutral terms of sum type. It comes with an update operation defined up to the circular conversions π_c^e.

Definition 4.15 (Postdiction Environment)**.** *We define the set $\mathbf{P}$ of* postdiction environment *as the set $\mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp$ together with an update function $(-)^- : \mathbf{P} \to (\mathsf{Ne} \times \mathsf{Var}) \to \mathbf{P}$:*

$$\epsilon^{n \mapsto \mathsf{in}_i\, x} n' ::= \begin{cases} \mathsf{in}_i\, x & \text{if } n =_{\pi_c^e} n' \\ \epsilon(n') & \text{otherwise} \end{cases}$$

Interpretation Monad

We will now define the monad used in the interpretation, a computation should take place in the name generation environment E and the postdiction environment

$\mathbf{P}(= \mathsf{Ne} \to (\mathsf{Var}+\mathsf{Var})_\perp)$, and it can be aborted by throwing an exception containing a neutral term. Hence the computation type $\mathbf{T}A$ for a result type A is:

$$\mathbf{T} ::= \mathbf{St}_E(\mathbf{St}_{\mathbf{P}}(A_{\perp(\mathsf{Ne})})) = E \to (\mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp) \to \mathsf{Ne} + A$$

The unit and multiplication of the monad of the interpretation are defined quite naturally from the definition of the unit and multiplication of its monad components.

Definition 4.16 (Interpretation monad $\mathbf{T} ::= \mathbf{St}_E \circ \mathbf{St}_{\mathsf{Ne}\to(\mathsf{Var}+\mathsf{Var})_\perp} \circ -_{\perp(\mathsf{Ne}_+)}$)**.** *The (set operator of) the interpretation monad* $\mathbf{T}$ *is defined by*

$$\begin{aligned}\mathbf{T}A ::= \mathbf{St}_E(&\mathbf{St}_{\mathsf{Ne}\to(\mathsf{Var}+\mathsf{Var})_\perp}(A_{\perp(\mathsf{Ne}_+)}))\\ &(= E \to (\mathsf{Ne} \to (\mathsf{Var} + \mathsf{Var})_\perp) \to A_{\perp(\mathsf{Ne}_+)})\end{aligned}$$

Given $m \in \mathbf{T}A$ *and* $f \in A \to \mathbf{T}B$*, we define the unit* $\nu : A \to \mathbf{T}A$ *and multiplication* $\star : \mathbf{T}A \to (A \to \mathbf{T}B) \to \mathbf{T}B$ *of the interpretation monad by:*

$$\begin{aligned}\nu^{\mathbf{T}} a &::= \nu^{\mathbf{St}_E}(\nu^{\mathbf{St}_{\mathsf{Ne}\to(\mathsf{Var}+\mathsf{Var})_\perp}}(\nu^{A_{\perp(\mathsf{Ne}_+)}} a))\\ m \star^{\mathbf{T}} f(e)(s) &::= m(e)(s) \star^{-\perp(\mathsf{Ne}_+)} \lambda a.f(a)(e)(s)\end{aligned}$$

The update operation for the name environment monad and the postdiction monad extend to the interpretation monad.

The update operations for the name environment monad and the postdiction monad extends component-wise to the interpretation monad.

Definition 4.17 (update for the Postdiction Monad)**.** *We define an update function* $(-)^- : \mathbf{St}_{\mathbf{P}(A)} \to (\mathsf{Ne} \times (\mathsf{Var}+\mathsf{Var})) \to \mathbf{St}_{\mathbf{P}(A)}$ *operation by extending the update operation for postdiction environment by:*

$$m^{n\mapsto \mathsf{in}_i\, x}(\epsilon) ::= m(\epsilon^{n\mapsto \mathsf{in}_i\, nx})$$

This update operation extends in turn into an operation $(-)^- : \mathbf{T}A \to (\mathsf{Ne} \times (\mathsf{Var} + \mathsf{Var})) \to \mathbf{T}A$ *to the interpretation monad just in the same way by:*

$$m^{n\mapsto \mathsf{in}_i\, x}(e)(\epsilon) ::= m(e)(\epsilon^{n\mapsto \mathsf{in}_i\, x})$$

Definition 4.18 (update for the Name Generation Monad)**.** *We define an update* $(-)^- : \mathbf{T}(A) \to \mathsf{Var} \to \mathbf{T}(A)$ *operation by extending the update operation for the name generation monad by:*

$$m^v(e) ::= m(e^v)$$

The interpretation is a call by value interpretation for computations in the monad $\mathbf{T}$:

Definition 4.19 (interpretation)**.** *The* call-by-value interpretation *of the* Λ_+*-calculus is given on types by*

$$\begin{aligned}
[\![o]\!] &::= \mathsf{Ne} \\
[\![\tau_1 \to \tau_2]\!] &::= [\![\rho]\!] \to \mathbf{T}([\![\sigma]\!]) \\
[\![\tau_1 + \tau_2]\!] &::= [\![\sigma_0]\!] + [\![\sigma_1]\!]
\end{aligned}$$

Given a valuation η *on a context* Γ *(*$\eta \vDash \Gamma$*), the* interpretation of a typed term $\Gamma \vdash t : \tau$ *is the monadic value* $[\![t]\!]_\eta \in \mathbf{T}([\![\tau]\!])$ *defined by:*

$$\begin{aligned}
[\![x]\!]_\eta &::= \nu(\eta(x)) \\
[\![rs]\!]_\eta &::= [\![r]\!]_\eta \star \lambda f.[\![s]\!]_\eta \star \lambda a.fa \\
[\![\lambda x.r]\!]_\eta &::= \nu(\lambda a.[\![r]\!]_{\eta, x\mapsto a}) \\
[\![\mathsf{case}(r, x^\rho.s_0, y^\sigma.s_1)]\!]_\eta &::= [\![r]\!]_\eta \star \lambda(\iota_i c).[\![s_i]\!]_{\eta, x\mapsto c} \\
[\![\mathsf{in}_i\, t]\!]_\eta &::= [\![t]\!]_\eta \star \lambda c.\nu(\iota_i c)
\end{aligned}$$

Code 12 (call-by-value *NbE*)**.** *The function* $\downarrow_\tau : [\![\tau]\!] \to \mathbf{T}(\mathsf{PNF}_\tau)$ *and* $\uparrow^\tau : \mathsf{Ne}_\tau \to \mathbf{St_P}([\![\tau]\!]_{\perp(\mathsf{Ne})})$ *are defined simultaneously by:*

$$\begin{aligned}
\downarrow_o n &::= \nu(n) \\
\downarrow_{\rho\to\sigma} f &::= \mathsf{new} \star \mathbb{\lambda} x.\mathsf{doAcase?}(\nu^{\mathbf{St}_E}(\uparrow^\rho x) \star f \star \downarrow_\sigma, \{x\})^x \star \mathbb{\lambda} e.\nu(\lambda x.e) \\
\downarrow_{\rho_0+\rho_1} \iota_i c &::= \downarrow_{\rho_i} c \star \mathbb{\lambda} e.\nu(\mathsf{in}_i\, e) \\
\uparrow^o n &::= \nu(n) \\
\uparrow^{\rho\to\sigma} n &::= \nu(\mathbb{\lambda} v.\downarrow_\rho v \star \mathbb{\lambda} P.\nu(\uparrow^\sigma (nP))) \\
\uparrow^{\rho+\sigma} n &::= \mathbb{\lambda}\epsilon. \begin{cases} \mathsf{case}\ \ \epsilon(n)\ \mathsf{of} \\ \qquad \perp \Rightarrow \mathsf{throw}(n) \\ \qquad \iota_i x \Rightarrow (\uparrow^{\rho_i} x)\epsilon \star \mathbb{\lambda} v.\nu(\iota_i v) \end{cases}
\end{aligned}$$

where the auxiliary function doAcase? *is defined by*

$$\begin{aligned}
&\mathsf{doAcase?}(f, X) ::= \mathsf{catch} f \\
&\qquad \mathbb{\lambda} n.\ \mathsf{if} \quad \mathsf{FV}(n) \cap X = \emptyset\ \mathsf{then}\ \mathsf{throw}(n) \\
&\qquad\quad \mathsf{else} \quad \mathsf{new} \star \mathbb{\lambda} x. \\
&\qquad\qquad \mathsf{doAcase?}(\mathsf{upd}(f, n, x, L), X \cup \{x\})^x \star \mathbb{\lambda} t1. \\
&\qquad\qquad \mathsf{doAcase?}(\mathsf{upd}(f, n, x, R), X \cup \{x\})^x \star \mathbb{\lambda} t2. \\
&\qquad\qquad\quad \mathsf{if} \quad x \notin \mathsf{FV}(t1)\ \wedge\ t1 =_{\pi_c^e} t2 \\
&\qquad\qquad\quad \mathsf{then} \quad \nu(t1) \\
&\qquad\qquad\quad \mathsf{else} \quad \nu(\mathsf{case}(n, x.t1, x.t2))
\end{aligned}$$

Code 13 (Normalization function)**.** *The normalization function* $\mathsf{nf} : \mathsf{Tm}^{\tau}_{\Gamma} \to \mathsf{NF}^{\tau}_{\Gamma}$ *is defined by:*

$$\mathsf{nf}^{\tau}(r) = \mathsf{doAcase?}(\nu(\uparrow \Gamma) \star \mathbb{\lambda}\overrightarrow{a}.[\![r]\!]_{\overrightarrow{x} \to \overrightarrow{a}} \star \downarrow_{\tau}) e_r(\mathbb{\lambda}n.\bot)$$

where the multiplication $\star$ *of the interpretation monad is extended to a list of monadic values by:*

$$(\overrightarrow{m}, m) \star \mathbb{\lambda}(\overrightarrow{a}, a).m' ::= \overrightarrow{m} \star \mathbb{\lambda}\overrightarrow{a}.m \star \mathbb{\lambda}a.m'$$
$$\varepsilon \star \mathbb{\lambda}\varepsilon.m' ::= m'$$

The claims of this section state the correctness and completeness of the algorithm (6) above.

Claim 3 (Correctness)**.**

$$r =_{\beta\eta} \mathsf{nf}(r)$$

Claim 4 (Completeness)**.**

$$r =_{\beta\eta} s \Longrightarrow \mathsf{nf}(r) =_{\mathsf{cc}} \mathsf{nf}(s)$$

The results in the previous chapter (2) were developed to be a basis to prove these results. A first exploration lead us to use logical monadic relations whose monadic part itself is defined inductively. This monadic part has to encompass the notion of fresh variables, as in the previous chapter, but also the notion of attempts needed to run the *NbE* until no exceptions are raised (the base case corresponding to a monadic value returning directly a value, and the step case corresponding to a monadic value returning an exception.) For time reason, we have let a formal study for further research. The conceptual simplicity of the approach using exceptions allowed us to implement the *NbE* algorithm in the pure functional language Haskell without using any primitive operators and we are confident that proofs of the claims (3) and (4) above will for the same reason be feasible.

Chapter 5

Inductive Types

Parametric inductive types can be seen as functions taking type parameters as arguments and returning an instantiated inductive type. Correspondingly given functions between parameters one can construct by iteration a copy function between the corresponding instantiated inductive types which corresponds to the change of parameters along these functions. This suggests the question whether this construction defines a functor.

Inductive types are traditionally interpreted as universal objects. This means that functions which verify the same recursive or iterative equations are equal. Although this interpretation is very useful when proving for example the correctness of a function with respect to its specification, or for transformations of programs, it is a very strong requirement known to be undecidable (see [76]). The implementation of inductive type as universal object is therefore not possible. The commonly adopted solution has been to design system where only the existential part of the universal property is retained. This already, allows to define functions from inductive types by iteration or recursion. This interpretation presupposes also a positive answer to the previous question ; that parametric inductive types are indeed functors. However, it is not the case with respect to the equality generated from standard reductions. We investigate a minimal type system with inductive types and show by means of modular rewriting techniques that reductions to make this construction a functor on a subcategory of the system can already be safely added while preserving the decidability of the internal conversion relation.

5.1 System

5.1.1 Types and Schemas

Definition 5.1 (types). *Given a countable set* TVar *of type variables and* Const *of constructors, the set* Ty *of* types *is defined simultaneously together with the set* $\mathsf{KT}_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)$ *of* constructor types *over* α *with* type parameters $\overrightarrow{\rho}, \overrightarrow{\sigma}$*:*

$$\mathsf{Ty} \ni \rho, \sigma ::= \alpha \mid \rho \to \sigma \mid \mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$$
$$\mathsf{KT}_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) \ni \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) ::= \overrightarrow{\rho} \to \overrightarrow{(\overrightarrow{\sigma_i} \to \alpha)}_{1 \leqslant i \leqslant n} \to \alpha$$

where $\alpha \in \mathsf{TVar}$, $\overrightarrow{\mathsf{c}} \subseteq \mathsf{Const}$, $\overrightarrow{\sigma} = \overrightarrow{\sigma_1}, \cdots, \overrightarrow{\sigma_n}$. *We assume* $\alpha \notin \mathsf{FV}(\overrightarrow{\rho}, \overrightarrow{\sigma})$, *where the set of free variable* $\mathsf{FV}(\rho)$ *of a type* ρ *is defined by:*

$$\mathsf{FV}(\alpha) ::= \alpha$$
$$\mathsf{FV}(\rho \to \sigma) ::= \mathsf{FV}(\rho) \cup \mathsf{FV}(\sigma)$$
$$\mathsf{FV}(\mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})) ::= \mathsf{FV}(\kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)) \setminus \{\alpha\}$$

The types with $\to$ as main symbol are called arrow or functional types, those with the binding symbol μ are called *inductive types*. We require the list of constructors $\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)}$ of an inductive type to be non empty. We assume moreover that the names of constructors are uniquely determined by their inductive type and that the constructors within an inductive type are different. Within the definition of constructor type above, the types $\rho \in \overrightarrow{\rho}$ and $\overrightarrow{\sigma_i} \to \alpha \in \overrightarrow{(\overrightarrow{\sigma_i} \to \alpha)}_{1 \leqslant i \leqslant n}$ stand for the types of the arguments of a constructor, they are called respectively *parametric operators*, and *recursive operators* (0-*recursive* if $\overrightarrow{\sigma}$ is empty and 1-*recursive* otherwise).

Parametric and recursive operator verify the so-called *strict positivity condition* (inductive type can not occur in the domain of the type of an argument of their constructor). It is a syntactical condition frequently used to ensure good property of the system such as termination or well-foundation of inductive predicate defined over inductive types. With this restriction, the inductive types that can be expressed in the system correspond essentially to well-founded trees (arbitrary branching and of finite depth).

Note that we have fixed a particular order on the arguments of a constructor (first parametric and then recursive), it doesn't influence the expressivity of the system and simplifies the presentation.

Notation 19. *A constructor type* $\kappa(\alpha)$ *has always the form* $\overrightarrow{\tau} \to \alpha$. *We shall write* $\kappa^-(\alpha)$ *for the list of types* $\overrightarrow{\tau}$. *For* $\mu = \mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$, *we will write* $\mathsf{c_k} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) \in \mu$ *if* $\mathsf{c_k} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) \in \overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)}$

To explain the interpretation of parametric inductive type as functor of their parameter, we need to single out the following notion of *Schema of Inductive Type*.

Definition 5.2 (Schema of Inductive Type)**.** *Given a list of variables* $\overrightarrow{k}$ *for inductive type constructors, type variables* $\overrightarrow{\pi}, \overrightarrow{\theta}, \alpha$ *(with* $\alpha \notin \overrightarrow{\pi} \cup \overrightarrow{\theta}$*), and constructor types* $\overrightarrow{\kappa_{\overrightarrow{\pi},\overrightarrow{\theta}}(\alpha)}$*, we define a* schema of inductive type $\mathbf{S}$ *by*

$$\mathbf{S}_{\overrightarrow{\pi},\overrightarrow{\theta}}(\overrightarrow{k}) ::= \mu\alpha(\overrightarrow{k : \kappa_{\overrightarrow{\pi},\overrightarrow{\theta}}(\alpha)})$$

Each inductive type $\mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$ is obtained by instantiation of the constructor variables $\overrightarrow{k}$ and type parameters $\overrightarrow{\pi}, \overrightarrow{\theta}$ of a schema of inductive type $\mu\alpha(\overrightarrow{k : \kappa_{\overrightarrow{\pi},\overrightarrow{\theta}}(\alpha)})$.

We now give representative examples of schema and inductive types definable in our system.

Example 17. *A schema can concern the name of the constructors only,*

$$\mathbf{N} = \mu\alpha(k_1 : \alpha, k_2 : \alpha \to \alpha) \textit{(schema of the natural numbers),}$$

it can have a type variable in a parametric operator

$$\mathbf{L}_\pi = \mu\alpha(k_1 : \alpha, k_2 : \pi \to \alpha \to \alpha) \textit{(schema of list)}$$

or type variables in both parametric and 1-recursive operator:

$$\mathbf{T}_{\pi,\theta} = \mu\alpha(k_1 : \alpha, k_2 : \pi \to (\theta \to \alpha) \to \alpha) \textit{(schema of tree),}$$

Instantiations of these schemas may be $\mathsf{N} = \mathbf{N}(0, \mathsf{s}) = \mu\alpha(0 : \alpha, \mathsf{s} : \alpha \to \alpha$ *(natural numbers),* $\mathsf{N}' = \mathbf{N}(0', \mathsf{s}')$ *(a "copy" of N),* $\mathsf{L}(\mathsf{N}') = \mathbf{L}_{\mathsf{N}'}(\mathsf{nil}, \mathsf{cons})$ *(lists over* N' *with standard names of constructors),* $\mathsf{T}(\mathsf{N}, \mathsf{N}) = \mathbf{T}_{\mathsf{N},\mathsf{N}}(\mathsf{leaf}, \mathsf{node})$ *(infinitely branching tree over* N*),* $\mathbf{T}_{\mathsf{N},\mathsf{N}'}(\mathsf{leaf}', \mathsf{node}')$ *etc.*

5.1.2 Terms

The terms of our systems are those of the simply typed λ-calculus extended by constructor constants from Const and iteration operators (*iterators*) $(\!|\, \overrightarrow{t} \,|\!)^{\mu,\tau}$ for an inductive type μ and a type τ (μ stands for the source and τ for the target type).

Definition 5.3 (Terms)**.** *The set of* terms Tm *is generated by the following grammar:*

$$\mathsf{Tm} \ni r, s, t ::= x \mid \lambda x^\tau r \mid (r\ s) \mid \mathsf{c_k} \mid (\!|\, \overrightarrow{t} \,|\!)^{\mu,\tau}$$

with $x \in \mathsf{Var}$*,* $\mathsf{c_k} \in \mathsf{Const}$ *and* $\tau, \mu \subseteq \mathsf{Ty}$*.*

Definition 5.4 (Typing). *the* typing relation *is defined by*

$$\frac{(x,\rho)\in\Gamma}{\Gamma\vdash x:\rho}\ (\textsc{Var}) \qquad \frac{\Gamma,x:\rho\vdash r:\sigma}{\Gamma\vdash\lambda x^{\rho}.r:\rho\to\sigma}\ (\to\mathrm{I}) \qquad \frac{\Gamma\vdash s:\rho \qquad \Gamma\vdash r:\rho\to\sigma}{\Gamma\vdash rs:\sigma}\ (\to\mathrm{E})$$

$$\frac{(\mathsf{c_k}:\kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)\in\mu) \qquad \Gamma\vdash\overrightarrow{r}:\kappa^{-}_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\mu)}{\Gamma\vdash\mathsf{c_k}\,\overrightarrow{r}:\mu}\ (\mu I)$$

$$\frac{\mu\alpha\,(\overrightarrow{\mathsf{c}:\kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})=\mu \qquad \Gamma\vdash\overrightarrow{t:\kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\tau)}}{\Gamma\vdash(\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau}:\mu\to\tau}\ (\mu E)$$

An argument of a constructor is called *parameter argument* if its type is a parametric operator and *recursive argument* if its type is a recursive operator.

Example 18. *Let be given the following inductive types* $\mu=\mathsf{N},\mathsf{L}(\rho),\mathsf{T}(\rho,\sigma)$ *obtained by instantiation of the schema in the previous example on types* ρ *and* σ*, and a type* τ *to be the "target-type" in the typing rule* (μE) *above. The types of the iterator terms* $\overrightarrow{t}$ *(step types) must be:*

- τ *and* $\tau\to\tau$ *in case of* N*;*
- τ *and* $\rho\to\tau\to\tau$ *in case of lists* $\mathsf{L}(\rho)$*;*
- τ *and* $\rho\to(\sigma\to\tau)\to\tau$ *in case of trees* $\mathsf{T}(\rho,\sigma)$*.*

Their particularly simple form is due to the use of iteration (as opposed to primitive recursion where the step type should contain also the types of the arguments of the constructor).

5.1.3 Reductions

To define a capture avoiding substitution, we begin by defining a contextual substitution, which will allow us to define renaming of bound variables (α-conversion).

Definition 5.5 (contextual substitution)**.** *Given a list of distinct variables* $\overrightarrow{x} = x_1, \ldots, x_n$ *and of terms* $\overrightarrow{r} = r_1, \ldots r_n$ *of same length, the effect of the* contextual substitution $[\overrightarrow{x}/\overrightarrow{r}]$ *is defined by induction on* Tm *as follows:*

$$x[\overrightarrow{x}/\overrightarrow{r}] ::= \begin{cases} r_i & \text{if } x = x_i \wedge x_i \in \overrightarrow{x}, \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.r)[\overrightarrow{x}/\overrightarrow{r}] ::= \begin{cases} \lambda x.r[x_1,\ldots,x_{i-1},x_{i+1},\ldots x_n/r_1,\ldots,r_{i-1},r_{i+1},\ldots,r_n] & \text{if } x = x_i \wedge x_i \in \overrightarrow{x} \\ \lambda x.r[\overrightarrow{x}/\overrightarrow{r}] & \text{otherwise} \end{cases}$$

$$(rs)[\overrightarrow{r}/\overrightarrow{x}] ::= (r[\overrightarrow{r}/\overrightarrow{x}])(s[\overrightarrow{r}/\overrightarrow{x}])$$

$$(\!|\overrightarrow{t}|\!)^{\mu,\tau}[\overrightarrow{s}/\overrightarrow{y}] ::= (\!|\overrightarrow{t[\overrightarrow{s}/\overrightarrow{y}]}|\!)^{\mu,\tau}$$

The α-conversion is a congruence on the terms, i.e., an equivalence relation contextually closed, we define therefore here the notion of *contextual closure*.

Definition 5.6 (contextual closure)**.** *A conversion relation* $=_R$ *is contextually closed if the following rules hold:*

$$\frac{r =_R s}{rt =_R st}\ (=_R\text{-AppL}) \qquad \frac{r =_R s}{tr =_R ts}\ (=_R\text{-AppR}) \qquad \frac{r =_R s}{\lambda x.r =_R \lambda x.s}\ (=_R\text{-}\xi)$$

$$\frac{t =_R t'}{(\!|\overrightarrow{r}, t, \overrightarrow{s}|\!)^{\mu,\tau} =_\alpha (\!|\overrightarrow{r}, t', \overrightarrow{s}|\!)^{\mu,\tau}}\ (=_R\text{-It})$$

Definition 5.7 (α-conversion)**.** *The* α*-conversion* $=_\alpha$ *is the smallest congruence relation generated from the following axiom:*

$$\lambda x.r =_\alpha \lambda y.r[x/y] \qquad y \notin \mathsf{FV}(r)$$

Now that we have defined α-conversion, we are able to define correct (or capture avoiding) *substitution*.

Definition 5.8 (correct substitution)**.** *Simultaneous (correct) substitution of terms* $\overrightarrow{s}$ *to variables* $\overrightarrow{y}$ *in a term* t*,* $t\{\overrightarrow{s}/\overrightarrow{y}\}$ *is defined by:*

$$r\{\overrightarrow{x}/\overrightarrow{s}\} = r'[\overrightarrow{x}/\overrightarrow{s}]$$

where $r =_\alpha r'$ *and no bound variable of* r' *is free in a term* $s \in \overrightarrow{s}$

We will now define the reductions of the systems which are relations between terms typable with a same type in a same context. As a rule of thumb, to lighten notations, we will neither precise the type nor the context involved if they can be easily inferred.

Definition 5.9 (β-reduction)**.** *We define the relation of* β-reduction *by the following rule:*

$$(\lambda x^{\tau}.\, r)\ s \longmapsto_{\beta} r\{^{s}/_{x}\} \qquad (\longmapsto_{\beta})$$

Definition 5.10 (η-expansion)**.** *We define the relation of* η-expansion *by the following rule:*

$$r \longmapsto_{\eta} \lambda x^{\rho}.\, r\, x \qquad (\longmapsto_{\eta})$$

where $r : \rho \to \sigma$*,* r *is not an abstraction,* $x \notin \mathsf{FV}(r)$*.*

The condition that r is not an abstraction rules out a source of non-termination. The usual restriction stating that r should not appear in applicative position (this would also causes non-termination) is incorporated in the definition of one-step reduction below.

We will define two variants of the reduction for iteration; first the traditional one:

Definition 5.11 (ι-reduction)**.** *Let* $\mu \equiv \mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$*,* $\mathsf{c_k} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) \in \mu$*, and* $\kappa \equiv \overrightarrow{\rho} \to (\overrightarrow{\sigma}_i \to \alpha)_{1 \leqslant i \leqslant n} \to \alpha$ *over* α *in* μ*. Given a term* $\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}$*, where* $\overrightarrow{p}$ *(with* p_i *of type* ρ_i*) denotes the* parameter arguments *and* $\overrightarrow{r}$ *(with* r_i *of type* $\overrightarrow{\sigma}_i \to \mu$*) the* recursive arguments*, and the terms* $\overrightarrow{t}$ *of step type* $\overrightarrow{\kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\tau)}$*, the* ι-reduction *is defined by:*

$$(\iota) \quad (\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau}(\mathsf{c_k}^{\mu}\,\overrightarrow{p}\,\overrightarrow{r}) \longmapsto_{\iota} t_k\,\overrightarrow{p}\,\overrightarrow{((\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau} \circ r)} = t_k\,\overrightarrow{p}\,\overrightarrow{(\lambda\overrightarrow{x}.(\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau}(r\,\overrightarrow{x}))}\ .$$

This reduction may create β-redexes. Obvious redexes appear if the iteration terms t_k above is an abstraction $\lambda\overrightarrow{y}.s$. Moreover if an abstracted variable $y \in \overrightarrow{y}$ corresponding to a 1-recursive argument is in applicative position inside this iteration term, this too will produce subsequent a β-redex (it will be substituted by a term $\lambda\overrightarrow{x}(\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau}(r\,\overrightarrow{x}) \in \overrightarrow{(\lambda\overrightarrow{x}.(\!|\,\overrightarrow{t}\,|\!)^{\mu,\tau}(r\,\overrightarrow{x}))}$).

Since our system is equipped with η-expansion, one can require functional iteration terms to be abstractions and 1-recursive variables inside iteration terms to be applied as a pre-condition and then define a modified ι-reduction carrying out all these administrative β-reductions in one step.

We begin by defining sets of terms $\mathcal{I}t(\overrightarrow{y})$ where variables $\overrightarrow{y}$, meant to correspond to 1-recursive arguments in an iteration term, are in applicative position.

Definition 5.12. *Let typed variables* $\overrightarrow{y} = y_1^{\overrightarrow{\sigma_1}\to\tau}, \ldots, y_n^{\overrightarrow{\sigma_n}\to\tau}$ *be given, we define inductively the set of terms* $\mathcal{I}t(\overrightarrow{y})$ *where these variables always appear applied to a maximal number of arguments*

$$\mathcal{I}t(\overrightarrow{y}) \ni t ::= (y_i\,\overrightarrow{t})^{\tau} \mid x \mid \lambda z.t \mid tt \mid (\!|\,\overrightarrow{t}\,|\!) \mid \mathsf{c_k}\,\overrightarrow{t} \qquad (x \notin \overrightarrow{y})$$

To carry out administrative reductions due to variables corresponding to 1-recursive argument in applicative position, we need to define first a modified substitution.

Definition 5.13 (Modified substitution)**.** *The effect of* modified simultaneous substitution $t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle$ *of variables* $\overrightarrow{y}$ *by compositions of* $\overrightarrow{u}$ *and* $\overrightarrow{r}$ *(with* $u_i : \mu_i \to \tau$*,* $r_i : \overrightarrow{\sigma_i} \to \mu_i$*) into a term* $t \in \mathcal{I}t(\overrightarrow{y})$*, is defined recursively on* $\mathcal{I}t(\overrightarrow{y})$*:*

$$
\begin{array}{rll}
y_i\,\overrightarrow{t}\,\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= u(r\overrightarrow{t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle}) & \textit{(the essential case)} \\
x\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= x & \\
(\lambda z.t)\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= \lambda z.t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & \textit{where } z \notin \overrightarrow{y} \textit{ and } z \notin FV(\overrightarrow{u\bullet r}) \\
(tt)\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & \\
(\!|\,\overrightarrow{t}\,|\!)\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= (\!|\overrightarrow{t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle}|\!) & \\
\mathsf{c_k}\,\overrightarrow{t}\,\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle & ::= \mathsf{c_k}\overrightarrow{t\langle\overrightarrow{u\bullet r}/\overrightarrow{y}\rangle} &
\end{array}
$$

(At the third line α*-conversion may be needed.)*

Definition 5.14 (ι_2-reduction)**.** *Let* $\mu \equiv \mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$*,* $\mathsf{c_k} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha) \in \mu$*, and* $\kappa \equiv \overrightarrow{\rho} \to (\overrightarrow{\sigma}_i \to \alpha)_{1\leqslant i\leqslant n} \to \alpha$ *over* α *in* μ*. Let a term* $\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}$ *be given, where* $\overrightarrow{p}$ *(*p_i *of type* ρ_i*) denote the* parameter arguments *and* $\overrightarrow{r}$ *(*r_i *of type* $\overrightarrow{\sigma}_i \to \mu$*) the* recursive arguments*. Let the terms* $\overrightarrow{t}$ *of step-type (iteration terms) be fully* η*-expanded externally, i.e.,* $t_k = \lambda\overrightarrow{x^\rho}\overrightarrow{y^{\overrightarrow{\sigma}_i\to\tau}}.s_k$*, and moreover* $s_k \in \mathcal{I}t(\overrightarrow{y_i^{\overrightarrow{\sigma}_i\to\tau}})$*. Under these conditions the* ι_2-reduction *is defined by:*

$$(\!|\ldots,\lambda\overrightarrow{x}\,\overrightarrow{y}.s_k|\!),\ldots(\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}) \longmapsto_\iota s_k\{\overrightarrow{p}/\overrightarrow{x}\}\langle\overrightarrow{(\!|t|\!)\bullet r}/\overrightarrow{y}\rangle$$

Example 19 (multiplication by 2)**.** *Although primitive recursion is encodable in our system, for sake of simplicity we present here functions definable using iteration,*

- *the multiplication by* 2 *of natural numbers:*

  $$\times 2 ::= (\!|0, \lambda x.\mathsf{s}(\mathsf{s}x)|\!)$$

  the associated ι_2*-reduction for the term* $\mathsf{s}\,t$ *is:*

  $$(\times 2)(\mathsf{s}\,t) \longrightarrow_{\iota_2} \mathsf{s}(\mathsf{s}(\times 2\,t))$$

- *the pointwise multiplication by two of a list of natural number*

  $$\mathsf{map}\times 2 ::= (\!|\mathsf{nil}, \lambda xy.\mathsf{cons}(\times 2\,x)y|\!)$$

  the associated ι_2*-reduction for the term* $\mathsf{cons}\,a\,l$ *is:*

  $$\mathsf{map}\times 2(\mathsf{cons}\,a\,l) \longrightarrow_{\iota_2} \mathsf{cons}\,(\times 2\,a)\,(\mathsf{map}\times 2\ l)$$

- *the selection of even branches in a tree can be defined as:*

$$\mathsf{sel2} \equiv (\!|\mathsf{leaf}, \lambda xy.\mathsf{node}\, x\, (\lambda z.y(\times 2\, z))|\!)$$

the associated ι_2-reduction for the term $\mathsf{node}\, t\, f$ is:

$$\mathsf{sel2}(\mathsf{node}\, t\, f) \longrightarrow_{\iota_2} \mathsf{node}\, t\, (\lambda z.\mathsf{sel2}(f(\times 2\, z)))$$

The reductions defined above rewrite redex appearing at the exterior of the term, these reductions are extended to subterms via a contextual closure incorporating the usual restriction on applicative position in η-expansion.

Definition 5.15 (One-step Reduction)**.** *The One-step reduction $\longrightarrow_R$ is defined as the smallest relation such that:*

$$\frac{r \longmapsto_R r'}{r \longrightarrow_R r'}\ (R\text{-Ax}) \qquad \frac{r \longrightarrow_R r' \qquad r \not\longrightarrow_\eta r'}{rs \longrightarrow_R r's}\ (R\text{-AppL}) \qquad \frac{s \longrightarrow_R s'}{rs \longrightarrow_R rs'}\ (R\text{-AppR})$$

$$\frac{r \longrightarrow_R r'}{\lambda x.r \longrightarrow_R \lambda x.r'}\ (R\text{-}\xi) \qquad \frac{t \longrightarrow_R t'}{(\!|\overrightarrow{r}, t, \overrightarrow{s}|\!) \longrightarrow_R (\!|\overrightarrow{r}, t', \overrightarrow{s}|\!)}\ (R\text{-Rec})$$

R can be for example β, η, ι, ι_2.

Notation 20. *The transitive closure of $\longrightarrow_R$ will be written $\xrightarrow{+}_R$, its reflexive transitive closure $\xrightarrow{*}_R$, its inverse relation $\longleftarrow_R$. The composition $\{(r,s) \mid \exists t,\ r \longrightarrow_R t \wedge t \longrightarrow_S s\}$ of two reduction relations $\longrightarrow_R$ and $\longrightarrow_S$ will be written $\longrightarrow_R;\longrightarrow_S$ their union $\{(x,y) \mid x \longrightarrow_R y \vee x \longrightarrow_S y\}$ will be denoted by $\longrightarrow_{RS}$. Alternatively we will use the name of the relations so that R, R^+, R^*, R^{-1}, $R;S$, RS will stand for $\longrightarrow_R$, $\xrightarrow{+}_R$, $\xrightarrow{*}_R$, $\longleftarrow_R$, $(\longrightarrow_R;\longrightarrow_S)$, $\longrightarrow_{RS}$ respectively. The reduction relation $R \setminus S$ stands for the set of pair $(r,s) \in R$ but $(r,s) \notin S$.*

*The R-*derivations *(sequences of terms such that two successive terms are in a one step reduction relation $\longrightarrow_R$) will be denoted by d, e.... The expression $t \xrightarrow{\infty}_R$ will denote an infinite derivation beginning at t.*

5.2 Extended Conversions

5.2.1 General Results

The rewriting system generated from β, η and ι reductions has been proven to be *convergent*, i.e., strongly normalizing and confluent, in numerous works (for example [23]) and we will not repeat the proof here.

Theorem 2. *The reduction $\beta\eta\iota$ is convergent.*

The system with the alternative ι-reduction ι_2 is proved convergent using a (particularly simple) embedding (i.e.,, a reduction preserving encoding of a system within another).

Theorem 3. *1. The system equipped with $\beta\eta\iota_2$-reduction is embeddable in he one equipped with $\beta\eta\iota$-reduction,*

2. the reduction $\beta\eta\iota_2$ is convergent.

Proof. 1. The embedding is the identity on terms and one $\beta\eta\iota_2$-reduction is simulated by one ι-reduction followed by several β-reductions.

2. The embedding (1) above means that for each reduction in $\beta\eta\iota_2$ there exists a sequence of reductions in $\beta\eta\iota$, hence $\beta\eta\iota_2$ is strongly normalizing, otherwise $\beta\eta\iota$ would not be strongly normalizing.

   If t is a normal form in $\beta\eta\iota$, then t is a normal form in $\beta\eta\iota_2$ (the reductions of $\beta\eta\iota_2$ apply on redexes of $\beta\eta\iota$ under some restrictions). If t is a normal form in $\beta\eta\iota_2$, then t is a normal form in $\beta\eta\iota$. The set of normal form for $\beta\eta\iota$ and $\beta\eta\iota_2$ are the same. Moreover $\xrightarrow{+}_{\beta\eta\iota_2}$ is a subrelation of $\beta\eta\iota$, hence the set of normal form of a term t are the same in the two systems, hence $\beta\eta\iota_2$ is confluent.

□

5.2.2 Inductive Type Schemas as Functors

We will now define a subcategory $\mathcal{I}$ of the whole system for which we will prove the functorial laws of inductive types to be decidable. The set of object $\mathcal{I}_0$ of this category $\mathcal{I}$ will be inductive types and the set $\mathcal{I}_1$ of arrows is constituted of identity terms, iterators, and compositions thereof.

Definition 5.16. *The subset $\mathcal{I}_0$ is the set of all inductive types $\mu\alpha\,(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)})$. The subset of typable terms $\mathcal{I}_1 : \rho \to \sigma$ for $\rho, \sigma \in \mathcal{I}_0$ is defined as follows:*

$$\mathcal{I}_1 \ni a, a' ::= \lambda x^{\mu}.x \mid (\!|\overrightarrow{t}|\!) \mid a \circ a'$$

Definition 5.17 (Instantiation of schemas and copy of inductive type)**.** *The* instantiation function $\mathbf{Cp}^0_{\mathbf{S}} : \mathsf{Const}^{|\overrightarrow{k}|} \times \mathcal{I}_0^{|\overrightarrow{\pi}|+|\overrightarrow{\theta}|} \to \mathcal{I}_0$ *for a schema of inductive type* $\mathbf{S} = \mu\alpha(\overrightarrow{k : \kappa_{\overrightarrow{\pi},\overrightarrow{\theta}}(\alpha)})$ *is defined by:*

$$\mathbf{Cp}^0_{\mathbf{S}}(\overrightarrow{\mathsf{c}}, \overrightarrow{\rho}, \overrightarrow{\sigma}) ::= \mu\alpha(\overrightarrow{\mathsf{c} : \kappa_{\overrightarrow{\rho},\overrightarrow{\sigma}}(\alpha)}).$$

If moreover, given inductive types $\overrightarrow{\rho'}$, $\overrightarrow{\sigma'}$ *there exists terms* f, f' *in* $\mathcal{I}_1$*, with typing* $f_i : \rho_i \to \rho'_i$ *and* $f'_i : \sigma'_i \to \sigma_i$*, we define the* copy function $\mathbf{Cp}^1_{\mathbf{S}}$ *taking these terms and the relabelling function* $l : \overrightarrow{\mathsf{c}} \mapsto \overrightarrow{\mathsf{c}'}$ *as arguments by:*

$$\mathbf{Cp}^1_{\mathbf{S}}(l, \overrightarrow{f}, \overrightarrow{f'}) := (\!|\overrightarrow{t}|\!) : \mathbf{Cp}^0_{\mathbf{S}}(\overrightarrow{\mathsf{c}}, \overrightarrow{\rho}, \overrightarrow{\sigma}) \to \mathbf{Cp}^0_{\mathbf{S}}(\overrightarrow{l(\mathsf{c})}, \overrightarrow{\rho'}, \overrightarrow{\sigma'})$$

with $t_k \in \overrightarrow{t}$ *given by:*

$$t_k ::= \lambda\overrightarrow{x^{\rho}}\overrightarrow{y^{\sigma_i \to \varphi}} \cdot \mathsf{c}'_{\mathsf{k}}\ \overrightarrow{\circ_x(f)}\ \overrightarrow{\lambda\overrightarrow{z}.y\circ_z(f')}$$

where the function $\circ_x(f_k)$ *which returns a* β*-reduced form of* $f_k x$ *is defined recursively:*

- $\circ_x(\lambda x^{\mu}.x) = x$
- $\circ_x((\!|\overrightarrow{t}|\!)) = (\!|\overrightarrow{t}|\!)x$
- $\circ_x(a \circ a') = \circ_x(a)\{^{\circ_x(a')}/_x\}$

With the notation of the definition above and given constructor names $\overrightarrow{\mathsf{c}''}$, types $\overrightarrow{\rho''}, \overrightarrow{\sigma''}$, relabelling functions $l' : \overrightarrow{\mathsf{c}'} \to \overrightarrow{\mathsf{c}''}$, and terms $g : \overrightarrow{\rho} \to \overrightarrow{\rho''} \in \mathcal{I}_1$ and $g' : \overrightarrow{\sigma''} \to \overrightarrow{\sigma} \in \mathcal{I}_1$, the following equalities are provable:

$$\begin{aligned} \mathbf{Cp}^1_{\mathbf{S}}(l, \overrightarrow{g}, \overrightarrow{g'}) \circ \mathbf{Cp}^1_{\mathbf{S}}(l, \overrightarrow{f}, \overrightarrow{f'}) &= \mathbf{Cp}^1_{\mathbf{S}}(l' \circ l, \overrightarrow{g \circ f}, \overrightarrow{f' \circ g'}) \\ \mathbf{Cp}^1_{\mathbf{S}}(\mathsf{id}, \overrightarrow{\mathsf{id}}, \overrightarrow{\mathsf{id}}) &= \mathsf{id}_{\mathbf{Cp}^0_{\mathbf{S}}} \end{aligned}$$

This means that with respect to an extensional model the pair $(\mathbf{Cp}^0_{\mathbf{S}}, \mathbf{Cp}^1_{\mathbf{S}})$ defines a functor. This result is well known, and a categorical proof (of a generalisation of this result) can be found for example in Varmo Vene's doctoral thesis ([89]). However, these equalities do not hold w.r.t. the conversion relation. We shall extend the reduction relation in order to obtain a functor w.r.t. the conversion relation while preserving confluence and strong normalization of the underlying rewrite system.

In the following we will lighten the notation and omit all unnecessary material. By analogy with category theory we will write $\mathbf{Cp_S}$ instead of $\mathbf{Cp}^0_{\mathbf{S}}$, $\mathbf{Cp}^1_{\mathbf{S}}$ and often just $\mathbf{Cp}$ when $\mathbf{S}$ will be clear from the context. In the same way we will not write the relabelling functions which we will not consider as part of the calculus.

Definition 5.18 (χ-reductions). *The* χ-reductions *are given by:*

$$\begin{aligned}\mathbf{Cp}(\overrightarrow{g},\overrightarrow{g'})(\mathbf{Cp}(\overrightarrow{f},\overrightarrow{f'})t) &\longmapsto_{\chi_\circ} \mathbf{Cp}(\overrightarrow{g\circ f},\overrightarrow{f'\circ g'})t\\ \mathbf{Cp}(\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}})t &\longmapsto_{\chi_{\mathsf{id}}} t\end{aligned}$$

Example 20. *The function* $\mathsf{map}\times 2$ *of the previous example can be written as* $\mathbf{Cp_L}(\times 2)$. *The* $\chi_\circ$*-reduction for the composition of two functions* $\mathsf{map}\times 2$ *states that doubling every element of a list from a list where every element has already been doubled should reduce to doubling the double of every element of the initial list.*

$$\mathbf{Cp_L}(\times 2)\mathbf{Cp_L}(\times 2)t \longrightarrow_{\chi_\circ} \mathbf{Cp_L}(\times 2\circ\times 2)t$$

The function $\mathsf{sel2}$ *of the previous example can be written as* $\mathbf{Cp}_T(\mathsf{id},\times 2)$. *The* $\chi_\circ$*-reduction for the composition of two functions* $\mathsf{sel2}$ *states that selecting even branches from a tree where one has already selected even branches should reduce to selecting the even branches of the even branches of this tree.*

$$\mathbf{Cp}_T(\mathsf{id},\times 2)(\mathbf{Cp}_T(\mathsf{id},\times 2)t) \longrightarrow_{\chi_\circ} \mathbf{Cp}_T(\mathsf{id},\times 2\circ\times 2)t$$

An advantage is, for example, for a function $\times 4$*, in order to prove that we have*

$$\mathbf{Cp}_T(\mathsf{id},\times 2)(\mathbf{Cp}_T(\mathsf{id},\times 2)t) = \mathbf{Cp}_T(\mathsf{id},\times 4)t \text{ or}$$

$$\mathbf{Cp_L}(\times 2)\mathbf{Cp_L}(\times 2) = \mathbf{Cp_L}(\times 4),$$

one only has to prove $\times 2\circ\times 2 = \times 4$.

Remark 18. *The restriction of the function* $\mathbf{Cp}$ *to functions generated by iterators or compositions thereof is clearly a limitation of our results. Although our results are still interesting in themselves, an extension to the the general case is a topic of current research.*

5.3 Main Theorems

In this sections we will derive the convergence of the system augmented by the $\chi_\circ$ and the χ_{id} reductions by using modular properties of abstract reduction systems.

5.3.1 Adjournment

Definition 5.19 (Adjournment). *Given two reduction relations* S *and* R*, we say that* S *is* adjournable *w.r.t.* R *in a derivation* d*, if*

$$d = t \longrightarrow_S \longrightarrow_R \overset{\infty}{\longrightarrow}_{RS} \Rightarrow \exists\, e = t \longrightarrow_R \overset{\infty}{\longrightarrow}_{RS}$$

If S *is adjournable w.r.t. to* R *in all derivation* d*, then we say that* S *is adjournable w.r.t. to* R.

Remark 19. *S is adjournable w.r.t. R in particular in the case: $\longrightarrow_S;\longrightarrow_R \subseteq \longrightarrow_R;\longrightarrow_{RS}$. The notion of adjournability is traditionally expressed with this weaker condition (where d is not taken into account) (cf. [7]).*

Lemma 22 (Adjournment)**.** *If R and S are strongly normalizing and S is adjournable w.r.t. to R then RS is strongly normalizing.*

Proof. Let us suppose that there is an infinite RS-derivation beginning from a term t. As R and S are strongly normalizing, this derivation consists of an alternation of finite R- and S-derivations. In particular the derivation is of the form

$$t \longrightarrow_X t' \longrightarrow_S \longrightarrow_R \xrightarrow{\infty}_{RS}$$

with an initial fragment $X = R^*$ or $X = R^*S^+$. We can adjourn the derivation following t' to obtain a derivation

$$t \longrightarrow_X t' \longrightarrow_R \xrightarrow{\infty}_{RS}$$

The iteration of this process will infinitely increase the number of R-reductions in the beginning, preserving the infinite tail and we shall have a contradiction with the assumption that R is strongly normalizing. □

5.3.2 Convergence of $\beta\eta\iota_2\chi_\circ$

Theorem 4 (Strong normalization of $\chi_\circ$)**.** *The $\chi_\circ$-reduction is strongly normalizing and adjournable with respect to $\beta\eta\iota_2$-reduction.*

To ease the exposition we will need to single out a particular occurrence of a subterm t' of a term t, we will then use the notation $C[t']$ for the term t, in this notation C is called a context.

This notation can be defined formally by adding an extra symbol [] to the definition of terms, this symbol [] called a hole can be considered as a special variable not allowed to be bound.

Definition 5.20 (context)**.** *A* context with multiple occurrences *K is given by the grammar:*

$$K, L ::= [] \mid x \mid \lambda x^\tau K \mid (K\ L)$$

A simple context *C (or just* context*) is a context with multiple occurrences K where the symbol [] occurs once and only once. The notation $C[t]$ is an abbreviation for the contextual substitution $C[^{[]}/_t]$ of a term t in the simple context C.*

Proof. Since $\chi_\circ$ is strongly normalizing (by a simple argument on the size of the term), it remains to show that $\chi_\circ$ is adjournable w.r.t. $\beta\eta\iota_2$. It is adjournable if and only if it is adjournable w.r.t. all infinite derivations $d = \longrightarrow_{\chi_\circ} \longrightarrow_{\beta\eta\iota_2} d'$. We consider the reduction $\longrightarrow_{\beta\eta\iota_2}$ following the first $\longrightarrow_{\chi_\circ}$-segment in d.

Let us consider a $\chi_\circ$-reduction followed by a ι_2-reduction. In this case, the adjournment of $\chi_\circ$-reduction with respect to traditional ι-reduction is not possible.

$$\begin{array}{rcl} C[\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r})] & \longrightarrow_{\chi_\circ} & C[\mathbf{Cp}_{\overrightarrow{g\circ f},\overrightarrow{f'\circ g'}}\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}] \\ & \longrightarrow_{\iota_2} & C[\overrightarrow{\mathsf{c_k}\circ_x(g\circ f)\{^p/_x\}}\overrightarrow{\lambda\overrightarrow{z}.(y\circ_z(f'\circ g'))}\langle\overrightarrow{\mathbf{Cp}_{\overrightarrow{g\circ f},\overrightarrow{f'\circ g'}}\bullet r}/\overrightarrow{y}\rangle] \\ & \equiv & C[\overrightarrow{\mathsf{c_k}\circ_p(g\circ f)}\overrightarrow{\lambda z.\mathbf{Cp}_{\overrightarrow{g\circ f},\overrightarrow{f'\circ g'}}(r\circ_z(f'\circ g'))}] \end{array}$$

It can be adjourned as follows:

$$\begin{array}{rcl} C[\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}\mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r})] & & \\ & \longrightarrow_{\iota_2} & C[\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\overrightarrow{\mathsf{c_k}\circ_x(f)\{^p/_x\}}\overrightarrow{\lambda\overrightarrow{z}.(y\circ_z(\overrightarrow{f'}))}\langle\overrightarrow{\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}\bullet r}/\overrightarrow{y}\rangle)] \\ & \equiv & C[\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\overrightarrow{\mathsf{c_k}\circ_p(f)}\overrightarrow{\lambda\overrightarrow{z}.(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(r\circ_z(\overrightarrow{f'})))})] \\ & \longrightarrow_{\iota_2} & C[\overrightarrow{\mathsf{c_k}\circ_x(g)\{^{\circ_p(f)}/_x\}}\overrightarrow{\lambda\overrightarrow{z}.y\circ_z(g')}\langle\overrightarrow{\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}\bullet(\lambda\overrightarrow{z}.(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(r\circ_z(\overrightarrow{f'}))))}/\overrightarrow{y}\rangle] \\ & \equiv & C[\overrightarrow{\mathsf{c_k}\circ_p(g\circ f)}\overrightarrow{\lambda\overrightarrow{z}.\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}((\lambda\overrightarrow{z}.(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(r\circ_z(\overrightarrow{f'}))))\circ_z(g'))}] \\ & \longrightarrow_{\beta} & C[\overrightarrow{\mathsf{c_k}\circ_p(g\circ f)}\overrightarrow{\lambda\overrightarrow{z}.\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(r\circ_z(f'\circ g')))}] \\ & \longrightarrow_{\chi_\circ} & C[\overrightarrow{\mathsf{c_k}\circ_p(g\circ f)}\overrightarrow{\lambda z.\mathbf{Cp}_{\overrightarrow{g\circ f},\overrightarrow{f'\circ g'}}(r\circ_z(f'\circ g'))}] \end{array}$$

For the other cases, the adjournment method works without complication. (We will abbreviate in the following $\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}(\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}t)$ and $\mathbf{Cp}_{\overrightarrow{g\circ f},\overrightarrow{f'\circ g'}}t$ by $L_{\chi_\circ}(t)$ and $R_{\chi_\circ}(t)$):

1. For β-conversion:

   (a) For $C \equiv C[(\lambda x \cdot p[L_{\chi_\circ}(t)])\ q]$, we have

   $$C \longrightarrow_{\chi_\circ} C[(\lambda x \cdot p[R_{\chi_\circ}(t)])\ q] \longrightarrow_{\beta_\rightarrow} C[p[R_{\chi_\circ}(t)]\{^q/_x\}]$$

   In this case, we can adjourn $\chi_\circ$ by

   $$C \longrightarrow_{\beta_\rightarrow} C[p[L_{\chi_\circ}(t)]\{^q/_x\}] \longrightarrow_{\chi_\circ} C[p[R_{\chi_\circ}(t)]\{^q/_x\}]$$

   (b) For $C \equiv C[(\lambda x \cdot p)\ q[L_{\chi_\circ}(t)]]$, we have

   $$C \longrightarrow_{\chi_\circ} C[(\lambda x \cdot p)\ q[R_{\chi_\circ}(t)]] \longrightarrow_{\beta_\rightarrow} C[p\{^{q[R_{\chi_\circ}(t)]}/_x\}]$$

   In this case, we can adjourn $\chi_\circ$ by

   $$C \longrightarrow_{\beta_\rightarrow} C[p\{^{q[L_{\chi_\circ}(t)]}/_x\}] \xrightarrow{*}_{\chi_\circ} C[p\{^{q[R_{\chi_\circ}(t)]}/_x\}]$$

2. For η-conversion, there is no interesting overlap with $\chi_\circ$-conversion as we have the following facts about a $\chi_\circ$-redex $L_{\chi_\circ}()$:

   - $\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}$ and $\mathbf{Cp}_{\overrightarrow{g},\overrightarrow{g'}}$ are iterators ;
   - and t and $L_{\chi_\circ}(t)$ inhabit an *inductive* type.

□

Theorem 5 (Strong normalization of $\beta\eta\iota_2\chi_\circ$)**.** *The reduction $\beta\eta\iota_2\chi_\circ$ is strongly normalizing.*

Proof. As the reductions $\beta\eta\iota_2$ and $\chi_\circ$ are strongly normalizing and $\chi_\circ$ is adjournable with respect the reduction $\beta\eta\iota_2$, we can apply the adjournment lemma (22). □

By Newman's lemma (see [74] for the original proof), a strongly normalizing and locally confluent system is confluent, so we need only to check local confluence.

Theorem 6 (Confluence of $\beta\eta\iota_2\chi_\circ$)**.** *The reduction $\beta\eta\iota\chi_\circ$ is confluent.*

Proof. As $\beta\eta\iota_2\chi_\circ$-conversion is strongly normalizing, it is enough to show that $\beta\eta\iota_2\chi$-conversion is locally confluent, by Newman's Lemma. As $\beta\eta\iota_2$- and $\chi_\circ$-conversions are both confluent, the proof is by verification (case analysis) that $\longleftarrow_{\chi_\circ};\longrightarrow_{\beta\eta\iota_2} \subseteq \overset{*}{\longleftarrow}_{\beta\eta\iota_2\chi_\circ};\overset{*}{\longrightarrow}_{\beta\eta\iota_2\chi_\circ}$. □

Theorem 7 (Convergence of $\beta\eta\iota_2\chi_\circ$)**.** *The reduction $\beta\eta\iota_2\chi_\circ$ is convergent.*

Proof. The reduction $\beta\eta\iota_2\chi_\circ$ is strongly normalizing by theorem (5) and convergent by theorem (6). □

5.3.3 Pre-adjusted Adjournment

We would like to be able to adjourn χ_{id}-reduction w.r.t. ι_2-reduction. However, there is a difficulty, let us consider an example:

Example 21. *Trying to adjourn directly the following derivation, where* node *is the constructor for arbitrary branching tree:*

$$\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\mathsf{node}\,p\,x) \longrightarrow_{\chi_{\mathsf{id}}} \mathsf{node}\,p\,x \longrightarrow_{\beta\eta\iota} \mathsf{node}\,p'\,x$$

results in first applying a ι_2-contraction and then a χ_{id}-contraction:

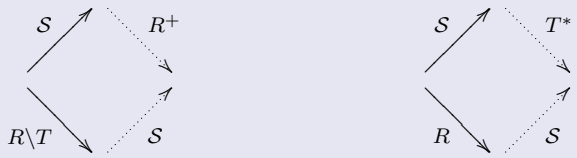
$$\begin{array}{rcl} \mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\mathsf{node}\,p\,x) & \longrightarrow_{\iota_2} & \mathsf{node}\,x\,\lambda z.yz\{^p/_x\}\langle^{\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}\bullet x}/_y\rangle \\ & \equiv & \mathsf{node}\,p\,\lambda z.\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(xz) \\ & \longrightarrow_{\chi_{\mathsf{id}}} & \mathsf{node}\,p\,\lambda z.xz, \end{array}$$

and there is no way to close the fork with the initial derivation.

The solution is to incorporate some η-expansion in the first derivation before applying the adjournment lemma.

Definition 5.21 (insertability)**.** *Given two reduction relation R, T, with $T \subset R$, T is said to be* insertable *in R if there exists a relation $\mathcal{S}$ on the support of R with $T \subseteq \mathcal{S}$ and the two following conditions hold:*

$$\mathcal{S}^{-1}; (R \setminus T) \subseteq R^+; \mathcal{S}^{-1} \qquad \mathcal{S}^{-1}; R \subseteq T^*; \mathcal{S}^{-1}$$



Lemma 23 (insertion)**.** *Given two reduction relations R, T such that T is insertable in R and T is strongly normalizing. If there exists an infinite derivation d from t and an object t' with $t \to_T t'$, then there exists an infinite derivation d' from t'.*

Proof. As T is strongly normalizing, d contains infinitely many $R \setminus T$-reductions (possibly interleaved with finite sequences of T-reductions). As T is insertable in R there exists a relation $\mathcal{S}$ with $(t, t') \in T \subseteq \mathcal{S}$, and we can construct a derivation d' where every reduction $R \setminus T$ of d are repercussed along $\mathcal{S}$ by a R^+-reduction in d'. Hence R^+ contains infinitely many R^+-reductions (possibly interleaved with finite sequences of T^*-reductions) and is therefore infinite □

Definition 5.22 (Conditional Adjournment)**.** *Let R, S be reduction relations, an infinite derivation $d = t \longrightarrow_S \longrightarrow_R \xrightarrow{\infty}_{RS}$ beginning with t and $\mathcal{P}$ a predicate on the terms. Then S is* adjournable *w.r.t. R in d* under condition $\mathcal{P}$*, if*

$$d = t \longrightarrow_S \longrightarrow_R \xrightarrow{\infty}_{RS} \wedge \mathcal{P}(t) \Rightarrow \exists\, e = t \longrightarrow_R \xrightarrow{\infty}_{RS},$$

S is adjournable w.r.t. R under condition $\mathcal{P}$, if S is adjournable w.r.t. R in d under condition $\mathcal{P}$ for all d.

We now introduce the notion of *realisation* of a condition by a reduction relation T, which says that starting from some term t we will always find after a finite number of reductions from T a term t' satisfying the condition.

Definition 5.23 (realization)**.** *Let T be a reduction relation and $\mathcal{P}$ a predicate on the terms. T* realises *$\mathcal{P}$ for t if $\exists t', t \to_T^* t' \wedge \mathcal{P}(t')$. T realizes $\mathcal{P}$ if T realizes $\mathcal{P}$ for all terms.*

Lemma 24 (pre-adjusted adjournment)**.** *Given reductions relations R, S, T with $S, T \subseteq R$, S is adjournable with respect to R under condition $\mathcal{P}$ and T is insertable in R, strongly normalizing and realizes $\mathcal{P}$, then S is adjournable w.r.t. R.*

Proof. Given a derivation $d = t \longrightarrow_S \longrightarrow_R \xrightarrow{\infty}_{RS}$ beginning with t,

- either $\mathcal{P}(t)$ and then S is adjournable w.r.t. R in d
- or as T realizes $\mathcal{P}$, $\exists t', t \to_T^+ t' \wedge \mathcal{P}(t')$. But T is insertable and strongly normalizing so by lemma 23, there exists an infinite derivation from t'. Hence, as $T \subseteq R\ S$ is adjournable w.r.t. R.

□

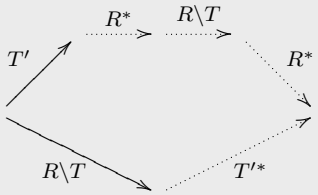
Definition 5.24 (unrestricted η-expansion $\overline{\eta}$)**.** *we define the rewrite rule for* unrestricted η-expansion $\overline{\eta}$ *by:*

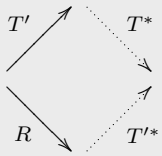
$$t \longmapsto_{\overline{\eta}} \lambda x^{\rho}.tx \text{ if } t : \rho \to \sigma$$

The one step reduction reduction relation $\longrightarrow_{\overline{\eta}}$ *is defined as the contextual closure of* $\longmapsto_{\overline{\eta}}$

Lemma 25 (weak condition for insertability)**.** *In the definition of the insertability, if the relation* $\mathcal{S}$ *is the transitive reflexive closure of a reduction relation* T'*, we establish some sufficient condition for* T *to be insertable. Given reduction relations* R, T, T'*, if the relation* T' *verifies the two conditions:*

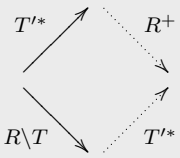
$$T'^{-1}; R \setminus T \subseteq R^*; (R \setminus T); R^*; (T'^{-1})^* \qquad T'^{-1}; T \subseteq R^*; (T'^{-1})^*$$

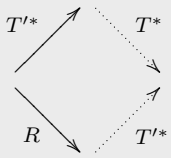




then T *is insertable:*

$$(T'^{-1})^*; R \setminus T \subseteq R^+; (T'^{-1})^* \qquad (T'^{-1})^*; R \subseteq T^*; (T'^{-1})^*$$





Proof. cf lemma 6.4 in [23]. □

Lemma 26. *η-expansion is insertable.*

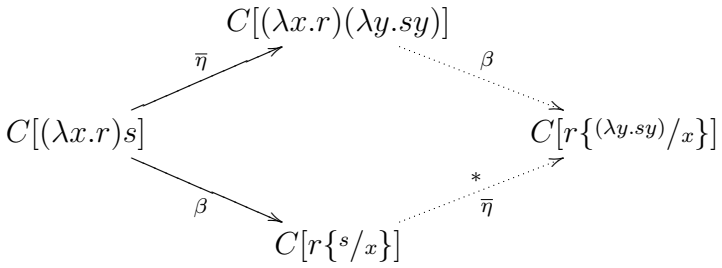
Proof. Take the transitive reflexive closure of $\overline{\eta}$ as relation $\mathcal{S}$ in the definition of insertability 5.21.

By lemma 25, it is enough to show:

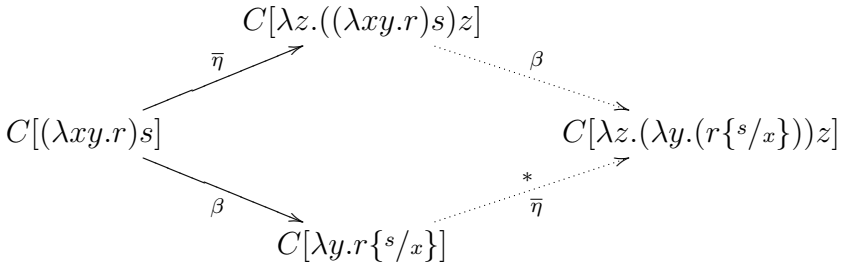
$$\longleftarrow_{\overline{\eta}}; R\backslash \longrightarrow_{\eta} \subseteq R^*; R\backslash \longrightarrow_{\eta}; R^*; \longleftarrow^*_{\overline{\eta}} \qquad \wedge \qquad \longleftarrow_{\overline{\eta}}; \longrightarrow_{\eta} \subseteq R^*; \longleftarrow^*_{\overline{\eta}}$$

The proof is simple and we will only discuss the critical cases (why we take the reflexive transitive closure of $\overline{\eta}$ instead of merely η for $\mathcal{S}$).

- for β, there are two non trivial cases:

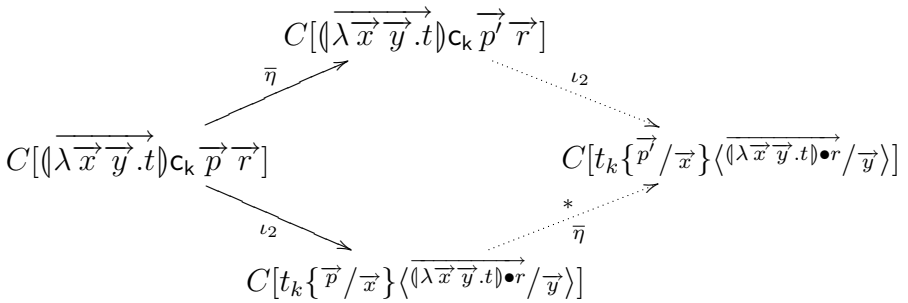


  As the subterm s can be substituted in an applicative position, this case illustrates the need for $\overline{\eta}$ to expand a term in applicative position. Remark also that the variable x can occur several times in r or not occurs at all, so that we need to take the reflexive transitive closure of $\overline{\eta}$.

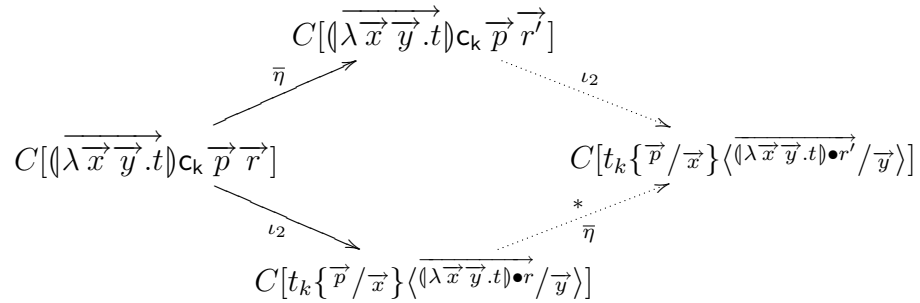


  This case illustrates the need for the $\overline{\eta}$-expansion to be applicable to a subterm in abstraction form.

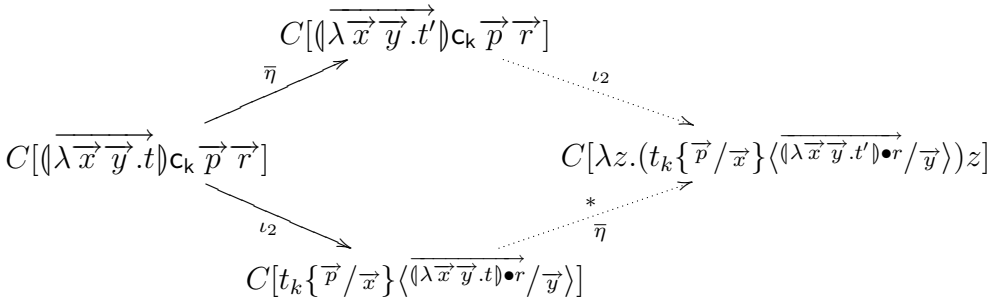
- for ι_2, the situation is similar to β, the $\overline{\eta}$ can take place in a parameter argument: for $\overrightarrow{p} = p_0, \ldots, p_i, \ldots, p_n$ $(i \in [0, \ldots, n])$ we write $\overrightarrow{p'}$ for $p_0, \ldots, \lambda x.p_i x, \ldots, p_n$.



or in a recursive argument: for $\overrightarrow{r} = r_0, \ldots, r_i, \ldots, r_n$ $(i \in [0, \ldots, n])$ we write $\overrightarrow{r'}$ for $r_0, \ldots, \lambda x.r_i x, \ldots, r_n$.



or in the iterator: for $\overrightarrow{t} = t_0, \ldots, t_i, \ldots, t_n$ $(i \in [0, \ldots, n])$ we note $\overrightarrow{t'} := t_0, \ldots, \lambda z.t_i z, \ldots, t_n$. The non trivial case is $i = k$:



□

5.3.4 Convergence of $\beta\eta\iota_2\chi$

Theorem 8 (Strong Normalization of χ_{id}). *1. χ_{id}-reduction is strongly normalizing,*

2. χ_{id}-reduction is adjournable with respect to $\beta\eta\iota_2\chi_\circ$ under the condition that 1-recursive arguments $\overrightarrow{r} = r_1 \ldots r_n$ of a constructor $\mathsf{c}\,\overrightarrow{p}\,\overrightarrow{r}$ of type μ are fully eta-expansed externally, i.e., $r_i = \overrightarrow{r'_i} = \lambda\overrightarrow{x}.(r'_i\,\overrightarrow{x})^\mu$.

proof of theorem 8. We will abbreviate $\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}t$ by $L_{\chi_{\mathsf{id}}}(t)$

1. For ι_2-conversion:

   (a) For $C \equiv C[\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\mathsf{c}_k\ \overrightarrow{p}\ \overrightarrow{r})]$, by η-insertion we can put the terms $\overrightarrow{r}$ in full externally eta-expansed form. We will write $\overline{r}$ for a term r fully η-expansed externally. We can then always adjourn

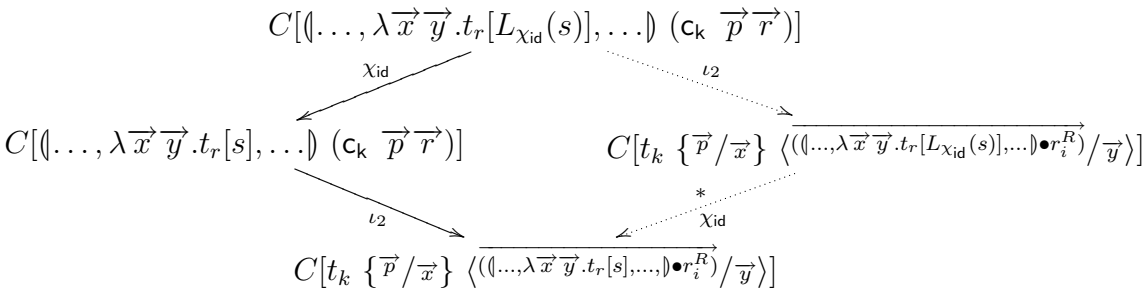
$$C[\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\mathsf{c}_k\ \overrightarrow{p}\ \overrightarrow{\overline{r}})] \longrightarrow_{\chi_{\mathsf{id}}} C[\mathsf{c}_k\ \overrightarrow{p}\ \overrightarrow{\overline{r}}] \longrightarrow_{\beta\eta\iota} \cdots$$

by

$$\begin{array}{rcl} C[\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\mathsf{c}_k\ \overrightarrow{p}\ \overrightarrow{r})] & \longrightarrow_{\iota_2} & C[(\mathsf{c}_k\,\overrightarrow{x}\,\overrightarrow{\lambda\,\overrightarrow{z}\,.y\,\overrightarrow{z}})\{\overrightarrow{p}/\overrightarrow{x}\}\langle\overrightarrow{\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}\bullet\overrightarrow{r}}/\overrightarrow{y}\rangle] \\ & \equiv & C[\mathsf{c}_k\,\overrightarrow{p}\,\overrightarrow{\lambda\,\overrightarrow{z}\,.\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}(\overrightarrow{r}\,\overrightarrow{z})}] \\ & \longrightarrow^*_\beta\longrightarrow_{\chi_{\mathsf{id}}} & C[\mathsf{c}_k\,\overrightarrow{p}\,\overrightarrow{r}\,] \end{array}$$

(b) For $C \equiv C[(\!|\ldots,\lambda\,\overrightarrow{x}\,\overrightarrow{y}\,.t_r[L_{\chi_{\mathsf{id}}}(s)],\ldots|\!)\ (\mathsf{c_k}\ \overrightarrow{p}\,\overrightarrow{r})]$ and $\overrightarrow{t} \in It(y^{\overrightarrow{\overrightarrow{\sigma_i}\to\tau}})$ (one remarks that $It(y^{\overrightarrow{\overrightarrow{\sigma_i}\to\tau}})$ is stable under $\longleftarrow_{\chi_{\mathsf{id}}}$), we have three possibilities:

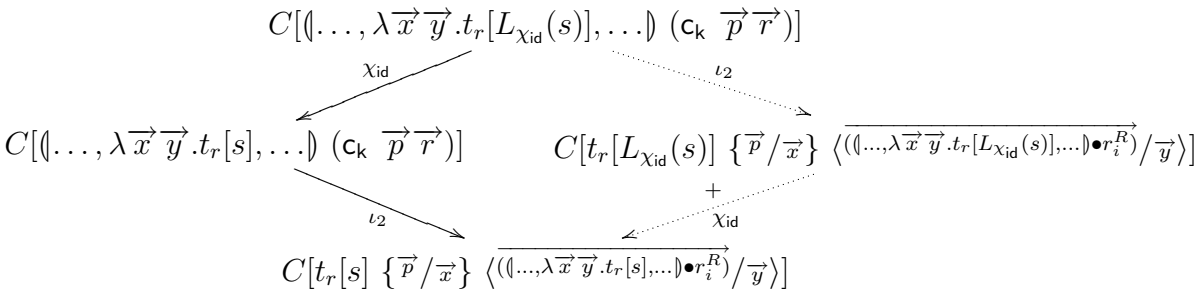
i. Either $r \neq k$ and then we have



with as many χ_{id}-conversions as there are variables $\overrightarrow{y}$ occurring in t_k.

ii. or $r = k$, and then

- either $L_{\chi_{\mathsf{id}}}(s)$ is a strict subterm in $t_r[L_{\chi_{\mathsf{id}}}(s)]$:



with as many χ_{id}-conversions as there are variable $\overrightarrow{y}$ occurring in t_k plus one for the $L_{\chi_{\mathsf{id}}}(s)$ occurring already in t_r before ι_2-reduction.

- or $L_{\chi_{\mathsf{id}}}(s)$ is not a strict subterm but the whole term itself. Let us therefore rewrite the original term as $C[(\!|\ldots,\lambda\,\overrightarrow{x}\,\overrightarrow{y}\,.L_{\chi_{\mathsf{id}}}(s),\ldots|\!)\ \mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}]$. Then we have

$$C \longrightarrow_{\chi_{\mathsf{id}}} C[(\!|\ldots,\lambda\,\overrightarrow{x}\,\overrightarrow{y}\,.s,\ldots|\!)\ \mathsf{c_k}\,\overrightarrow{p}\,\overrightarrow{r}] \longrightarrow_{\iota_2} C[s]$$

(because $L_{\chi_{\mathsf{id}}}(s)$ inhabits necessarily an inductive type, therefore can't be of functional type and accept arguments, and hence

$\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{p}, \overrightarrow{r}$ are the empty lists). In this case, we can adjourn χ_{id} by

$$C \longrightarrow_{\iota_2} C[L_{\chi_{\mathsf{id}}}(s)] \longrightarrow_{\chi_{\mathsf{id}}} C[s] \ .$$

(c) For $C \equiv C[(\!|\ldots, \lambda\overrightarrow{x}\,\overrightarrow{y}.t_k, \ldots|\!)\ (\mathsf{c_k}\ \ldots, p_r[L_{\chi_{\mathsf{id}}}(s)], \ldots\ \overrightarrow{r})]$, we have:

$$\begin{array}{rl} C & \longrightarrow_{\chi_{\mathsf{id}}} C[(\!|\ldots, \lambda\overrightarrow{x}\,\overrightarrow{y}.t_k, \ldots|\!)\ (\mathsf{c_k}\ \ldots p_r[s] \ldots\ \overrightarrow{r})] \\ & \longrightarrow_{\iota_2} C[t_k\ \{^{\ldots p_r[s] \ldots}/_{\overrightarrow{x}}\}\ \langle^{\overrightarrow{((\!|\lambda\overrightarrow{x}\,\overrightarrow{y}.\,\overrightarrow{t}\,|\!)\bullet r_i)}}/_{\overrightarrow{y}}\rangle \ . \end{array}$$

In this case, we can adjourn χ_{id} by

$$\begin{array}{rl} C & \longrightarrow_{\iota_2} C[t_k\ \{^{\ldots p_r[L_{\chi_{\mathsf{id}}}(s)] \ldots}/_{\overrightarrow{x}}\}\ \langle^{\overrightarrow{((\!|\lambda\overrightarrow{x}\,\overrightarrow{y}.\,\overrightarrow{t}\,|\!)\bullet r_i^R)}}/_{\overrightarrow{y}}\rangle \\ & \longrightarrow_{\chi_{\mathsf{id}}} C[t_k\ \{^{\ldots p_r[s] \ldots}/_{\overrightarrow{x}}\}\ \langle^{\overrightarrow{((\!|\lambda\overrightarrow{x}\,\overrightarrow{y}.\,\overrightarrow{t}\,|\!)\bullet r_i^R)}}/_{\overrightarrow{y}}\rangle \ . \end{array}$$

(d) For $C \equiv C[(\!|\ldots, \lambda\overrightarrow{x}\,\overrightarrow{y}.t_k, \ldots|\!)\ (\mathsf{c_k}\ \overrightarrow{p}\ \ldots r_r[L_{\chi_{\mathsf{id}}}(s)] \ldots)]$ (writing a for the number of recursive arguments), we have:

$$\begin{array}{rl} C & \longrightarrow_{\chi_{\mathsf{id}}} C[(\!|\ldots, \lambda\overrightarrow{x}\,\overrightarrow{y}.t_k, \ldots|\!)\ (\mathsf{c_k}\ \overrightarrow{p}\ \ldots r_r[L_{\chi_{\mathsf{id}}}(s)] \ldots)] \\ & \longrightarrow_{\iota_2} C[t_k\ \{\overrightarrow{p}/_{\overrightarrow{x}}\}\ \langle^{\ldots((\!|\lambda\overrightarrow{x}\,\overrightarrow{y}.\,\overrightarrow{t}\,|\!)\bullet r_r[s])\ \ldots}/_{\overrightarrow{y}}\rangle \ . \end{array}$$

In this case, we can adjourn χ_{id} by

$$\begin{array}{rl} C & \longrightarrow_{\iota_2} C[t_k\ \{\overrightarrow{p}/_{\overrightarrow{x}}\}\langle^{\ldots((\!|\lambda\overrightarrow{x}\,\overrightarrow{y}.\,\overrightarrow{t}\,|\!)\bullet r_r[L_{\chi_{\mathsf{id}}}(s)])\ \ldots}/_{\overrightarrow{y}}\rangle \\ & \longrightarrow_{\chi_{\mathsf{id}}} C[t_k\ \{\overrightarrow{p}/_{\overrightarrow{x}}\}\ \langle^{\ldots((\!|\,\overrightarrow{t}\,|\!)\bullet r_r[s])\ \ldots}/_{\overrightarrow{y}}\rangle \ . \end{array}$$

2. For β-conversion:

(a) For $C \equiv C[(\lambda x \cdot p[L_{\chi_{\mathsf{id}}}(t)])\ q]$, we have

$$C \longrightarrow_{\chi_{\mathsf{id}}} C[(\lambda x \cdot p[t])\ q] \longrightarrow_{\beta_\rightarrow} C[p[t]\{^q/_x\}].$$

In this case, we can adjourn χ_{id} by

$$C \longrightarrow_{\beta_\rightarrow} C[p[L_{\chi_{\mathsf{id}}}(t)]\{^q/_x\}] \longrightarrow_{\chi_{\mathsf{id}}} C[p[t]\{^q/_x\}].$$

(b) For $C \equiv C[(\lambda x \cdot p)\ q[L_{\chi_{\mathsf{id}}}(t)]]$, we have

$$C \longrightarrow_{\chi_{\mathsf{id}}} C[(\lambda x \cdot p)\ q[t]] \longrightarrow_{\beta_\rightarrow} C[p\{^{q[t]}/_x\}].$$

In this case, we can adjourn χ_{id} by

$$C \longrightarrow_{\beta_\rightarrow} C[p\{^{q[L_{\chi_{\mathsf{id}}}(t)]}/_x\}] \xrightarrow{*}_{\chi} C[p\{^{q[t]}/_x\}].$$

3. For η-conversion, there is no interesting overlap with χ_{id}-conversion as we have the following facts about a χ_{id}-redex $L_{\chi_{\mathsf{id}}}(t)$:

- $L_{\chi_{\mathsf{id}}}$ is a recursor ;
- and t and $L_{\chi_{\mathsf{id}}}(t)$ inhabit an *inductive* type.

4. For $\chi_\circ$-conversion The only non trivial case is:

$$C[\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}t)] \longrightarrow_{\chi_{\mathsf{id}}} C[\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(t)]$$

In this case we can adjourn by

$$C[\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}(\mathbf{Cp}_{\overrightarrow{\mathsf{id}},\overrightarrow{\mathsf{id}}}t)] \longrightarrow_{\chi_\circ} C[\mathbf{Cp}_{\overrightarrow{f},\overrightarrow{f'}}t]$$

□

Theorem 9 (Strong Normalization of $\beta\eta\iota_2\chi$)**.** *The reduction $\beta\eta\iota_2\chi$ is strongly normalizing.*

Proof. Obviously the eta expansion η realizes the condition that 1-recursive arguments occurring in an iteration term are totally expansed externally. (the terms to be expansed are neither in application position nor λ-abstraction). Moreover the eta expansion η is insertable by lemma (26). Theorem (8) sates that χ_{id}-reduction is adjournable with respect to $\beta\eta\iota_2\chi_\circ$ under the condition that 1-recursive arguments $\overrightarrow{r} = r_1 \dots r_n$ of a constructor $\mathsf{c}\,\overrightarrow{p}\,\overrightarrow{r}$ of type μ are fully eta-expansed externally, i.e., $r_i = \overrightarrow{r'_i} = \lambda\overrightarrow{x}.(r'_i\overrightarrow{x})^\mu$. Hence we can conclude by lemma (24), that χ_{id} is adjournable with respect to $\beta\eta\iota_2\chi_\circ$. Finally as both $\beta\eta\iota_2\chi_\circ$ and χ_{id} are strongly normalizing we can apply lemma (22) to conclude that $\beta\eta\iota_2\chi$ is strongly normalizing. □

Theorem 10 (Confluence of $\beta\eta\iota_2\chi$)**.** *The reduction $\beta\eta\iota\chi$ is confluent.*

proof of theorem 10. As $\beta\eta\iota\chi$-conversion is strongly normalizing, it is enough, by Newman's Lemma to show that $\beta\eta\iota\chi$-conversion is locally confluent. As $\beta\eta\iota\chi_\circ$- and χ_{id}-conversions are both confluent, The proof is a verification by case analysis that $\longleftarrow_{\chi_{\mathsf{id}}}; \longrightarrow_{\beta\eta\iota\chi_\circ} \subseteq \overset{*}{\longrightarrow}_{\beta\eta\iota\chi}; \overset{*}{\longleftarrow}_{\beta\eta\iota\chi}$. □

Theorem 11 (Convergence of $\beta\eta\iota_2\chi$)**.** *The reduction $\beta\eta\iota_2\chi$ is convergent.*

Proof. The reduction $\beta\eta\iota_2\chi$ is strongly normalizing by theorem (9) and confluent by theorem (10). □

Conclusions

This work allowed us to study several deciding algorithms for the theories of typed λ-calculus. We have gathered our conclusions and suggestions for further work under the following themes.

Reduction based versus reduction free normalization

At the heart of the thesis is the opposition between reduction (or rewriting) driven and reduction free algorithms. While the conceptual simplicity of rewriting theory is attractive to understand the computational behaviour of normalization, reduction free algorithms provides decidability results, where application of rewriting theory is not obvious. The two approaches are related and we have tried to explain the latter in terms of the former. However the relationship between the two approaches has not yet been enough explored. For example one could further explore the extraction of reduction proofs which as shown in [18] can produce reduction free algorithm. As shown in this thesis, there is several possible NbE algorithms for a same system according to the model of computation used (call-by-value, call-by-name, Algol-like). Does there exist a proof (or several different proofs) from which these algorithms can be extracted? The extraction method used in [18] was modified realizability, does a different method (e.g., functional interpretation) give a different algorithm?

Fresh variables

We have developed a monadic formalisation of fresh variables generation for the NbE algorithm. We believe that this formalisation is natural in the sense that it follows closely an intuitive description of what it means for a variable to be fresh. This treatment reduces the gap between informal description and actual mathematical formalisation of the algorithm. The main change between informal description to a pure functional implementation is to pass from simple type to monadic ones. We are conscious that the simplicity of our solution is due to its specialization for a single problem, namely the generation of fresh names in NbE algorithm. Nevertheless an interesting question is how this approach incorporates or relates in the broad body of research concerning formalisation of system with names and bindings (nominal reasoning).

Sequent calculus and natural deduction

Applying the normalization by evaluation for the calculus Λ_J allowed us to tightly relate cut-free derivations in sequent calculus and natural deduction. Normal forms for permutative conversion in the Λ_J calculus can be seen as a notation for cut-free proof of the sequent calculus. Further restrictions on normal forms in the Λ_J-calculus can be seen as defining normal forms among cut-free proofs. These restricted normal forms are in turn isomorphic to long normal forms in the simply typed λ-calculus modulo certain decidable conversions. This sharpens previous results relating sequent calculus and natural deduction.

Normalization by evaluation for λ-calculus with extensional permutative conversions

We have given a purely functional algorithm for deciding extensional conversions of terms in a system with generalized applications Λ_J and one with sum types Λ_+. As already advocated, algorithms described in a pure functional style are much closer to a mathematical formalisation. In an imperative settings proofs are much more involved as one has to take into account the existence of a global state, influencing the behaviour of primitive operations. Hence proofs of correctness and completeness for these algorithms should be feasible. The conversions of these systems are very similar, and so are the *NbE* algorithms we have described. This suggests to carry out the proofs of correctness and completeness for the simplest system Λ_J-calculus and then adapt them for the Λ_+-calculus. This is still work in progress.

Functoriality of inductive types

Our studies show that functorial laws are admissible for a subcategory of the system with inductive types, in the sense that properties derived from these laws, if expressed as equalities are decidable. Although these results are already interesting in themselves, an obvious direction for further research is the study of the general case, both for reduction based and reduction free algorithms.

Index

Bibliography

[1] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.

[2] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14(4):587–611, August 2004.

[3] Thorsten Altenkirch. α-conversion is easy. Under Revision, 2002.

[4] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.

[5] Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\rightarrow,2}$. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Proc. of 7th Int. Symp. on Functional and Logic Programming, FLOPS 2004 (Nara, Japan, 7–9 Apr. 2004)*, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275. Springer, Berlin, 2004.

[6] Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In Dexter Kozen and Carron Shankland, editors, *MPC*, volume 3125 of *Lecture Notes in Computer Science*, pages 32–53. Springer, 2004.

[7] Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 1986.

[8] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. Reproduced in *Selected Reprints on Dataflow and Reduction Architectures*, ed. S. S. Thakkar, IEEE, 1987, pages 215–243, and in *ACM Turing Award Lectures: The First Twenty Years*, ACM Press, 1987, pages 63–130.

[9] Vincent Balat. *Une étude des sommes fortes : isomorphismes et formes normales*. PhD thesis, Université Paris 7, 2002.

[10] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, January 2004.

[11] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. ACM, Springer.

[12] Henk Barendregt. The impact of the lambda calculus on logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[13] Freiric Barral, David Chemouil, and Sergei Soloviev. Non-standard reductions and categorical models in typed lambda-calculus. *Logitcheskie Issledovaniya (Studies in logic)*, 12:300–315, December 2005. Moscow, Nauka.

[14] Freiric Barral and Sergei Soloviev. Inductive type schemas as functors. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 35–45. Springer, 2006.

[15] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In F. Honsell and M. Miculan, editors, *Proc. of 4th Int. Conf. on Found. of Software Science and Computation Structures, FoSSaCS'01, Genova, Italy, 2–6 Apr. 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 57–71. Springer, Berlin, 2001.

[16] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2000.

[17] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer.

[18] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

[19] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings 6'th Symposium on Logic in Computer Science (LICS'91)*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[20] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer, 1987.

[21] Frédéric Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris XI Orsay, 2001.

[22] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, June 1992.

[23] David Chemouil. *Types inductifs, isomorphismes et récriture extensionnelle*. Thèse de doctorat, Université Paul Sabatier, Toulouse, September 2004.

[24] David Chemouil and Sergei Soloviev. Remarks on isomorphisms of simple inductive types. In *Mathematics, Logic and Computation , Eindhoven*, pages 1–19, Electronic Notes in Theoretical Computer Science 85, 7, July 2003. Elsevier,.

[25] Alonzo Church. A set of postulates for the foundations of logic. *Annals of Mathematics*, 33:346–366, 1932.

[26] Alonzo Church. An unsolvable problem in elementary number theory. *Amer. J. Math.*, 58:345–363, 1936.

[27] Alonzo Church. A simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[28] Catarina Coquand. A machine assisted semantical analysis of simply typed λ–calculus. Programming Methodology Group, Chalmers University of Technology and University Göteborg, 1993.

[29] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 14(1):57–90, 2002.

[30] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. In *International Workshop TYPES'93*, Nijmegen, The Netherlands, May 1993.

[31] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:73–94, 1997.

[32] Thierry Coquand and Gérard Huet. Constructions - A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *Eurocal 85, Volume 1*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, Berlin-Heidelberg-New York, 1985.

[33] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[34] Thierry Coquand and Christine Paulin. Inductively defined types (preliminary version). In P. Martin-Löf and G. Mints, editors, *Proc. of Int. Conf. on Computer Logic, COLOG'88, Tallinn, USSR, 12–16 Dec 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.

[35] Roberto Di Cosmo. *Isomorphisms of Types: from λ–calculus to information retrieval and language design.* PTCS. Birkhäuser, Boston, 1995.

[36] Haskell. B. Curry. Functionality in combinatory logic. In Proceedings of the National Academy of Sciences, U.S.A., volume 20, pages 584–590, 1934.

[37] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM, ACM Press.

[38] Philippe de Groote. Strong norm of classical natural deduction with disjunction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2001.

[39] Daniel J. Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. *Information and Computation*, 157(1-2):52–83, 2000.

[40] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer, 2000.

[41] Andrzej Filinski. Representing layered monads. In *Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, 1999.

[42] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer. Extended version available as technical report BRICS RS-99-17.

[43] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer, May 2001.

[44] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1934.

[45] Neil Ghani. ßn-equality for coproducts. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1995.

[46] Neil Ghani and Christoph Lüth. Composing monads using coproducts. *Intl. Conference on Functional Programming 2002*, 37(9):133–144, 2002.

[47] Jean-Yves Girard. *Interprétation functionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[48] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[49] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *CTCS*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 1987.

[50] Hugo Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic. 8th Workshop, CSL'94. Kazimierz, Poland, September 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995.

[51] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[52] Martin Hofmann. Nbe with a logical relation. Unpublished work, 1999.

[53] W.A. Howard. The formulae–as–types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[54] Gérard Huet. A unification algorithm for typed λ–calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[55] Felix Joachimski and Ralph Matthes. Short proofs of normalisation for the simply-typed λ-calculus, permutative conversions and Gödel's T. To appear: Archive for Mathematical Logic, 2002.

[56] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[57] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 134–143. Springer, 1992.

[58] Stephen C. Kleene. Permutability of inferences in Gentzen's calculi LK and LJ. Memoirs of the American Mathematical Society No. 10, pp. 1–26, Providence, 1952.

[59] Stephen C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Ann. of Math. (2)*, 36:630–636, 1935.

[60] Joachim Lambek. Deductive systems and categories. I. syntactic calculus and residuated categories. *Math. Systems Theory*, 2:287–318, 1968.

[61] Joachim Lambek. Deductive systems and categories: II: Standard constructions and closed categories. In *Category theory, homology theory and their applications*, number 86 in Lecture Notes in Mathematics, pages 76–122. Springer, 1969.

[62] Joachim Lambek. Deductive systems and categories III: Cartesian closed categories, intuitionist propositional calculus, and combinatory logic. In *Toposes, Algebraic Geometry and Logic*, pages 57–82. 1981. Lecture Notes in Mathematics 274.

[63] Joachim Lambek and Philip J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[64] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[65] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer, New York, 1976.

[66] Christopher R. Mann. The connection between equivalence of proofs and Cartesian closed categories. *Proceedings of the London Mathematical Society*, 21(3):289–310, 1975.

[67] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[68] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Mathematisches Institut der Universität München, 1998.

[69] Ralph Matthes. Interpolation for Natural Deduction with Generalized Eliminations. In R. Kahle, P. Schröder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2001.

[70] Ralph Matthes. Non-strictly positive fixed points for classical natural deduction. *ANNALSPAL: Annals of Pure and Applied Logic*, 133, 2005.

[71] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[72] Grigory E. Mints. Closed categories and the theory of proofs. *Journal of Soviet Mathematics*, 15(1):45–62, 1981.

[73] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996.

[74] Maxwell H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[75] Karim Nour and René David. A short proof of the strong normalization of classical natural deduction with disjunction. *The Journal ofSymbolic Logic*, 68(4):1277–1288, December 2003.

[76] Mitsuhiro Okada and Phillip J. Scott. A note on rewriting theory on for uniqueness of iteration. *Theory and Applications of Categories*, 6(4):47–64, 1999.

[77] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*, volume 3 of *Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy*. Almqvist & Wiksell, Stockholm, 1965.

[78] Dag Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307. North–Holland, Amsterdam, 1971.

[79] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer.

[80] John C. Reynolds. The essence of ALGOL. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland, Amsterdam, 1981.

[81] John C. Reynolds. Design of the programming language forsythe. Technical report, Carnegie Mellon University, July 22 1996.

[82] Mikael Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, volume 449 of LNAI*, pages 603–617, Berlin, Heidelberg, 1990. Springer.

[83] Jonathan P. Seldin. The logic of Curry and Church, 2006. To appear in the Handbook of the History of Logic, volume 5, edited by Dov Gabbay and John Woods, to be published by Elsevier.

[84] Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

[85] William W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[86] Makoto Tatsuta. Second order permutative conversions with Prawitz's strong validity. *Progress in Informatics*, 2:41–56, 2005.

[87] Makoto Tatsuta and Grigori Mints. A simple proof of second-order strong normalization with permutative conversions. *Ann. Pure Appl. Logic*, 136(1-2):134–155, 2005.

[88] Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.

[89] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis (Diss. Math. Univ. Tartuensis 23), Dept. of Computer Science, Univ. of Tartu, August 2000.

[90] Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001.

[91] Phil Wadler. Theorems for free! In *Proc. 1989 ACM Conf. on Lisp and Functional Programming*, pages 347–359, 1989.

[92] Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of construction. Under consideration for publication in J.Functional Programming.

[93] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, England, first edition, 1910–1913. Three volumes.

Décidabilité pour Conversions Non-Standards en Lambda-Calculs Typés

Freiric Barral

Thèse soutenue à Munich, le 6 Juin 2008.

Directeurs : Prof. Dr. Martin Hofmann et Prof. Dr. Sergeï Soloviev.

Résumé : Cette thèse étudie la décidabilité des conversions des lambda-calculs typés ainsi que les algorithmes permettant cette décidabilité. Notre étude prend en consideration des conversions qui vont au-delà des traditionnelles conversions que sont les conversions beta, eta ou encore les conversions permutatives (encore appelées conversions commutatives). Pour décider ces conversions deux classes d'algorithmes s'opposent, ceux basés sur la récriture où le but est de décomposer et d'orienter les conversions afin d'obtenir un système convergent, l'algorithme revient alors à récrire les termes jusqu'à ce qu'ils atteignent une forme irréductible et les algorithmes dits « libres de réduction » ou la conversion est décidée par un détour dans un métalangage ; tout au long de cette thèse, nous nous efforçons d'expliquer cette deuxième classe grâce à la première.

Mots-clés : lambda-calcul typé, conversion, réduction, décidabilité, normalisation par evaluation.

Discipline : Informatique Théorique.

Institut de recherche en Informatique de Toulouse
118 route de Narbonne,
31 062 Toulouse Cedex 4,
France

Ludwig-Maximilians-Universität München,
Informatik Institut,
Lehr- und Forschungseinheit Theoretische Informatik,
Oettingenstraße 67,
80 538 München,
Allemagne

Entscheidbarkeit für Nicht-Standard Konversionen in Getypten Lambda-Kalkülen

Freiric Barral

Doktorarbeit verteidigt in München am 6. Juni 2008.

Betreuer: Prof. Dr. Martin Hofmann und Prof. Dr. Sergeï Soloviev.

Zusammenfassung: Diese Arbeit befasst sich mit der Entscheidbarkeit von Konversionen in getypten Lambda-Kalkülen, und den Algorithmen, die diese Entscheidbarkeit ermöglichen. Unsere Studie betrachtet Konversionen, die über die traditionelle beta, eta oder permutative Konversionen (auch kommutative Konversionen genannt) hinaus gehen. Um diese Konversionen zu entscheiden, gibt es zwei ganz verschiedene Klassen von Algorithmen, die zum Einsatz kommen können: die auf Termersetzungssystem basierende Algorithmen, wobei das Ziel ist, die Konversionen zu zerlegen und zu orientieren um ein konvergentes System zu erhalten, und die so genannte „reduktionsfreien“ Algorithmen, die die Konversion rekursiv durch ein Umweg in einer Meta-Sprache entscheiden. Wir bemühen uns in dieser Arbeit, die zweite Klasse mit Hilfe der erste zu erklären.

Schlüsselwörter: getypter lambda-Kalkül, Konversion, Reduktion, Entscheidbarkeit, Normalisierung durch Auswertung.

Fachgebiet: Theoretische Informatik.

Institut de recherche en Informatique de Toulouse
118 route de Narbonne,
31 062 Toulouse Cedex 4,
Frankreich

Ludwig-Maximilians-Universität München,
Informatik Institut,
Lehr- und Forschungseinheit Theoretische Informatik,
Oettingenstraße 67,
80 538 München,
Deutschland

Decidability for Non-Standard Conversions in Typed Lambda-Calculi

Freiric Barral

Thesis defended in Munich on 6 June 2008.

Supervisors: Prof. Dr. Martin Hofmann and Prof. Dr. Sergeï Soloviev.

Abstract: This thesis studies the decidability of conversions in typed lambda-calculi, along with the algorithms allowing for this decidability. Our study takes in consideration conversions going beyond the traditional beta, eta, or permutative conversions (also called commutative conversions). To decide these conversions, two classes of algorithms compete, the algorithms based on rewriting, where the goal is to decompose and orient the conversion so as to obtain a convergent system, these algorithms then boil down to rewrite the terms until they reach an irreducible forms; and the "reduction free" algorithms where the conversion is decided recursively by a detour via a meta-language. Throughout this thesis, we strive to explain the latter thanks to the former.

Keywords: typed lambda calculus, conversion, reduction, decidability, normalization by evaluation.

Field: Theoretical Computer Science.

Institut de recherche en Informatique de Toulouse
118 route de Narbonne,
31 062 Toulouse Cedex 4,
France

Ludwig-Maximilians-Universität München,
Informatik Institut,
Lehr- und Forschungseinheit Theoretische Informatik,
Oettingenstraße 67,
80 538 München,
Germany