Normalisation in Lambda Calculus and its relation to Type Inference

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op dinsdag 18 juni 1996 om 16.00 uur

door

Paula Gabriela Severi

geboren te Montevideo, Uruguay

Dit proefschrift is goedgekeurd
door de promotoren:
prof.dr. J.C.M. Baeten
prof.dr. H.P. Barendregt
en de copromotor:
dr. J.H. Geuvers

Contents

Chapter 1

Introduction

In this thesis we consider typed and untyped lambda calculi. In this introduction, firstly, we give an informal explanation of the untyped lambda calculus, of the concept of type and an overview of the lambda calculi with types. Finally, we summarise the contents of this thesis.

1.1 Lambda Calculus

The lambda calculus is a formal system based on a function notation invented by A. Church [Chu41]. It captures the most basic aspects of the manners in which operators are combined to form other operators.

We give some motivations for the syntax of the λ-calculus. In mathematics, a function that given x produces x^2 is written as follows.

$$f : x \mapsto x^2$$

This notation is not adequate when higher-order functions are involved (functions which admit other functions as arguments). Church used a notation involving the special symbol λ to construct functions in a systematic way. In Church's notation the example above is written as $\lambda x.x^2$.

The λ-terms are formed with two constructors, namely the *application* ( ) and the *abstraction* λ. We write $(f\ a)$ to express that a function f is applied to the argument a. Let $b(x)$ be an expression containing x. A function that assigns the value $b(a)$ to the argument a is denoted by $(\lambda x.\ b(x))$. This justifies the definition of a relation on the set of λ-terms called *β-reduction*.

$$((\lambda x.\ b(x))\ a) \to_\beta b(a) \quad \beta\text{-reduction}$$

In the example, we have that $((\lambda x.x^2)\ 3)$ β-reduces to 3^2. We usually omit a pair of parenthesis when this does not cause confusion, for example we write $(\lambda x.x^2)\ 3$ for the term above.

The equivalence relation generated by the β-reduction is called *β-conversion* and is denoted by $\twoheadleftarrow\!\!\twoheadrightarrow_\beta$ and the transitive closure of $\rightarrow_\beta$ is denoted by $\twoheadrightarrow_\beta$.

An important property of β-reduction is confluence. Since a term may contain several subterms of the form $(\lambda x.\ b(x))\ a$ (called *redexes*), the β-reduction is not functional. This is not an obstacle for the theory of the lambda calculus due to the fact that it has the property of confluence. Confluence means that if we have the solid arrows of the diagram then we also have the dotted ones.

$$\begin{tikzcd}
 & b \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
b_1 \arrow[dr, dotted, "\beta"'] & & b_2 \arrow[dl, dotted, "\beta"] \\
 & d &
\end{tikzcd}$$

A term that can not be reduced any further is said to be a *normal form* (or *to be in normal form*). For example $(\lambda x.x)\ y \rightarrow_\beta y$ and y is in normal form.

An important property of λ-terms is normalisation. Normalisation and termination are synonymous. In λ-calculus, termination of the reduction is not guaranteed, e.g. $\Omega = (\lambda x.(x\ x))\ (\lambda x.(x\ x))$ reduces to itself. A term is said to be *weakly normalising* (or *normalising*) if there is a reduction sequence that ends in a normal form. For example, the term $(\lambda x.x)\ y$ is weakly normalising. There may be λ-terms that are normalising but they have some reduction sequence that is not finite, e.g. the term $(\lambda x.y)\Omega$ reduces to y if we contract the leftmost redex and to itself if we contract Ω. A term is said to be *strongly normalising* if any reduction sequence starting from the term terminates. Clearly, $(\lambda x.y)\Omega$ is weakly normalising but not strongly normalising.

The trace of a redex can be followed by marking the corresponding abstraction. In the term $(\lambda x.(x\ x))\ ((\underline{\lambda} x.x)\ y)$, we have marked the second redex that we want to trace. When this term is reduced to $((\underline{\lambda} x.x)\ y)((\underline{\lambda} x.x)\ y)$, two redexes are created which are called *residuals* of the initial marked redex. The β-reduction restricted to the marked redexes is called $\underline{\beta}$-reduction. *Developments* are reduction sequences where only residuals of redexes that are present in the initial term are contracted. In other words, a development is a $\underline{\beta}$-rewrite sequence. For example, a development is the rewrite sequence

$$\begin{aligned}
(\underline{\lambda} x.(x\ x))\ ((\underline{\lambda} x.x)\ y) &\rightarrow_{\underline{\beta}} ((\underline{\lambda} x.x)\ y)\ ((\underline{\lambda} x.x)\ y) \\
&\rightarrow_{\underline{\beta}} y\ ((\underline{\lambda} x.x)\ y) \\
&\rightarrow_{\underline{\beta}} (y\ y).
\end{aligned}$$

An important classical result in the lambda calculus is that all developments are finite (finiteness of developments) [Bar85]. To prove that the developments are finite is to prove

that the $\underline{\beta}$-reduction, the β-reduction restricted to the marked redexes, is strongly normalising.

Even though the confluence property of the lambda calculus guarantees that the system is 'functional', it is important from the practical point of view to find adequate reduction strategies, i.e. functions that determine 'a way of reducing'. A strategy is called *normalising* if it finds the normal form of a weakly normalising λ-term. For example, the strategy that reduces only the leftmost redex of a λ-term is normalising. The importance of the existence of normalising strategies is that it implies the decidability of β-conversion restricted to the weakly normalising terms. In order to check if two weakly normalising terms are convertible, we compute their normal forms by applying the normalising strategy and we check whether the two normal forms are syntactically equal. Due to the fact that there exists a normalising strategy for the lambda calculus, β-conversion restricted to the β-weakly normalising λ-terms is decidable.

We also consider perpetual strategies, i.e. strategies that preserve the property of non-termination. In the term $(\lambda x.y)\Omega$, a perpetual strategy has to reduce Ω and not the leftmost redex. To have a perpetual strategy is important because a term is strongly normalising if and only if the perpetual strategy yields a finite reduction sequence.

In this thesis, we present new proofs of finiteness of developments and superdevelopments (a generalisation of developments) and of the fact that some strategies are perpetual.

1.2 Lambda Calculi with Types

Types were introduced for the first time in the combinatory logic (a variant of the lambda calculus) in [Cur34] and in the lambda calculus itself in [Chu40]. Nowadays types are used for many purposes as will be explained later. In order to give a first insight of the notion of type, we give one simple motivation related to set theory. The λ-terms do not represent the mathematical set-theoretic notion of function, with domain and range as part of the definition of the function. However, they can be modified to fit this notion by adding the notion of type.

We want to say that in the term $\lambda x.x^2$, the variable x ranges over the set of natural numbers. This could be expressed by considering a special symbol **Nat** which could be interpreted in set theory as the set of natural numbers. The λ-term is now written as $\lambda x : \mathbf{Nat}.x^2$. We read $x : \mathbf{Nat}$ as x has type **Nat** and the intended meaning of $x : \mathbf{Nat}$ is that the variable x ranges over the set of natural numbers.

We need another symbol $\rightarrow$ to express that this is a function from the natural numbers into the natural numbers.

$$\lambda x : \mathbf{Nat}.x^2 : \mathbf{Nat} \rightarrow \mathbf{Nat}$$

This is read as follows: the term $\lambda x : \mathbf{Nat}.x^2$ has type $\mathbf{Nat} \rightarrow \mathbf{Nat}$ and its intended meaning is that $\lambda x : \mathbf{Nat}.x^2$ is a function from the set of natural numbers into itself.

The types **Nat** and $\mathbf{Nat} \rightarrow \mathbf{Nat}$ should be syntactic expressions defined in a formal language with the notion of set as a possible interpretation for them. The lambda calculus with this kind of types is called *simply typed lambda calculus* (or $\lambda_{\rightarrow}$).

The two original papers by Curry and Church introduced two different families of lambda calculi with types. In the systems à la Church, the variable of the abstraction contains the information of its type like in the example $\lambda x : \mathbf{Nat}.x^2$. The terms of these systems usually have the property of uniqueness of types since the types of the variables determine the type of the whole term.

In the systems à la Curry, the types for the variables of the abstractions are not declared and the types assigned to the λ-terms are not unique. The term $\lambda x.x$ may have the type $\mathbf{Nat} \to \mathbf{Nat}$ but also may have the type $\alpha \to \alpha$ for an arbitrary type α.

Types may become more complex if we want to enrich the expressiveness of our language. The identity function $\lambda x{:}\mathbf{Nat}.x$ is 'the same function' as $\lambda x{:}\mathbf{Bool}.x$ except for the type. The language of types may be extended to express that the identity function may be applied to *any type*. The language of terms is extended with a new abstraction Λ and a quantifier on types $\forall$. The identity is written as

$$\Lambda\alpha.\lambda x{:}\alpha.\ x$$

The type of the identity is

$$\forall\alpha.(\alpha \to \alpha)$$

The fact that $\Lambda\alpha.\lambda x{:}\alpha.\ x$ has type $\forall\alpha.(\alpha \to \alpha)$ means that for any type α, the function $\lambda x{:}\alpha.\ x$ has type $(\alpha \to \alpha)$ The abstraction Λ and the quantifier $\forall$ are applied to type variables, i.e. a variable that can be substituted by types. In this example, the type variable α can be substituted by types like **Nat** and **Bool**.

$$(\Lambda\alpha.\lambda x{:}\alpha.\ x)\mathbf{Nat} \quad \to_\beta \quad \lambda x{:}\mathbf{Nat}.x$$

A function constructed with Λ is called *polymorphic*, i.e. the argument of the function is a type. For example, $(\Lambda\alpha.\lambda x{:}\alpha.\ x)$ is the polymorphic identity.

The extension of the simply typed lambda calculus with polymorphism is called *polymorphic typed lambda calculs* (F or $\lambda 2$ or *second order typed lambda calculus*) and it was introduced independently by [Gir72] and [Rey74].

Implementations of Lambda calculi

The λ-calculus is the foundation of functional programming languages. The λ-calculus itself could be considered as an abstract programming language. It contains the concept of computation in full generality and strength but in a pure form with a very simple syntax. The first typed programming languages developed to avoid typing errors at compile-time, were ALGOL-60 and PASCAL where variables have to be declared in the programs (typing à la Church). More sophisticated typed languages appeared later, like ML [HMM86], Miranda [Tur85] and Haskell [HW88]. The last mentioned languages are functional programming languages based on fragments of F à la Curry. In this approach, types are introduced in the programming language to ensure correctness of programs. Types are a way of classifying the objects to use them in a correct way. Hence the type of a program

gives a partial specification of the program. Terms are viewed as programs and types as specifications for the programs. Another approach to the notion of type is the so-called propositions-as-types interpretation [Bru70] [How80]. A type is viewed as a proposition and a term as its proof. The first systems of proof checking (type checking) based on this interpretation of propositions-as-types and proofs-as-terms were the systems of AUTOMATH [NGdV94]. Modern systems that also provide computer-assistance for the construction of proofs are Coq [Dow91], Lego [LP92], Constructor [HA91], Nuprl [Con86] and Alf [Mag94]. Coq is based on the calculus of constructions [CH88] extended with inductive definitions [CP90]. Lego is a proof assistant for the extended calculus of constructions [Luo90] with inductive types. Constructor is a partly automated proof assistant for pure type systems. The first version of Alf [ACN90] is based on Martin-Löf's type theory [NPS90] and the actual version [Mag94] is based on the monomorphic type theory with explicit substitution [Tas93].

1.2.1 Pure Type Systems

In this thesis we work with pure type systems. They provide a framework to describe a large class of type systems à la Church in a uniform way. They were introduced independently by S. Berardi [Ber88] (see also [Ber90]) and J. Terlouw [Ter89]. Many systems can be described in this way, for instance the simply and the polymorphic typed lambda calculus, the systems of the AUTOMATH family [NGdV94], the Calculus of Constructions (and all the systems of the λ-cube [Bar92]) and the inconsistent system $\lambda*$ [Gir72].

They are called 'pure' because there is only one type constructor and only one reduction rule, namely the type constructor Π and the β-reduction.

If A is a type and $B(x)$ is a family of types indexed over A then $\Pi x{:}A.\ B(x)$ is also a type. The term $\lambda x{:}A.\ b(x)$ has type $\Pi x{:}A.\ B(x)$ if for every a of type A, the term $b(a)$ has type $B(a)$.

In case the expression $B(x)$ does not depend on x, $\Pi x{:}A.\ B$ is equivalent to the ordinary function type $A \to B$.

In pure type systems we have only one rule for all possible 'functions types'. The typing rule for a product depends on some parameters. By instantiating the parameters, we obtain different product rules that allow to have different kinds of functions. For example the abstractions Λ and λ of the polymorphic typed lambda calculus are replaced by the unique abstraction λ of pure type systems and the type constructors $\to$ and $\forall$ are replaced by the unique type constructor Π. Since we have the same symbol for the abstraction of type variables and of term variables, we have to distinguish them in other way. When we write $\alpha : *$, we mean that α is a type variable. The polymorphic identity is written in a pure type system style using the abstraction λ instead of Λ and writing $\alpha : *$ for α.

$$\lambda\alpha{:}*.\lambda x{:}\alpha.x$$

The type of the polymorphic identity is written using the product Π instead of the quantifier

$\forall$ and writing $\alpha : *$ for α.

$$\Pi\alpha{:}*.(\Pi x{:}\alpha.\alpha)$$

One instantiation of the product rule allows to type the product $(\Pi x{:}\alpha.\alpha)$ that corresponds to $\alpha \to \alpha$ in the polymorphic typed lambda calculus. Another instantiation of the same rule allows to type $\Pi\alpha : *.(\Pi x : \alpha.\alpha)$ that corresponds to $\forall\alpha.(\alpha \to \alpha)$ in the polymorphic typed lambda calculus.

1.2.2 Definitions

Any programming language provides a mechanism to introduce definitions, i.e. an abbreviation or name for a larger term that can be used several times in a program. Definitions are usually considered in a meta-level and not as part of the language of the lambda calculus.

In the systems of the AUTOMATH family [NGdV94] definitions are considered as part of the formal language. The meta-theory of these systems is treated in detail in [Daa80]. However, some of the proofs apply only to the particular type system that they consider and do not extend to other type systems.

We consider it important to include definitions in the syntax of the lambda calculus and to study the properties that are preserved by the extension. This study can be done in a very general manner if we use the framework of pure type systems.

We suppose that we have a context or environment where we can introduce definitions. The definitions of the context are called *global definitions* and they can be used anywhere in the program or term. A global definition is written as $x{=}a{:}A$. This means that the name x is an abbreviation for the term a whose type is A. There are definitions that have a restricted scope and they are called *local definitions*. A local definition is written as $x{=}a{:}A$ *in* b. This means that x is an abbreviation for the term a whose type is A but it can be used only inside the term b. In our opinion it is important for practical use to have both ways of introducing definitions: global and local.

The intended meaning of a definition $x{=}a{:}A$ is that the *definiendum* x can be unfolded by the *definiens* a in its scope, either globally or locally. The unfolding of a definition is not the substitution of all the occurrences of the definition at once like for the β-reduction. Instead one occurrence of the definition is unfolded at a time. The reduction that performs the unfolding of definitions is called *δ-reduction*.

The equality $x = a$ can be used not only in the evaluation of a term but also in the typing of a term. We can use the fact that the definiendum and the definiens are equal in order to type terms that could not be typed otherwise. Suppose that the definition $x{=}a{:}*$ is a type and x occurs in another type B of the expression b, the typing of b may depend on the fact that we can use that x is equal to a.

A definition $x{=}a{:}A$ *in* b has a similar behaviour as $(\lambda x{:}A.\ b)a$. However the two facts mentioned above suggest that they are in fact different, both from the point of view of the evaluation and the typing.

We give an example in $\lambda 2$ that shows the importance and usage of definitions in the

evaluation and in the typing. First we introduce the global definition '**Bool**' as follows.

$$\mathbf{Bool} = \Pi\alpha : *.\alpha \to \alpha \to \alpha : *$$

The elements of **Bool** are **true** and **false** and they are given by the following global definitions.

$$\mathbf{true} \;=\; \lambda\alpha{:}*.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ x : \mathbf{Bool}$$

$$\mathbf{false} \;=\; \lambda\alpha{:}*.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ y : \mathbf{Bool}$$

Observe that in a definition we can make use of the definitions introduced previously, in the definitions of **true** and **false**, we make use of **Bool** that was defined before.

The **if-then-else** is also introduced as a global definition.

$$\mathbf{if\text{-}then\text{-}else} \;=\; \lambda\alpha{:}*.\ \lambda b{:}\mathbf{Bool}.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ (b\ \alpha\ x\ y)$$

$$: (\Pi\alpha{:}*.\ \mathbf{Bool} \to \alpha \to \alpha \to \alpha)$$

The function that yields the negation of a boolean is abbreviated by the global definition **not**.

$$\mathbf{not} = \lambda b : \mathbf{Bool}.\ (\mathbf{if\text{-}then\text{-}else}\ \mathbf{Bool}\ \ b\ \mathbf{false}\ \mathbf{true}) : \mathbf{Bool} \to \mathbf{Bool}$$

The term (**not true**) evaluates to **false** and the term (**not false**) evaluates to **true**.

In the evaluation of (**not true**) it is convenient to unfold one occurrence of **true** and to leave the other without being unfolded. In the following $\beta\delta$-rewrite sequence, we first unfold the definition of **not**, then we perform one β-reduction step, we unfold the definition of **if-then-else** and perform some β-reduction steps.

$$\begin{array}{ll}
(\mathbf{not\ true}) & \to_\delta \\
((\lambda b : \mathbf{Bool}.\ \mathbf{if\text{-}then\text{-}else}\ \mathbf{Bool}\ b\ \mathbf{false}\ \mathbf{true})\ \mathbf{true}) & \to_\beta \\
(\mathbf{if\text{-}then\text{-}else}\ \mathbf{Bool}\ \mathbf{true}\ \mathbf{false}\ \mathbf{true}) & \to_\delta \\
((\lambda\alpha{:}*.\ \lambda b{:}\mathbf{Bool}.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ (b\ \alpha\ x\ y))\ \mathbf{Bool}\ \mathbf{true}\ \mathbf{false}\ \mathbf{true}) & \twoheadrightarrow_\beta \\
(\mathbf{true}\ \mathbf{Bool}\ \mathbf{false}\ \mathbf{true}) &
\end{array}$$

In order to get the result, the first occurrence of **true** in (**true Bool false true**) should be unfolded but the second occurrence need not be unfolded. This shows the convenience of unfolding one occurrence of the definition at a time.

The terms defined above are all typable because we can use the fact that the definiendum and the definiens are equal.

For example, **true** would not have type **Bool** if we were not allowed to use the fact that the defiendum **Bool** is equal to its definiens $\Pi\alpha : *.(\alpha \to \alpha \to \alpha)$. The type of **true** is $\Pi\alpha{:}*.\alpha \to \alpha \to \alpha$ and by using the fact that **Bool** is equal to $\Pi\alpha : *.\alpha \to \alpha \to \alpha$, we deduce that **true** has type **Bool**.

The **if-then-else** would not be typable at all if we could not use the fact that the definiendum **Bool** is equal to its definiens $\Pi\alpha{:}*.(\alpha \to \alpha \to \alpha)$. In the definition of **if-then-else**, the type of the argument b is **Bool**. Since b is the operator of an application,

the type of b should be a function type and the symbol **Bool** is far from being a function type unless we can unfold its definition.

The typing rules for pure type systems are extended to include definitions. The rules have to be extended in such a way that the fact that the definiendum and the definiens are equal can be used in type derivation. This is achieved by adding the δ-conversion rule. In this thesis we study the behaviour of the δ-reduction and the combination of the δ with the β-reduction in pure type systems with definitions. The extension is done in such a way that almost all the properties that are valid for a pure type system are also valid for its extension with definitions.

1.2.3 Normalisation

We are interested in the property of normalisation in type systems. If all the terms that are typable in the type system are (strongly) normalising then the system itself is said to be (*strongly*) *normalising* .

Normalisation is important due to the fact that if the type system is normalising the conversion when restricted to the typable terms is decidable. If we want to check if two typable terms are convertible, we apply a reduction strategy to find their normal forms and check if the normal forms are syntactically equal. The operation that computes the normal form can be exponential (or even worse!) and so in practical situations it is advisable to find a good strategy for computing a common-reduct of two terms (not necessarily the normal form) in a reasonable time.

Strong normalisation is also important. The normalising strategies in lambda calculus, e.g. the leftmost reduction, are not always efficient. They may even take longer paths to the normal forms than other strategies. We would like to have the freedom to choose the strategy to compute the normal form (or some partial value). In this case, to ensure termination we have to prove that any strategy is normalising, i.e. strong normalisation.

Examples of pure type systems that are strongly normalising are the simply typed lambda calculus [Tai67] [Tro73], the systems of the AUTOMATH family [NGdV94], the polymorphic typed lambda calculus [Gir72], the Calculus of Constructions [GN91] (and all the systems of the λ-cube) and the 'pure' part of the extended calculus of constructions [Luo90]. However the pure type systems that are inconsistent like $\lambda*$ are not weakly normalising [Gir72]. We do not know any pure type system that is weakly normalising but not strongly normalising.

In this thesis we consider two extensions of pure type systems (one that is obtained by weakening the abstraction rule and the other by adding definitions) and prove that normalisation is preserved by these extensions. We also prove that the δ-reduction, the unfolding of definitions, is strongly normalising.

1.2.4 Type Inference

Look at the following problems related to type systems.

1. *Type checking.* Given a and b, does a have type b?

2. *Type inference.* Given a, does a have any type?

3. *Inhabitation.* Given b, is b the type of some term?

They are important in the implementations of functional programming languages, proof checkers and proof assistants.

Type inference for systems à la Curry is not always decidable, not even for normalising ones. For example, type inference for the simply typed lambda calculus is decidable [Cur69] [Mil78] but for $\lambda\omega$ (or $F\omega$) is undecidable [Urz93b]. An incomplete overview of the decidability of type inference in the systems à la Curry can be found in a table on page 183 in [Bar92]. This table can be filled in completely by now: type checking and inference in $\lambda 2$ (or F) have been proved undecidable in [Wel94] and the inhabitation of $\lambda\cap$ has been proved undecidable in [Urz93a]. Type inference for ML [DM82] is decidable because polymorphism can be used only in a weaker form (the universal quantifiers can occur only in the outermost position of a type).

For systems à la Church, in all known cases the problems of type checking and type inference are equivalent. Moreover decidability of type inference is very closely related to normalisation. Decidability of type inference for normalising pure type systems whose set of sorts is finite is proved in [BJ93] and decidability for normalising pure type systems whose specification is recursive and singly sorted (also semi-full) is proved in [Pol96]. Undecidability of type inference for inconsistent impredicative pure type systems is proved in [CH94] generalising the result in [MR86].

The problem of inhabitation in systems à la Church presents the same complications as for systems à la Curry. For the inconsistent systems, inhabitation is trivial since all the types have at least one inhabitant. In the systems of the λ-cube, inhabitation is decidable only for the simply typed lambda calculus($\lambda_{\rightarrow}$) and for $\lambda\underline{\omega}$. For the rest of the systems in the λ-cube inhabitation is undecidable [Spr95].

In this thesis, we give solutions for the problem of type inference in pure type systems (also with definitions). We present a type inference procedure (it is not an algorithm, i.e. a program that always terminates) for pure type systems. It can be applied to any singly sorted pure type system (systems with the uniqueness of types property), including the non-normalising ones. In order to prove that it behaves correctly (it terminates and yields the type of a term if the term is typable and it may not terminate otherwise), we use the results proved in this same thesis concerning normalisation.

1.3 Summary of the Contents of this Thesis

This thesis is divided into three parts: 1) an abstract presentation of rewriting and typing, 2) lambda calculus and 3) pure type systems with definitions.

An abstract presentation of rewriting and typing

In this part, the concepts of computation and typing are formalised in an abstract way as binary relations on an arbitrary set. The word *abstract* is due to the fact that we do not fix a set of terms or expressions, instead we take an arbitrary set. In this way, we give a common framework for the systems presented in the rest of the thesis. We believe that this abstract presentation gives uniformity and clarity to the exposition.

In **chapter 2**, we start by recalling the notion of *abstract rewriting systems*. The rewrite relation intends to model the concept of computation. This setting has been used to capture some properties of the concept of computation like confluence and normalisation in an abstract way.

In **chapter 3**, with the intention to clarify the basic abstract properties of confluence and normalisation we compare abstract rewriting systems with topological structures and find the relationship of the mentioned properties with well known topological concepts.

In **chapter 4**, as one of the main purposes of the thesis is to study the behaviour of confluence and normalisation under different kinds of extensions of pure type systems, we introduce the notion of *abstract rewriting systems with typing* which intends to model the interaction between the concepts of computation and typing.

Chapters 2 and 4 are partly joined work with Femke van Raamsdonk. We were both interested in an abstract notion for type systems in order to have a common framework for pure type systems and for higher order rewriting systems. For the use of these abstract notions in the context of higher order rewriting systems, we refer to [Raa96].

Chapter 3 is based on a paper with Walter Ferrer [FS93].

Lambda Calculus

In this part, we give some new characterisations of the set of weakly and strongly normalising λ-terms focusing on expansion rather than reduction. These characterisations of the set of strongly normalising λ-terms permit us to give new and simple proofs of classical results about λ-calculus. In most cases the new proofs are essentially simpler than the already existing ones and help us to understand not only *the mechanics* of the proofs of the results but also *the reasons* for their validity.

In **chapter 5**, we give new characterisations of the set of weakly and strongly normalising λ-terms.

In **chapter 6**, we define two perpetual strategies G_{bk} and G_∞ similar to F_{bk} [BK82] and F_∞ [BBKV76]. In order to prove that a strategy F is perpetual, we prove that if $F(M)$ is strongly normalising then so is M and we use one of the characterisations of the set of strongly normalising λ-terms given before. We also prove that the strategies G_{bk} and F_{bk} are maximal, i.e. the length of the G_{bk} and F_{bk}-reduction sequences are maximal.

In **chapter 7**, we give two proofs of finiteness of developments and superdevelopments. We define the $\underline{\beta}$-reduction as the β-reduction restricted to marked redexes. In order to prove finiteness of developments we have to prove that the $\underline{\beta}$-reduction is strongly normalising. In the first proof we define by induction a set that coincides with the set of $\underline{\beta}$-strongly

normalising terms, then we prove that this set is equal to the set of $\underline{\lambda}$-terms. In other words, we prove that all the $\underline{\lambda}$-terms are $\underline{\beta}$-strongly normalising. In the second proof we write a function from the set of $\underline{\lambda}$-terms to the set of strongly normalising λ-terms that preserves the reduction.

In **chapter 8**, we prove strong normalisation for the simply typed lambda calculus. We use again the new characterisation of the strongly normalising λ-terms.

This part is based on a paper with Femke van Raamsdonk [RS95].

Pure Type Systems with Definitions

In this part, we study the meta-theory of pure type systems with definitions in detail. We also give semi-algorithms of type inference for singly sorted pure type systems with and without definitions. A semi-algorithm of type inference is a program that terminates and infers the type if the term is typable, otherwise it may not terminate.

In **chapter 9** we recall the notion of *pure type systems*. First we look at the the notion of specification (the parameters of the typing rules for pure type systems.) We define the notion of morphism between specifications. Then we recall the typing rules for pure type systems. Pure type systems are obtained from the typing rules by instantianting the specification. We give some examples of specifications, and show which lambda calculi with types correspond to the typing rules of pure type systems instantiated with these specifications.

In **chapter 10** we define a function that infers the type for singly sorted pure type systems (systems with the uniqueness of types property). We weaken the rules of pure type systems by removing a premise from the abstraction rule. This premise, called the Π-*condition*, states that $(\Pi x{:}A.\ B)$ should be well-typed in order to be able to give the type $(\Pi x{:}A.\ B)$ to the abstraction $\lambda x{:}A.\ b$. We study the metatheory of the *pure type systems without the* Π-*condition*. We prove that if a singly sorted pure type system is normalising then so is the corresponding pure type system without the Π-condition. Using this result we define a set of rules for pure type systems that are *syntax directed*, i.e. the last rule in a type derivation is determined by the shape of the term and the context. We prove the equivalence between the syntax directed set of rules and the original ones. Finally, we define a function that infers the type in a singly sorted pure type system based on this syntax directed set of rules. This chapter is based on the paper [Sev96].

In **chapter 11**, we study the metatheory for *pure type systems extended with definitions*. We prove properties like confluence and subject reduction for the combination of the β and δ-reduction. We prove strong normalisation for the δ-reduction and define a perpetual and maximal strategy for the δ-reduction similar to F_{bk} for β-reduction. Also, we prove that weak normalisation is preserved by the extension, i.e. if a pure type system is weakly normalising then so is its extension with definitions. Moreover, we prove for certain pure type systems, including the Calculus of Constructions, that strong normalisation is preserved by the extension. This chapter is based on a paper with Erik Poll [SP93, SP94].

In **chapter 12**, we define a function that infers the type of a term in a singly sorted pure type system with definitions. Similarly to pure type systems, we define a set of rules

for pure type systems with definitions that is *syntax directed* and prove the equivalence between the syntax directed set of rules and the original ones. We define a function that *infers the type* in a singly sorted pure type system with definitions based on this syntax directed set of rules.

Part I

An Abstract Presentation of Rewriting and Typing

Chapter 2

Abstract Rewriting Systems

2.1 Introduction

The concept of **computation** can be modelled in an abstract way as a binary relation on a set. This relation is called the *rewrite relation*, usually denoted by $\rightarrow$. A pair consisting of a set and a binary relation is called an *abstract rewriting system*. We are interested, for example, in computing the value of an element. A computation is represented by a sequence of elements $a_0 \rightarrow a_1 \rightarrow a_2 \ldots \rightarrow a_n$ whose last element a_n represents the value of the computation. Several properties can be studied in this abstract setting, like confluence, weak and strong normalisation. A final value of a computation can be represented by an element that cannot be reduced any further and is called *normal form*. If all the elements have a normal form then the system is said to be *weakly normalising*. The intuitive meaning of *confluence* is that any procedure that computes the value of an element yields the same result, the intuitive meaning of *strong normalisation* is that any procedure is finite.

This setting allows us to give criteria to prove properties like confluence, weak and strong normalisation in a very general way.

We summarise the contents of the sections of this chapter. In section 2.2, we recall the basic concepts concerning abstract rewriting systems. In section 2.3 we define the notions of morphism between abstract rewriting systems and of rewrite sequences. In section 2.4 we recall the concepts of confluence, weak and strong normalisation. We also give some general criteria to prove confluence, weak and strong normalisation.

2.2 Abstract Rewriting Systems

In this section we recall the definition of abstract rewriting system. The abstract notion of rewrite relation was first formalised by Newman (see [New42]) under the name of 'move' or indexed 1-complex. They are also called abstract reduction systems in [Klo80]. We call them abstract rewriting systems as in [Oos94].

Definition 2.2.1. An *abstract rewriting system* is a structure $(A, \rightarrow)$ where A is a set of objects and $\rightarrow$ is a subset of $A \times A$ called a *rewrite relation* (or *reduction*).

Definition 2.2.2. We say that $(a, b) \in \rightarrow$ is a *rewrite step* in the abstract rewriting system $(A, \rightarrow)$. We write $a \rightarrow b$ instead of $(a, b) \in \rightarrow$.

The reflexive closure of $\rightarrow$ is denoted by $\rightarrow^{=}$. The transitive closure of $\rightarrow$ is denoted by $\rightarrow^{+}$. The transitive-reflexive closure of $\rightarrow$ is written as $\twoheadrightarrow$. The inverse relation of $\rightarrow$ is denoted as $\leftarrow$. The equivalence relation generated by $\rightarrow$ is written as $\leftrightarrow$ and called *conversion*.

Example 2.2.3.

1. For each $n \in \mathbb{N}$ we define the abstract rewriting system,

$$\mathcal{I}_n = (\{i \mid 0 \leq i \leq n\}, \{(i, i+1) \mid 0 \leq i \leq (n-1)\})$$

   Diagrammatically,

$$\begin{array}{ll} \mathcal{I}_1 & 0 \rightarrow 1 \\ \mathcal{I}_2 & 0 \rightarrow 1 \rightarrow 2 \\ \vdots & \\ \mathcal{I}_n & 0 \rightarrow 1 \rightarrow 2 \rightarrow \ldots \rightarrow n \end{array}$$

2. $\mathcal{I} = (\mathbb{N}, \{(i, i+1) \mid i \geq 0\})$. Diagrammatically,

$$0 \rightarrow 1 \rightarrow 2 \rightarrow \ldots$$

2.3 Morphisms

In this section we introduce the notion of morphism for abstract rewriting systems (see also [Raa96]). We think that the concept of morphism is the natural one. It allows us to express several concepts like the notions of rewrite sequence and extension as morphisms.

A morphism should preserve the structure. For abstract rewriting systems a morphism is a function between sets that preserves the rewrite relation $\rightarrow$. These are the morphisms associated to the category of abstract rewriting systems. In case the function preserves other relations like $\rightarrow^{+}$, $\twoheadrightarrow$, we call it refining and implementing morphism respectively. Morphisms that preserve conversion $\leftrightarrow$ are defined later.

Definition 2.3.1. Let $(A, \rightarrow_\alpha)$ $(B, \rightarrow_\beta)$ be two abstract rewriting systems. A *morphism* from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ is a function $f : A \rightarrow B$ such that for all $a, a' \in A$ if $a \rightarrow_\alpha a'$ then $f(a) \rightarrow_\beta f(a')$.

The category whose objects are the abstract rewriting systems and the morphisms are as defined above is denoted as **Ars**.

We define different kinds of morphisms between abstract rewriting systems depending on the relation they preserve.

Definition 2.3.2. Let $(A, \rightarrow_\alpha)$ and $(B, \rightarrow_\beta)$ be two abstract rewriting systems.

A *refining morphism* from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ is a function $f : A \rightarrow B$ such that for all $a, a' \in A$ if $a \rightarrow_\alpha a'$ then $f(a) \rightarrow_\beta^+ f(a')$.

An *implementing morphism* from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ is a function $f : A \rightarrow B$ such that for all $a, a' \in A$ if $a \rightarrow_\alpha a'$ then $f(a) \twoheadrightarrow_\beta f(a')$.

A *forgetting morphism* from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ is a function $f : A \rightarrow B$ such that for all $a, a' \in A$ if $a \rightarrow_\alpha a'$ then $f(a) = f(a')$. By $=$, we mean equality between elements in a set.

Definition 2.3.3. Let $(A, \rightarrow_\alpha)$ and $(B, \rightarrow_\beta)$ be two abstract rewriting systems.

We say that $(B, \rightarrow_\beta)$ is an *extension* of $(A, \rightarrow_\alpha)$ if $A \subset B$ and the inclusion mapping is a morphism from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$.

We say that $(B, \rightarrow_\beta)$ is a *conservative extension of* $(A, \rightarrow_\alpha)$ if $(B, \rightarrow_\beta)$ is an extension of $(A, \rightarrow_\alpha)$ and for all $a, a' \in A$, if $a \rightarrow_\beta a'$ then $a \rightarrow_\alpha a'$.

We say that $(B, \rightarrow_\beta)$ is a *strong conservative extension of* $(A, \rightarrow_\alpha)$ if $(B, \rightarrow_\beta)$ is a conservative extension of $(A, \rightarrow_\alpha)$ and A is closed under $\rightarrow_\beta$, i.e. if $a \in A$ and $a \rightarrow_\beta a'$ then $a' \in A$.

A finite computation is represented by a finite rewrite sequence of elements $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots a_n$ in an abstract rewriting system. We will formally define a finite rewrite sequence in an abstract rewriting system $(A, \rightarrow)$ as a morphism from $\mathcal{I}_n$ to $(A, \rightarrow)$.

A computation can also be infinite and it is represented by an infinite rewrite sequence $a_0 \rightarrow a_1 \rightarrow \ldots$. We will formally define an infinite rewrite sequence in an abstract rewriting system $(A, \rightarrow)$ as a morphism from $\mathcal{I}$ to $(A, \rightarrow)$.

Definition 2.3.4. Let $(A, \rightarrow)$ be an abstract rewriting system, $a \in A$ and $n \in \mathbb{N}$.

A *rewrite sequence of length n starting at a* is a triple (a, n, σ) such that σ is a morphism from $\mathcal{I}_n$ to $(A, \rightarrow)$ and $\sigma(0) = a$. In a diagram:

$$\begin{array}{ccccccccc} 0 & \rightarrow & 1 & \rightarrow & 2 & \ldots & \rightarrow & n \\ \sigma(0) & \rightarrow & \sigma(1) & \rightarrow & \sigma(2) & \ldots & \rightarrow & \sigma(n) \end{array}$$

We denote a rewrite sequence (a, n, σ) as $\sigma : \sigma(0) \rightarrow \sigma(1) \rightarrow \ldots \rightarrow \sigma(n)$ or $\sigma : \sigma(0) \twoheadrightarrow \sigma(n)$.

A *rewrite sequence of infinite length starting at* a is a pair (a, σ) such that σ is a morphism from $\mathcal{I}$ to $(A, \rightarrow)$ and $\sigma(0) = a$. In a diagram:

$$\begin{array}{ccccccc} 0 & \rightarrow & 1 & \rightarrow & 2 & \ldots \\ \\ \sigma(0) & \rightarrow & \sigma(1) & \rightarrow & \sigma(2) & \ldots \end{array}$$

We denote an infinite rewrite sequence (a, σ) as $\sigma : \sigma(0) \rightarrow \sigma(1) \rightarrow \ldots$.

The *length* of σ is denoted by $\|\sigma\|$. We have that $\|\sigma\|$ is either a natural number or ∞.

The *domain* of σ is denoted by $dom(\sigma)$.

Note that a morphism $f : A \rightarrow B$ from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ 'preserves rewrite sequences'. If (a, n, σ) is a finite rewrite sequence in $(A, \rightarrow_\alpha)$ then $(f(a), n, f \circ \sigma)$ is a rewrite sequence in $(B, \rightarrow_\beta)$. If (a, σ) is an infinite rewrite sequence in $(A, \rightarrow_\alpha)$ then $(f(a), f \circ \sigma)$ is an infinite rewrite sequence.

In the following definition, we introduce the notion of lifting which will be used to define the notion of development in chapter 7.

Definition 2.3.5. Let $f : A \rightarrow B$ be a morphism from the abstract rewriting system $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$. A rewrite sequence σ in $(A, \rightarrow_\alpha)$ is an *f-lifting* of a rewrite sequence ρ if $f \circ \sigma = \rho$.

The notion of lifting for rewrite sequence has been defined in [RS95] in the context of indexed abstract rewriting systems. Note that this is the categorical notion of lifting for morphisms.

Definition 2.3.6. A rewrite sequence $\sigma : a \twoheadrightarrow b$ is *maximal* if for all $\rho : a \twoheadrightarrow b$ we have $\|\sigma\| \geq \|\rho\|$.

2.4 Properties of Abstract Rewriting Systems

In this section we define the basic properties of confluence, weak and strong normalisation in an abstract rewriting system.

An element that cannot be reduced any further is called *normal form* and it can be viewed as the final value of the computation. If all the elements have a computation that ends in a normal form then the system is said to be weakly normalising. Other important properties of abstract rewriting systems are confluence and strong normalisation.

Definition 2.4.1. Let $(A, \rightarrow)$ be an abstract rewriting system and $a \in A$.

We say that a is *confluent* if for all $b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, there exists an element d such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

We say that a is a *normal form* (or *$\rightarrow$-normal form*) if there is no b such that $a \rightarrow b$.

We say that a *has a normal form* (or *has a* $\rightarrow$*-normal form*) if there exists a normal form b such that $a \twoheadrightarrow b$.

We say that a is *weakly normalising* (or $\rightarrow$*-weakly normalising*) if a has a normal form.

We say that a is *strongly normalising* (or $\rightarrow$*-strongly normalising*) if there is no infinite rewrite sequence starting at a.

The notions of confluence, weak and strong normalisation can be extended to abstract rewriting systems.

Definition 2.4.2. Let $(A, \rightarrow)$ be an abstract rewriting system.

We say that $(A, \rightarrow)$ or $\rightarrow$ is *confluent* if for all $a \in A$, a is confluent.

We say that $(A, \rightarrow)$ or $\rightarrow$ is *weakly normalising* if for all $a \in A$, a is $\rightarrow$-weakly normalising.

We say that $(A, \rightarrow)$ or $\rightarrow$ is *strongly normalising* if for all $a \in A$, a is $\rightarrow$-strongly normalising.

In the following we define the simple and transitive reductions graphs of an element which represent the set of values of all the computations starting from this element. We also define the simple and transitive expansion graphs of an element which represent the set of inputs whose computation yields the element.

Definition 2.4.3. Let $(A, \rightarrow)$ be an abstract rewriting system and $a \in A$. We define the following subsets of A.

a) $\mathcal{G}_{\rightarrow}(a) = \{b \in A \mid a \rightarrow b\}$ and $\mathcal{G}_{\twoheadrightarrow}(a) = \{b \in A \mid a \twoheadrightarrow b\}$. We call them *the simple and the transitive reduction graphs of* a.

b) $\mathcal{E}_{\rightarrow}(a) = \{b \in A \mid b \rightarrow a\}$ and $\mathcal{E}_{\twoheadrightarrow}(a) = \{b \in A \mid b \twoheadrightarrow a\}$. We call them *the simple and the transitive expansion graphs of* a.

Note that $\mathcal{E}_{\rightarrow}(a) = \mathcal{G}_{\leftarrow}(a)$ and that $\mathcal{E}_{\twoheadrightarrow}(a) = \mathcal{G}_{\twoheadleftarrow}(a)$.

Definition 2.4.4. Let $(A, \rightarrow_\alpha)$ and $(B, \rightarrow_\beta)$ be two abstract rewriting systems. The *union* of $(A, \rightarrow_\alpha)$ and $(B, \rightarrow_\beta)$ is defined by $(A \cup B, \rightarrow_\alpha \cup \rightarrow_\beta)$. We write $\rightarrow_{\alpha\beta}$ instead of $\rightarrow_\alpha \cup \rightarrow_\beta$.

2.5 Strategies

In this section, we define the notion of strategy. A strategy is a procedure that determines the way we reduce an element.

Definition 2.5.1.

1. A *strategy for the rewrite relation* $\rightarrow$ is a mapping $f : A \rightarrow A$ such that for all $a \in A$ $a \twoheadrightarrow f(a)$.

2. A *one-step strategy for the rewrite relation* $\rightarrow$ is a mapping $f : A \rightarrow A$ such that for all $a \in A$ not in normal form, we have that $a \rightarrow f(a)$.

In chapter 6, we need to consider strategies that yield a set of reducts instead of only one. For that reason, we introduce the following definition.

Definition 2.5.2.

1. A *non-deterministic strategy for the rewrite relation* $\rightarrow$ is a mapping $F : A \rightarrow \mathcal{P}(A)$ such that for all $a \in A$, $b \in F(a)$ we have that $a \twoheadrightarrow b$.

2. A *non-deterministic one-step strategy for the rewrite relation* $\rightarrow$ is a mapping $F : A \rightarrow \mathcal{P}(A)$ such that for all $a \in A$ not in normal form, we have that $F(a) \neq \emptyset$ and for all $b \in F(a)$, $a \rightarrow b$.

For example, the function defined by $F(a) = \{a\}$ for all $a \in A$ is a (trivial) non-deterministic strategy.

Let f be a one-step strategy. We define an f-rewrite sequence starting from a. Intuitively, an f-rewrite sequence is a sequence of the form

$$a \rightarrow f(a) \rightarrow f^2(a) \rightarrow \ldots$$

possibly ending in the normal form of a.

Definition 2.5.3. Let F be a (non-deterministic) one-step strategy for $\rightarrow$. We say that a rewrite sequence σ is an *F-rewrite sequence* if for all $n \in dom(\sigma)$ such that $n > 0$, $\sigma(n) \in F(\sigma(n-1))$.

Definition 2.5.4. A (non-deterministic) one-step strategy F is *maximal* if all the F-rewrite sequence starting are maximal.

Definition 2.5.5. A non-deterministic strategy $F : A \rightarrow \mathcal{P}(A)$ is called *normalising* if for all $a \in A$ such that a is weakly normalising, there exists an F-rewrite sequence from a to a normal form.

The importance of the existence of normalising strategies in weakly normalising abstract rewriting systems is that the decidability of the equality on normal forms implies the decidability of the conversion. In order to check if two elements are convertible, we compute their normal forms by applying the normalising strategy and we check that the two normal forms are equal.

Definition 2.5.6. A non-deterministic strategy $F : A \rightarrow \mathcal{P}(A)$ is called *perpetual* if for all $a \in A$ such that a is not strongly normalising we have that all the elements in $F(a)$ are not strongly normalising.

For a perpetual strategy F, if a is not strongly normalising then all the F-rewrite sequences starting at a are infinite.

Definition 2.5.7. A non-deterministic strategy $F : A \to \mathcal{P}(A)$ is called *confluent* if for all $a, b \in A$ such that $a \twoheadleftarrow\!\twoheadrightarrow b$ implies that the F-rewrite sequences starting at a and b intersect.

If $(A, \to)$ is confluent then the simple graph $\mathcal{G}_{\to}$ is a confluent strategy.

Definition 2.5.8. A *common-reduct strategy* is a function $F : A \times A \to \mathcal{P}(A)$ if for all $a, b \in A$ such that $a \twoheadleftarrow\!\twoheadrightarrow b$ then for all $c \in F(a, b)$ we have that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.

The importance of the existence of a common-reduct strategy F is that we can check if two elements are convertible in a confluent abstract rewriting system. In order to check if two elements a and b are convertible, we check if $F(a, b)$ is not the empty set.

In the next lemma we prove that confluent strategies are particular cases of common-reduct strategies. This is evident since a confluent strategy has only one argument and a common-reduct strategy has two.

Lemma 2.5.9. If $F : A \to \mathcal{P}(A)$ is a confluent strategy then $G : A \times A \to \mathcal{P}(A)$ defined by $G(a, b) = F^n(a) \cap F^m(b)$ where n, m are the least natural numbers such that $F^n(a) \cap F^m(b) \neq \emptyset$ is a common-reduct strategy.

An example of a common-reduct strategy is $F : A \times A \to \mathcal{P}(A)$ defined from the simple graph $\mathcal{G}_{\to}$ by $F(a, b) = \mathcal{G}^n_{\to}(a) \cap \mathcal{G}^m_{\to}(b)$ where n, m are the least natural numbers such that $\mathcal{G}^n_{\to}(a) \cap \mathcal{G}^m_{\to}(b) \neq \emptyset$.

More examples of common-reduct strategies are given in chapters 10 and 12.

2.6 Criteria

In this section we give some criteria to prove confluence and normalisation that will be used in the following chapters of this thesis. They are expressed in terms of the notion of morphism. We study the manner in which the properties of confluence, weak and strong normalisation are transported from one abstract rewriting system $(A, \to_\alpha)$ to the abstract rewriting system $(B, \to_\beta)$ depending on the class of morphism we can find from $(A, \to_\alpha)$ to $(B, \to_\beta)$.

Forgetting morphisms give rise to a criterion for confluence provided they are strategies. This criterion is used in chapter 11 in the proof of confluence for the δ-reduction.

Lemma 2.6.1. (Confluence Criterion) Let $(A, \to_\alpha)$ be an abstract rewriting system. If there is a forgetting morphism $f : A \to A$ which is a strategy for $\to_\alpha$, then $\to_\alpha$ is confluent.

Proof: The proof is illustrated by the following diagram.

$$\begin{tikzcd}
 & a \arrow[dl, "\alpha"'] \arrow[dr, "\alpha"] \arrow[dd, "\alpha"] & \\
a_1 \arrow[d, "\alpha"] & & a_2 \arrow[d, "\alpha"] \\
f(a_1) = & f(a) & = f(a_2)
\end{tikzcd}$$

□

Implementing morphisms give rise to a criteria for confluence provided they are strategies. This criterion is used in chapter 11 in the proof of confluence for the $\beta\delta$-reduction.

Lemma 2.6.2. (Confluence Criteria) Let $(A, \rightarrow_\alpha)$ and $(B, \rightarrow_\beta)$ be abstract rewriting systems and suppose that there are implementing morphisms $f : A \rightarrow B$ and $g : B \rightarrow A$ such that $f \circ g : B \rightarrow B$ is a strategy for $\rightarrow_\beta$. If $\rightarrow_\alpha$ is confluent then so is $\rightarrow_\beta$.

Proof: Suppose that $b \twoheadrightarrow_\beta b_1$ and $b \twoheadrightarrow_\beta b_2$. Since g is an implementing morphism from $(B, \rightarrow_\beta)$ to $(A, \rightarrow_\alpha)$ we have that $g(b) \twoheadrightarrow_\alpha g(b_1)$ and $g(b) \twoheadrightarrow_\alpha g(b_2)$. Since $\rightarrow_\alpha$ is confluent we have that there exists $a \in A$ such that:

$$\begin{tikzcd}
 & g(b) \arrow[dl, "\alpha"'] \arrow[dr, "\alpha"] & \\
g(b_1) \arrow[dr, dotted, "\alpha"'] & & g(b_2) \arrow[dl, dotted, "\alpha"] \\
 & a &
\end{tikzcd}$$

Since $f \circ g$ is a strategy for $\rightarrow_\beta$ and f is an implementing morphism from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ we have the following picture:

$$\begin{tikzcd}
 & b \arrow[dl, "\beta"'] \arrow[dr, "\beta"] \arrow[dd, "\beta"] & \\
b_1 \arrow[dd, "\beta"'] & & b_2 \arrow[dd, "\beta"] \\
 & f \circ g(b) \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
f \circ g(b_1) \arrow[dr, dotted, "\beta"'] & & f \circ g(b_2) \arrow[dl, dotted, "\beta"] \\
 & f(a) &
\end{tikzcd}$$

□

Strategies from one abstract rewriting system to a conservative extension give rise to a weak normalisation criterion.

Lemma 2.6.3. (Weak Normalisation Criterion) Let $(B, \rightarrow_\beta)$ be a strong conservative extension of the abstract rewriting system $(A, \rightarrow_\alpha)$. Suppose there exists a strategy $f : B \rightarrow B$ for $\rightarrow_\beta$ such that $f(B) \subset A$. If $\rightarrow_\alpha$ is weakly normalising then so is $\rightarrow_\beta$.

Proof: Suppose $b \in B$. Then $b \twoheadrightarrow_\beta f(b)$ and $f(b) \in A$. Since $\rightarrow_\alpha$ is weakly normalising there exists an $\rightarrow_\alpha$-normal form $c \in A$ of $f(b)$. Since $\rightarrow_\alpha \subset \rightarrow_\beta$ we have that $f(b) \twoheadrightarrow_\beta c$.

Since $(B, \rightarrow_\beta)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$ and $c \in A$ is an $\rightarrow_\alpha$-normal form, we have that c is an $\rightarrow_\beta$-normal form. Suppose towards a contradiction that c is not an $\rightarrow_\beta$-normal form. Then there exists d such that $c \rightarrow_\beta d$. Since A is closed under $\rightarrow_\beta$, we have that $d \in A$. Since $\rightarrow_\beta \cap (A \times A) \subset \rightarrow_\alpha$ we have that $c \rightarrow_\alpha d$. This contradicts the fact that c is an $\rightarrow_\alpha$-normal form. □

For proving preservation of weak normalisation from one abstract rewriting system $(A, \rightarrow_\alpha)$ to another abstract rewriting system $(B, \rightarrow_\rho)$, the requirement that $(B, \rightarrow_\rho)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$ can be weakened. Assume we can split the rewrite relation $\rightarrow_\rho$ into two relations $\rightarrow_\beta$ and $\rightarrow_\delta$. We require that only $(B, \rightarrow_\beta)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$. In this case we need $\rightarrow_\delta$ to be weakly normalising. This criterion is used in chapter 4 to prove a criterion that is applied in chapter 11 in the proof of weak normalisation for pure type systems with definitions.

Lemma 2.6.4. (Weak Normalisation Criterion) Let $(B, \rightarrow_\beta)$ be a strong conservative extension of the abstract rewriting system $(A, \rightarrow_\alpha)$ and let $(B, \rightarrow_\delta)$ be an abstract rewriting system. Suppose the following conditions are verified.

a) The relation $\rightarrow_\delta$ is weakly normalising.

b) For all $b \in B$ such that b is a δ-normal form, we have that $b \in A$.

c) If a is in δ-normal form and $a \rightarrow_\alpha a'$ then a' is in δ-normal form.

If $\rightarrow_\alpha$ is weakly normalising then so is $\rightarrow_{\beta\delta}$.

Proof: Suppose $b \in B$. By a) there exists d such that d is the δ-normal form of b. Then $b \twoheadrightarrow_\delta d$. It follows from b) that $d \in A$. Since $\rightarrow_\alpha$ is weakly normalising there exists c an $\rightarrow_\alpha$-normal form of d. Since $\rightarrow_\alpha \subset \rightarrow_\beta$ we have that $d \twoheadrightarrow_\beta c$. Since $(B, \rightarrow_\beta)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$, c is in $\rightarrow_\beta$-normal form. By c) we also have that c is in $\rightarrow_\delta$-normal form. □

Refining morphisms give rise to a criterion for strong normalisation since they map infinite rewrite sequences into infinite rewrite sequences. This criterion is used in chapter 7 in the second proof of finiteness of developments and superdevelopments.

Lemma 2.6.5. (Strong Normalisation Criterion) Let $(A, \rightarrow_\alpha)$, $(B, \rightarrow_\beta)$ be two abstract rewriting systems and suppose there is a refining morphism $f : B \rightarrow A$.

If $\rightarrow_\alpha$ is strongly normalising then so is $\rightarrow_\beta$.

In order to prove the preservation of strong normalisation from one abstract rewriting system $(A, \rightarrow_\alpha)$ to another abstract rewriting system $(B, \rightarrow_\rho)$, in the case it is not possible to find a refining morphism from $(B, \rightarrow_\rho)$ to $(A, \rightarrow_\alpha)$, a method is to split the reduction $\rightarrow_\rho$ into two reductions $\rightarrow_\beta$ and $\rightarrow_\delta$ and find a refining morphism from $(B, \rightarrow_\beta)$ to $(A, \rightarrow_\alpha)$. In the case that $\rightarrow_\delta$ is strongly normalising, we have a strong normalisation criterion. This criterion is used to prove the criterion of chapter 4 that is used in the proof of strong normalisation for pure type systems with definitions in chapter 11.

Lemma 2.6.6. (Strong Normalisation Criterion) Let $(A, \rightarrow_\alpha)$, $(B, \rightarrow_{\beta\delta})$ be abstract rewriting systems. Suppose there is a mapping $f : B \rightarrow A$ such that:

a) f is an implementing morphism from $(B, \rightarrow_\delta)$ to $(A, \rightarrow_\alpha)$.

b) f is a refining morphism from $(B, \rightarrow_\beta)$ to $(A, \rightarrow_\alpha)$.

If $\rightarrow_\alpha$ and $\rightarrow_\delta$ are strongly normalising then $\rightarrow_{\beta\delta}$ is strongly normalising.

Proof: Suppose towards a contradiction that $\rightarrow_{\beta\delta}$ is not strongly normalising, i.e. there is an infinite $\rightarrow_{\beta\delta}$-rewrite sequence starting at $b \in B$.

Observe that the number of β-rewrite steps in this sequence is infinite, i.e. $\forall n \in \mathbb{N} \;\; \exists m > n : b_m \rightarrow_\beta b_{m+1}$. Otherwise it would follow that there is $n_0 \in \mathbb{N}$ such that $\forall m > n_0 \; b_m \rightarrow_\delta b_{m+1}$. Hence the sequence $b_{n_0+1} \rightarrow_\delta b_{n_0+2} \rightarrow_\delta \ldots$ would be infinite. As $\rightarrow_\delta$ is strongly normalising, this can not happen. Hence the number of β-rewrite steps in the sequence $b \rightarrow_{\beta\delta} b_1 \rightarrow_{\beta\delta} b_2 \ldots$ is infinite. Then this sequence is of the form

$$b \twoheadrightarrow_\delta b_{n_1} \rightarrow_\beta b_{n_2} \twoheadrightarrow_\delta b_{n_3} \rightarrow_\beta b_{n_4} \twoheadrightarrow_\delta b_{n_5} \rightarrow_\beta b_{n_6} \twoheadrightarrow_\delta \ldots$$

By hypothesis **a)** and **b)** there is an infinite α-rewrite sequence starting at $f(b)$:

$$f(b) \twoheadrightarrow_\alpha f(b_{n_1}) \rightarrow_\alpha^+ f(b_{n_2}) \twoheadrightarrow_\alpha f(b_{n_3}) \rightarrow_\alpha^+ f(b_{n_4}) \twoheadrightarrow_\alpha f(b_{n_5}) \rightarrow_\alpha^+ f(b_{n_6}) \twoheadrightarrow_\alpha \ldots$$

which contradicts the assumption that $\rightarrow_\alpha$ is strongly normalising. $\square$

2.7 Conclusions and Related Work

In this chapter, we have presented the notion of abstract rewriting system in a categorical way, by introducing different types of morphisms and we have proved some general lemmas concerning confluence and normalisation.

In the literature abstract rewriting systems are not usually presented in a categorical manner. We have introduced the notion of morphism and considered the category of abstract rewriting systems. In our opinion, the categorical presentation is more elegant. For

example the notion of rewrite sequence can be defined economically in terms of morphisms. The application of a function to a rewrite sequence is the composition of two morphisms. The notion of lifting of a development coincides in this way with the categorical notion of lifting.

We have introduced the general notion of common-reduct strategy in order to use it later in the definitions of the type inference semi-algorithms. A common-reduct strategy can be used to check conversion in confluent abstract rewriting systems, if F is a common-reduct strategy, the terms a and b are convertible if and only if $F(a, b) \neq \emptyset$.

The Church-Rosser strategy presented in [Bar85] is a particular case of a common-reduct strategy for β-reduction. The main difference between a common-reduct strategy and a Church-Rosser strategy is that the former depends on two arguments and the latter on one. The normalising strategies are important for checking conversion but they ensure termination only on the set of weakly normalising elements.

The way in which conversion is checked in [Coq91] supposes implicitly the existence of a common-reduct strategy. In [Coq91], an algorithm for checking conversion is defined. For that purpose a computable binary relation is defined which is equivalent to the $\beta\eta$-reduction. This relation depending on two arguments can be considered as a common-reduct strategy. We define this strategy for β in chapter 10 and a similar one for $\beta\delta$ in chapter 12. Also in these chapters we give more examples of common-reduct strategies for β and $\beta\delta$-reduction.

The criteria to prove confluence, weak and strong normalisation have been obtained generalising the original proofs done for pure type systems with definitions (see [SP94]). Many other criteria came up as generalisations of existing proofs (see [New42], [Klo90], [Oos94] and [GLM92]). The proofs of [SP94] are rewritten as applications of the new criteria in chapter 11. Moreover these criteria will also be applied to other particular cases.

Chapter 3

Topology

3.1 Introduction

In this chapter we dress many of the basic properties of abstract rewriting systems in a topological costume. It will not be formally used later and the readers who are not interested can skip it.

We define a topology $(A, \mathcal{T}_{\rightarrow})$ associated to an abstract rewriting system $(A, \rightarrow)$. The closed sets of this topology are those subsets of A that are closed under the rewrite relation $\rightarrow$ and the open sets are those subsets of A that are closed under the inverse relation $\leftarrow$. The closure operator applied to an element $a \in A$ is the transitive reduction graph of the element, $\mathcal{G}_{\rightarrow}(a)$. This is the smallest closed set that contains a. In the context of preorders, the topology associated to a preorder is called the Alexandroff topology. This topology is very special in the sense that the open sets are closed under arbitrary intersections (not only finite ones) and there is another operator besides the closure that yields the expansions of a set.

We give some topological characterisations of confluence. An abstract rewriting system $(A, \rightarrow)$ is confluent if and only if the topology associated to $(A, \rightarrow)$ verifies that the intersection of any pair of closed sets that are subsets of $\mathcal{G}_{\rightarrow}(a)$ is non-empty or in informal language if the closed subsets of $\mathcal{G}_{\rightarrow}(a)$ are 'large'. Moreover, we prove that an abstract rewriting system $(A, \rightarrow)$ is confluent if and only if the two operators in the topology associated to $(A, \rightarrow)$ verify a subcommutation condition.

We also give characterisations of normal forms and the properties of weak and strong normalisation for what we call irreflexive abstract rewriting systems.

A normal form in an irreflexive abstract rewriting system is a closed point in the topology associated to the abstract rewriting system.

An irreflexive abstract rewriting system is weakly normalising if and only if all the closed sets of the associated topology contain a closed point. Another characterisation states that an irreflexive abstract rewriting system is weakly normalising if and only if A is the result of applying the expansion operator to the set of closed points.

An irreflexive abstract rewriting system is strongly normalising if and only if the fol-

lowing two conditions are verified:

1. The family of principal closed open sets is noetherian.
2. The closure of two points is the same if and only if they are the same point.

This chapter is organised as follows. In section 3.2 we define the topology associated to an abstract rewriting system and the abstract rewriting system associated to a topology. In section 3.3, we show that there is an equivalence between the preorders and symmetric topological spaces. In section 3.4, we give topological characterisations of confluence, weak and strong normalisation.

3.2 Topology

In this section we associate a topology $\mathcal{T}_{\rightarrow}$ to an abstract rewriting system $(A, \rightarrow)$. The closed sets of this topology are the subsets of A that are closed under the rewrite relation $\rightarrow$. The closure operator is the extension of the transitive graph of an element of A to subsets of A. Conversely, we associate an abstract rewriting system to a topology. The rewrite relation is defined as follows: an element a rewrites to b if b is in the closure of the element a. This rewrite relation is reflexive and transitive.

We recall the definition and the basic properties of closure operators on a set (see [Kel55]).

Definition 3.2.1. A map $\mathbf{C} : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ is called a *closure operator* if it verifies:

1. $\emptyset^{\mathbf{c}} = \emptyset$,
2. $X \subseteq X^{\mathbf{c}}$,
3. $X^{\mathbf{c}} = (X^{\mathbf{c}})^{\mathbf{c}}$,
4. $(X_1 \cup X_2)^{\mathbf{c}} = {X_1}^{\mathbf{c}} \cup {X_2}^{\mathbf{c}}$ for all $X, X_1, X_2 \subset A$.

Definition 3.2.2. If $\mathbf{C} : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ is a closure operator then $\mathcal{F} = \{X \subseteq A \mid X = X^{\mathbf{c}}\}$ and $\mathcal{T} = \{X \subseteq A \mid A - X \in \mathcal{F}\}$ form the family of *closed and open sets of a topology in* A.

We associate a topology $\mathcal{T}_{\rightarrow}$ to an abstract rewriting system $(A, \rightarrow)$. The closed sets of this topology are the subsets of A that are closed under the rewrite relation $\rightarrow$. The closure operator is the extension of the transitive graph of an element of A to subsets of A.

Definition 3.2.3. Let $(A, \rightarrow)$ be an abstract rewriting system. We define a mapping $\mathbf{C} : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ defined for $X \subseteq A$ by $\mathbf{C}(X) = \{b \in A \mid \exists x \in X \quad x \twoheadrightarrow b\}$.

For each subset X of A, $\mathbf{C}(X)$ is the set of all the elements in A that are obtained by rewriting some element in X. It is easy to prove that this mapping is a closure operator.

Lemma 3.2.4. Let $(A, \rightarrow)$ be an abstract rewriting system. The operator $\mathbf{C} : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ is a closure operator.

We denote by $\mathcal{F}_{\mathcal{A}}$ (or $\mathcal{F}_{\rightarrow}$) and $\mathcal{T}_{\mathcal{A}}$ (or $\mathcal{T}_{\rightarrow}$) the family of closed and open sets with respect to the topology associated to an abstract rewriting system. This topology is sometimes called the Alexandroff topology associated to $\rightarrow$. It has been considered especially in the case in which $\rightarrow$ is a preorder on A, see [GLSH92].

In the case that $X = \{a\}$ we write $\mathbf{C}(a)$ instead of $\mathbf{C}(\{a\})$. The closure of a point is the transitive reduction graph of the point and it is a closed set. In other words, if $a \in A$ then $\mathbf{C}(a) = \{x \in A \mid a \twoheadrightarrow x\} = \mathcal{G}_{\twoheadrightarrow}(a)$. Observe also that $\mathbf{C}(X) = \bigcup_{a \in X} \mathbf{C}(a) = \bigcup_{a \in X} \mathcal{G}_{\twoheadrightarrow}(a)$. Note also that $a \twoheadrightarrow b \Leftrightarrow b \in \mathbf{C}(a) \Leftrightarrow \mathbf{C}(b) \subseteq \mathbf{C}(a)$.

Also the transitive expansion graph of a point is an open set. In other words, $\mathcal{E}_{\twoheadrightarrow}(a) = \{x \in A \mid x \twoheadrightarrow a\}$ is an open set. It is the smallest set that is open and contains the element a. Note also that $b \twoheadrightarrow a \Leftrightarrow b \in \mathcal{E}_{\twoheadrightarrow}(a) \Leftrightarrow \mathcal{E}_{\twoheadrightarrow}(b) \subseteq \mathcal{E}_{\twoheadrightarrow}(a)$.

In the following lemma, we prove that the closed sets are invariant under the rewrite relation and the open sets are invariant under the inverse relation.

Lemma 3.2.5.

1. A subset $X \subseteq A$ is closed if and only if it is invariant under the rewrite relation $\rightarrow$. In other words X is closed if and only if $x \in X, x \rightarrow y$ implies $y \in X$ for all $x, y \in A$.

2. A subset $X \subseteq A$ is open if and only if it is invariant under the rewrite relation $\leftarrow$. In other words X is open if and only if $x \in X, y \rightarrow x$ implies $y \in X$ for all $x, y \in A$.

3. A subset of A is $\mathcal{T}_{\rightarrow}$ open if and only if it is $\mathcal{T}_{\leftarrow}$ closed, i.e. $\mathcal{T}_{\rightarrow} = \mathcal{F}_{\leftarrow}$ and $\mathcal{F}_{\rightarrow} = \mathcal{T}_{\leftarrow}$.

Definition 3.2.6.

The family of all sets of the form $\mathbf{C}(a)$ with $a \in A$ is called *the family of principal closed sets* and is denoted as $\mathcal{P}_{\rightarrow} \subseteq \mathcal{F}_{\rightarrow}$.

The family of all sets of the form $\mathcal{E}_{\twoheadrightarrow}(a)$ with $a \in A$ is called *the family of principal open sets* and is denoted as $\mathcal{O}_{\rightarrow} \subseteq \mathcal{T}_{\rightarrow}$.

Note that the family of open sets $\mathcal{O}_{\rightarrow}$ is a basis for the topology $\mathcal{T}_{\rightarrow}$.

We associate to each abstract rewriting system $(A, \rightarrow)$ the topological space $(A, \mathcal{T}_{\rightarrow})$ by means of a functor. It is easy to prove that if $f : A \rightarrow B$ is a morphism from $(A, \rightarrow_{\alpha})$ to $(B, \rightarrow_{\beta})$ then f is a continuous function from $(A, \mathcal{T}_{\rightarrow_{\alpha}})$ into $(B, \mathcal{T}_{\rightarrow_{\beta}})$.

Definition 3.2.7. The functor $\mathcal{H} : \mathbf{Ars} \rightarrow \mathbf{Top}$ is defined as follows:

$\mathcal{H}(A, \rightarrow) = (A, \mathcal{T}_{\rightarrow})$ for $(A, \rightarrow)$ an abstract rewriting system,

$\mathcal{H}(g) = g$ for g a morphism between abstract rewriting systems.

We associate an abstract rewriting system to a topology. The rewrite relation is defined as follows: an element a rewrites to b if b is in the closure of the element a.

Definition 3.2.8. Let $(A, \mathcal{T})$ be a topological space whose closure operator is $\_^{\mathrm{c}}$. We define a binary relation $\rightarrow_{\mathcal{T}}$ on A as follows.

$$a \rightarrow_{\mathcal{T}} b \text{ if } b \in a^{\mathrm{c}}.$$

This binary relation is the rewrite relation associated to $\mathcal{T}$. The abstract rewriting system associated to $\mathcal{T}$ is $(A, \rightarrow_{\mathcal{T}})$.

Note that $\rightarrow_{\mathcal{T}}$ is reflexive and transitive.

We associate to each topological space $(A, \mathcal{T})$ an abstract rewriting system $(A, \rightarrow_{\mathcal{T}})$ by means of a functor. It is easy to prove that if f is a continuous function from $(A, \mathcal{T})$ into $(B, \mathcal{T}')$ then $f : A \rightarrow B$ is a morphism from $(A, \rightarrow_{\mathcal{T}})$ to $(B, \rightarrow_{\mathcal{T}'})$.

Definition 3.2.9. The functor $\mathcal{G} : \mathbf{Top} \rightarrow \mathbf{Ars}$ is defined as follows:

$\mathcal{G}(A, \mathcal{T}) = (A, \rightarrow_{\mathcal{T}})$ for $(A, \mathcal{T})$ a topological space,

$\mathcal{G}(f) = f$ for $f : A \rightarrow B$ a continuous function of topological spaces.

3.3 Equivalence

The abstract rewriting systems whose rewrite relation verifies reflexivity and transitivity are called *preorders* and the topological spaces whose open sets are closed under intersections are called *symmetric topological spaces*. In this section, we prove that there is an equivalence between the preorders and the symmetric topological spaces.

We can associate the reflexive-transitive closure of an abstract rewriting system by means of a functor.

Definition 3.3.1. The functor *transitive closure* for abstract rewriting systems is denoted as $\mathcal{TC} : \mathbf{Ars} \rightarrow \mathbf{Ars}$ and defined as follows:

$\mathcal{TC}(A, \rightarrow) = (A, \twoheadrightarrow)$ for $(A, \rightarrow)$ an abstract rewriting system,

$\mathcal{TC}(f) = f$ for f a morphism between abstract rewriting systems.

Lemma 3.3.2. The functor $\mathcal{TC}$ verifies that $\mathcal{TC}^2 = \mathcal{TC}$.

We define the notion of symmetric topological space.

Definition 3.3.3. Let $(A, \mathcal{T})$ be a topology on A. We say that $(A, \mathcal{T})$ is *a symmetric topology* if the family of open sets is closed by intersections.

Definition 3.3.4. Let $(A, \mathcal{T})$ be a topological space and $\mathcal{T}_0 \subseteq \mathcal{T}$ be an arbitrary subfamily of $\mathcal{T}$ and call $O_{\mathcal{T}_0}$ the set $O_{\mathcal{T}_0} = \bigcap_{O \in \mathcal{T}_0} O$. We define *the symmetric topology associated to* $\mathcal{T}$ (denoted as $\mathcal{T}_r$) as the topology whose basis is the set $\{O_{\mathcal{T}_0} \mid \mathcal{T}_0 \subseteq \mathcal{T}\}$.

Note that $\mathcal{T}_r \supseteq \mathcal{T}$. The construction above can be expressed as a functor.

Definition 3.3.5. Define a functor $\mathcal{R}$ from the category of topological spaces into itself, $\mathcal{R} : \mathbf{Top} \to \mathbf{Top}$, as follows:

$\mathcal{R}(A, \mathcal{T}) = (A, \mathcal{T}_r)$ for $(A, \mathcal{T})$ a topological space,

$\mathcal{R}(f) = f$ for f a continuous function.

Lemma 3.3.6. The functor $\mathcal{R}$ verifies that $\mathcal{R}^2 = \mathcal{R}$.

The symmetric topology is very special in the sense that there is another operator that yields the smallest open set that contains a given element. This set is just the transitive expansion graph $\mathcal{E}_{\leftarrow}(a)$.

Lemma 3.3.7. The following statements are equivalent.

a) $\mathcal{T}$ is a symmetric topology.

b) $\mathcal{T}$ is a topology and there exists another topology $\mathcal{T}'$ in A , such that if $\mathcal{F}'$ denotes the family of closed sets of $\mathcal{T}'$, then $\mathcal{F}' = \mathcal{T}$.

c) $\mathcal{T}$ is a topology and the following condition is verified.

(M) For all $a \in A$, there exists a unique set $\mathbf{S}(a)$ that is the smallest open set that contains the point a, i.e. $\mathbf{S}(a) \subseteq X$ for all $X \in \mathcal{T}$ such that $a \in X$.

Proof: To prove **a)** $\Rightarrow$ **c)** we take a symmetric topology $(A, \mathcal{T})$ and observe that it verifies **(M)** because $\bigcap_{\{U \mid a \in U \in \mathcal{T}\}} U$ is the smallest open set containing a.

Conversely, a topology $(A, \mathcal{T})$ that verifies **(M)** is a symmetric topology. Suppose that $\mathcal{T}_0 \subseteq \mathcal{T}$ and $X = \bigcap_{U \in \mathcal{T}_0} U \neq \emptyset$. Take $a \in X$ then for any $U \in \mathcal{T}_0$, $\mathbf{S}(a) \subseteq U$ and then $\mathbf{S}(a) \subseteq X$. So that $X = \bigcup_{a \in X} \mathbf{S}(a)$ is an open set. $\square$

Lemma 3.3.8. Suppose that $\mathcal{T}$ is a topology on the set A and $\mathcal{T}_r$ the associated symmetric topology. For every point $a \in A$ we have that $\mathcal{C}_{\mathcal{T}}(a) = \mathcal{C}_{\mathcal{T}_r}(a)$.

Lemma 3.3.9. Let $\mathcal{T}$ be a symmetric topology on the set A.

a) For any $X \subseteq A$ there exists an open set $\mathbf{S}(X)$ that is the smallest open subset of A that contains X. Moreover $\mathbf{S}(X) = \bigcup_{a \in X} \mathbf{S}(a)$.

b) Let $a, b \in A$. We have that $b \in a^{\mathrm{c}} \Leftrightarrow a \in \mathbf{S}(b) \Leftrightarrow \mathbf{S}(a) \subseteq \mathbf{S}(b)$.

c) The family of open sets $\mathbf{S}(a)$ with $a \in A$ form a basis for the topology $\mathcal{T}$.

Note that if $(A, \rightarrow)$ is an abstract rewriting system, the topology $\mathcal{T}_{\rightarrow}$ is a symmetric topology.

In the following theorem we prove that there is an equivalence between the preorders and the symmetric topologies.

Theorem 3.3.10. Let **Preord** be the subcategory of **Ars** consisting of the preorders and **Tops** the subcategory of **Top** consisting of the topological spaces whose topology is a symmetric topology. Let $\mathcal{H}, \mathcal{G}, \mathcal{TC}$ and $\mathcal{R}$ be the functors defined before.

a) The composition $\mathcal{G} \circ \mathcal{H}$ satisfies $\mathcal{G} \circ \mathcal{H} = \mathcal{TC}$.

b) The composition $\mathcal{H} \circ \mathcal{G}$ satisfies $\mathcal{H} \circ \mathcal{G} = \mathcal{R}$.

c) The functors $\mathcal{H}$ and $\mathcal{G}$ are inverses of each other when respectively restricted to **Preord** and **Tops**.

d) The functor $\mathcal{H}$: **Preord** $\rightarrow$ **Tops** is an equivalence of categories. Its inverse is the functor $\mathcal{G}$: **Tops** $\rightarrow$ **Preord**.

Proof: Parts a) and b) are easy to prove. Part c) follows immediately from parts a), b) and the fact that $\mathcal{R}$ and $\mathcal{TC}$ are projection functors onto **Tops** and **Preord** respectively. Part d) follows immediately from the previous parts. □

3.4 Topological Characterisations

In this section, we give topological characterisations of confluence, weak and strong normalisation.

In the following theorem, we give some topological characterisations of confluence. An abstract rewriting system $(A, \rightarrow)$ is confluent if and only if in the topology associated to $(A, \rightarrow)$, the intersection of any pair of closed subsets of $\mathcal{G}_{\rightarrow}(a)$ is non-empty. Moreover, we prove that an abstract rewriting system $(A, \rightarrow)$ is confluent if and only if the two operators in the topology associated to $(A, \rightarrow)$ verify some subcommutation condition.

Theorem 3.4.1. (Topological Characterisation of Confluence)
The following statements are equivalent.

1. The abstract rewriting system $(A, \rightarrow)$ is confluent.
2. For all a in A and for every pair C and D of non empty $\mathcal{T}_{\rightarrow}$ closed subsets of $\mathbf{C}(a)$, $C \cap D \neq \emptyset$.
3. For all $X \subseteq A$, $\mathbf{C}(\mathbf{S}(X)) \subseteq \mathbf{S}(\mathbf{C}(X))$.

Proof:

$(1 \Leftrightarrow 2)$. Suppose $(A, \rightarrow)$ is confluent and C and D are as above. Take $c \in C$ and $d \in D$. As $c, d \in \mathbf{C}(a)$ we have that $d \twoheadleftarrow a \twoheadrightarrow c$. Then there exists an $x \in A$ such that $d \twoheadrightarrow x \twoheadleftarrow c$. As $d \in D$, $\mathbf{C}(d) \subseteq D$ and as $d \twoheadrightarrow x$, $x \in \mathbf{C}(d) \subseteq D$. Similarly, $x \in \mathbf{C}(c) \subseteq C$. Hence $x \in C \cap D$.

Conversely, suppose that we have $a, d, c \in A$ such that $d \twoheadleftarrow a \twoheadrightarrow c$. Then $\mathbf{C}(d) \subseteq \mathbf{C}(a)$ and $\mathbf{C}(c) \subseteq \mathbf{C}(a)$. By hypothesis, there exists an element $x \in \mathbf{C}(d) \cap \mathbf{C}(c)$. That means that $d \twoheadrightarrow x \twoheadleftarrow c$ and hence that $(A, \rightarrow)$ is confluent.

$(1 \Leftrightarrow 3)$. Observe that

$$\mathbf{C}(\mathbf{S}(X)) = \{z : \exists y, y \twoheadrightarrow z \ \ \& \ \ y \twoheadrightarrow x \ \ \& \ \ x \in X\}$$

$$\mathbf{S}(\mathbf{C}(X)) = \{u : \exists v, u \twoheadrightarrow v \ \ \& \ \ x \twoheadrightarrow v \ \ \& \ \ x \in X\}$$

□

Next we give a topological characterisation of normal forms. The closed points in the topology associated to the abstract rewriting systems are either elements that rewrite to itself or normal forms.

First we recall the notion of loop and irreflexive abstract rewriting systems.

Definition 3.4.2. Let $(A, \rightarrow)$ be an abstract rewriting system.

We say that a rewrite sequence starting at a is a *loop* if it is of the form $a \rightarrow^+ a$.

We say that a *one-step loop* is a loop of length 1.

For example, in the λ-calculus $< \Lambda, \rightarrow_\beta >$ one-step loops are of the form $C[\Omega] \rightarrow C[\Omega]$ with $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$.

Definition 3.4.3. We say that an abstract rewriting system is *irreflexive* if there is not any one-step loop.

We define the notion of terminal loops as loops that cannot rewrite to anything else than itself.

Definition 3.4.4. A *terminal loop* is a loop $a \rightarrow^+ a$ such that there is no $b \neq a$ with $a \twoheadrightarrow b$.

Note that the only terminal loops are loops of one step.

Definition 3.4.5. An abstract rewriting system $(A, \rightarrow)$ is *weakly irreflexive* if there is not any terminal loop.

For example, the pure type systems defined in chapter 9 are weakly irreflexive. There are no terminal loops.

In order to give a topological characterisation of normal forms we require that the abstract rewriting system should be weakly irreflexive. In this case the normal forms are exactly the closed points in the topology associated to the abstract rewriting system.

In the following lemma, we give a topological characterisation of normal forms for weakly irreflexive abstract rewriting systems: an element of a weakly reflexive abstract rewriting system is a normal form if it is a closed point in the associated topology.

Lemma 3.4.6. (Topological characterisation of normal forms)
Let $(A, \rightarrow)$ be a weakly irreflexive abstract rewriting system. An element $a \in A$ is a normal form iff $\{a\}$ is a $\mathcal{T}_{\rightarrow}$ closed set.

In the following theorem, we give some topological characterisations of weak normalisation for weakly irreflexive abstract rewriting systems.

The first characterisation says that an abstract rewriting system is weakly normalising if and only if every closed set has a closed point.

Note that the weakly normalising elements are the expansion of some normal form. This give us the other characterisation: $(A, \rightarrow)$ is weakly normalising if A can be obtained by applying the operator $\mathbf{S}$ to the set of closed points.

Theorem 3.4.7. (Topological characterisations of weak normalisation)
Let $(A, \rightarrow)$ be a weakly irreflexive abstract rewriting system. The following statements are equivalent.

1. $(A, \rightarrow)$ is weakly normalising.

2. Every non empty closed subset of A with respect to the topology associated to $(A, \rightarrow)$ has a closed point.

3. If X is the set of closed points of A then $A = \mathbf{S}(X)$.

Proof:

$(1 \Leftrightarrow 2)$. Suppose $(A, \rightarrow)$ is weakly normalising. Take $C \neq \emptyset$ a closed subset of A. Take $c \in C$ and consider $a \in A$ such that a is the normal form of c. Hence $a \in C$ because C is closed and a is a closed point because of lemma 3.4.6. Conversely, suppose that every non empty closed set has a closed point. Then for any $b \in A$ the closed set $\mathbf{C}(b)$ contains a closed point c. Then $b \twoheadrightarrow c$ and c is in normal form.

$(1 \Leftrightarrow 3)$. Easy. $\square$

To characterise strong normalisation we need the concept of noetherian family of subsets.

Definition 3.4.8. Let A be an arbitrary set and $\mathcal{S}$ a family of subsets of A, i.e. $\mathcal{S} \subseteq \mathcal{P}(A)$. We say that $\mathcal{S}$ is *noetherian* if and only if all decreasing subfamilies of $\mathcal{S}$ stabilize, i.e. for an arbitrary family $\{S_i : i \in \mathbb{N}\} \subseteq \mathcal{S}$ such that $S_1 \supseteq S_2 \supseteq S_3 \supseteq .. \supseteq S_n \supseteq .. \Rightarrow \exists m \in \mathbb{N}$ such that $S_m = S_{m+1} = ...$

Theorem 3.4.9. (Topological characterisation of strong normalisation)
Let $(A, \rightarrow)$ be a weakly irreflexive abstract rewriting system.
$(A, \rightarrow)$ is strongly normalising if and only if the following two conditions are verified:

1. The family $\mathcal{P}_{\rightarrow} \subseteq \mathcal{F}_{\rightarrow}$ of principal closed sets of A is noetherian.
2. $\mathbf{C}(a) = \mathbf{C}(b) \Leftrightarrow a = b$.

Proof: Suppose $\rightarrow$ is strongly normalising. We prove the two conditions:

1. Suppose that we have a decreasing family of sets in $\mathcal{P}_{\rightarrow}$, i.e. a family of the form:
$$\mathbf{C}(a_1) \supseteq \mathbf{C}(a_2) \supseteq \ldots \supseteq \mathbf{C}(a_n) \supseteq \mathbf{C}(a_{n+1}) \supseteq \ldots$$
This family produces a sequence of reductions $a_1 \twoheadrightarrow a_2 \twoheadrightarrow ...a_n \twoheadrightarrow a_{n+1} \ldots$. There exists $m \in \mathbb{N}$ such that $a_m = a_{m+1} = \ldots$. Then $\mathbf{C}(a_m) = \mathbf{C}(a_{m+1}) = \ldots$ and hence the family stabilizes.
2. If $\mathbf{C}(a) = \mathbf{C}(b)$ with $a \neq b$, we would have a reduction of infinite length $a \rightarrow^+ b \rightarrow^+ a \rightarrow^+ b \ldots$.

Conversely, any reduction $a_1 \rightarrow a_2 \rightarrow \ldots a_n \rightarrow a_{n+1} \ldots$ produces a family of principal closed sets, $\mathbf{C}(a_1) \supseteq \mathbf{C}(a_2) \supseteq \ldots \supseteq \mathbf{C}(a_n) \supseteq \mathbf{C}(a_{n+1}) \supseteq \ldots$. By the noetherian hypothesis we conclude that there exists an $m \in \mathbb{N}$ such that $\mathbf{C}(a_m) = \mathbf{C}(a_{m+1}) = \ldots$. Hence by hypothesis 2 we conclude that $a_m = a_{m+1} = \ldots$. $\square$

3.5 Conclusions and Related Work

The topology associated to an abstract rewriting system is the well-known Alexandroff topology that has been considered mainly for preorders. The comparison between abstract rewriting systems and topological structures has the novelty of finding topological characterisations for confluence and normalisation.

We see that in these topological characterisations of confluence, weak and strong normalisation expansion is as important as reduction. Expansion has also played a role in the characterisation of the set of strongly normalising λ-terms in chapter 5.

This chapter is logically independent of the rest of the thesis. Although we did not apply these results later, we think that it is relevant because to illuminate the same object with light from different angles can be sometimes very productive.

This topology does not look at the structure of the term (neither does the notion of abstract rewriting system) but only at the reduction graph of the term. In earlier work, other topologies associated to a reduction relation have been considered. In all these cases the topology depends on the structure of the terms (see [Bar85], [KKSdV91] and [KKS95]).

Chapter 4

Abstract Typing Systems

4.1 Introduction

In computer science the notions of computation and typing are basic, and play an essential role in the theory and its applications. In chapter 2 we have formalised the notion of computation in an abstract way as a binary relation on a set.

The concept of **typing** can be modelled also as a binary relation. In this case the relation is called the *typing relation* (usually denoted by $:$). An element a is typable if there exists b such that $a : b$ and it is inhabited if there exists b such that $b : a$.

In the case that the binary relation represents the rewrite relation, we are interested in looking at properties associated to its transitive closure like confluence and normalisation. In the case of a binary relation that represents the typing relation, we are interested in other kinds of properties like for example uniqueness of types. A typing relation does not have much use by itself and we believe it makes sense in an abstract setting only when considered together with a rewrite relation.

We consider a triple $(A, \rightarrow, :)$ consisting of a set and two binary relations, one representing the rewrite relation and the other representing the typing relation. There should be some interaction between these two relations. Now we discuss the kind of interaction we consider interesting.

We use the typing relation in order to restrict the domain of the rewrite relation to the set of elements that are typable or inhabited. This is useful when the abstract rewriting system does not verify weak or strong normalisation.

We want the set of typable elements together with the rewrite relation to be a subsystem of the original. This is verified when the set of typable elements and the set of inhabited elements are closed under the rewrite relation. These properties are called *subject reduction* and *type reduction* respectively.

Hence, we need that the rewrite and typing relations verify the subject and the type reduction properties. We say that the triple $(A, \rightarrow, :)$ is an *abstract rewriting system with typing* if it verifies the subject and the type reduction properties.

We want that the abstract rewriting system obtained by restricting the domain A to

the set of typable or inhabited elements verifies weak and strong normalisation. In this chapter we prove these properties for abstract rewriting systems with typing under certain hypothesis.

Finally, we introduce an abstract notion of *environment* (also called *context*). The typing relation and the rewrite relation may depend on environments. We consider indexed families of abstract rewriting systems with typing where the indices represent the environments. These families are called *environmental abstract rewriting systems with typing.*

We summarise the contents of the sections of this chapter. In section 4.2, we introduce the notions of abstract typing system and of type. In section 4.3, we introduce the abstract concepts of subject reduction and type reduction. Also we define the notions of abstract rewriting system with typing. We define the properties of uniqueness of types and of weak and strong normalisation. Also we give some general criteria to prove these properties. In section 4.4, we introduce the notion of environment. We add this feature to all the abstract structures defined in the previous section. In section 4.5, we introduce a general notion of semantics.

4.2 Abstract Typing Systems

In this section, we introduce the notion of abstract typing systems (see also [Raa96]) to formalise the notion of typing. As we said before a typing relation is formalised in an abstract way as a binary relation.

Definition 4.2.1. An *abstract typing system* is a pair $(A, :)$ consisting of a set A and a relation $: \subset A \times A$ called *typing relation.*

Definition 4.2.2. Let $(A, :)$ be an abstract typing system.

We say that a *has type* b if $a : b$.

We say that a is a *term* (or *typable*) if there exists b such that $a : b$.

We say that b is a *type* (or *inhabited*) if there exists a such that $a : b$.

We say that a is a *toptype* if a is a type that is not typable.

Definition 4.2.3. We define a *morphism* from $(A, :_\alpha)$ to $(B, :_\beta)$ as a function $f : A \to B$ such that if $a :_\alpha a'$ then $f(a) :_\beta f(a')$ for all $a, a' \in A$.

The category whose objects are the abstract typing systems and whose morphisms are the ones defined above is denoted by **Ats**.

Definition 4.2.4. We say that $(B, :_\beta)$ is an *extension* of $(A, :_\alpha)$ if $A \subset B$ and the inclusion mapping is a morphism from $(A, :_\alpha)$ to $(B, :_\beta)$.

Definition 4.2.5. We say that an extension $(B, :_\beta)$ of $(A, :_\alpha)$ is *conservative* if $a :_\beta a'$ implies $a :_\alpha a'$ for all $a, a' \in A$.

4.3 Abstract Rewriting Systems with Typing

In this section, we consider triples $(A, \rightarrow, :)$ consisting of a set A and two binary relations, one representing the rewrite relation and the other representing the typing relation. There should be some interaction between these two relations. We will require that the rewrite relation and the typing relation commute in different ways.

Definition 4.3.1. (Commutation of the rewrite and the typing relations)
Let $(A, \rightarrow, :)$ be a set A and two binary relations $\rightarrow$ and $:$ on the set A.

1. **Reducing the term.** If a has type b then we can rewrite the subject a to a' and a' should have a type b' related to b. According to the relation between b and b', we classify the way in which these two relations interact as follows.

   (a) We say that $(A, \rightarrow, :)$ satisfies *very weak subject reduction* if for all a, b, a' such that $a : b$ and $a \rightarrow a'$, there exists b' such that $a' : b'$ and $b \twoheadleftarrow\!\twoheadrightarrow b'$.

   (b) We say that $(A, \rightarrow, :)$ satisfies *weak subject reduction* if for all a, b, a' such that $a : b$ and $a \rightarrow a'$, there exists b' such that $a' : b'$ and $b \twoheadrightarrow b'$.

$$\begin{tikzcd} a \arrow[d] & : & b \arrow[d, dotted, two heads] \\ a' & : & b' \end{tikzcd}$$

   (c) We say that $(A, \rightarrow, :)$ satisfies *subject reduction* if for all a, b, a' such that $a : b$ and $a \rightarrow a'$ we have that $a' : b$.

$$\begin{tikzcd} a \arrow[d] & : & b \arrow[d, equal] \\ a' & : & b \end{tikzcd}$$

   Note that $(A, \rightarrow, :)$ satisfies (weak) subject reduction if and only if $(A, \twoheadrightarrow, :)$ satisfies (weak) subject reduction.

2. **Reducing the type.** If a has type b then we can rewrite the type b to b' and b' should have an inhabitant a' related to a. According to the relation between a and a', we classify the way in which these two relations interact as weak type reduction and type reduction.

   (a) We say that $(A, \rightarrow, :)$ satisfies *weak type reduction* if for all a, b, b' such that $a : b$ and $b \rightarrow b'$, there exists a' such that $a' : b'$ and $a \twoheadrightarrow a'$.

$$\begin{array}{ccc} a & : & b \\ \vdots & & | \\ \not\parallel & & \swarrow \\ a' & : & b' \end{array}$$

(b) We say that $(A, \rightarrow, :)$ satisfies *type reduction* if for all a, b, b' such that $a : b$ and $b \rightarrow b'$ we have that $a : b'$.

$$\begin{array}{ccc} a & : & b \\ \| & & | \\ & & \swarrow \\ a & : & b' \end{array}$$

Note that $(A, \rightarrow, :)$ satisfies (weak) type reduction if and only if $(A, \twoheadrightarrow, :)$ satisfies (weak) type reduction.

3. **Expanding.** If a has type b then we can expand a or b. We introduce the notions of subject expansion and type expansion.

   (a) We say that $(A, \rightarrow, :)$ satisfies *subject expansion* if for all a, b, a' such that $a : b$, a' is typable and $a' \rightarrow a$ we have that $a' : b$.

   (b) We say that $(A, \rightarrow, :)$ satisfies *type expansion* if for all a, b, b' such that $a : b$, b' is typable and $b' \rightarrow b$ we have that $a : b'$.

An abstract rewriting system with (very, weak) typing consists of a set, two binary relations which satisfy some commutation requirements.

Definition 4.3.2.

We say that $(A, \rightarrow, :)$ is an *abstract rewriting system with very weak typing* if $(A, \rightarrow, :)$ satisfies very weak subject reduction.

We say that $(A, \rightarrow, :)$ is an *abstract rewriting system with weak typing* if $(A, \rightarrow, :)$ satisfies weak subject and weak type reduction.

We say that $(A, \rightarrow, :)$ is an *abstract rewriting system with typing* if $(A, \rightarrow, :)$ satisfies subject and type reduction.

A morphism between abstract rewriting systems with typing is a function between sets that preserves the rewrite and the typing relations.

Definition 4.3.3. We define a *morphism* from $(A, \rightarrow_\alpha, :_\alpha)$ to $(B, \rightarrow_\beta, :_\beta)$ as a function $f : A \rightarrow B$ such that f is a morphism in **Ars** from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ and a morphism in **Ats** from $(A, :_\alpha)$ to $(B, :_\beta)$.

The category whose objects are the abstract rewriting systems with (weak) typing and whose morphisms are the ones defined above is denoted by $(\mathbf{Arst}_\omega)$ $\mathbf{Arst}$.

In a similar way as above, we can define the notions of refining, implementing and forgetting morphism.

If $(A, \rightarrow, :)$ is an abstract rewriting system with (weak) typing then the set $A_l = \{a \mid a$ is typable or a is inhabited$\}$ is closed under the rewrite relation $\rightarrow$ and the pair $(A_l, \rightarrow)$ is an abstract rewriting system.

Definition 4.3.4. Let $(A, \rightarrow, :)$ be an abstract rewriting system with (weak) typing. We define *the restriction of* $(A, \rightarrow)$ *by* $(A, :)$ as the abstract rewriting system $(A_l, \rightarrow)$.

The most commonly used abstract rewriting systems do not satisfy the necessary properties of weak or strong normalisation. Abstract rewriting systems are combined with abstract typing systems in order to restrict the original system $(A, \rightarrow)$ to $(A_l, \rightarrow)$ to have these properties. Intuitively, an abstract rewriting system with (weak) typing $(A, \rightarrow, :)$ verifies some property if the restriction of $(A, \rightarrow)$ by $(A, :)$ verifies this property.

Definition 4.3.5. Let $(A, \rightarrow, :)$ be an abstract rewriting system with (weak) typing. We say that $(A, \rightarrow, :)$ is *(weakly) strongly normalising* if the restriction of $(A, \rightarrow)$ by $(A, :)$ is (weakly) strongly normalising.

The restriction of an abstract rewriting system by an abstract typing system defined above can be expressed by means of a functor.

Definition 4.3.6. We define a functor $\mathcal{L} : \mathbf{Arst}_\omega \rightarrow \mathbf{Ars}$ as follows.

$$\mathcal{L}(A, \rightarrow, :) \quad = \quad (A_l, \rightarrow) \quad \text{for } (A, \rightarrow, :) \in \mathbf{Arst}_\omega,$$

where $A_l = \{a \mid a$ is typable or a is inhabited$\}$. This functor is defined for morphisms in the obvious way.

Lemma 4.3.7. A (refining, implementing, forgetting) morphism from $(A, \rightarrow_\alpha, :_\alpha)$ to $(B, \rightarrow_\beta, :_\beta)$ is a (refining, implementing, forgetting) morphism from $\mathcal{L}(A, \rightarrow_\alpha, :_\alpha)$ to $\mathcal{L}(B, \rightarrow_\beta, :_\beta)$.

Notice that, a morphism from $\mathcal{L}(A, \rightarrow_\alpha, :_\alpha)$ to $\mathcal{L}(B, \rightarrow_\beta, :_\beta)$ is a function that preserves the rewrite relation on a restricted domain. This function may not preserve the rewrite relation on the whole set and hence it may not be a morphism between abstract rewriting systems with typing.

The criteria for weak and strong normalisation given for abstract rewriting systems can be adapted to abstract rewriting system with (weak) typing. The weak normalisation criteria are used in the proof of weak normalisation for pure type systems with definitions of chapter 11. The second strong normalisation criterion is used in the proof of strong normalisation for pure type systems with definitions in chapter 11.

Lemma 4.3.8. (Weak Normalisation Criterion) Let $(A, \rightarrow_\alpha, :_\alpha)$ and $(B, \rightarrow_\beta, :_\beta)$ be abstract rewriting systems with (weak) typing. Suppose the following conditions are verified.

a) $(B, \rightarrow_\beta)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$.

b) There exists a function $f : B \rightarrow A$ that is a strategy for $\rightarrow_\beta$ and a morphism from $(B, :_\beta)$ to $(A, :_\alpha)$.

If $(A, \rightarrow_\alpha, :_\alpha)$ is weakly normalising then so is $(B, \rightarrow_\beta, :_\beta)$.

Lemma 4.3.9. (Weak Normalisation Criterion) Let $(A, \rightarrow_\alpha, :_\alpha)$ and $(B, \rightarrow_{\beta\delta}, :_\beta)$ be abstract rewriting systems with (weak) typing. Suppose the following conditions are verified.

a) $(B, \rightarrow_\beta)$ is a strong conservative extension of $(A, \rightarrow_\alpha)$.

b) If a is in δ-normal form and $a \rightarrow_\alpha a'$ then a' is in δ-normal form.

c) The relation $\rightarrow_\delta$ is weakly normalising.

d) The δ-normal form is a morphism $nf_\delta : B \rightarrow A$ from $(B, :_\beta)$ to $(A, :_\alpha)$.

If $(A, \rightarrow_\alpha, :_\alpha)$ is weakly normalising then so is $(B, \rightarrow_{\beta\delta}, :_\beta)$.

Lemma 4.3.10. (Strong Normalisation Criterion) Let $(A, \rightarrow_\alpha, :_\alpha)$, $(B, \rightarrow_\beta, :_\beta)$ be two abstract rewriting systems with typing and suppose there is a refining morphism $f : B \rightarrow A$ between them. If $(A, \rightarrow_\alpha, :_\alpha)$ is strongly normalising then so is $(B, \rightarrow_\beta, :_\beta)$.

Lemma 4.3.11. (Strong Normalisation Criterion) Let $(A, \rightarrow_\alpha, :_\alpha)$ to $(B, \rightarrow_{\beta\delta}, :_\beta)$ be abstract rewriting systems with (weak) typing. Suppose there is a mapping $f : B \rightarrow A$ such that:

a) f is an implementing morphism from $(B, \rightarrow_\delta, :_\beta)$ to $(A, \rightarrow_\alpha, :_\alpha)$.

b) f is a refining morphism from $(B, \rightarrow_\beta, :_\beta)$ to $(A, \rightarrow_\alpha, :_\alpha)$.

c) $(B, \rightarrow_\delta, :_\beta)$ is strongly normalising.

If $(A, \rightarrow_\alpha, :_\alpha)$ is strongly normalising then so is $(B, \rightarrow_{\beta\delta}, :_\beta)$.

Proof: This follows from lemma 2.6.6. We consider the abstract rewriting systems: $\mathcal{L}(A, \rightarrow_\alpha, :_\alpha)$, $\mathcal{L}(B, \rightarrow_\beta, :_\beta)$ and $\mathcal{L}(B, \rightarrow_\delta, :_\beta)$. $\square$

We define uniqueness of types property (up to conversion).

Definition 4.3.12. Let $(A, \rightarrow, :)$ be an abstract rewriting system with (weak) typing. We say that $(A, \rightarrow, :)$ verifies *uniqueness of types* if for all a such that $a : b$ and $a : b'$, we have that $b \twoheadleftrightarrow b'$.

We give a criterion to prove that the uniqueness of types property is preserved from one system to its extension. This criterion is used to prove uniqueness of types for singly sorted pure type systems with definitions in chapter 11.

Lemma 4.3.13. (Uniqueness of Types Criterion) Let $(B, \rightarrow_\beta)$ be an extension of $(A, \rightarrow_\alpha)$. Suppose there is a mapping $f : B \rightarrow A$ that is a strategy for $\rightarrow_\beta$ and a morphism from $(B, :_\beta)$ to $(A, :_\alpha)$. If $(A, \rightarrow_\alpha, :_\alpha)$ verifies uniqueness of types so does $(B, \rightarrow_\beta, :_\beta)$.

Proof: Let $b \in B$ be such that $b :_\beta c$ and $b :_\beta c'$. Since f is a morphism, we have that $f(b) :_\alpha f(c)$ and $f(b) :_\alpha f(c')$. By uniqueness of types for $(A, :_\alpha)$ we have that $f(c) \twoheadleftrightarrow_\alpha f(c')$. Since $\rightarrow_\alpha \subset \rightarrow_\beta$, we have that $f(c) \twoheadleftrightarrow_\beta f(c')$. Since f is a strategy we have that $c \twoheadleftrightarrow_\beta c'$. $\square$

4.4 Environments

Both the rewrite relation and the typing relation may depend on environments (also called contexts). The typing relation for pure type systems defined in chapter 9 and the reduction of global definitions defined in chapter 11 are examples of this dependency. In order to have an abstract picture of that situation for the reduction of global definitions we consider indexed families of abstract rewriting systems and for pure type systems we consider indexed families of abstract rewriting systems with typing where the indices represent the environments.

Definition 4.4.1. Let A and C be sets. We say that the triple $(A, C, \rightarrow)$ is an *environmental abstract rewriting system* if $\rightarrow$ is a function from C to $\mathcal{P}(A \times A)$.

We write $\rightarrow_\Gamma$ instead of $\rightarrow (?)$.

Note that $(A, \rightarrow_\Gamma)$ is an abstract rewriting system for $? \in C$.

The elements of C might be called *pseudoenvironments* (or *pseudocontexts*). The definition of environment appears later.

Definition 4.4.2. Let $(A, C, \rightarrow)$ be as above.

A *rewrite step in* $?$ is a $\rightarrow_\Gamma$-rewriting step. We write $? \vdash a \rightarrow b$ instead of $a \rightarrow_\Gamma b$.

A *rewrite sequence in* $?$ is defined as a $\rightarrow_\Gamma$-rewriting sequence. We write $? \vdash a_1 \rightarrow a_2 \rightarrow a_3 \ldots$ instead of $a_1 \rightarrow_\Gamma a_2 \rightarrow_\Gamma a_3 \ldots$.

Definition 4.4.3. Let $(A, C, \rightarrow_\alpha)$ and $(A, C, \rightarrow_\beta)$ be two environmental abstract rewriting systems. The *union of* $\rightarrow_\alpha$ *and* $\rightarrow_\beta$ *in* $?$ is defined as the union of $\rightarrow_{\alpha_\Gamma}$ and $\rightarrow_{\beta_\Gamma}$. We write $? \vdash a \rightarrow_{\alpha\beta} b$ instead of $a \rightarrow_{\alpha_\Gamma \beta_\Gamma} b$.

We define the notion of morphism for environmental abstract rewriting systems. A morphism is a pair of functions, one of the functions transforms the environments (or indices) and the other transforms the elements.

Definition 4.4.4. Let $(A, C, \rightarrow_\alpha)$ and $(B, D, \rightarrow_\beta)$ as above. We say that the pair (f, g) with $f : C \rightarrow D$ and $g : C \times A \rightarrow B$ is a *morphism* from $(A, C, \rightarrow_\alpha)$ to $(B, D, \rightarrow_\beta)$ if for all $a, b \in A$ and $? \in C$, if $? \vdash a \rightarrow_\alpha b$ then $f(?) \vdash g(?, a) \rightarrow_\beta g(?, b)$.

In a similar way, we can define the notions of refining, implementing and forgetting morphism.

Definition 4.4.5. Let $(A, C, \rightarrow)$ as above. We say that $f : C \times A \rightarrow A$ is a *strategy* if $? \vdash a \twoheadrightarrow f(?, a)$ for all $? \in C$ and $a \in A$. (Sometimes we write $f_\Gamma(a)$ instead of $f(?, a)$).

Intuitively, an element a of A verifies a property in $?$ if a verifies this property in the abstract rewriting system $(A, \rightarrow_\Gamma)$. Moreover, an environmental abstract rewriting system $(A, C, \rightarrow)$ verifies a property if $(A, \rightarrow_\Gamma)$ verifies this property for all $? \in C$.

Definition 4.4.6. Let $(A, C, \rightarrow)$ as above and $? \in C$.

We say that a is (*weakly*) *strongly normalising in* $?$ if a is (weakly) strongly normalising in the abstract rewriting system $(A, \rightarrow_\Gamma)$.

We say that $(A, C, \rightarrow)$ is (*weakly*) *strongly normalising* if $(A, \rightarrow_\Gamma)$ is (weakly) strongly normalising for all $? \in C$.

We say that $(A, C, \rightarrow)$ is *confluent* if $(A, \rightarrow_\Gamma)$ is confluent for all $? \in C$.

Definition 4.4.7. Let A and C be sets. We say that the triple $(A, C, :)$ is an *environmental abstract typing system* if $:$ is a function from C to $\mathcal{P}(A \times A)$.

Note that $(A, :_\Gamma)$ is an abstract typing system for $? \in C$.

Definition 4.4.8. Let $(A, C, :)$ be as above and $? \in C$. We say that a *has type* b *in* $?$ if $a :_\Gamma b$. We write $? \vdash a : b$ instead of $a :_\Gamma b$.

Definition 4.4.9. We say that $? \in C$ is an *environment* (or a *context*) if there are a and b such that $? \vdash a : b$.

Definition 4.4.10. Let $(A, C, :)$ as above and $? \in C$.

We say that a is a *term* (or *typable*) *in* $?$ if a is a term in the abstract typing system $(A, :_\Gamma)$. We denote that a is not typable in $?$ by $? \nvdash a : -$.

We say that a is a *term* (or *typable*) if there exists $?$ such that a is a term in $?$.

We say that a is a *type* (or *inhabited*) *in* $?$ if a is a type in the abstract typing system $(A, :_\Gamma)$.

We say that a is a *type* (or *inhabited*) if there exists $?$ such that a is a type in $?$.

We say that a *is a toptype in* $?$ if a is a toptype in the abstract typing system $(A, :_\Gamma)$.

We say that a *is a toptype* if there exists $?$ such that a is a toptype in $?$.

Definition 4.4.11. We say that the quadruple $(A, C, \rightarrow, :)$ is an *environmental abstract rewriting system with* (*very weak* , *weak*) *typing* if $\rightarrow$ and $:$ are functions from C to $\mathcal{P}(A \times A)$ and for all $? \in C$, $(A, \rightarrow_\Gamma, :_\Gamma)$ is an abstract rewriting system with (very weak, weak) typing.

Most of the examples of environmental abstract rewriting systems have a rewrite relation $\rightarrow$ that does not depend on the set C. In these cases, $a \rightarrow a'$ can be considered as an abbreviation of $? \vdash a \rightarrow a'$, for all $? \in C$.

An environmental abstract rewriting system with (weak) typing $(A, C, \rightarrow, :)$ verifies a property if $(A, \rightarrow_\Gamma, :_\Gamma)$ verifies this property for all $? \in C$.

Definition 4.4.12. Let $(A, C, \rightarrow, :)$ as above and $? \in C$.

We say that $(A, C, \rightarrow, :)$ is (*weakly*) *strongly normalising* if $(A, \rightarrow_\Gamma, :_\Gamma)$ is (weakly) strongly normalising for all $? \in C$.

We say that $(A, C, \rightarrow, :)$ verifies *uniqueness of types* if $(A, \rightarrow_\Gamma, :_\Gamma)$ verifies uniqueness of types for all $? \in C$.

Definition 4.4.13. Let $\mathcal{A} = (A, C, \rightarrow_\alpha, :_\alpha)$ and $\mathcal{B} = (B, D, \rightarrow_\beta, :_\beta)$ as above.
We say that $\mathcal{B}$ is an *extension* of $\mathcal{A}$ if the following conditions are verified.

1. $A \subset B$.
2. $C \subset D$.
3. $? \vdash a \rightarrow_\alpha a'$ then $? \vdash a \rightarrow_\beta a'$ for all $a, a' \in A$ and $? \in C$.
4. $? \vdash a :_\alpha a'$ then $? \vdash a :_\beta a'$ for all $a, a' \in A$ and $? \in C$.

We define the notion of morphism for environmental abstract rewriting systems with typing as a pair of functions. One of the functions transforms the contexts and the other transforms the elements. These functions preserve the rewrite relation and the typing relation. Note that they also preserve rewrite sequences.

Definition 4.4.14. Let $\mathcal{A} = (A, C, \rightarrow_\alpha, :_\alpha)$ and $\mathcal{B} = (B, D, \rightarrow_\beta, :_\beta)$ as above.
We say that (f, g) is a *morphism* from $\mathcal{A}$ to $\mathcal{B}$ if $f : C \rightarrow D$, $g : C \times A \rightarrow B$ and the following conditions are verified.

1. If $? \vdash a \rightarrow_\alpha b$ then $f(?) \vdash g(?, a) \rightarrow_\beta g(?, b)$.
2. $? \vdash a :_\alpha b$ then $f(?) \vdash g(?, a) :_\beta g(?, b)$.

If the rewrite relation $\rightarrow_\beta$ in the definition above does not depend on D then the first clause is replaced by the following one: if $? \vdash a \rightarrow_\alpha b$ then $g(?\,, a) \rightarrow_\beta g(?\,, b)$.

We denote the category whose objects are the environmental abstract rewriting systems with typing as **Carst**, the category whose objects are the environmental abstract rewriting systems with weak typing as $\mathbf{Carst}_\omega$ and the category whose objects are the environmental abstract rewriting with very weak typing as $\mathbf{Carst}_{\nu\omega}$) (in all these cases, the morphisms are the ones considered above).

In a similar way, we can define the notions of implementing, refining and forgetting morphisms for environmental abstract rewriting systems with (weak) typing.

4.5 Semantics

The notion of interpretation can be formalised in an abstract way by means of a morphism that preserves the conversion relation. The codomain of the interpretation can be seen as the *semantics* of the respective domain. For all the categories defined in the previous sections, we define the notions of interpretation and semantics. This section has been introduced to state and prove formally the corollary 11.4.16.

Definition 4.5.1. Let $(A, \rightarrow_\alpha)$ $(B, \rightarrow_\beta)$ be two abstract rewriting systems.

We define an *interpretation* (or a *converting morphism*) from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ as a function $f : A \rightarrow B$ such that if $a \twoheadleftrightarrow_\alpha a'$ then $f(a) \twoheadleftrightarrow_\beta f(a')$ for all $a, a' \in A$. If there exists an interpretation from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$, we say that $(B, \rightarrow_\beta)$ is a *semantics* for $(A, \rightarrow_\alpha)$.

Definition 4.5.2. We define an *interpretation* (or *converting morphism*) from $(A, \rightarrow_\alpha, :_\alpha)$ to $(B, \rightarrow_\beta, :_\beta)$ as a function $f : A \rightarrow B$ such that f is an interpretation in **Ars** from $(A, \rightarrow_\alpha)$ to $(B, \rightarrow_\beta)$ and a morphism in **Ats** from $(A, :_\alpha)$ to $(B, :_\beta)$. If there exists an interpretation from $(A, \rightarrow_\alpha, :_\alpha)$ to $(B, \rightarrow_\beta, :_\beta)$, we say that $(B, \rightarrow_\beta, :_\beta)$ is a *semantics* for $(A, \rightarrow_\alpha, :_\alpha)$.

Definition 4.5.3. Let $\mathcal{A} = (A, C, \rightarrow_\alpha, :_\alpha)$ and $\mathcal{B} = (B, D, \rightarrow_\beta, :_\beta)$ be environmental abstract rewriting systems with typing.

We say that (f, g) is an *interpretation* (or *converting morphism*) from $\mathcal{A}$ to $\mathcal{B}$ if $f : C \rightarrow D$, $g : C \times A \rightarrow B$ and the following conditions are verified.

1. If $? \vdash a \rightarrow_\alpha b$ then $f(?) \vdash g(?\,, a) \twoheadleftrightarrow_\beta g(?\,, b)$.

2. $? \vdash a :_\alpha b$ then $f(?) \vdash g(?\,, a) :_\beta g(?\,, b)$.

If there exists an interpretation from $(A, C, \rightarrow_\alpha, :_\alpha)$ to $(B, D, \rightarrow_\beta, :_\beta)$, we say that $(B, D, \rightarrow_\beta, :_\beta)$ is a *semantics* for $(A, C, \rightarrow_\alpha, :_\alpha)$.

In the following, we define the notion of weak converting morphism as a pair of functions. The function that transforms contexts depends on the context and on the element.

Definition 4.5.4. Let $\mathcal{A} = (A, C, \rightarrow_\alpha, :_\alpha)$ and $\mathcal{B} = (B, D, \rightarrow_\beta, :_\beta)$ as above.

We say that $f : C \times A \rightarrow D \times B$ is a *weak converting morphism* from $\mathcal{A}$ to $\mathcal{B}$ if the following conditions are verified for all $a, b \in A$ and $? \in C$. Suppose $f(?, a) = (\Delta, c)$ and $f(?, b) = (\Delta', d)$.

1. If $? \vdash a \twoheadleftrightarrow_\alpha b$ then $\Delta \vdash c \twoheadleftrightarrow_\beta d$.
2. If $? \vdash a :_\alpha b$ then $\Delta \vdash c :_\beta d$.

This special kind of morphisms is used in chapter 10. Note that the contexts Δ and Δ' have no relationship and the first clause cannot be replaced by '$? \vdash a \rightarrow_\alpha b$ then $\Delta \vdash c \twoheadleftrightarrow_\beta d$'.

A weak converting morphism from $\mathcal{A}$ to $\mathcal{B}$ could be seen as a weak interpretation from $\mathcal{A}$ to $\mathcal{B}$.

4.6 Conclusions and Related Work

In this chapter, we have introduced the notions of abstract typing system and abstract rewriting system with typing. The concepts will not surprise the specialists in the subject, firstly because they are extremely natural and secondly because even though they appear as 'new', they were already 'there' in a sort of ghostly manner. All these definitions and properties are used in the rest of the thesis.

We think it is necessary to have a formal basis for type systems in the same fashion as the notion of abstract rewriting systems is a formal basis for the lambda calculus. For example, the whole section 4.5 was introduced in order to be able to formally describe a property for models of pure type systems with definitions. Intuitively, the interpretation of a pure type system with definitions is obtained by computing first the δ-normal form and then applying the interpretation of the original system without definitions. In [Pol94], this result is stated in an informal way. Since we introduce the general setting of abstract typing systems, we can say formally what an interpretation is and the considered property can be formalised adequately.

Abstract formalisations of the notion of **logic** can be found in [Bar74], [Mes89], [HST89], [Avr92] and [Acz95]. These formalisations are aimed to model different aspects of logic. For example, the notion of a proof is modelled in [Mes89] but the rewrite relation between proofs is not considered important. The notion of abstract rewriting systems with typing intends to capture the notions of terms (proofs), types (propositions) and the reduction of terms (the simplification of proofs). Hence our formalisation is aimed to model the interaction between the typing relation and the rewriting relation. This is because we are interested in properties of the typing systems like weak and strong normalisation.

We think that it could be possible to define the abstract notion of derivation and relate it with the abstract logics defined in [Avr92] and [Acz95]. In that case we should introduce an abstract notion of typing rules and of typing relation generated by these rules. Also we should probably add more structure to the notion of environment.

Part II

Lambda Calculus

Chapter 5

Strongly Normalising λ-terms

5.1 Introduction

In this chapter we give some characterisations of the set of β-strongly normalising λ-terms. We use these characterisations (see chapters 6-8) to give new proofs of some results concerning normalisation in λ-calculus.

This chapter is organised as follows. In section 5.2 we define the set of λ-terms and the β-reduction. In section 5.3, first we define a set $\mathcal{SN}'$ by induction that reflects the intuition of what should be the set of strongly normalising terms. Then we define another set $\mathcal{SN}$ that is equal to $\mathcal{SN}'$. Finally we prove that $\mathcal{SN}$ is the set of β-strongly normalising terms.

5.2 Lambda Calculus

In this section we recall the definition of the untyped lambda calculus with β-reduction.

The set of variables is denoted by $V = \{v, v', v'', \ldots\}$ and arbitrary variables in V are denoted by $x, y, z, \ldots$.

Definition 5.2.1. The set Λ of λ-*terms* is defined as the smallest set satisfying the following clauses.

1. $V \subset \Lambda$,
2. if $M \in \Lambda$ then $\lambda x.M \in \Lambda$,
3. if $M \in \Lambda$ and $N \in \Lambda$ then $(M\ N) \in \Lambda$.

Definition 5.2.2. The mapping $FV : \Lambda \subset \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M) - \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \end{array}$$

A variable x is said to occur *free* in M if $x \in FV(M)$.

Definition 5.2.3. The mapping $BV : \Lambda \subset \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} BV(x) & = & \emptyset \\ BV(\lambda x.M) & = & BV(M) \cup \{x\} \\ BV(MN) & = & BV(M) \cup BV(N) \end{array}$$

A variable x is said to occur *bound* in M if $x \in BV(M)$.

We define substitution as in [CF58].

Definition 5.2.4. The result of the *substitution* of N for x in M is defined as follows.

$$\begin{array}{rcll} x[x := N] & = & N & \\ y[x := N] & = & y & \text{if } x \neq y \\ (PQ)[x := N] & = & (P[x := N]Q[x := N]) & \\ (\lambda x.P)[x := N] & = & (\lambda x.P) & \\ (\lambda y.P)[x := N] & = & (\lambda z.P[y := z][x := N]) & \text{if } y \neq x,\ y \in FV(N) \text{ and } z \text{ is fresh} \\ (\lambda y.P)[x := N] & = & (\lambda y.P[x := N]) & \text{otherwise} \end{array}$$

Definition 5.2.5. A *change of a bound variable* in the term M is the replacement of a subterm $(\lambda x.N)$ by $(\lambda y.N[x := y])$ where $y \notin FV(N)$.

The relation of α-conversion between λ-terms is defined as follows.

Definition 5.2.6. The term M is *α-convertible* to N if N is the result of applying to M a series of changes of bound variables or vice versa.

Convention 5.2.7. Two terms are identified if they are α-convertible.

We define now the notion of context in the lambda calculus as a term with holes in it.

Definition 5.2.8. We define the set $\mathcal{P}$ by induction as follows.

1. $x \in \mathcal{P}$,
2. $[\ ] \in \mathcal{P}$,
3. if $C_1[\ ] \in \mathcal{P}$ and $C_2[\ ] \in \mathcal{P}$ then $(C_1[\ ]\ C_2[\ ]) \in \mathcal{P}$,
4. if $C_1[\ ] \in \mathcal{P}$ then $(\lambda x.C_1[\ ]) \in \mathcal{P}$.

A *context* is an element of $\mathcal{P}$ and is denoted by $C[\ ]$. If $C[\ ] \in \mathcal{P}$ and $M \in \Lambda$ then $C[M]$ denotes the result of placing M in the holes of $C[\ ]$.

The essential feature of a context $C[\ ]$ is that a free variable in M may become bound in $C[M]$.

Definition 5.2.9. The β-reduction (or β-rewrite relation) is defined as follows.

$$C[(\lambda x.M)N] \to_\beta C[M[x := N]]$$

where $C[\ ] \in \mathcal{P}$ has only one occurrence of $[\ ]$.

We use the usual abbreviations: $I = \lambda x.x$, $K = (\lambda x.\lambda y.x)$, $\omega = \lambda x.(xx)$ and $\Omega = (\omega\omega)$.

There are terms in the λ-calculus that are not strongly normalising like Ω and $(K\ I\ \Omega)$.

5.3 The set $\mathcal{SN}$

In this section we give two definitions by induction of the set of strongly normalising λ-terms. These definitions use β-expansion.

An easy observation is that the set that contains all normal forms and that is closed under expansion is exactly the set of all weakly normalising terms. So we have the following definition.

Definition 5.3.1. The set $\mathcal{W}$ is the smallest set of λ-terms satisfying the following:

1. all normal forms are in $\mathcal{W}$,
2. if $C[P[x := Q]] \in \mathcal{W}$, then $C[(\lambda x.P)Q] \in \mathcal{W}$.

The first naive attempt to obtain the set of all strongly normalising terms, is to add the requirement that the argument of the redex introduced by the expansion is strongly normalising. The set $\mathcal{S}$ is the smallest set that satisfies

1. all normal forms are in $\mathcal{S}$,
2. if $C[P[x := Q]] \in \mathcal{S}$ and $Q \in \mathcal{S}$, then $C[(\lambda x.P)Q] \in \mathcal{S}$.

However, it is easy to see that there are terms that are not strongly normalising that belong to $\mathcal{S}$. For example, take $\omega = \lambda y.(yy)$ and the rewrite sequence

$$\begin{array}{lll} (\lambda x.(\lambda y.z)(xx))\omega & \to_\beta & (\lambda x.z)\omega \\ & \to_\beta & \omega \end{array}$$

The last term in the sequence, ω, is in normal form then $\omega \in \mathcal{S}$. If we go backwards in the sequence we get that $(\lambda x.z)\omega \in \mathcal{S}$ and also $(\lambda x.(\lambda y.z)(xx))\omega \in \mathcal{S}$. But this term is not strongly normalising because

$$\begin{array}{lll} (\lambda x.(\lambda y.z)(xx))\omega & \to_\beta & (\lambda y.z)(\omega\omega) \\ & \to_\beta & (\omega\omega) \end{array}$$

The problem is that expansions cannot be allowed to take place just everywhere. The expansion as in the second clause of the definition of $\mathcal{S}$ above will be required to create a *spine redex*, i.e. a head redex or if there is no head redex an outermost redex.

Definition 5.3.2. The set $\mathcal{O}$ of contexts with a hole at a *spine position* is defined as the minimal set that satisfies

1. if $C[\,] \in \mathcal{O}$ then $x\ M_1\ \ldots C[\,]\ldots M_n \in \mathcal{O}$,
2. if $C[\,] \in \mathcal{O}$ then $\lambda x.C[\,] \in \mathcal{O}$,
3. $[\,]P_1 \ldots P_n \in \mathcal{O}$.

The redex in $C[(\lambda x.M)N]$ is called a *spine redex* if $C[\ ] \in \mathcal{O}$ [BKKS87].

Definition 5.3.3. The set $\mathcal{SN}'$ is defined as the smallest set that satisfies

1. all normal forms are in $\mathcal{SN}'$,
2. if $C[P[x := Q]] \in \mathcal{SN}'$, $Q \in \mathcal{SN}'$ and $C[\ ] \in \mathcal{O}$, then $C[(\lambda x.P)Q] \in \mathcal{SN}'$.

In order to obtain another definition of the set of strongly normalising λ-terms, observe that the set of normal forms can be defined by induction in the following way.

Definition 5.3.4. The set $\mathcal{NF}$ is the smallest set of λ-terms satisfying the following:

1. if x is a variable and $M_1, \ldots, M_n \in \mathcal{NF}$ for some $n \geq 0$, then $xM_1 \ldots M_n \in \mathcal{NF}$,
2. if $M \in \mathcal{NF}$ then $\lambda x.M \in \mathcal{NF}$,

We define the set $\mathcal{SN}$ as follows.

Definition 5.3.5. The set $\mathcal{SN}$ is the smallest set of λ-terms satisfying the following:

1. if x is a variable and $M_1, \ldots, M_n \in \mathcal{SN}$ for some $n \geq 0$, then $xM_1 \ldots M_n \in \mathcal{SN}$,
2. if $M \in \mathcal{SN}$ then $\lambda x.M \in \mathcal{SN}$,
3. if $M[x := N]P_1 \ldots P_n \in \mathcal{SN}$ and $N \in \mathcal{SN}$, then $(\lambda x.M)NP_1 \ldots P_n \in \mathcal{SN}$.

In the following theorem, we prove that the set $\mathcal{SN}$ characterises the set of strongly normalising terms.

Theorem 5.3.6. (Characterisation of the strongly normalising λ-terms)
M is strongly normalising if and only if $M \in \mathcal{SN}$.

Proof:

$\Rightarrow$. Let M be a strongly normalising term. The proof proceeds by induction on the pair $(\mathsf{maxred}(M), M)$, lexicographically ordered by the usual ordering on $\mathbb{N}$ and the subterm ordering. Here we denote by $\mathsf{maxred}(M)$ the length of a maximal rewrite sequence from M to normal form.

The base case is trivial since it is easy to see that all normal forms are in $\mathcal{SN}$.

Suppose the maximal reduction of M to normal form takes $k+1$ steps. Let $M = \lambda x_1 \ldots \lambda x_n.PQ_1 \ldots Q_m$. There are two cases.

Case 1. $P = y$. Then the normal form of M is of the form $\lambda x_1 \ldots \lambda x_n.yQ'_1 \ldots Q'_m$ with $Q_i \twoheadrightarrow_\beta Q'_i$ for $i = 1, \ldots, m$. By induction hypothesis, $Q_1 \in \mathcal{SN}, \ldots, Q_m \in \mathcal{SN}$. By the first and second clause of the definition of $\mathcal{SN}$, we have $M = \lambda x_1 \ldots \lambda x_n.yQ_1 \ldots Q_m \in \mathcal{SN}$.

Case 2. $P = \lambda y.P_0$. We have $M = \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1Q_2 \ldots Q_m \to \lambda x_1 \ldots \lambda x_n.P_0[y := Q_1]Q_2 \ldots Q_m$. By induction hypothesis, $\lambda x_1 \ldots \lambda x_n.P_0[y := Q_1]Q_2 \ldots Q_m \in \mathcal{SN}$. Also by induction hypothesis, $Q_1 \in \mathcal{SN}$. By the last clause of the definition of $\mathcal{SN}$, we have $M = \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1 \ldots Q_m \in \mathcal{SN}$.

$\Leftarrow$. Suppose $M \in \mathcal{SN}$. We prove by induction on the derivation of $M \in \mathcal{SN}$ that M is strongly normalising.

1. If $M = xM_1 \ldots M_n$ with $M_1, \ldots, M_n \in \mathcal{SN}$, then the statement follows easily by induction hypothesis.

2. If $M = \lambda x.M_0$ with $M_0 \in \mathcal{SN}$, then by induction hypothesis M_0 is strongly normalising. Then also $M = \lambda x.M_0$ is strongly normalising.

3. Let $M = (\lambda x.M_0)M_1M_2 \ldots M_n$ with $M_0[x := M_1]M_2 \ldots M_n \in \mathcal{SN}$ and $M_1 \in \mathcal{SN}$. Consider an arbitrary rewrite sequence $\rho : M = P_0 \to_\beta P_1 \to_\beta P_2 \to_\beta \ldots$ starting in M. There are two possibilities: in ρ either the head redex of M is contracted or the head redex of M is not contracted.

   In the first case, there is an i such that $P_i = M_0'[x := M_1']M_2' \ldots M_n'$, with $M_0 \twoheadrightarrow_\beta M_0', \ldots, M_n \twoheadrightarrow_\beta M_n'$. Then P_i is a result of rewriting the term $M_0[x := M_1]M_2 \ldots M_n$. The latter is by induction hypothesis strongly normalising. Hence P_i is strongly normalising so ρ is finite.

   In the second case, all terms in ρ are of the form $(\lambda x.M_0')M_1'M_2' \ldots M_n'$ with $M_0 \twoheadrightarrow_\beta M_0', \ldots, M_n \twoheadrightarrow_\beta M_n'$. By induction hypothesis, the term $M_0[x := M_1]M_2 \ldots M_n$ is strongly normalising. Therefore $M_0, M_2, \ldots, M_n$ are strongly normalising. Moreover, we have by induction hypothesis that M_1 is strongly normalising. Hence all the terms in the rewrite sequence are strongly normalising and hence ρ is finite.

□

Theorem 5.3.7. $\mathcal{SN}' = \mathcal{SN}$.

Proof: $\mathcal{SN}' \subset \mathcal{SN}$ is proved by induction on $\mathcal{SN}'$. For $\mathcal{SN} \subset \mathcal{SN}'$, we prove that the set of strongly normalising λ-terms is a subset of $\mathcal{SN}'$ by induction on $(\mathsf{maxred}(M), M)$. □

5.4 Conclusions and Related work

We have defined the sets $\mathcal{SN}$ and $\mathcal{SN}'$ and proved that they are equal to the set of β-strongly normalising λ-terms. In the chapters 6-8, we use the set $\mathcal{SN}$ to give new proofs of classical results in lambda calculus. The use of $\mathcal{SN}$ to prove normalisation properties is very convenient because its definition is by induction and furthermore it recalls the notion of saturated set.

A *saturated set* is a subset X of the set of strongly normalising λ-terms that satisfies the following properties.

1. if x is a variable and $M_1, \ldots, M_n$ are strongly normalising terms then the term $xM_1 \ldots M_n \in X$,

2. if $M[x := N]P_1 \ldots P_n \in X$ and N is strongly normalising then the term $(\lambda x.M)NP_1 \ldots P_n \in X$.

The first and second clauses in the definition of $\mathcal{SN}$ are also conditions in the definition of a saturated set. In the definition of $\mathcal{SN}$, we have an additional clause for abstractions, and in the definition of saturated set it is necessary to add the requirement that the set should be a subset of the set of strongly normalising terms.

Our definition of $\mathcal{SN}$ first appeared in [RS95] and more or less simultaneously a similar definition appeared in [Loa95].

Chapter 6

Perpetual Strategies

6.1 Introduction

In this chapter we define two strategies G_{bk} and G_∞ similar to F_{bk} [BK82] and F_∞ [BBKV76]. These strategies are *perpetual*, which means that they yield an infinite rewrite sequence whenever possible. We prove that G_{bk} and G_∞ are perpetual by using the characterisation of the set of strongly normalising terms. As a consequence, we deduce that F_{bk} and F_∞ are perpetual.

This chapter is organised as follows. In section 6.2 we prove that the strategies G_{bk} and F_{bk} are perpetual. Then we prove that the strategies G_∞ and F_∞ are perpetual. For the strategies G_∞ and F_∞, we prove in section 6.4 that they are not only perpetual but also maximal. That is, they yield the longest possible reduction to normal form whenever the initial term is strongly normalising, and an infinite rewrite sequence otherwise. This is done by computing the length of the rewrite sequence to the normal form.

6.2 The Strategies F_{bk} and G_{bk}

First we consider the strategy F_{bk} as introduced in [BK82].

Definition 6.2.1. Suppose that $M \in \Lambda$ is not in normal form.
Let $M = C[(\lambda x.P)Q]$ where $(\lambda x.P)Q$ is the leftmost redex of M.

$$F_{bk}(C[(\lambda x.P)Q]) \;=\; \begin{cases} C[P[x := Q]] & \text{if } Q \text{ is strongly normalising} \\ C[(\lambda x.P)F_{bk}(Q)] & \text{otherwise} \end{cases}$$

We define the strategy G_{bk} as a variant of F_{bk}. We reduce any spine redex instead of just the leftmost redex. This yields a non-deterministic strategy.

Definition 6.2.2. We define $G_{bk} : \Lambda \to \mathcal{P}(\Lambda)$ as follows.

$$
\begin{array}{lcl}
G_{bk}(xM_1 \ldots M_n) & = & \bigcup_{i=1}^{i=n} \{(xM_1 \ldots N_i \ldots M_n) | N_i \in G_{bk}(M_i)\} \\
G_{bk}(\lambda x.M) & = & \{(\lambda x.N) | N \in G_{bk}(M)\} \\
G_{bk}((\lambda x.M)NP_1 \ldots P_n) & = & \begin{cases} \{M[x := N]P_1 \ldots P_n\} & \text{if } N \text{ is strongly normalising} \\ \{(\lambda x.M)QP_1 \ldots P_n | Q \in G_{bk}(N)\} & \text{otherwise} \end{cases}
\end{array}
$$

Theorem 6.2.3. G_{bk} is a perpetual strategy.

Proof: We prove that if $G_{bk}(M) \subset \mathcal{SN}$ then $M \in \mathcal{SN}$ by induction on the structure of M.

1. Suppose that the term is $(xM_1 \ldots M_n)$. Since $G_{bk}(xM_1 \ldots M_n) \subset \mathcal{SN}$, we have that $G_{bk}(M_i) \subset \mathcal{SN}$ for all $i = 1 \ldots n$. By induction hypothesis we have that $M_i \in \mathcal{SN}$. Hence $(xM_1 \ldots M_n) \in \mathcal{SN}$.

2. Suppose that the term is $(\lambda x.M)$. Since $G_{bk}(\lambda x.M) \subset \mathcal{SN}$, we have that $G_{bk}(M) \subset \mathcal{SN}$. Moreover, by induction hypothesis $M \in \mathcal{SN}$. Therefore $(\lambda x.M) \in \mathcal{SN}$.

3. Suppose that the term is $(\lambda x.M)NP_1 \ldots P_n$. We have two cases:

   (a) If N is strongly normalising then $N \in \mathcal{SN}$. Since $M[x := N]P_1 \ldots P_n \in \mathcal{SN}$ we have that $(\lambda x.M)NP_1 \ldots P_n$.

   (b) If N is not strongly normalising then there exists non-strongly normalising term Q in $G_{bk}((\lambda x.M)NP_1 \ldots P_n)$.

□

The strategy F_{bk} is contained in G_{bk}.

Lemma 6.2.4. Let $M \in \Lambda$ not in normal form. Then $F_{bk}(M) \in G_{bk}(M)$.

Theorem 6.2.5. F_{bk} is a perpetual strategy.

This follows from lemma 6.2.4 and theorem 6.2.3.

6.3 The Strategies F_∞ and G_∞

We now consider the strategy F_∞ that is defined in [BBKV76]. This strategy does not check whether the argument of the leftmost redex is strongly normalising or not. Instead, it is checked whether the leftmost redex is an I-redex. If it is, it is contracted. If it is not, contracting it could imply loosing the possibility of having an infinite reduction sequence. Therefore, in that case, the leftmost redex is only contracted if the argument is a normal form. If the argument is not a normal form, the strategy is applied to the argument.

Definition 6.3.1. Suppose that $M \in \Lambda$ is not in normal form.
Let $M = C[(\lambda x.P)Q]$ where $(\lambda x.P)Q$ is the leftmost redex of M.

$$F_\infty(C[(\lambda x.P)Q]) = \begin{cases} C[P[x := Q]] & \text{if } x \in FV(P) \text{ or } Q \in \mathcal{NF} \\ C[(\lambda x.P)F_\infty(Q)] & \text{otherwise} \end{cases}$$

The merit of F_∞ is that it is decidable.

We define the strategy G_∞ as a variant of F_∞. We do not only reduce the leftmost redex but also any spine redex. This yields a non-deterministic strategy.

Definition 6.3.2. We define $G_\infty : \Lambda \to \mathcal{P}(\Lambda)$ as follows.

$$G_\infty(xM_1 \ldots M_n) = \bigcup_{i=1}^{i=n} \{(xM_1 \ldots N_i \ldots M_n) | N_i \in G_\infty(M_i)\}$$

$$G_\infty(\lambda x.M) = \{(\lambda x.N) | N \in G_\infty(M)\}$$

$$G_\infty((\lambda x.M)NP_1 \ldots P_n) = \begin{cases} \{M[x := N]P_1 \ldots P_n\} & \text{if } x \in FVP \text{ or } Q \in \mathcal{NF} \\ \{(\lambda x.M)QP_1 \ldots P_n | Q \in G_\infty(N)\} & \text{otherwise} \end{cases}$$

Theorem 6.3.3. G_∞ is a perpetual strategy.

We prove that if $G_\infty(M) \subset \mathcal{SN}$ then $M \in \mathcal{SN}$ by induction on the structure of M.

Lemma 6.3.4. Let $M \in \Lambda$ not in normal form. Then $F_\infty(M) \in G_\infty(M)$.

As an immediate consequence of the previous lemma we have that F_∞ is perpetual.

Theorem 6.3.5. F_∞ is a perpetual strategy.

6.4 Maximal Strategies

In this section we prove that the strategies F_∞ and G_∞ are *maximal*, which means that they compute for each term M the longest possible rewrite sequence. In particular, a maximal strategy is perpetual. The converse is not necessarily true, as witnessed by the strategy F_{bk} defined in [BK82].

Example 6.4.1. The F_{bk}-rewrite sequence starting at $(\lambda x.z)(II)$ has length 1.

$$(\lambda x.z)(II) \to_\beta z$$

However the length of the maximal rewrite sequence is 2.

$$\begin{aligned} (\lambda x.z)(II) &\to_\beta (\lambda x.z)I \\ &\to_\beta z \end{aligned}$$

Our proof that G_∞ is a maximal strategy makes use of the characterisation of strongly normalising terms. We define a mapping h that computes the length of a G_∞-rewrite sequence of a term. Then it is proved that the mapping h computes the length of a maximal rewrite sequence to normal form.

We define a map $h : \Lambda \to \mathbb{N} \cup \{\infty\}$ that computes for each term the length of its F_∞-rewrite sequence.

Definition 6.4.2.

1. The map $h : \mathcal{SN} \to \mathbb{N}$ is defined by induction on the definition of $\mathcal{SN}$.

$$\begin{aligned} h(xM_1 \ldots M_n) &= \begin{cases} 0 & \text{if } n = 0 \\ \sum_{i=1}^{n} h(M_i) & \text{if } n \neq 0 \end{cases} \\ h(\lambda x.M) &= h(M) \\ h((\lambda x.M)NP_1 \ldots P_n) &= \begin{cases} h(M[x := N]P_1 \ldots P_n) + 1 & \text{if } x \in FV(M) \\ h(M\ P_1 \ldots P_n) + h(N) + 1 & \text{if } x \notin FV(M) \end{cases} \end{aligned}$$

2. We extend $h : \mathcal{SN} \to \mathbb{N}$ to $h : \Lambda \to \mathbb{N} \cup \{\infty\}$ by defining $h(M) = \infty$ if $M \notin \mathcal{SN}$.

We prove that the map h has the following two properties:

- it computes the length of all the G_∞-rewrite sequences of a term M,
- it computes the length of a maximal rewrite sequence starting in M.

From these we conclude that G_∞ and F_∞ are maximal strategies.
First we prove the following lemma.

Lemma 6.4.3. Let $M \in \mathcal{SN}$.

1. If $M \in \mathcal{NF}$ then $h(M) = 0$.
2. If $M \notin \mathcal{NF}$ then $h(M) = h(N) + 1$ for all $N \in G_\infty(M)$.

Proof:

1. Trivial.
2. Suppose that M is not in normal form. We prove that $h(M) = h(N) + 1$ for all $N \in G_\infty(M)$ by induction on $M \in \mathcal{SN}$.
   We consider these two cases:

(a) The term M is of the form $yQ_1 \ldots Q_m$.
By induction hypothesis we have $h(Q_i) = h(N_i) + 1$ for all $N_i \in G_\infty(Q_i)$.
Take i and $N_i \in G_\infty(Q_i)$. Hence we have

$$\begin{aligned} h(M) &= \sum_{k=i}^{m} h(Q_k) \\ &= h(Q_i) + \sum_{k \neq i} h(Q_k) \\ &= h(N_i) + 1 + \sum_{k \neq i} h(Q_k) \\ &= h(yQ_1 \ldots N_i \ldots Q_n) + 1 \end{aligned}$$

(b) The term M is $(\lambda y.P_0)Q_1 \ldots Q_m$. Two cases are distinguished.

i. $y \in FV(P_0)$. Then $G_\infty(M) = \{P_0[y := Q_1]\ Q_2 \ldots Q_m\}$.
We have that $h(M) = h(P_0[y := Q_1]\ Q_2 \ldots Q_m) + 1$.

ii. $y \notin FV(P_0)$. Again two cases are distinguished.

A. If Q_1 is not in normal form then

$$G_\infty(M) = \{(\lambda y.P_0)N\ Q_2 \ldots Q_m | N \in G_\infty(Q_1)\}$$

By induction hypothesis, $h(Q_1) = h(N) + 1$ for all $N \in G_\infty(Q_1)$. Hence we have

$$\begin{aligned} h(M) &= h(P_0\ Q_2 \ldots Q_m) + h(Q_1) + 1 \\ &= h(P_0\ Q_2 \ldots Q_m) + h(N) + 1 + 1 \\ &= h((\lambda y.P_0)N\ Q_2 \ldots Q_m) + 1 \end{aligned}$$

B. If Q_1 is in normal form then

$$\begin{aligned} h(M) &= h((\lambda y.P_0)Q_1 Q_2 \ldots Q_m) \\ &= h(P_0\ Q_2 \ldots Q_m) + h(Q_1) + 1 \\ &= h(P_0\ Q_2 \ldots Q_m) + 0 + 1 \end{aligned}$$

□

Theorem 6.4.4. The map $h : \Lambda \to \mathbb{N} \cup \{\infty\}$ computes the length of all the G_∞-rewrite sequence of a term M.

Proof: If $M \in \mathcal{SN}$ then a G_∞-rewrite sequence is of the form

$$M \to_\beta M_1 \to_\beta \ldots \to_\beta M_n \text{ with } M \text{ in normal form.}$$

It follows by induction on n that $h(M) = n$ using lemma 6.4.3.
If $M \notin \mathcal{SN}$ then a G_∞-rewrite sequence of M is infinite and indeed $h(M) = \infty$. □

Now we prove that $h : \Lambda \to \mathbb{N} \cup \{\infty\}$ computes the length of a maximal rewrite sequence starting at M. Here $\mathsf{maxred}(M)$ denotes the length of a maximal rewrite sequence starting in M.

Theorem 6.4.5. Let $M \in \Lambda$. We have

$$h(M) = \mathsf{maxred}(M)$$

Proof: If $M \notin \mathcal{SN}$, then $h(M) = \infty$ so it is clear that the statement holds.

Suppose that $M \in \mathcal{SN}$ is not in normal form. We will prove that the length of an arbitrary reduction to normal form is less than or equal to $h(M)$. The proof proceeds by induction on the number of steps in the derivation of $M \in \mathcal{SN}$. The term M is of the form $\lambda x_1 \ldots x_n.PQ_1 \ldots Q_m$ where P can be either a variable y or an abstraction $\lambda y.P_0$. We consider these two cases:

1. $P = y$. An arbitrary reduction from M to normal form can be transformed into a reduction sequence of the same length such that:

$$\begin{array}{rcl} \lambda x_1 \ldots \lambda x_n.yQ_1 \ldots Q_m & \overset{n_1}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.y\ \mathsf{nf}(Q_1)Q_2 \ldots Q_m \\ & \overset{n_2}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.y\ \mathsf{nf}(Q_1)\ \mathsf{nf}(Q_2) \ldots Q_m \\ & \twoheadrightarrow_\beta & \ldots \\ & \overset{n_m}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.y\ \mathsf{nf}(Q_1)\ \mathsf{nf}(Q_2) \ldots \mathsf{nf}(Q_m) \end{array}$$

   Here $\mathsf{nf}(M)$ denotes the normal form of M.

   The number of steps of this sequence is $n_1 + \ldots + n_m$. By induction hypothesis, we have $h(Q_i) \geq n_i$ for $i = 1, \ldots, m$. Hence we have

$$\begin{array}{rcl} h(M) & = & \displaystyle\sum_{i=1}^{m} h(Q_i) \\ & \geq & \displaystyle\sum_{i=1}^{m} n_i \end{array}$$

2. $P = \lambda y.P_0$. Two cases are distinguished.

   (a) $y \in FV(P_0)$. An arbitrary reduction sequence from M to normal form is of the form

$$\begin{array}{rcl} M & = & \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1Q_2 \ldots Q_m \\ & \overset{p}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0')Q_1'Q_2' \ldots Q_m' \\ & \to_\beta & \lambda x_1 \ldots \lambda x_n.P_0'[y := Q_1']Q_2' \ldots Q_m' \\ & \overset{l}{\twoheadrightarrow}_\beta & \mathsf{nf}(M) \end{array}$$

It can be transformed into a rewrite sequence of the form

$$\begin{array}{rcl} M & = & \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1Q_2 \ldots Q_m \\ & \to_\beta & \lambda x_1 \ldots \lambda x_n.P_0[y := Q_1]Q_2 \ldots Q_m \\ & \overset{k}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.P_0'[y := Q_1']Q_2' \ldots Q_m' \\ & \overset{l}{\twoheadrightarrow}_\beta & \mathsf{nf}(M) \end{array}$$

with $k \geq p$. By induction hypothesis, $h(P_0[y := Q_1]Q_2 \ldots Q_m) \geq k + l$. Hence

$$\begin{array}{rcl} h(M) & = & h(P_0[y := Q_1]Q_2 \ldots Q_m) + 1 \\ & \geq & k + l + 1 \\ & \geq & p + l + 1 \end{array}$$

(b) $y \notin P_0$. An arbitrary reduction sequence from M to normal form can be transformed into a reduction sequence of the same length of the form:

$$\begin{array}{rcl} M & = & \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1Q_2 \ldots Q_m \\ & \overset{p}{\twoheadrightarrow}_\beta & \lambda x_1 \ldots \lambda x_n.(\lambda y.P_0)Q_1'Q_2 \ldots Q_m \\ & \to_\beta & \lambda x_1 \ldots \lambda x_n.P_0\ Q_2 \ldots Q_m \\ & \overset{l}{\twoheadrightarrow}_\beta & \mathsf{nf}(M) \end{array}$$

By induction hypothesis we have that $h(Q_1) \geq p$ and $h(P_0Q_2 \ldots Q_m) \geq l$. Hence

$$\begin{array}{rcl} h(M) & = & h(P_0Q_2 \ldots Q_m) + h(Q_1) + 1 \\ & \geq & l + p + 1 \end{array}$$

□

Theorem 6.4.6. (Maximal Strategy) The strategy G_∞ is maximal.

Proof:

1. By theorem 6.4.4 we have that $h(M)$ is the length of all the G_∞-rewrite sequences of M.

2. By theorem 6.4.5 we have that $h(M) = \mathsf{maxred}(M)$ is the maximum length of all reductions sequences starting at M.

Hence the strategy G_∞ is maximal. □

As a consequence of the previous theorem we have that the strategy F_∞ is also maximal.

6.5 Conclusions and Related Work

In this chapter the strategies G_{bk} and G_∞ are defined which are similar to F_{bk} [BK82] and F_∞ [BBKV76]. Instead of looking at the leftmost redex, the new strategies look at the spine redexes. As a consequence, these strategies are non-deterministic.

The original proofs of the facts that F_{bk} and F_∞ are perpetual proceed by a case analysis [Bar85]. In order to prove that a strategy F is perpetual it is proved that $F(M)$ admits an infinite rewrite sequence if M does so. In order to prove that the strategies G_{bk} and G_∞ are perpetual, we use the set $\mathcal{SN}$. In order to prove that a strategy G is perpetual, we prove that $G(M) \subset \mathcal{SN} \Rightarrow M \in \mathcal{SN}$. The deterministic strategies F_{bk} and F_∞ are particular cases of G_{bk} and G_∞ and so they are perpetual. Proving $G(M) \subset \mathcal{SN} \Rightarrow M \in \mathcal{SN}$ and using the definition of $\mathcal{SN}$ make our proofs more perspicuous.

The fact that F_∞ is a maximal strategy has been proved by Régnier [Reg94] using a relation that permits the permutation of redexes. Much more in the spirit of the present work is a paper by Sørensen ([Sor94]), who gives a proof that is very similar to ours. His work was developed independently and simultaneously. In our case we have proved that G_∞ is maximal. Since F_∞ is a particular case of G_∞, we deduce that F_∞ is also maximal.

Chapter 7

Developments and Superdevelopments

7.1 Introduction

In this chapter we give two new and short proofs of the fact that in λ-calculus all β-developments terminate. In order to prove that all β-developments are finite it is sufficient to prove that the $\underline{\beta}$ is strongly normalising. For the first proof we define a set that characterises the $\underline{\beta}$-strongly normalising terms. Then we prove that any term belongs to this set. As a consequence of this we have that all the terms are $\underline{\beta}$-strongly normalising. For the second proof we define a mapping from the underlined λ-terms to the set $\mathcal{SN}$. We prove that this is a morphism between abstract rewriting systems.

Applying similar methods, we give two new and short proofs of the fact that in λ-calculus all β-superdevelopments terminate.

This chapter is organised as follows. In section 7.2 we recall the definition of development. In section 7.3 we give a short and simple proof of finiteness of developments. In section 7.4 we give another proof of finiteness of developments that makes direct use of the set $\mathcal{SN}$. In section 7.5 we recall the definition of superdevelopments. In section 7.6 we prove that all superdevelopments are finite. In section 7.7 another proof of finiteness of superdevelopments that makes direct use of the set $\mathcal{SN}$ and similar to the one in 7.4.

7.2 Developments

We shortly recall some definitions, for a complete formal treatment see [Bar85]. A development is a rewrite sequence in which only descendants of redexes that are present in the initial term may be contracted.

Usually, β-developments are defined via a set of underlined λ-terms and an underlined β-reduction rule.

Definition 7.2.1. The set of underlined λ-terms $\underline{\Lambda}$ is defined by induction as follows.

1. $x \in \underline{\Lambda}$ for every variable x,
2. if $M \in \underline{\Lambda}$, then $\lambda x.M \in \underline{\Lambda}$,
3. if $M \in \underline{\Lambda}$ and $N \in \underline{\Lambda}$, then $MN \in \underline{\Lambda}$,
4. if $M \in \underline{\Lambda}$ and $N \in \underline{\Lambda}$, then $(\underline{\lambda} x.M)N \in \underline{\Lambda}$.

The notion of context $C[\ ]$ with holes is defined similar to definition 5.2.8.
The $\underline{\beta}$-reduction is defined as follows.

$$C[(\underline{\lambda} x.M)N] \to_{\underline{\beta}} C[M[x := N]]$$

where $C[\ ]$ is a context with only one occurrence of $[\ ]$.
Note that $\underline{\Lambda}$ is closed under $\underline{\beta}$-reduction.
We define a mapping e that erases underlinings.

Definition 7.2.2. The mapping $e : \underline{\Lambda} \to \Lambda$ is defined as follows.

$$\begin{array}{rcl} e(x) & = & x \\ e(\lambda x.M) & = & \lambda x.e(M) \\ e(MN) & = & e(M)e(N) \\ e((\underline{\lambda} x.M)N) & = & (\lambda x.e(M))e(N) \end{array}$$

Lemma 7.2.3. The mapping e is a morphism from $(\underline{\Lambda}, \to_{\underline{\beta}})$ to $(\Lambda, \to_\beta)$.

Definition 7.2.4. A rewrite sequence $\sigma : M \twoheadrightarrow_\beta N$ in Λ is a *development* if there is a rewrite sequence ρ in $\underline{\Lambda}$ that is an e-lifting of σ. Diagrammatically,

$$\begin{tikzcd} \mathcal{J} \arrow[r, "\hat{\sigma}"] \arrow[dr, "\sigma"'] & (\underline{\Lambda}, \to_{\underline{\beta}}) \arrow[d, "e"] \\ & (\Lambda, \to_\beta) \end{tikzcd}$$

The abstract rewriting system $\mathcal{J}$ can be either $\mathcal{I}_n$ for some n or $\mathcal{I}$ (see example 2.2.3).

Example 7.2.5. The following β-rewrite sequence is a development.

$$\lambda x.(x\ x)\ \lambda x.(x\ x) \to_\beta \lambda x.(x\ x)\ \lambda x.(x\ x)$$

The e-lifting of this development is the following $\underline{\beta}$-rewrite sequence.

$$\underline{\lambda} x.(x\ x)\ \lambda x.(x\ x) \to_{\underline{\beta}} \lambda x.(x\ x)\ \lambda x.(x\ x)$$

The term $\lambda x.(x\ x)\ \lambda x.(x\ x)$ is in $\underline{\beta}$-normal form.

We write a function that computes the $\underline{\beta}$-normal form.

Definition 7.2.6. The mapping $\mathrm{nf}_{\underline{\beta}} : \underline{\Lambda} \to \Lambda$ is defined as follows.

$$\begin{aligned}
\mathrm{nf}_{\underline{\beta}}(x) &= x \\
\mathrm{nf}_{\underline{\beta}}(\lambda x.M) &= \lambda x.\mathrm{nf}_{\underline{\beta}}(M) \\
\mathrm{nf}_{\underline{\beta}}(MN) &= \mathrm{nf}_{\underline{\beta}}(M)\mathrm{nf}_{\underline{\beta}}(N) \\
\mathrm{nf}_{\underline{\beta}}((\underline{\lambda} x.M)N) &= \mathrm{nf}_{\underline{\beta}}(M)[x := \mathrm{nf}_{\underline{\beta}}(N)]
\end{aligned}$$

Note that $\mathrm{nf}_{\underline{\beta}}(M)$ does not contain $\underline{\lambda}$'s so it is in $\underline{\beta}$-normal form.

Lemma 7.2.7.

1. $M \twoheadrightarrow_{\underline{\beta}} \mathrm{nf}_{\underline{\beta}}(M)$.
2. If $M \to_{\underline{\beta}} N$ then $\mathrm{nf}_{\underline{\beta}}(M) = \mathrm{nf}_{\underline{\beta}}(N)$.

So we have that $\mathrm{nf}_{\underline{\beta}}(M)$ is the $\underline{\beta}$-normal form of M and is unique. The functions e and $\mathrm{nf}_{\underline{\beta}}$ are used in [Bar92] to prove confluence for β-reduction.

In the following sections, we prove that $\underline{\beta}$ is strongly normalising.

7.3 First Proof of Finiteness of Developments

We give a new and short proof of finiteness of developments by considering another inductive definition $\mathcal{D}$ of the set of all underlined λ-terms. Like in the definition of the set $\mathcal{SN}$, we make use of the expansion. We prove that all the terms in $\mathcal{D}$ are $\underline{\beta}$-strongly normalising. Finally, we prove that $\mathcal{D} = \underline{\Lambda}$.

Definition 7.3.1. The set $\mathcal{D}$ is the smallest set of λ-terms satisfying

1. $x \in \mathcal{D}$ for all variables x,
2. if $M \in \mathcal{D}$, then $\lambda x.M \in \mathcal{D}$,
3. if $M \in \mathcal{D}$ and $N \in \mathcal{D}$, then $MN \in \mathcal{D}$.
4. if $M[x := N] \in \mathcal{D}$ and $N \in \mathcal{D}$, then $(\underline{\lambda} x.M)N \in \mathcal{D}$.

The proof of the following lemma is immediate.

Lemma 7.3.2. If P in PQ is not of the form $\underline{\lambda} x.P_0$, then all $\underline{\beta}$-reducts of PQ are of the form $P'Q'$ with $P \twoheadrightarrow_{\underline{\beta}} P'$ and $Q \twoheadrightarrow_{\underline{\beta}} Q'$.

Theorem 7.3.3. If $M \in \mathcal{D}$, then all $\underline{\beta}$-rewrite sequences starting in M are finite.

Proof: The proof proceeds by induction on the derivation of $M \in \mathcal{D}$.

1. If M is a variable then it is trivial.

2. Let $M = \lambda x.P$ with $P \in \mathcal{D}$. By induction hypothesis, we have that P is strongly $\underline{\beta}$-normalising. So M is strongly $\underline{\beta}$-normalising.

3. Let $M = PQ$ with $P \in \mathcal{D}$ and $Q \in \mathcal{D}$. Note that P is not of the form $\underline{\lambda}x.P_0$. By lemma 7.3.2, every $\underline{\beta}$-reduct of M is of the form $P'Q'$ with $P \twoheadrightarrow_{\underline{\beta}} P'$ and $Q \twoheadrightarrow_{\underline{\beta}} Q'$. By induction hypothesis there are no infinite $\underline{\beta}$-rewrite sequences starting in P or in Q. Therefore M is strongly $\underline{\beta}$-normalising.

4. Let $M = (\underline{\lambda}x.P)Q$ with $P[x := Q] \in \mathcal{D}$ and $Q \in \mathcal{D}$. Consider an arbitrary $\underline{\beta}$-rewrite sequence $\rho : M = M_0 \twoheadrightarrow_{\underline{\beta}} M_1 \twoheadrightarrow_{\underline{\beta}} M_2 \twoheadrightarrow_{\underline{\beta}} \ldots$. There are two possibilities: in ρ the head redex of M is contracted or the head redex of M is not contracted.

   In the first case there is an i such that $M_i = P'[x := Q']$, with $P \twoheadrightarrow_{\underline{\beta}} P'$ and $Q \twoheadrightarrow_{\underline{\beta}} Q'$. The term M_i is a result of rewriting $P[x := Q]$, and the latter is by induction hypothesis strongly $\underline{\beta}$-normalising. Hence ρ is finite.

   In the second case all terms in ρ are of the form $(\underline{\lambda}x.P')Q'$ with $P \twoheadrightarrow_{\underline{\beta}} P'$ and $Q \twoheadrightarrow_{\underline{\beta}} Q'$. By induction hypothesis, $P[x := Q]$ is strongly $\underline{\beta}$-normalising, which yields that P is strongly $\underline{\beta}$-normalising, and moreover Q is strongly $\underline{\beta}$-normalising. Hence all terms in ρ are strongly normalising so ρ is finite.

□

Lemma 7.3.4. If $M \in \mathcal{D}$ and $N \in \mathcal{D}$ then $M[x := N] \in \mathcal{D}$.

This lemma is proved by induction on $M \in \mathcal{D}$.

Lemma 7.3.5. If $M[x := N] \in \underline{\Lambda}$ then $M \in \underline{\Lambda}$.

Theorem 7.3.6. $\underline{\Lambda} = \mathcal{D}$.

Proof:

$\subset$. Let $M \in \underline{\Lambda}$. We prove by induction on M that $M \in \mathcal{D}$. We prove the case that $M = (\underline{\lambda}x.P)Q$. By induction hypothesis, $P \in \mathcal{D}$ and $Q \in \mathcal{D}$. By lemma 7.3.4 we have that $P[x := Q] \in \mathcal{D}$ and by the definition of $\mathcal{D}$ we have that $(\underline{\lambda}x.P)Q \in \mathcal{D}$.

$\supset$. Let $M \in \mathcal{D}$. By induction on the derivation of $M \in \mathcal{D}$ we prove that $M \in \underline{\Lambda}$. We prove the case that $M = (\underline{\lambda}x.P)Q$. By induction hypothesis, $P[x := Q] \in \underline{\Lambda}$ and $Q \in \underline{\Lambda}$. By lemma 7.3.5, $P \in \underline{\Lambda}$. Hence $(\underline{\lambda}x.P)Q \in \underline{\Lambda}$.

□

Corollary 7.3.7. (Finiteness of Developments)
All β-developments are finite.

7.4 Second Proof of Finiteness of Developments

It is possible to prove in a different way, also using the set $\mathcal{SN}$, that all developments are finite. We define a morphism

$$(\underline{\Lambda}, \rightarrow_{\underline{\beta}}) \xrightarrow{l} (\mathcal{SN}, \rightarrow_{\beta})$$

Let Abs denote a distinguished variable.

Definition 7.4.1. We define $l : \underline{\Lambda} \rightarrow \mathcal{SN}$ as follows.

$$\begin{array}{rcl} l(x) & = & x \\ l(\lambda x.M) & = & \mathsf{Abs}\lambda x.l(M) \\ l(MN) & = & l(M)l(N) \\ l((\underline{\lambda} x.M)N) & = & (\lambda x.l(M))l(N) \end{array}$$

Lemma 7.4.2. $l(M[x := N]) = l(M)[x := l(N)]$.

Theorem 7.4.3.

1. if $M \in \underline{\Lambda}$ then $l(M) \in \mathcal{SN}$,
2. if $M \in \underline{\Lambda}$ and $M \rightarrow_{\underline{\beta}} N$, then $l(M) \rightarrow_{\beta} l(N)$.

Corollary 7.4.4. The mapping l is a morphism from $(\underline{\Lambda}, \rightarrow_{\underline{\beta}})$ into $(\mathcal{SN}, \rightarrow_{\beta})$.

Theorem 7.4.5. The rewrite relation $\underline{\beta}$ is strongly normalising.

Proof: This follows from lemma 2.6.5 and corollary 7.4.4. □

Corollary 7.4.6. (Finiteness of Developments)
All β-developments are finite.

7.5 Superdevelopments

In [Raa93], *superdevelopments* were introduced and proved to be finite. Superdevelopments form an extension of the notion of development. In a superdevelopment not only redexes that descend from the initial term may be contracted, but also some redexes that are created during reduction.

There are three ways of creating new redexes (see [Lev78]):

1. $((\lambda x.\lambda y.M)N)P \rightarrow_{\beta} (\lambda y.M[x := N])P$
2. $(\lambda x.x)(\lambda y.M)N \rightarrow_{\beta} (\lambda y.M)N$

3. $(\lambda x.C[xM])(\lambda y.N) \to_\beta C'[(\lambda y.N)M']$ where C' and M' are obtained from C and M by replacing all free occurrences of x by $(\lambda y.N)$.

The first two kinds of created redexes are 'innocent' and they may be contracted in a superdevelopment. The result that all superdevelopments are finite shows that infinite β-reduction sequences are due to the presence of the third type of redexes.

In the following two sections we give two new proofs of the fact that in λ-calculus all β-superdevelopments terminate.

First we shortly repeat the definition of a superdevelopment. The definition makes use of a set of labelled λ-terms and a notion of labelled β-reduction. Since application nodes will be labelled, we write them explicitly.

Definition 7.5.1. The set Λ_l^ω of labelled λ-terms is defined by induction as follows.

1. $x \in \Lambda_l^\omega$ for every variable x,
2. if $M \in \Lambda_l^\omega$ and $i \in \mathbb{N}$, then $\lambda_i x.M \in \Lambda_l^\omega$,
3. if $M, N \in \Lambda_l^\omega$ and $X \subset \mathbb{N}$, then $@^X(M,N) \in \Lambda_l^\omega$.

Sometimes we write i instead of $\{i\}$ for $i \in \mathbb{N}$.

The notion of context $C[\ ]$ with holes is defined similar to definition 5.2.8.

On the set Λ_l^ω, the β_l-reduction is defined as follows.

$$C[@^X(\lambda_i x.M, N)] \to C[M[x := N]] \quad \text{if } i \in X$$

where $C[\ ]$ has only one occurrence of $[\ ]$.

We define a mapping from Λ_l to Λ that erases the labels.

Definition 7.5.2. The mapping $e_l : \Lambda_l^\omega \to \Lambda$ is defined by induction on the definition of Λ_l as follows.

$$\begin{aligned} e_l(x) &= x \\ e_l(\lambda_i x.M) &= \lambda x.e_l(M) \\ e_l(@^X(M,N)) &= e_l(M)e_l(N) \end{aligned}$$

The proof of the following lemma is straightforward.

Lemma 7.5.3. The mapping e_l is a morphism from $(\Lambda_l^\omega, \to_{\beta_l})$ to $(\Lambda, \to_\beta)$.

The β_l-reduction is not strongly normalising on the set Λ_l^ω since any β-rewrite sequence can be lifted in a β_l-rewrite sequence. This is illustrated by the following example.

Example 7.5.4. Let $\omega = \lambda_1 x.@^1(x,x)$.

$$@^1(\omega,\omega) \quad \to_{\beta_l} \quad @^1(\omega,\omega)$$

The term $@^1(\omega,\omega)$ is not β_l-strongly normalising.

We restrict the set Λ_l^ω to a set Λ_l of well-labelled terms.

Definition 7.5.5.

1. A term $M \in \Lambda_l$ is said to be *well-labelled* if the label X of an application node never contains the label i of a λ outside the scope of the application node. The set of well-labelled λ-terms is denoted by Λ_l.

2. A term $M \in \Lambda_l$ is *initially labelled* if it is well-labelled and all λ's have a different label.

The set Λ_l of well-labelled terms is closed under β_l-reduction.

Lemma 7.5.6. **(β_l-closure)** If $M \in \Lambda_l$ and $M \to_{\beta_l} N$ then $N \in \Lambda_l$.

This is proved by induction on $M \in \Lambda_l^\omega$.

Definition 7.5.7. A rewrite sequence $\sigma : M \twoheadrightarrow_\beta N$ in Λ is a *superdevelopment* if there is a rewrite sequence ρ in Λ_l that starts in an initially labelled term and that is an e_l-lifting of σ. Diagrammatically,

$$\begin{tikzcd}
\mathcal{J} \arrow[r, "\hat{\sigma}"] \arrow[dr, "\sigma"'] & (\Lambda_l, \to_{\beta_l}) \arrow[d, "e_l"] \\
& (\Lambda, \to_\beta)
\end{tikzcd}$$

The abstract rewriting system $\mathcal{J}$ is either $\mathcal{I}_n$ or $\mathcal{I}$ (see example 2.2.3 and definition 2.3.4).

Example 7.5.8. The following β-rewrite sequence is a superdevelopment.

$$\begin{array}{lll}
(\lambda x.\lambda y.xy)(\lambda z.z)u & \to_\beta & (\lambda y.(\lambda z.z)y)u \\
 & \to_\beta & (\lambda z.z)u
\end{array}$$

An e_l-lifting for this β-rewrite sequence is, for example, the following β_l-rewrite sequence.

$$\begin{array}{lll}
@^2(@^1(\lambda_1 x.\lambda_2 y.@^3(x,y), \lambda_4 z.z), u) & \to_{\beta_l} & @^2(\lambda_2 y.@^3(\lambda_4 z.z, y), u) \\
 & \to_{\beta_l} & @^3(\lambda_4 z.z, u)
\end{array}$$

In the first step, the variable x is replaced by $\lambda_4 z.z$. This corresponds to the third way of creating a redex and so $@^3(\lambda_4 z.z, y)$ is not a β_l-redex.

We define a function that computes the β_l-normal form of a term.

Definition 7.5.9. The function $\mathrm{nf}_{\beta_l} : \Lambda_l \to \Lambda_l$ is defined by induction on the definition of Λ_l as follows.

$$\begin{aligned} nf_{\beta_l}(x) &= x \\ nf_{\beta_l}(\lambda_i x.M) &= \lambda_i x.nf_{\beta_l}(M) \\ nf_{\beta_l}(@^X(M,N)) &= \begin{cases} M_0[x := nf_{\beta_l}(N)] & \text{if } nf_{\beta_l}(M) = \lambda_i x.M_0 \text{ and } i \in X \\ @^X(nf_{\beta_l}(M), nf_{\beta_l}(N)) & \text{otherwise} \end{cases} \end{aligned}$$

Lemma 7.5.10. If $@^X(M,N) \in \Lambda_l$ then $nf_{\beta_l}(M[x := N]) = nf_{\beta_l}(M)[x := nf_{\beta_l}(N)]$.

Lemma 7.5.11. Let $M \in \Lambda_l$.

1. $M \twoheadrightarrow_{\beta_l} nf_{\beta_l}(M)$ and $nf_{\beta_l}(M)$ is in β_l-normal form.
2. If $M \to_{\beta_l} M'$ then $nf_{\beta_l}(M) = nf_{\beta_l}(M')$.

So we have $nf_{\beta_l}(M)$ is the β_l-normal form of M and it is unique. As a consequence of this we have that β_l is confluent.

In the following sections, we prove that β_l is strongly normalising.

7.6 First Proof of Finiteness of Superdevelopments

We give a new proof of the fact that all superdevelopments are finite. It is similar to the proof of finite developments in section 7.3. We define a set $\mathcal{SD}$ of underlined λ-terms by induction. In this case we make use of the expansion looking at the cases 1) and 2) of creating new redexes. We prove that the terms in $\mathcal{SD}$ are $\underline{\beta}$-strongly normalising. Finally, we write a morphism

$$(\Lambda_l, \to_{\beta_l}) \xrightarrow{u} (\mathcal{SD}, \to_{\underline{\beta}})$$

Definition 7.6.1. The set $\mathcal{SD}$ is defined by induction as follows.

1. $x \in \mathcal{SD}$ for all variables x,
2. if $M \in \mathcal{SD}$, then $\lambda x.M \in \mathcal{SD}$,
3. if $M \in \mathcal{SD}$ and $N \in \mathcal{SD}$, then $MN \in \mathcal{SD}$,
4. if $M[x := N]P_1 \ldots P_n \in \mathcal{SD}$ and $N \in \mathcal{SD}$, then $(\underline{\lambda} x.M)NP_1 \ldots P_n \in \mathcal{SD}$,
5. if $(\underline{\lambda} y.M)NP_1 \ldots P_n \in \mathcal{SD}$, then $(\underline{\lambda} x.x)(\underline{\lambda} y.M)NP_1 \ldots P_n \in \mathcal{SD}$.

The notion of context $C[\,]$ with holes is defined similar to definition 5.2.8.
The $\underline{\beta}$-rewrite relation is defined as follows.

$$C[(\underline{\lambda}x.M)N] \to_{\underline{\beta}} C[M[x := N]]$$

where $C[\,]$ has only one occurrence of $[\,]$.

Lemma 7.6.2. If $M, N \in \mathcal{SD}$ then $M[x := N] \in \mathcal{SD}$.

Lemma 7.6.3. **($\underline{\beta}$-Closure)** Let $M \in \mathcal{SD}$. If $M \to_{\underline{\beta}} M'$ then $M' \in \mathcal{SD}$.

Lemma 7.6.4. Let $M = PQ$ with $P \in \mathcal{SD}$ and $Q \in \mathcal{SD}$. If $M \twoheadrightarrow_{\underline{\beta}} M'$, then $M' = P'Q'$ with $P \twoheadrightarrow_{\underline{\beta}} P'$ and $Q \twoheadrightarrow_{\underline{\beta}} Q'$.

Theorem 7.6.5. If $M \in \mathcal{SD}$, then all $\underline{\beta}$-rewrite sequences starting at M are finite.

Proof: The proof proceeds by induction on the derivation of $M \in \mathcal{SD}$.

1. If M is a variable then it is trivial.

2. Let $M = \lambda x.P$ with $P \in \mathcal{SD}$. By induction hypothesis, we have that P is strongly $\underline{\beta}$-normalising. So M is strongly $\underline{\beta}$-normalising.

3. Let $M = PQ$ with $P \in \mathcal{SD}$ and $Q \in \mathcal{SD}$. By induction hypothesis P and Q are $\underline{\beta}$-strongly normalising. It follows from lemma 7.6.4 that any $\underline{\beta}$-sequence starting at M is finite.

4. Let $M = (\underline{\lambda}x.P)QN_1 \ldots N_n$ with $P[x := Q]N_1 \ldots N_n \in \mathcal{SD}$. Consider an arbitrary $\underline{\beta}$-rewrite sequence $\rho : M = M_0 \twoheadrightarrow_{\underline{\beta}} M_1 \twoheadrightarrow_{\underline{\beta}} M_2 \twoheadrightarrow_{\underline{\beta}} \ldots$. There are two possibilities: in ρ the head redex of M is contracted or the head redex of M is not contracted.

   In the first case, there is an i such that $M_i = P'[x := Q']N'_1 \ldots N'_n$ with $P \twoheadrightarrow_{\underline{\beta}} P'$, $Q \twoheadrightarrow_{\underline{\beta}} Q', N_1 \twoheadrightarrow_{\underline{\beta}} N'_1, \ldots, N_n \twoheadrightarrow_{\underline{\beta}} N'_n$.

   The term M_i is obtained by rewriting $P[x := Q]N_1 \ldots N_n$ and the latter term is by induction hypothesis strongly $\underline{\beta}$-normalising. Hence ρ is finite.

   In the second case, all terms in ρ are of the form $(\underline{\lambda}x.P')Q'N'_1 \ldots N'_n$ with $P \to \to_{\underline{\beta}} P', Q \twoheadrightarrow_{\underline{\beta}} Q', N_1 \twoheadrightarrow_{\underline{\beta}} N'_1, \ldots, N_n \twoheadrightarrow_{\underline{\beta}} N'_n$. Since $P[x := Q]N_1 \ldots N_n$ and Q are by induction hypothesis strongly $\underline{\beta}$-normalising, we have that $P, Q, N_1, \ldots, N_n$ are strongly $\underline{\beta}$-normalising. So all terms in ρ are strongly $\underline{\beta}$-normalising and hence ρ is finite.

5. Let $M = (\underline{\lambda}x.x)(\underline{\lambda}y.N)PN_1 \ldots N_n$ with $(\underline{\lambda}y.N)PN_1 \ldots N_n \in \mathcal{SD}$. Consider an arbitrary $\underline{\beta}$-rewrite sequence $\rho : M = M_0 \twoheadrightarrow_{\underline{\beta}} M_1 \twoheadrightarrow_{\underline{\beta}} M_2 \twoheadrightarrow_{\underline{\beta}} \ldots$. There are two possibilities: in ρ the head redex of M is contracted or the head redex of M is not contracted.

In the first case, there is an i such that $M_i = (\underline{\lambda} y.N')P'N'_1 \ldots N'_n$ with $N \twoheadrightarrow_{\underline{\beta}} N', P \twoheadrightarrow_{\underline{\beta}} P', N_1 \twoheadrightarrow_{\underline{\beta}} N'_1, \ldots, N_n \twoheadrightarrow_{\underline{\beta}} N'_n$. The term M_i is obtained by rewriting the term $(\underline{\lambda} y.N)PN_1 \ldots N_n$ and the latter term is by induction hypothesis strongly $\underline{\beta}$-normalising. So M_i is strongly $\underline{\beta}$-normalising and hence ρ is finite.

In the second case, all terms in ρ are of the form $(\underline{\lambda} x.x)(\underline{\lambda} y.N')P'N'_1 \ldots N'_n$ with $N \twoheadrightarrow_{\underline{\beta}} N', P \twoheadrightarrow_{\underline{\beta}} P', N_1 \twoheadrightarrow_{\underline{\beta}} N'_1, \ldots, N_n \twoheadrightarrow_{\underline{\beta}} N'_n$.

By induction hypothesis, $(\underline{\lambda} y.N)PN_1 \ldots N_n$ is strongly $\underline{\beta}$-normalising.

Hence $N, P, N_1, \ldots, N_n$ are all strongly $\underline{\beta}$-normalising. This yields that ρ is finite.

□

We define a set $\underline{\Lambda}^\omega$ of 'liberal' underlined λ-terms.

Definition 7.6.6. The set $\underline{\Lambda}^\omega$ is the smallest set satisfying the following.

1. $x \in \underline{\Lambda}^\omega$ for every variable x,
2. if $M \in \underline{\Lambda}^\omega$, then $\lambda x.M \in \underline{\Lambda}^\omega$,
3. if $M \in \underline{\Lambda}^\omega$ then $(\underline{\lambda} x.M) \in \underline{\Lambda}^\omega$,
4. if $M \in \underline{\Lambda}^\omega$ and $N \in \underline{\Lambda}^\omega$, then $MN \in \underline{\Lambda}^\omega$.

We define a mapping $(-)^* : \mathcal{SD} \to \underline{\Lambda}^\omega$ that underlines one special λ of a term. If $M \twoheadrightarrow_{\underline{\beta}} \lambda x.M_0$ then the value M^* is obtained from M by underlining the λ that descends to the head lambda in $\lambda x.M_0$.

Definition 7.6.7. We define $(-)^* : \mathcal{SD} \to \underline{\Lambda}^\omega$ by induction on the length of the maximal $\underline{\beta}$-rewrite sequence to normal form.

$$
\begin{array}{rcll}
(\lambda x.M)^* & = & \underline{\lambda} x.M & \\
((\underline{\lambda} x.M)NP_1 \ldots P_n)^* & = & (\underline{\lambda} x.M')N'P'_1 \ldots P'_n & \text{if} \quad (M[x := N]P_1 \ldots P_n)^* = \\
 & & & \qquad M'[x := N']P'_1 \ldots P'_n \\
M^* & = & M & \qquad \text{otherwise}
\end{array}
$$

Lemma 7.6.8. If $M, N \in \mathcal{SD}$ then $M^*N \in \mathcal{SD}$.

Proof: Let $M \in \mathcal{SD}$. Then M is either of the form $(xU_1 \ldots U_n)$, or $(\lambda x.P)U_1 \ldots U_n$ or $(\underline{\lambda} x.P)QU_1 \ldots U_n$. We proceed by induction on the length of the maximal $\underline{\beta}$-rewrite sequence to normal form.

1. Let $M = (\lambda x.P)$. Then $(\lambda x.P)^* = \underline{\lambda} x.P$. By lemma 7.6.2, we have that $P[x := N] \in \mathcal{SD}$. Hence $(\lambda x.P)^*N \in \mathcal{SD}$.

2. Let $M = xU_1 \ldots U_n$. Then $M^* = M$ and $MN \in \mathcal{SD}$.
3. Let $M = (\lambda x.P)QU_1 \ldots U_n$. Then $M^* = M$ and $MN \in \mathcal{SD}$.
4. Let $M = (\underline{\lambda} x.P)QU_1 \ldots U_n$. By induction hypothesis, we have that $(P[x := Q]U_1 \ldots U_n)^* N \in \mathcal{SD}$. Hence $M^* N \in \mathcal{SD}$.

□

Lemma 7.6.9. Let $M \in \mathcal{SD}$. If $M \neq (xN_1 \ldots N_n)$ then $(M[x := P])^* = M^*[x := P]$.

The proof proceeds by induction on the length of a maximal $\underline{\beta}$-rewrite sequence to normal form.

Definition 7.6.10. The mapping $u : \Lambda_l \to \underline{\Lambda}^\omega$ is defined as follows.

$$\begin{aligned} u(x) &= x \\ u(\lambda_i x.M) &= \lambda x.u(M) \\ u(@^X(M,N)) &= \begin{cases} u(M)^*\ u(N) & \text{if } nf_{\beta_l}(M) = \lambda_i x.M_0 \text{ and } i \in X \\ u(M)\ u(N) & \text{otherwise} \end{cases} \end{aligned}$$

Theorem 7.6.11. Let $M \in \Lambda_l$. Then $u(M) \in \mathcal{SD}$.

The previous theorem is proved by induction on $M \in \Lambda_l$ and using lemma 7.6.8.

Lemma 7.6.12. Let $M \in \Lambda_l$. If $u(M) = (xN_1 \ldots N_n)$ then $nf_{\beta_l}(M)$ is of the form $(xP_1 \ldots P_n)$ for some terms $P_1, \ldots, P_n$.

This lemma is proved by induction on $M \in \Lambda_l$.

Lemma 7.6.13. Let $@^X(M,N) \in \Lambda_l$. Then $u(M[x := N]) = u(M)[x := u(N)]$.

Proof: The proof proceeds by induction on $M \in \Lambda_l$. We prove only the case of the application $M = @^X(P,Q)$ with $nf_{\beta_l}(P) = \lambda_i x.P_0$ and $i \in X$.

It follows from lemma 7.5.10 and the fact that the terms are well-labelled that

$$nf_{\beta_l}(P) = \lambda_i x.P_0 \text{ if and only if } nf_{\beta_l}(P[x := N]) = \lambda_i x.P_0[x := nf_{\beta_l}(N)].$$

Therefore

$$\begin{aligned} u(@^X(P,Q))[x := u(N)] &= u(P)^*\ u(Q)[x := N] && \text{by definition 7.6.10} \\ &= u(P[x := N])^*\ u(Q[x := N]) && \text{by lemmas 7.6.9 and 7.6.12} \\ &= u(@^X(P,Q)[x := N]) && \text{by definition 7.6.10} \end{aligned}$$

□

Theorem 7.6.14. Let $M \in \Lambda_l$. If $M \to_{\beta_l} N$ in Λ_l then $u(M) \to_{\underline{\beta}} u(N)$ in $\mathcal{SD}$.

This theorem is proved by using lemma 7.6.13.

Theorem 7.6.15. The mapping u is a morphism from $(\Lambda_l, \to_{\beta_l})$ to $(\mathcal{SD}, \to_{\underline{\beta}})$.

The proof follows from theorems 7.6.11 and 7.6.15.

Corollary 7.6.16. (Finiteness of Superdeveloments)
All superdevelopments are finite.

7.7 Second Proof of Finiteness of Superdevelopments

Another proof of the fact that all superdevelopments are finite can be given in a way similar to the one in section 7.4. We define a morphism

$$(\Lambda_l, \to_{\beta_l}) \xrightarrow{\jmath} (\mathcal{SN}, \to_\beta)$$

Let App denote a distinguished variable.

Definition 7.7.1. The mapping $\jmath : \Lambda_l \to \mathcal{SN}$ is defined as follows.

$$\begin{aligned}
\jmath(x) &= x \\
\jmath(\lambda_i x.M) &= \lambda x.\jmath(M) \\
\jmath(@^X(M,N)) &= \begin{cases} \jmath(M)\jmath(N) & \text{if } nf_{\beta_l}(M) = \lambda_i x.M \text{ and } i \in X \\ \mathsf{App}\jmath(M)\jmath(N) & \text{otherwise} \end{cases}
\end{aligned}$$

Lemma 7.7.2. Let $M \in \Lambda_l$, $\jmath(M) \in \mathcal{SN}$ and $N \in \mathcal{SN}$. Then $\jmath(M)[x := N] \in \mathcal{SN}$.

Proof: The proof proceeds by induction on the derivation of $\jmath(M) \in \mathcal{SN}$.

1. Suppose $\jmath(M) = yP_1 \ldots P_n$ with $P_i \in \mathcal{SN}$ for $i = 1, \ldots, n$. If $n > 0$ then $y = \mathsf{App}$. By induction hypothesis, $\jmath(P_i)[x := N] \in \mathcal{SN}$ for $i = 1, \ldots, n$. Hence $\jmath(M)[x := N] \in \mathcal{SN}$.

2. Suppose $\jmath(M) = \lambda y.P$ with $P \in \mathcal{SN}$. Using induction hypothesis we obtain that $\jmath(M)[x := N] \in \mathcal{SN}$.

3. Suppose $\jmath(M) = (\lambda y.P)Q_1Q_2 \ldots Q_n$ with $P[y := Q_1]Q_2 \ldots Q_n \in \mathcal{SN}$ and $Q_1 \in \mathcal{SN}$. By induction hypothesis, we have $(P[y := Q_1]Q_2 \ldots Q_n)[x := N] \in \mathcal{SN}$ and $Q_1[x := N] \in \mathcal{SN}$. This yields $\jmath(M)[x := N] \in \mathcal{SN}$.

□

Lemma 7.7.3. Let $M \in \Lambda_l$, $\jmath(M) \in \mathcal{SN}$ and $N \in \mathcal{SN}$. Then $\jmath(M)N \in \mathcal{SN}$.

Proof: The proof proceeds by induction on the derivation of $\jmath(M) \in \mathcal{SN}$.

1. Suppose $\jmath(M) = xP_1 \ldots P_n$ with $P_i \in \mathcal{SN}$ for $i = 1, \ldots, n$. Then $\jmath(M)N \in \mathcal{SN}$.

2. Suppose $\jmath(M) = \lambda x.P$ with $P \in \mathcal{SN}$. Then $M = \lambda_i x.M_0$ and $\jmath(M_0) = P$. By the previous lemma we have $P[x := N] \in \mathcal{SN}$. Hence $\jmath(M)N \in \mathcal{SN}$.

3. Suppose $\jmath(M) = (\lambda x.P)Q_1Q_2 \ldots Q_n$ with $P[x := Q_1]Q_2 \ldots Q_n \in \mathcal{SN}$ and $Q_1 \in \mathcal{SN}$. By induction hypothesis, we have $P[x := Q_1]Q_2 \ldots Q_nN \in \mathcal{SN}$. Moreover $Q_1 \in \mathcal{SN}$, hence $\jmath(M)N \in \mathcal{SN}$.

□

Lemma 7.7.4. Let $@^X(M, N) \in \Lambda_l$. Then $\jmath(M)[x := \jmath(N)] = \jmath(M[x := N])$.

This lemma is proved by induction on $M \in \Lambda_l$.

Theorem 7.7.5. Let $M \in \Lambda_l$.

1. $\jmath(M) \in \mathcal{SN}$.

2. If $M \to_{\beta_l} N$ in Λ_l then $\jmath(M) \to_\beta \jmath(N)$ in $\mathcal{SN}$.

Proof: The proof of the first part proceeds by induction on $M \in \Lambda_l$ and makes use of the lemmas 7.7.2 and 7.7.3. The second part uses lemma 7.7.4. □

Corollary 7.7.6. The mapping $\jmath$ is a morphism from $(\Lambda_l, \to_{\beta_l})$ to $(\mathcal{SN}, \to_\beta)$.

Theorem 7.7.7. $\to_{\beta_l}$ is strongly normalising.

This theorem follows from lemma 2.6.5 and corollary 7.7.6.

Corollary 7.7.8. (Finiteness of Superdeveloments)
All superdevelopments are finite.

7.8 Conclusions and Related Work

The result that all β-developments are finite is a classical result in λ-calculus and various proofs already exist. Church and Rosser proved finiteness of developments for the λI-calculus with β-reduction in [CR36]. The first proof for the full λ-calculus is given by Schroer in [Sch65]. Other proofs have been given in [Hyl73] and [Bar85]. In [Klo80], finiteness of developments is proved from strong normalisation for a β-reduction with 'memory' [Ned73] (see also section 11.5). There is a short and elegant proof by de Vrijer [Vri85], in which an exact bound for the length of a development is computed. For proving that the bound is an exact bound, he makes in fact use of the strategy F_∞. Another proof can be found in [Par90] (see also [Kri93]) that uses strong normalisation of the simply typed lambda calculus with intersection types. In this proof a morphism from the set of underlined lambda terms to the simply typed lambda calculus with intersection types is defined similar to our morphism used in the second proof of finiteness of developments (see definition 7.4.1). A similar proof using strong normalisation of simply typed lambda calculus appears in [Ghi94]. In [Mel96] an axiomatic and general proof of finiteness of developments is given.

In [Raa93] the proof that the superdevelopments are finite uses the method of minimalisation.

Superdevelopments are related to the so-called 'generalised β-reduction'. The generalised β-reduction first appers in [Ned73] (see also [KN95]) as a natural generalisation of the β-reduction in the item notation. A β-redex in item notation is a δ-item (an application) followed by a λ-item (an abstraction) like if they were a pair () of parentheses. The notion of β-redex is generalised to include more complicated structures of parentheses like (()). For example, a generalised β-rewrite step (or β_g-rewrite step) in our notation is the following:

$$(\lambda x.\lambda y.M)NP \rightarrow_{\beta_g} (\lambda x.M[y := P])N$$

If we underline the β_g-redexes, we get the underlined β-redexes of a superdevelopment. An way of labelling alternative to the one presented in [Raa93] (see definition 7.5.5) is to represent the term in item notation and to mark all the δ and λ-items that match as if they were parentheses.

Chapter 8

Simply Typed Lambda Calculus

8.1 Introduction

In this section we give a new proof of the fact that the simply typed λ-calculus is β-strongly normalising. In the proof we make use of the characterisation of the strongly normalising λ-terms.

This chapter is organised as follows. In section 8.2 we recall the definition of the simply typed lambda calculus à la Church. In section 8.3, we give a new proof of the fact that the simply typed lambda calculus is strongly normalising.

8.2 Simply Typed λ-calculus

In this section we shortly recall the definition of simply typed λ-calculus.

Definition 8.2.1. The set $Type$ is defined as follows.

1. $0 \in Type$,
2. if $\tau \in Type$ and $\sigma \in Type$ then $(\tau \to \sigma) \in Type$.

Types are written as $\tau, \sigma, \ldots$. We write $\tau_1 \to \tau_2 \to \tau_3$ instead of $(\tau_1 \to (\tau_2 \to \tau_3))$.

Note that a type τ is always of the form $\tau_1 \to \ldots \to \tau_n \to 0$.

We assume that we have an infinite number of variables, $V_\tau = \{v^\tau, v'^\tau \ldots\}$ for each $\tau \in Type$.

The set V of variables is $\bigcup_{\tau \in Type} V_\tau$.

A variable that ranges on V_τ is denoted by $x, y, \ldots$. If $x \in V_\tau$ and $y \in V_\sigma$ with $\tau \neq \sigma$ then $x \neq y$.

Definition 8.2.2. We define the family $\{\Lambda_\tau\}_{\tau \in Type}$ of subsets of Λ as follows.

1. $V_\tau \subset \Lambda_\tau$,
2. if $M \in \Lambda_{\tau\to\sigma}$ and $N \in \Lambda_\tau$ then $(M\ N) \in \Lambda_\sigma$,
3. if $M \in \Lambda_\sigma$ and $x \in V_\tau$ then $\lambda x.M \in \Lambda_{\tau\to\sigma}$.

Definition 8.2.3. The *simply typed lambda calculus* λ^τ (or $\lambda_\to$) is the abstract rewriting system with typing defined by

$$(\Lambda, \to_\beta, \{(M,\tau) \mid M \in \Lambda_\tau\})$$

Substitution and β-reduction are defined as in chapter 5.

Lemma 8.2.4. (Substitution Lemma)
Let $x \in V_\tau$. If $M \in \Lambda_\sigma$ and $N \in \Lambda_\tau$ then $M[x := N] \in \Lambda_\sigma$.

Definition 8.2.5. If $A \subset \Lambda_\tau$ and $B \subset \Lambda_\sigma$, we define

$$A \to B = \{M \in \Lambda_{\tau\to\sigma} \mid \forall N \in A : MN \in B\}$$

Note that if $A \subset A'$ then $A' \to B \subset A \to B$ and that if $B \subset B'$ then $A \to B \subset A \to B'$.

Lemma 8.2.6. $\Lambda_{\tau\to\sigma} = \Lambda_\tau \to \Lambda_\sigma$.

Proof: If $M \in \Lambda_{\tau\to\sigma}$ and $N \in \Lambda_\tau$ then $MN \in \Lambda_\sigma$. Conversely, if $M \in \Lambda_\tau \to \Lambda_\sigma$, then $(Mx) \in \Lambda_\sigma$ for $x \in V_\tau$. Therefore $M \in \Lambda_{\tau\to\sigma}$. $\square$

8.3 Strong Normalisation

In this section we prove that the simply typed λ-calculus is β-strongly normalising.

Definition 8.3.1. The set $\mathcal{SN}(\tau)$ is defined as follows.

$$\mathcal{SN}(\tau) = \Lambda_\tau \cap \mathcal{SN}$$

Theorem 8.3.2. $\mathcal{SN}(\tau \to \sigma) \supset \mathcal{SN}(\tau) \to \mathcal{SN}(\sigma)$.

Proof: Let $M \in \mathcal{SN}(\tau) \to \mathcal{SN}(\sigma)$. We have that $M \in \mathcal{SN}$, because $MN \in \mathcal{SN}(\sigma)$. If $(Mx) \in \Lambda_\sigma$ for $x \in V_\tau$ then $M \in \Lambda_{\tau\to\sigma}$. $\square$

The converse inclusion is not so easy to prove. First we need the following lemma.

Lemma 8.3.3. Let $N \in \mathcal{SN}(\tau_1) \to \ldots \to \mathcal{SN}(\tau_n) \to \mathcal{SN}(0)$.
If $P \in \mathcal{SN}(\sigma)$ and $x \in V_{\tau_1\to\ldots\to\tau_n\to 0}$ then $P[x := N] \in \mathcal{SN}(\sigma)$.

Proof: The proof proceeds by induction on the derivation of $P \in \mathcal{SN}$.

1. Suppose $P = yP_1 \dots P_k$ with $P_1, \dots, P_k \in \mathcal{SN}$. By induction hypothesis, we have $P_i[x := N] \in \mathcal{SN}$ for $i = 1, \dots, k$. We write P_i^* for $P_i[x := N]$ for $i = 1, \dots, k$.

   If $y \neq x$, then $P[x := N] \in \mathcal{SN}$ follows from the fact that $P_i^* \in \mathcal{SN}$ for $i = 1, \dots, k$. Using lemma 8.2.4, we obtain $P[x := N] \in \mathcal{SN}(\sigma)$.

   If $y = x$, then we have to prove that $NP_1^* \dots P_k^* \in \mathcal{SN}(\sigma)$. By the induction hypothesis and lemma 8.2.4, we have that $P_i^* \in \mathcal{SN}(\tau_i)$ for $i = 1, \dots, k$. Furthermore, $N \in \mathcal{SN}(\tau_1) \to \dots \to \mathcal{SN}(\tau_n) \to \mathcal{SN}(0) \subset \mathcal{SN}(\tau_1) \to \dots \to \mathcal{SN}(\tau_k) \to \mathcal{SN}(\sigma)$, by theorem 8.3.2. Hence we have $P[x := N] = NP_1^* \dots P_k^* \in \mathcal{SN}(\sigma)$.

2. Suppose $P = \lambda y.P_0$ with $P_0 \in \mathcal{SN}$. By induction hypothesis, we have $P_0[x := N] \in \mathcal{SN}$. Therefore $P[x := N] = (\lambda z.P_0)[x := N] \in \mathcal{SN}(\sigma)$.

3. Suppose $P = (\lambda y.P_0)P_1P_2 \dots P_k$ with $P_0[y := P_1]P_2 \dots P_k \in \mathcal{SN}$ and $P_1 \in \mathcal{SN}$. By induction hypothesis, we have $(P_0[y := P_1]P_2 \dots P_k)[x := N] \in \mathcal{SN}$ and $P_1[x := N] \in \mathcal{SN}$. Hence $P[x := N] = ((\lambda y.P_0)P_1P_2 \dots P_k)[x := N] \in \mathcal{SN}(\sigma)$.

□

Now we can prove the following theorem.

Theorem 8.3.4. $\mathcal{SN}(\tau \to \sigma) \subset \mathcal{SN}(\tau) \to \mathcal{SN}(\sigma)$.

Proof: Let $M \in \mathcal{SN}(\tau \to \sigma)$. We prove that for all $N \in \mathcal{SN}(\tau)$, we have $MN \in \mathcal{SN}(\sigma)$. Let $N \in \mathcal{SN}(\tau)$. Note that $MN \in \Lambda_\sigma$. It remains to prove that $MN \in \mathcal{SN}$. This is proven by induction on τ and for each τ by induction on the derivation of $M \in \mathcal{SN}$.

τ is 0. The proof of this part proceeds by induction on the derivation of $M \in \mathcal{SN}$.

1. Suppose $M = xM_1 \dots M_k$ with $M_1, \dots, M_k \in \mathcal{SN}$. We have $N \in \mathcal{SN}$ because $N \in \mathcal{SN}(\tau)$. This yields $MN = xM_1 \dots M_kN \in \mathcal{SN}$.

2. Suppose $M = \lambda x.P$ with $P \in \mathcal{SN}$. We have that $x \in V_\tau$ and $P \in \Lambda_\sigma$, so actually $P \in \mathcal{SN}(\sigma)$. For proving $(\lambda x.P)N \in \mathcal{SN}$, we need to prove $P[x := N] \in \mathcal{SN}$. This follows from an application of lemma 8.3.3.

3. Suppose $M = (\lambda x.M_0)M_1M_2 \dots M_k$ with $M_0[x := M_1]M_2 \dots M_k \in \mathcal{SN}$ and $M_1 \in \mathcal{SN}$. By induction hypothesis of the induction on the derivation of $M \in \mathcal{SN}$, we have $M_0[x := M_1]M_2 \dots M_kN \in \mathcal{SN}$. Moreover $M_1 \in \mathcal{SN}$.

   This yields $(\lambda x.M_0)M_1M_2 \dots M_kN \in \mathcal{SN}$.

τ is a composed type. The proof of this part proceeds as well by induction on the derivation of $M \in \mathcal{SN}$.

1. Suppose $M = xM_1 \ldots M_k$ with $M_1, \ldots, M_k \in \mathcal{SN}$. Since $N \in \mathcal{SN}$, we have that $MN \in \mathcal{SN}$.

2. Suppose $M = \lambda x.P$ with $P \in \mathcal{SN}$. For proving $(\lambda x.P)N \in \mathcal{SN}$, we need to prove that $P[x := N] \in \mathcal{SN}$. We have $\tau = \tau_1 \to \ldots \to \tau_n \to 0$. By the induction hypothesis of the induction on τ, we have $N \in \mathcal{SN}(\tau_1) \to \ldots \to \mathcal{SN}(\tau_n) \to \mathcal{SN}(0)$. Lemma 8.3.3 yields that $P[x := N] \in \mathcal{SN}$.

3. Suppose $M = (\lambda x.M_0)M_1M_2 \ldots M_k$ with $M_0[x := M_1]M_2 \ldots M_k \in \mathcal{SN}$ and $M_1 \in \mathcal{SN}$. By induction hypothesis of the induction on the derivation of $M \in \mathcal{SN}$, we have $M_0[x := M_1]M_2 \ldots M_kN \in \mathcal{SN}$. Moreover $M_1 \in \mathcal{SN}$. This yields $MN \in \mathcal{SN}$.

□

Corollary 8.3.5. $\mathcal{SN}(\tau \to \sigma) = \mathcal{SN}(\tau) \to \mathcal{SN}(\sigma)$.

Theorem 8.3.6. (Strong Normalisation for $\lambda_\to$)
For all $\tau \in Type$, if $M \in \Lambda_\tau$ then $M \in \mathcal{SN}(\tau)$.

Proof: The proof proceeds by induction on the derivation of $M \in \Lambda_\tau$.

1. Suppose $x \in \Lambda_\tau$ then $x \in \mathcal{SN}(\tau)$.

2. Suppose $M = \lambda x.P \in \Lambda_{\sigma\to\sigma'}$ with $x \in V_\sigma$ and $P \in \Lambda_{\sigma'}$. By induction hypothesis, we have $P \in \mathcal{SN}(\sigma')$. This yields $(\lambda x.P) \in \mathcal{SN}(\sigma \to \sigma')$.

3. Suppose $M = PQ \in \Lambda_\tau$. Then $P \in \Lambda_{\sigma\to\tau}$ and $Q \in \Lambda_\sigma$. By induction hypothesis, $P \in \mathcal{SN}(\sigma \to \tau)$ and $Q \in \mathcal{SN}(\sigma)$.

   By the previous theorem we have $\mathcal{SN}(\sigma \to \tau) = \mathcal{SN}(\sigma) \to \mathcal{SN}(\tau)$.

   Therefore $PQ \in \mathcal{SN}(\tau)$.

□

8.4 Conclusions and Related Work

An interesting proof of the normalisation of the simply typed lambda calculus is the one given by Tait in [Tai67]. Tait defined the class of *computable terms* (or *reducible terms*). Using Tait's method one can also prove strong normalisation for the simply typed lambda calculus and some of its extensions, like Gödel's T [Tro73]. Girard [Gir72] introduced the concept of *candidate of reducibility* to generalise Tait's method to include polymorphism

(for the systems F to F^ω). For an explanation of the method and applications to prove confluence see [Gal90].

The conditions in the definition of candidate of reducibility in [Gir72] were modified in [Tai75] and [Mit86] and a new definition was introduced: a saturated set. The definitions of the set $\mathcal{SN}$ and of saturated sets are very similar (see chapter 5).

Our proof differs from the proof by Tait in the fact that in his method an interpretation for types is used. A type τ is interpreted as a set of λ-terms and denoted by $[\![\tau]\!]$. Then, the interpretation of a type $\tau \to \sigma$ is defined to be $[\![\tau]\!] \to [\![\sigma]\!]$. So $[\![\tau \to \sigma]\!] = [\![\tau]\!] \to [\![\sigma]\!]$ by definition. In our proof the equality $\mathcal{SN}(\tau \to \sigma) = \mathcal{SN}(\tau) \to \mathcal{SN}(\sigma)$ needs to be proved. On the other hand, in Tait's method one has to prove that $[\![\tau]\!] \to [\![\sigma]\!]$ is a subset of the set of strongly normalising terms. In our proof, the set $\mathcal{SN}(\tau \to \sigma)$ is a subset of the set of strongly normalising terms by definition.

In [Vri87] strong normalisation for the simply typed lambda calculus is proved by giving a function that computes the length of the maximal rewrite sequence and by implictly following the strategy F_∞ of maximal length. In our proof we make use of the set $\mathcal{SN}$ which also implicitly uses this strategy.

It seems that the method used here cannot be extended to Gödel' s T nor to the combinatory version of $\lambda_\to$. Our method does not extend because it fails in theorem 8.3.4 which is proved by induction on the structure of the type. The type of the newly created redexes may be more complex and thus it is not possible to apply the induction hypothesis. This seems related to the proof of normalisation for the simply typed lambda calculus by Turing [Gan80]. In our opinion, this method, being too simple, will not be easily extensible.

It is possible to express the proof of strong normalisation of the simply typed lambda calculus in Peano Arithmetic (PA). On the other hand, strong normalisation for Gödel's T cannot be proved in PA and the system F cannot be proved in PA_2.

A topic for further investigation is to implement our proof in a proof checker like Coq, Lego or Alf.

Part III

Pure Type Systems with Definitions

Chapter 9

Pure Type Systems

9.1 Introduction

Pure type systems provide a way to describe a large class of type systems à la Church in a uniform way. They were introduced independently by S. Berardi [Ber88] (see also [Ber90]) and J. Terlouw [Ter89]. Important pure type systems are the systems of the λ-cube [Bar92]. They are called 'pure' because there is only one type constructor and only one reduction rule, namely the type constructor Π and the β-reduction.

This chapter is organised as follows. In section 9.2 we recall the notion of specification and of morphism between specifications. In section 9.3 we recall the definition of pure type systems.

9.2 Specifications

In this section we define the notion of specification. The specifications are 'the parameters' in the definition of pure type systems.

Definition 9.2.1. A *specification* is a triple $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ such that

1. $\boldsymbol{S}$ is a set of symbols called *sorts*,
2. $\boldsymbol{A} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ called set of *axioms*,
3. $\boldsymbol{R} \subseteq \boldsymbol{S} \times \boldsymbol{S} \times \boldsymbol{S}$ called set of *rules*.

Sorts are denoted by $s, s', \ldots, s_1, s_2, \ldots$.

The set of axioms and the set of rules are used in the typing rules of pure type systems (see section 9.3). The set $\boldsymbol{A}$ determines the axioms and the set $\boldsymbol{R}$ determines all 'the functions' we can form in the system.

Example 9.2.2. We give two examples of specifications. These specifications will make sense in section 9.3 after introducing the typing relation.

1. The specification $PRED$ is defined as follows.

$$PRED \quad \begin{array}{ll} \boldsymbol{S} & \{*^s, *^p, *^f, \Box^s, \Box^p\} \\ \boldsymbol{A} & \{(*^s, \Box^s), (*^p, \Box^p)\} \\ \boldsymbol{R} & \{(*^p, *^p, *^p), (*^s, *^p, *^p), (*^s, \Box^p, \Box^p), (*^f, *^s, *^f), (*^s, *^f, *^f)\} \end{array}$$

   The sort $*^p$ is for propositions, the sort $*^s$ is for sets and $*^f$ is for first order functions between the sets in $*^s$. The rule $(*^p, *^p, *^p)$ allows the formation of implication between propositions, the rule $(*^s, *^p, *^p)$ allows quantification over sets, the rule $(*^s, \Box^p, \Box^p)$ allows the formation of first order predicates, the rule $(*^f, *^s, *^f)$ allows the formation of function spaces between the basic set $*^s$ and the rule $(*^s, *^f, *^f)$ allows the formation of curried functions of several arguments in the basic set.

2. The specification P is defined as follows.

$$P \quad \begin{array}{ll} \boldsymbol{S} & \{*, \Box\} \\ \boldsymbol{A} & \{(*, \Box)\} \\ \boldsymbol{R} & \{(*, *, *), (*, \Box, \Box)\} \end{array}$$

   The sort $*$ is used for types and the sort $\Box$ is used for kinds. The rule $(*, *, *)$ allows the formation of types and $(*, \Box, \Box)$ allows the formation of kinds.

Definition 9.2.3. We define *a morphism* from the specification $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ to $S' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ as a function $f : \boldsymbol{S} \to \boldsymbol{S}'$ that satisfies the following conditions.

1. If $(s_1, s_2) \in \boldsymbol{A}$ then $(f(s_1), f(s_2)) \in \boldsymbol{A}'$.

2. If $(s_1, s_2, s_3) \in \boldsymbol{R}$ then $(f(s_1), f(s_2), f(s_3)) \in \boldsymbol{R}'$.

We denote the category whose objects are the specifications and morphisms the ones defined above by **Spec**.

Several examples of morphisms between specifications are given in [Geu93] and [Bar92].

Example 9.2.4. An important example of morphism between specifications is the following one used for the propositions-as-types interpretation.

We define $p : \{*^s, *^p, *^f, \Box^s, \Box^p\} \to \{*, \Box\}$ as follows.

$$\begin{array}{lclclcl} p(*^s) & = & * & p(\Box^s) & = & \Box \\ p(*^p) & = & * & p(\Box^p) & = & \Box \\ p(*^f) & = & * & & & \end{array}$$

It is easy to see that p is a morphism from $PRED$ to P.

Definition 9.2.5. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. A sort s in $\boldsymbol{S}$ is called *a topsort* if there is no $s_0 \in \boldsymbol{S}$ such that $(s, s_0) \in \boldsymbol{A}$.

Definition 9.2.6. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The specification S is called *singly sorted* if

1. $(s_1, s_2), (s_1, s_3) \in \boldsymbol{A}$ implies $s_2 = s_3$
2. $(s_1, s_2, s_3), (s_1, s_2, s_4) \in \boldsymbol{R}$ implies $s_3 = s_4$.

Definition 9.2.7. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification.

The specification $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is called *full* if $\boldsymbol{R} = \{(s_1, s_2, s_2) \mid s_1, s_2 \in \boldsymbol{S}\}$.

The specification $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is called *semi-full* if for all $(s_1, s_2, s_3) \in \boldsymbol{R}$ and $s_2' \in \boldsymbol{S}$ there exists $s_3' \in \boldsymbol{S}$ such that $(s_1, s_2', s_3') \in \boldsymbol{R}$.

Definition 9.2.8. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification.

The specification S is called *logical* if it verifies the following conditions.

1. The set $\boldsymbol{S}$ contains two distinguished sorts $*$ and $\Box$.
2. The set $\boldsymbol{A}$ contains the axiom $(*, \Box)$.
3. The set $\boldsymbol{R}$ contains the rule $(*, *, *)$.
4. There is no sort s such that $(s, *) \in \boldsymbol{A}$.

The specification S is called *non-dependent* if it is logical and the only rules concerning $*$ are of the form $(s, *, *)$ for some sort s.

The specification S is called *impredicative* if it is logical and $(\Box, *, *) \in \boldsymbol{R}$.

9.3 Pure Type Systems

We present the notion of pure type systems in a slightly different way than usual. We first define a functor λ from the category of specifications to the category of abstract rewriting systems with typing. We think that the presentation of pure type systems is more neat in this way.

A pure type system is a value $\lambda(S)$ of λ given by a 4-tuple:

1. a set $\mathcal{T}$ of pseudoterms,
2. a set $\mathcal{C}$ of pseudocontexts,
3. a reduction relation on pseudoterms, called β-reduction,
4. a typing relation denoted by $\vdash$.

First we define the components of this 4-tuple. Pseudoterms are expressions formed with a constructor λ for the abstractions, the brackets ( ) for the application and the constructor Π for the product (function space).

Definition 9.3.1. The set $\mathcal{T}_S$ (or $\mathcal{T}$ for short) of *pseudoterms* is defined as follows.

$$\mathcal{T} ::= V \mid \boldsymbol{S} \mid (\mathcal{T}\ \mathcal{T}) \mid (\lambda V{:}\mathcal{T}.\ \mathcal{T}) \mid (\Pi V{:}\mathcal{T}.\ \mathcal{T})$$

where V is a set of variables and $\boldsymbol{S}$ is the set of sorts.

Variables will be denoted as $x, y, z, \ldots, \alpha, \beta, \gamma \ldots$. Pseudoterms will be denoted as $a, b, c, d, \ldots, A, B, C, \ldots$. The usual parenthesis conventions for abstraction, application and product will be used (see [Bar92]).

Definition 9.3.2. The mapping $FV : \mathcal{T} \to \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(c) & = & \emptyset \\ FV(a\ b) & = & FV(a) \cup FV(b) \\ FV(\lambda x{:}A.\ a) & = & FV(A) \cup (FV(a) - \{x\}) \\ FV(\Pi x{:}A.\ a) & = & FV(A) \cup (FV(a) - \{x\}) \end{array}$$

We say that x is *free* in a if $x \in FV(a)$.

Definition 9.3.3. The mapping $BV : \mathcal{T} \to \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} BV(x) & = & \emptyset \\ BV(c) & = & \emptyset \\ BV(a\ b) & = & BV(a) \cup BV(b) \\ BV(\lambda x{:}A.\ a) & = & BV(A) \cup (BV(a) \cup \{x\}) \\ BV(\Pi x{:}A.\ a) & = & BV(A) \cup (BV(a) \cup \{x\}) \end{array}$$

We say that x is *bound* in a if $x \in BV(a)$.

Definition 9.3.4. The result of *substituting* d for (the free occurrences of) x in e is denoted as $e[x := d]$ and is defined as follows.

$$\begin{array}{rcll} s[x := d] & = & s & \\ x[x := d] & = & d & \\ y[x := d] & = & y & \text{if } x \neq y \\ (\lambda x{:}A.\ a)[x := d] & = & (\lambda x{:}A.\ a) & \\ (\lambda y{:}A.\ a)[x := d] & = & (\lambda y{:}A[x := d].\ a[x := d]) & \text{if } x \neq y \text{ and } y \notin FV(d) \\ (\lambda y{:}A.\ a)[x := d] & = & (\lambda z{:}A[x := d].\ a[y := z][x := d]) & \text{if } x \neq y,\ y \in FV(d) \text{ and } z \text{ is fresh} \\ (a\ \ b)[x := d] & = & (a[x := d]\ \ b[x := d]) & \\ (\Pi x{:}A.\ a)[x := d] & = & (\Pi x{:}A.\ a) & \\ (\Pi y{:}A.\ a)[x := d] & = & (\Pi y{:}A[x := d].\ a[x := d]) & \text{if } x \neq y \text{ and } y \notin FV(d) \\ (\Pi y{:}A.\ a)[x := d] & = & (\Pi z{:}A[x := d].\ a[y := z][x := d]) & \text{if } x \neq y,\ y \in FV(d) \text{ and } z \text{ is fresh} \end{array}$$

The set of pseudoterms with holes in it is defined by the following grammar.

Definition 9.3.5. The set $\mathcal{H}$ is defined as follows.

$$\mathcal{H} ::= [\ ] \mid V \mid \boldsymbol{S} \mid (\mathcal{H}\ \mathcal{H}) \mid (\lambda V{:}\mathcal{H}.\ \mathcal{H}) \mid (\Pi V{:}\mathcal{H}.\ \mathcal{H})$$

where V is the set of variables and $\boldsymbol{S}$ is the set of sorts.

An element in $\mathcal{H}$ is denoted by $C[\ ]$.

Pseudocontexts are lists of pairs consisting of one variable and one pseudoterm. The pseudocontexts are used in the definition of the typing relation in order to assign types to the variables.

Definition 9.3.6. The set $\mathcal{C}_S$ (or $\mathcal{C}$ for short) of *pseudocontexts* is defined as follows.

$$\mathcal{C} ::= \epsilon \mid < \mathcal{C}, V{:}\mathcal{T} >$$

Pseudocontexts will be denoted as $?, ?', \ldots, \Delta, \Delta', \ldots$.

The expression $?, x{:}A$ stands for $< ?, x{:}A >$.

Next we define a mapping Dom that gives the set of variables declared in a pseudocontext.

Definition 9.3.7. The mapping $Dom : \mathcal{C} \to \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} Dom(\epsilon) & = & \emptyset \\ Dom(?, x{:}A) & = & Dom(?) \cup \{x\} \end{array}$$

Definition 9.3.8. The result of *substituting* d for (the free occurrences of) a variable x in $?$ such that $x \notin Dom(?)$ is denoted as $?[x := d]$ and is defined as follows.

$$\begin{array}{rl} \epsilon[x := d] & = \epsilon \\ < ?, y{:}A > [x := d] & =< ?[x := d], y{:}A[x := d] > \end{array}$$

Definition 9.3.9. Let $d \in \mathcal{T}$. A *change of a bound variable* in the term d is the replacement of a subterm $(\lambda x{:}A.\ b)$ or $(\Pi x{:}A.\ b)$ by $(\lambda y{:}A.\ b[x := y])$ or $(\Pi y{:}A.\ b[x := y])$, respectively, where $y \notin FV(b)$.

Definition 9.3.10. The pseudoterm b is *α-convertible* to b' if b' is the result of applying to b a series of changes of variables or vice versa.

Convention 9.3.11. Two terms are identified if they are α-convertible.

Definition 9.3.12. The *β-reduction* is defined by the following rule:

$$C[(\lambda x{:}A.\ b)a] \to_\beta C[b[x := a]]$$

where $C[\ ] \in \mathcal{H}$ has only one occurrence of $[\ ]$.

Definition 9.3.13. The *typing relation* $\vdash_S$ (or $\vdash$ for short) is the smallest relation on $\mathcal{C} \times \mathcal{T} \times \mathcal{T}$ closed under the following rules.

$$
\begin{array}{lcl}
(axiom) & \epsilon \vdash s_1 : s_2 & \text{for } (s_1, s_2) \in \boldsymbol{A} \\
(start) & \dfrac{? \vdash A : s}{?, x : A \vdash x : A} & \text{where } x \text{ is } ?\text{-fresh} \\
(weakening) & \dfrac{? \vdash b : B \quad ? \vdash A : s}{?, x : A \vdash b : B} & \text{where } x \text{ is } ?\text{-fresh} \\
(formation) & \dfrac{? \vdash A : s_1 \quad ?, x : A \vdash B : s_2}{? \vdash (\Pi x{:}A.B) : s_3} & \text{for } (s_1, s_2, s_3) \in \boldsymbol{R} \\
(abstraction) & \dfrac{?, x : A \vdash b : B \quad ? \vdash (\Pi x{:}A.\ B) : s}{? \vdash (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)} & \\
(application) & \dfrac{? \vdash b : (\Pi x{:}A.\ B) \quad ? \vdash a : A}{? \vdash (b\ \ a) : B[x := a]} & \\
(conversion) & \dfrac{? \vdash b : B \quad ? \vdash B' : s \quad B \twoheadleftrightarrow_\beta B'}{? \vdash b : B'} &
\end{array}
$$

where s ranges over sorts, i.e. $s \in \boldsymbol{S}$.

Explanation of the typing rules. Variables are typable by means of the start rule, abstractions by means of the abstraction rule, applications by means of the application rule and the product by means of the formation rule.

The start and the weakening rules allows to enlarge the context. They ensure that all the components of a pseudocontext are typable. Besides they do not allow the repetition of variables in the context since a variable is added to the context only if it is fresh. One consequence of this is that we cannot have two nested abstractions with the same bound variable.

Note that the typing rules have two 'parameters'. One parameter is the set $\boldsymbol{A}$ in the first rule which determines the set of axioms we have in the system and the other is the set $\boldsymbol{R}$ in the formation rule which determines the products and hence the abstractions we can form in the system. When we fix the triple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$, we obtain a particular typing system.

The conversion rule ensures that types are 'closed under reduction and expansion'.

Definition 9.3.14. The functor $\lambda : \mathbf{Spec} \to \mathbf{Carst}$ is defined as follows.

$$
\begin{array}{lll}
\lambda(S) & = (\mathcal{T}, \mathcal{C}, \to_\beta, \vdash) & S \in \mathbf{Spec} \\
\lambda(f) & = (\hat{f}_1, \hat{f}_0) & f : \boldsymbol{S} \to \boldsymbol{S}' \in \mathbf{Spec}
\end{array}
$$

The functions $\hat{f}_0$ and $\hat{f}_1$ are the extension of f to the set of pseudoterms and pseudocontexts respectively.

$$\begin{array}{lcl}
\hat{f}_0(s) & = & f(s) \\
\hat{f}_0(\lambda x{:}A.\ b) & = & \lambda x{:}\hat{f}_0(A).\ \hat{f}_0(b) \\
\hat{f}_0(\Pi x{:}A.\ B) & = & \Pi x{:}\hat{f}_0(A).\ \hat{f}_0(B) \\
\hat{f}_0(F\ a) & = & (\hat{f}_0(F)\ \hat{f}_0(a))
\end{array}$$

$$\begin{array}{lcl}
\hat{f}_1(\epsilon) & = & \epsilon \\
\hat{f}_1(?\,, x{:}A) & = & \hat{f}_1(?\,), x{:}\hat{f}_0(A)
\end{array}$$

There are several things that we need to verify. Amongst the properties of pure type systems below, we list subject and type reduction. This means that $(\mathcal{T}, \mathcal{C}, \rightarrow_\beta, \vdash) \in$ **Carst**. Moreover the function $\lambda(f)$ is a morphism in **Carst** since it preserves the rewrite and the typing relations.

Definition 9.3.15. (Pure Type Systems)

- A *pure type system* (PTS) is an element of $\lambda(\mathbf{Spec}) = \{\lambda(S) \mid S \in \mathbf{Spec}\}$.
- A *singly sorted pure type system* is an element of $\{\lambda(S) \mid S \in \mathbf{Spec}\ \&\ S$ is singly sorted$\}$.

For example, $\lambda(PRED)$ and $\lambda(P)$ are pure type systems and $\lambda(p)$ is a morphism from $\lambda(PRED)$ to $\lambda(P)$.

The λ-cube consists of eight systems [Bar92] defined by the same set of sorts and the same set of axioms. They differ in the set of rules $\boldsymbol{R}$.

Definition 9.3.16. Let $\boldsymbol{S}_0 = \{*, \Box\}$ and $\boldsymbol{A}_0 = \{(*, \Box)\}$.

We will write $\boldsymbol{S}_0 \otimes \boldsymbol{S}_0$ for $\{(s_1, s_2, s_3) \mid s_2 = s_3\ \ \&\ \ s_1, s_2, s_3 \in \boldsymbol{S}_0\}$.

The λ-cube is a mapping from $\mathcal{P}(\boldsymbol{S}_0 \otimes \boldsymbol{S}_0)$ into **Carst** defined as follows.

System	$\boldsymbol{R}$			
$\lambda_{\rightarrow}$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
λP	$(*,*)$		$(*,\Box)$	
$\lambda P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda P\omega = \lambda C$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

The systems of the λ-cube correspond to some known systems with some variations: $\lambda_{\rightarrow}$ is the simply typed lambda calculus [Chu40], $\lambda 2$ is the second order typed lambda calculus [Gir72] and [Rey74], λP is AUT-QE and LF [Bru70] and [RHP87]. The system $\lambda\omega$ is POLYREC and $\lambda\omega$ is $F\omega$ [Gir72]. The last element in the table, λC, corresponds to the Calculus of Constructions.

All the systems in the λ-cube have only one topsort, namely $\Box$.

Definition 9.1. The system of higher order logic can be described by the following specification (see [Geu93]).

$$HOL \quad \begin{array}{|ll|} \hline \boldsymbol{S} & \{*, \Box, \triangle\} \\ \boldsymbol{A} & \{(*, \Box), (\Box, \triangle)\} \\ \boldsymbol{R} & \{(*, *), (\Box, *), (\Box, \Box)\} \\ \hline \end{array}$$

There is only one topsort in $\lambda(HOL)$ and that is $\triangle$.

Definition 9.2. The Calculus of Constructions extended with an infinite type hierarchy can be described by the following specification.

$$C_\infty \quad \begin{array}{|ll|} \hline \boldsymbol{S} & \mathbb{N} \\ \boldsymbol{A} & \{(n, n+1) \mid n \in \mathbb{N}\} \\ \boldsymbol{R} & \{(m, 0, 0) \mid m \in \mathbb{N}\} \cup \\ & \{(m, n, r) \mid \ m, n \in \mathbb{N} \ \& \ max(m, n) \leq r\} \\ \hline \end{array}$$

The system λC_∞ extended with strong Σ-types and cumulativity is the system **ECC** (see [Luo89]). We can see that λC_∞ is an extension of λC and of $\lambda(HOL)$ writing $*$ instead of 0, $\Box$ instead of 1 and $\triangle$ instead of 3.
Note that there is no topsort in λC_∞.

Definition 9.3.17. Let S be a logical specification. The pure type system $\lambda(S)$ is said to be *inconsistent* if for all A, there exists a such that a has type A in the context $A : *$.

For example, an inconsistent pure type system is $\lambda*$ determined by the specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ where $\boldsymbol{S} = \{*\}$, $\boldsymbol{A} = \{(*, *)\}$ and $\boldsymbol{R} = \{(*, *)\}$ (see [Gir72] and [Bar92]).

Properties of Pure Type Systems

The advantage of pure type systems is that we can do the metatheory of several type systems at the same time. There are several properties that can be proved for all these systems.

Theorem 9.3.18.

1. $(\mathcal{T}, \twoheadrightarrow_\beta)$ is confluent
2. Correctness of types, i.e. if $? \vdash d : D$ then there exists a sort s such that $? \vdash D : s$ or $D = s$ for all $? \in \mathcal{C}, d, D \in \mathcal{T}$.
3. Subject reduction, i.e. if $? \vdash d : D.$ and $d \rightarrow_\beta d'$ then $? \vdash d' : D$ for all $d, d', D \in \mathcal{T}$ and $? \in \mathcal{C}$.
4. Type reduction, i.e. if $? \vdash d : D$ and $D \rightarrow_\beta D'$ then $? \vdash d : D'$ for all $d, D, D' \in \mathcal{T}$ and $? \in \mathcal{C}$.
5. Strengthening, i.e. if $?_1, x{:}A, ?_2 \vdash b : B$ and $x \notin FV(?_2) \cup FV(b) \cup FV(B)$ then $?_1, ?_2 \vdash b : B$.
6. Let S be a singly sorted specification. Then $\lambda(S)$ verifies uniqueness of types.
7. Let S be a logical specification. If $\lambda(S)$ is inconsistent then $\lambda(S)$ is not normalising.

The proofs of these properties can be found in [GN91] and [Bar92] except for strengthening that is proved in [BJ93].

Theorem 9.3. (Strong Normalisation)

1. The system λC_∞ is β-strongly normalising.
2. The systems of the λ-cube are β-strongly normalising.
3. The system $\lambda(HOL)$ is β-strongly normalising.

Proof: The system **ECC** is strongly normalising (see [Luo89]) and contains λC_∞, the systems of the λ-cube and $\lambda(HOL)$. $\square$

There are pure type systems that are not normalising, e.g. the system $\lambda*$ which is inconsistent (see theorem 9.3.18 part 7).

Theorem 9.3.19. (Decidability of Type Inference and Type Checking)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

1. If $\boldsymbol{S}$ is finite then type inference and type checking in $\lambda(S)$ are decidable.
2. Suppose that S is singly sorted or semi-full and that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursive. Then type inference and type checking in $\lambda(S)$ are decidable.

The proof of the first part can be found in [BJ93] and the second one in [BJMP93] and [Pol96].

Theorem 9.3.20. (Undecidability of Type Inference and Type Checking)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a singly sorted, non-dependent and impredicative specification. If $\lambda(S)$ is inconsistent then type inference and type checking in $\lambda(S)$ are undecidable.

This is proved in [CH94] as a generalisation of the result of [MR86].

The problem of inhabitation in $\lambda_{\rightarrow}$ and in $\lambda\underline{\omega}$ are decidable but in the rest of the systems of the cube is undecidable [Spr95]. For the inconsistent systems, inhabitation is trivial.

9.4 Conclusions and Related Work

As we said before, pure type systems were introduced independently by S. Berardi [Ber88] and J. Terlouw [Ter89]. In the typing rules for pure type systems, the axiom depends on the set $\boldsymbol{A}$ and the product rule depends on the set $\boldsymbol{R}$. In other words, we have a family of axioms and product rules depending on $\boldsymbol{A}$ and $\boldsymbol{R}$. The typing rules are parametric and the parameters are the specifications, i.e. triples of the form $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. The dependency on the specification is expressed with the notation $\lambda(S)$ for the pure type system corresponding to the specification $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. We made a 'slight change' in the definition of pure type systems. We have written the functor $\lambda : \mathbf{Spec} \rightarrow \mathbf{Carst}$ instead of 'giving a set of values $\lambda(S)$'. We could think that now we have given 'a house' (or perhaps 'a type') for λ to live in and this house is the category of functors from **Spec** to **Carst**.

Extensions of pure type systems are also defined as functors from the category of specifications **Spec** into the category of environmental abstract rewriting systems with typing **Carst**. The category $\mathbf{Carst}^{\mathbf{Spec}}$ of functors from **Spec** to **Carst** is an adequate 'place' to put the pure type systems and all their extensions. Comparison of pure type systems and their extensions is made via morphisms in the category of functors from **Spec** to **Carst**, i.e. via natural transformations.

Chapter 10

Type Inference for Pure Type Systems

10.1 Introduction

In this chapter we define a partial function that infers the type of a term in a singly sorted pure type system. If the term has type in a singly sorted pure type system then this function terminates and yields the type of the term (up to β-conversion). This function can be constructed if we have a set of typing rules that is syntax directed. A set of rules is called *syntax directed* if the last rule in the derivation of a term is determined by the structure of the term and of the context. The rules of pure type systems are not syntax directed since the weakening and the conversion rules can be applied at any point in the derivation. In order to make this set of rules syntax directed, we should remove the non-structural rules (like the weakening and the conversion ones) and keep only the structural rules (the ones for term constructors). The system obtained by eliminating the non-structural rules should be equivalent to the original one. The proof of the equivalence (soundness and completeness) between the original rules and the corresponding syntax directed ones is problematic for some pure type systems (see [BJMP93] and also [Pol93a]). Here we present a *syntax directed set of rules for singly sorted pure type systems* similar to the one presented in [BJMP93]. As in [BJMP93], we use an auxiliary system to check for the Π-condition, i.e. the premise that the product $(\Pi x{:}A.\ B)$ should be well-typed in the abstraction rule. The auxiliary system we use to define the syntax directed set of rules is the corresponding pure type system without the Π-condition. The convenience of this system is that it permits us to prove soundness and completeness of the original rules for pure type systems with respect to the syntax directed ones.

This chapter is organised as follows. In section 10.2, we define a functor λ^w from the category of specifications to the category of contextual abstract rewriting systems with weak typing. We define the class of pure type systems without the Π-condition as the image of λ^w. A pure type system without the Π-condition can be considered as an extension of

the corresponding system with the Π-condition. The new terms of the extension are the abstractions $(\lambda x{:}A.\ b)$ whose type $(\Pi x{:}A.\ B)$ is not typable and applications of the form $(F\ a)$ whose operator F has a type that is not typable. Hence we are led to consider types that are not typable. We call them *toptypes*. In section 10.3, we prove the basic properties of pure type systems without the Π-condition, e.g. weak subject reduction theorem. In section 10.4, we analyse the shape of a toptype. We give a characterisation of the set of toptypes and prove weak type reduction. In the case of singly sorted specifications we give another characterisation of the set of toptypes and prove that it is closed under substitution and β-reduction. In section 10.5, we consider those β-redexes whose abstraction has a type that is not typable and we call them *illegal redexes*. We introduce a mapping φ that contracts all the illegal redexes of a term. Using φ we prove that weak normalisation is preserved by the extension from $\lambda(S)$ to $\lambda^\omega(S)$ in the case that S is singly sorted. Moreover φ can be used to define a morphism from $\lambda^\omega(S)$ to $\lambda(S)$ for each singly sorted specification S. In section 10.6, we define the notion of β_ι-reduction as the contraction of the illegal redexes. We prove that all the terms typable in a singly sorted pure type system are β_ι-strongly normalising. In section 10.7, we define a syntax directed set of rules for singly sorted pure type systems. In section 10.8, we define a partial function **type** that computes the type of a term in a singly sorted pure type system based upon the syntax directed set of rules defined in the previous section.

10.2 Pure Type Systems without the Π-condition

In this section we define the notion of pure type systems without the Π-condition. For pure type systems the type of an abstraction $(\lambda x{:}A.\ b)$ is a dependent product $(\Pi x{:}A.\ B)$ and the abstraction rule has the premise that the product $(\Pi x{:}A.\ B)$ should be well-typed. This premise is called the Π-condition. The problem with the Π-condition is that sometimes it is not well-adapted to induction. The idea is to remove this condition in order to define a syntax directed set of rules and prove the equivalence of the original rules with the syntax directed ones.

Definition 10.2.1. The Π-*condition* is the premise $? \vdash (\Pi x{:}A.\ B) : s$ of the abstraction rule in a pure type system.

Definition 10.2.2. We define a functor $\lambda^\omega : \mathbf{Spec} \to \mathbf{Carst}_\omega$ such that for $S \in \mathbf{Spec}$ we have that

$$\lambda^\omega(S) = (\mathcal{T}, \mathcal{C}, \to_\beta, \vdash^\omega).$$

We define the components of this 4-tuple as follows. The sets $\mathcal{T}$, $\mathcal{C}$ and the relation $\to_\beta$ are defined as in definition 9.3.14.

The typing relation $\vdash^\omega$ (or $\vdash^\omega_S$) is the smallest relation defined by the same rules as in definition 9.3.14 except for the abstraction rule that is replaced by the following one.

$$(abstraction)\frac{?\,, x : A \vdash^\omega b : B}{? \vdash^\omega (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

The functor $\lambda^\omega : \mathbf{Spec} \to \mathbf{Carst}_\omega$ is defined for morphisms in the obvious way.

There are two things that remain to be verified, i.e. $\lambda^\omega(S) \in \mathbf{Carst}_\omega$ and $\lambda^\omega(f) \in \mathbf{Carst}_\omega$. For $\lambda^\omega(S) \in \mathbf{Carst}_\omega$, we have to prove that a pure type system without the Π-condition verifies weak subject and weak type reduction. This will be proved in the next sections. For $\lambda^\omega(f) \in \mathbf{Carst}_\omega$, we have to prove that $\lambda^\omega(f)$ preserves the rewrite and the typing relation. This is very easy to prove.

Definition 10.2.3. (Pure Type Systems without the Π-condition)

A pure type system without the Π-condition is an element of the set

$$\lambda^\omega(\mathbf{Spec}) = \{\lambda^\omega(S) \mid S \in \mathbf{Spec}\}.$$

A singly sorted pure type system without the Π-condition is an element of the set

$$\{\lambda^\omega(S) \mid S \in \mathbf{Spec} \ \& \ S \text{ is singly sorted}\}.$$

Note that if a term is typable in $\lambda(S)$ then it is typable in $\lambda^\omega(S)$. Therefore $\lambda^\omega(S)$ is an extension of $\lambda(S)$.

The converse is not true. There may be terms typable in $\lambda^\omega(S)$ which are not typable in $\lambda(S)$. For example, the type $(A \to \Box)$ is not typable in the context $A : *$ either in $\lambda(C)$ or in $\lambda^\omega(C)$. However in $\lambda^\omega(C)$ we can derive that the term $\lambda x{:}A.(* \to *)$ has type $(A \to \Box)$ in the context $A : *$.

10.3 Basic Properties

In this section we prove the generation lemma, the thinning lemma and the weak subject reduction theorem for pure type systems without the Π-condition. Since correctness of types does not hold, we prove some weaker lemmas that we call correctness of contexts and correctness of types for variables. Finally we prove uniqueness of types for singly sorted specifications.

Lemma 10.3.1. (Generation Lemma)

1. If $? \vdash^\omega s : D$ then there exists s' such that $(s, s') \in \boldsymbol{A}$ and $D \twoheadleftarrow\twoheadrightarrow_\beta s'$.

2. If $? \vdash^\omega x : D$ then there are s, b and B such that $? \vdash^\omega B : s$, $x : B \in ?$ and $B \twoheadleftarrow\twoheadrightarrow_\beta D$.

3. If $? \vdash^\omega (\Pi x{:}A.B) : D$ then there is a rule $(s_1, s_2, s_3) \in \boldsymbol{R}$ such that $? \vdash^\omega A : s_1$, $?, x{:}A \vdash^\omega B : s_2$ and $D \twoheadleftarrow\twoheadrightarrow_\beta s_3$.

4. If $? \vdash^\omega (\lambda x : A.\ b) : D$ then there are s and B such that $?, x : A \vdash^\omega b : B$ and $D \twoheadleftarrow\twoheadrightarrow_\beta (\Pi x{:}A.\ B)$.

5. If $? \vdash^\omega (b\ \ a) : D$ then there are A, B such that $? \vdash^\omega b : (\Pi x{:}A.\ B)$, $? \vdash^\omega a : A$ and $D \twoheadleftarrow\twoheadrightarrow_\beta B[x := a]$.

All parts of the generation lemma are proved by induction on the derivation of a term.

Lemma 10.3.2. (Correctness of Contexts) If $?, x{:}A, ?' \vdash^{\omega} b : B$ then there exists an s such that $? \vdash^{\omega} A : s$.

This lemma is proved by induction on the derivation of $?, x{:}A, ?' \vdash^{\omega} b : B$.

Lemma 10.3.3. (Correctness of Types for Variables) If $? \vdash^{\omega} x : A$ then there exists s such that $? \vdash^{\omega} A : s$.

This lemma is proved by induction on the derivation of $? \vdash^{\omega} x : A$.

Lemma 10.3.4. Let $? \vdash^{\omega} (\lambda x{:}A.\ b) : D$. Then there exists a B such that $?, x : A \vdash^{\omega} b : B$. Moreover either $D = (\Pi x{:}A.\ B)$ or $? \vdash D : s$ and $D \twoheadleftarrow\twoheadrightarrow_{\beta} (\Pi x{:}A.\ B)$ for some sort s.

This lemma is proved by induction on the derivation of $? \vdash^{\omega} (\lambda x{:}A.\ b) : D$.

Lemma 10.3.5. If $? \vdash^{\omega} A : s$ and $? \vdash a : A$ then $? \vdash A : s'$ for some sort s'.

Proof: By the lemma of correctness of types for pure type systems, we have that $? \vdash A : s''$ or $A = s''$ for some s''. In case $A = s''$, we have that (s'', s) is an axiom. □

Lemma 10.3.6. (Thinning Lemma) Let $?' \vdash^{\omega} b : B$.

$$\text{If } ? \vdash^{\omega} a : A \text{ and } ? \subseteq ?' \text{ then } ?' \vdash^{\omega} a : A.$$

This lemma is proved by induction on the derivation of $? \vdash^{\omega} a : A$.

Lemma 10.3.7. (Substitution Lemma) If $? \vdash^{\omega} a : A$ and $?, x{:}A, ?' \vdash^{\omega} b : B$ then $?, ?'[x := a] \vdash^{\omega} b[x := a] : B[x := a]$.

This lemma is proved by induction on the derivation of $?, x{:}A, ?' \vdash^{\omega} b : B$.

Lemma 10.3.8. (Correctness of Domains) If $? \vdash^{\omega} F : \Pi x{:}A.\ B$ then $? \vdash^{\omega} A : s$ for some sort s.

This lemma is proved by induction on the derivation of $? \vdash^{\omega} F : \Pi x{:}A.\ B$.

Theorem 10.3.9. (Weak Subject Reduction Theorem)
Let $? \vdash^{\omega} e : E$.

a) If $e \rightarrow_{\beta} e'$ then there exists E' such that $E \twoheadrightarrow_{\beta} E'$ and $? \vdash^{\omega} e' : E'$.

b) If $? \rightarrow_{\beta} ?'$ then $?' \vdash^{\omega} e : E$.

Proof: The two statements are proved by simultaneous induction on the derivation of $? \vdash^{\omega} e : E$. We only consider the statement a) for the case of the application rule.

$$\text{(application)}\ \frac{?\ \vdash^\omega b : (\Pi x{:}A.\ B) \quad ?\ \vdash^\omega a : A}{?\ \vdash^\omega (b\ \ a) : B[x := a]}$$

There are several cases when we reduce the application $(b\ \ a)$.

1. Suppose $(b\ \ a) \to_\beta (b'\ \ a)$ with $b \to_\beta b'$. By induction we have that $?\ \vdash^\omega b' : (\Pi x{:}A'.\ B')$ for $A \twoheadrightarrow_\beta A'$ and $B \twoheadrightarrow_\beta B'$. By lemma 10.3.8 we have that $?\ \vdash^\omega A' : s$ for some sort s. Therefore we can apply the conversion rule and we obtain $?\ \vdash^\omega a : A'$. Then we apply the application rule and we have a derivation of $?\ \vdash^\omega (b\ \ a) : B'[x := a]$.

2. Suppose $(b\ \ a) \to_\beta (b\ \ a')$ with $a \to_\beta a'$. By induction we have that $?\ \vdash^\omega a' : A'$ for $A \twoheadrightarrow_\beta A'$. By lemma 10.3.8 we have that $?\ \vdash^\omega A : s$. Therefore we can apply the conversion rule and we have that $?\ \vdash^\omega a' : A$. Then we apply the application rule and we obtain a derivation of $?\ \vdash^\omega (b\ \ a') : B[x := a']$.

3. Suppose $(\lambda x{:}A'.\ d)a \to_\beta d[x := a]$. By lemma 10.3.4 we have that $?, x{:}A' \vdash^\omega d : B'$. Moreover by lemma 10.3.4 we know there are two possibilities, either $(\Pi x{:}A.\ B) = (\Pi x{:}A'.\ B')$ or $(\Pi x{:}A.\ B) \twoheadleftrightarrow_\beta (\Pi x{:}A'.\ B')$ and $?\ \vdash^\omega (\Pi x{:}A.\ B) : s$.

   (a) Suppose $(\Pi x : A.\ B) = (\Pi x : A'.\ B')$. Hence $A = A'$ and $B = B'$. By the substitution lemma we have that $?\ \vdash^\omega d[x := a] : B[x := a]$.

   (b) Suppose $(\Pi x{:}A.\ B) \twoheadleftrightarrow_\beta (\Pi x{:}A'.\ B')$ and $?\ \vdash^\omega (\Pi x{:}A.\ B) : s$ for some s. By lemma 10.3.8 we have that $?\ \vdash^\omega A' : s'$ for some s'. Therefore we can apply the conversion rule and we have that $?\ \vdash^\omega a : A'$. By the substitution lemma we have that $?\ \vdash^\omega d[x := a] : B'[x := a]$. Since $?\ \vdash^\omega (\Pi x{:}A.\ B) : s$, it follows from the generation lemma that $?, x : A \vdash^\omega B : s_2$ for some sort s_2. By the substitution lemma we have that $?\ \vdash^\omega B[x := a] : s_2$ and by conversion rule we have that $?\ \vdash^\omega d[x := a] : B[x := a]$.

The rest of the cases are easy to prove. □

The next example shows that subject reduction does not hold for pure type systems without the Π-condition. If $?\ \vdash^\omega e : E$ and $e \to_\beta e'$ then e' may not have type E. This happens when the type E is not typable and hence we cannot apply conversion rule.

Example 10.3.10. Let $? = < \gamma : * >$.

$$\begin{array}{rcll} e & = & (\lambda\alpha{:}*.\lambda x{:}\alpha.*)((\lambda\alpha{:}*.\alpha)\gamma) & \\ & \to_\beta & (\lambda\alpha{:}*.\lambda x{:}\alpha.*)\gamma & = \ e' \end{array}$$

In $\lambda^\omega(C)$, we have that $?\ \vdash^\omega e : (\lambda\alpha{:}*.\alpha)\gamma \to \Box$ and $?\ \vdash^\omega e' : \gamma \to \Box$ but not $?\ \vdash^\omega e' : (\lambda\alpha{:}*.\alpha)\gamma \to \Box$ (the latter can be proved by some meta-theoretical reasoning).

Theorem 10.3.11. (Uniqueness of Types) Let S be a singly sorted specification. If $?\ \vdash^\omega a : A$ and $?\ \vdash^\omega a : B$ then $A \twoheadleftrightarrow_\beta B$.

This theorem is proved by induction on the derivation of $?\ \vdash^\omega a : A$.

10.4 Description of Toptypes

Toptypes are types that are not typable. In this section we study the form of a toptype for pure type systems without the Π-condition and prove weak type reduction. Also we prove for singly sorted specifications that the set of toptypes is closed under β-reduction and under substitution.

Recall that A is *a toptype in* $?$ if there exists a such that $? \vdash^\omega a : A$ and $? \not\vdash^\omega A : -$ and that A is a *toptype* if there are a and $?$ such that $? \vdash^\omega a : A$ and $? \not\vdash^\omega A : -$ (see definitions 4.2.2 and 4.4.10).

Note that for pure type systems the only toptypes are the topsorts that are inhabited.

Next we give examples of toptypes in pure types systems without the Π-condition.

Example 10.4.1.

a) In $\lambda^\omega(C)$ we can derive

$$A : * \vdash^\omega_C \lambda x{:}A.(* \to *) : (A \to \Box)$$

We have that $(A \to \Box)$ is a toptype.

b) In $\lambda^\omega_\to$ we can derive

$$\vdash^\omega (\lambda\alpha{:}*.\lambda x{:}\alpha.x) : \Pi\alpha{:}*.\alpha \to \alpha$$

We have that $\Pi\alpha{:}*.\alpha \to \alpha$ is a toptype.

Definition 10.4.2. We say that the product $\Pi x{:}A.\ B$ *can be formed* in the context $?$ if there exists an s such that $? \vdash^\omega (\Pi x{:}A.\ B) : s$.

Note that the product $\Pi x{:}A.\ B$ cannot be formed in the context $?$ if for all s_1, s_2 such that $? \vdash^\omega A : s_1$ and $?, x{:}A \vdash^\omega B : s_2$ we have that there is no sort s with $(s_1, s_2, s) \in \boldsymbol{R}$.

We define the set $\mathcal{M}_\Gamma$ of 'potencial' toptypes in $?$. The set $\mathcal{M}_\Gamma$ is not exactly the set of toptypes. If an element is in the set $\mathcal{M}_\Gamma$ then it is not typable in $?$. However it may or may not be inhabited.

Definition 10.4.3. Let $? \in \mathcal{C}$. We define the set $\mathcal{M}_\Gamma$ as the smallest subset of $\mathcal{T}$ satisfying the following clauses.

1. if s is a topsort then $s \in \mathcal{M}_\Gamma$,

2. if $?, x{:}A \vdash^\omega B : s$ for some s and the product $(\Pi x{:}A.\ B)$ cannot be formed in $?$ then $(\Pi x{:}A.\ B) \in \mathcal{M}_\Gamma$.

3. if $B \in \mathcal{M}_{\Gamma, x:A}$ then $(\Pi x{:}A.\ B) \in \mathcal{M}_\Gamma$.

Definition 10.4.4. Let $? \in \mathcal{T}$. A mapping $depth_\Gamma : \mathcal{M}_\Gamma \to \mathbb{N}$ is defined as follows.

$$\begin{array}{rcll} depth_\Gamma(s) & = & 0 & \text{if } s \text{ is a topsort} \\ depth_\Gamma(\Pi x{:}A.\ B) & = & 1 & \text{if } ?, x{:}A \vdash^\omega B : s \text{ for some } s \\ & & & \text{and } (\Pi x{:}A.\ B) \text{ cannot be formed in } ? \\ depth_\Gamma(\Pi x{:}A.\ B) & = & depth_{\Gamma,x:A}(B) + 1 & \text{otherwise} \end{array}$$

We extend the mapping *depth* to pseudoterms as follows.

$$depth_\Gamma(A) = 0 \text{ for } A \in \mathcal{T} - \mathcal{M}_\Gamma \text{ and } ? \in \mathcal{C}.$$

Next we prove that a pseudoterm in the set $\mathcal{M}_\Gamma$ is a sequence of products whose 'heart' is either a topsort or it is a product that cannot be formed.

Lemma 10.4.5. Let $? \in \mathcal{C}$ and $A \in \mathcal{T}$. The following two statements are equivalent:

1. $A \in \mathcal{M}_\Gamma$, $n = depth_\Gamma(A)$,
2. there are $A_1, \ldots, A_n, B$ such that $A = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B$ where

   either $n \geq 0$ and B is a topsort s_0

   or $n \geq 1$ and $?, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^\omega B : s$ for some sort s and $(\Pi x_n{:}A_n.\ B)$ cannot be formed in $?, x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1}$.

Proof:

$(1 \Rightarrow 2)$. This is proved by induction on $A \in \mathcal{M}_\Gamma$.

1. Suppose that the pseudoterm is a topsort s. Take $n = 0$ and the second statement holds.
2. Suppose that the pseudoterm is $(\Pi x{:}A.\ B)$ with $?, x{:}A \vdash^\omega B : s$ for some s. We know that the product $\Pi x{:}A.\ B$ cannot be formed in $?, x{:}A$. Take $n = 1$ and the second statement holds.
3. Suppose that the pseudoterm is $(\Pi x{:}A.\ B)$ with $B \in \mathcal{M}_{\Gamma,x:A}$. By induction we have that $B = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B'$ where

   either $n \geq 0$ and B' is a topsort s_0
   or $n \geq 1$ and $?, x{:}A, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^\omega B' : s$ for some sort s and $(\Pi x_n{:}A_n.\ B')$ cannot be formed in $?, x{:}A, x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1}$.

   Therefore the second statement holds for $(\Pi x{:}A.\ B)$.

$(2 \Rightarrow 1)$. There are two cases:

1. Suppose $n \geq 0$ and B is a topsort s_0. We apply once the first clause and then n times the third clause.

2. Suppose that $?, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^\omega B : s$ for some sort s and the product $(\Pi x_n{:}A_n.\ B)$ cannot be formed in the context $?, x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1}$.

   We apply once the second clause and then n times the third clause.

□

Next we prove that the pseudoterms in the set $\mathcal{M}_\Gamma$ are not typable in the context $?$.

Lemma 10.4.6. If $A \in \mathcal{M}_\Gamma$ then $? \not\vdash^\omega A : -$.

Proof: This is proved by induction on $A \in \mathcal{M}_\Gamma$.

1. Suppose that the pseudoterm is a topsort s. Suppose towards a contradiction that $? \vdash^\omega s : D$ for some D. Using the generation lemma we deduce that there is a sort s' such that $(s, s') \in \boldsymbol{A}$. This is a contradiction. Hence $? \not\vdash^\omega s : -$.

2. Suppose that the pseudoterm is $(\Pi x{:}A.\ B)$ and that there are A and s such that $?, x{:}A \vdash^\omega B : s$. Also, we know that the product $\Pi x{:}A.\ B$ cannot be formed in $?, x{:}A$.

   Suppose towards a contradiction that $? \vdash^\omega (\Pi x{:}A.\ B) : D$ for some D. By the generation lemma, there are sorts s_1, s_2, s_3 such that $? \vdash^\omega A : s_1$, $?, x{:}A \vdash B : s_2$ and $(s_1, s_2, s_3) \in \boldsymbol{R}$. This is a contradiction.

3. Suppose that the pseudoterm is $(\Pi x{:}A.\ B)$ with $B \in \mathcal{M}_{\Gamma, x:A}$. By induction we have that $?, x{:}A \not\vdash^\omega B : -$.

   Suppose towards a contradiction that $? \vdash^\omega (\Pi x{:}A.\ B) : D$ for some D. By the generation lemma there is a sort s_2 such that $?, x{:}A \vdash B : s_2$. This is a contradiction.

□

Lemma 10.4.7. Let $? \vdash^\omega a : A$. If there is no sort s such that $? \vdash^\omega B[x := a] : s$ then there is no sort s such that $? \vdash^\omega (\Pi x{:}A.\ B) : s$.

Proof: Suppose $? \vdash^\omega (\Pi x{:}A.\ B) : s'$. By the generation lemma we have that $?, x{:}A \vdash^\omega B : s$. By the substitution lemma we have that $? \vdash^\omega B[x := a] : s$. □

Next we prove that a type in $?$ that does not have a sort as type is in the set $\mathcal{M}_\Gamma$.

Lemma 10.4.8. Suppose that $? \vdash^\omega a : A$.

If there is no sort s such that $? \vdash A : s$ then $A \in \mathcal{M}_\Gamma$.

Proof: This lemma is proved by induction on the derivation of $? \vdash^\omega a : A$. We will prove the cases of the abstraction and the application.

- (abstraction) $\dfrac{?, x{:}D \vdash^\omega e : E}{? \vdash^\omega (\lambda x{:}D.\ e) : (\Pi x{:}D.\ E)}$.

  1. Suppose that $?, x{:}D \vdash^\omega E : s$. The product $(\Pi x{:}D.\ E)$ cannot be formed in $?$. Hence $(\Pi x{:}D.\ E) \in \mathcal{M}_\Gamma$.
  2. Suppose that there is no sort s such that $?, x : D \vdash^\omega E : s$. By induction $E \in \mathcal{M}_{\Gamma, x:D}$. Hence $(\Pi x{:}D.\ E) \in \mathcal{M}_\Gamma$.

- (application) $\dfrac{? \vdash f : (\Pi x{:}D.\ E) \quad ? \vdash d : D}{? \vdash (f\ \ d) : E[x := d]}$.

  If there is no s such that $? \vdash^\omega E[x := d] : s$ then by lemma 10.4.7 there is no s such that $? \vdash (\Pi x{:}D.\ E) : s$.

  By induction we have that $(\Pi x : D.\ E) \in \mathcal{M}_\Gamma$. By lemma 10.4.5 we have that $E = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.H$ where either H is a topsort s_0 or $?, x{:}D, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^\omega H : s$ for some sort s.

  Then $E[x := d] = \Pi x_1{:}A_1[x := d] \ldots \Pi x_n{:}A_n[x := d].H[x := d]$ where $H[x := d]$ is either a topsort s_0 or $?, x_1{:}A_1[x := d], \ldots, x_n{:}A_n[x := d] \vdash^\omega H[x := d] : s$.

  1. Suppose that $H[x := d]$ is a topsort s_0. By lemma 10.4.5 we have that $E[x := d] \in \mathcal{M}_\Gamma$.
  2. Suppose that $?, x_1{:}A_1[x := d], \ldots, x_n{:}A_n[x := d] \vdash^\omega H[x := d] : s$. There should be a natural number k with $1 \leq k \leq n$ and a sort s' such that

     a) $?, x_1{:}A_1[x := d] \ldots x_k{:}A_k[x := d] \vdash^\omega (\Pi x_{k+1}{:}A_{k+1} \ldots \Pi x_n{:}A_n.\ B)[x := d] : s'$ and

     b) the product $(\Pi x_k{:}A_k \ldots \Pi x_n{:}A_n.\ B)[x := d]$ cannot be formed in the context $?, x_1{:}A_1[x := d] \ldots x_{k-1}{:}A_{k-1}[x := d]$.

     Therefore by lemma 10.4.5 we have that $E[x := d] \in \mathcal{M}_\Gamma$.

The rest of the cases are easy to prove. □

The next theorem says that a type is not typable in a context $?$ if and only if it is in the set $\mathcal{M}_\Gamma$. Hence, A is a toptype in $?$ if and only if A is a type in $?$ and it is in the set $\mathcal{M}_\Gamma$.

Theorem 10.4.9. (Description of Toptypes) Let $? \vdash^\omega a : A$. The following statements are equivalent:

1. $? \not\vdash^\omega A : -$,

2. There is no sort s such that $? \vdash A : s$,

3. $A \in \mathcal{M}_\Gamma$.

Proof: The proof of $(1 \Rightarrow 2)$ is trivial. The implication $(2 \Rightarrow 3)$ is lemma 10.4.8. The implication $(3 \Rightarrow 1)$ is lemma 10.4.6. □

Lemma 10.4.10. Let $? \vdash^\omega e : E$ and $? \not\vdash^\omega E : -$ be such that $depth(E) = n$ and $E = (\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ B)$. Then there exists $e' = (\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.\ b)$ such that $e \twoheadrightarrow_\beta e'$ and $? \vdash^\omega e' : E$.

Proof: This is proved by induction on the derivation of $? \vdash^\omega a : A$. Only one case is considered.

$$\frac{? \vdash^\omega b : (\Pi x{:}A.\ B) \quad ? \vdash^\omega a : A}{? \vdash^\omega (b\ \ a) : B[x := a]}.$$

By induction hypothesis we have that $b \twoheadrightarrow_\beta b'$, $b' = (\lambda x{:}A.\ \lambda x_1{:}A_1 \ldots \lambda x_m{:}A_m.\ d)$ and $? \vdash^\omega b' : (\Pi x{:}A.\ B)$ for some $A_1, \ldots A_m$ and d.

Note that $depth(B[x := a]) = n \leq m = depth(\Pi x{:}A.\ B)$.

Therefore $(b\ \ a) \twoheadrightarrow_\beta d$ and $d = (\lambda x_1{:}A_1[x := a] \ldots \lambda x_m{:}A_m[x := a].\ d[x := a])$. By the substitution lemma we have that $? \vdash^\omega d : B[x := a]$. □

Theorem 10.4.11. (Weak Type Reduction Theorem)
If $? \vdash^\omega e : E$ and $E \rightarrow_\beta E'$ then there exists e' such that $e \twoheadrightarrow_\beta e'$ and $? \vdash^\omega e' : E'$.

Proof: There are two possibilities, either E' is a toptype or not.

1. Suppose E' is not a toptype. By theorem 10.4.9, $? \vdash^\omega E' : s$. Applying the conversion rule, we have that $? \vdash e : E'$.

2. Suppose E' is a toptype. Then E is a toptype and $E = (\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ B)$ with $n = depth(E)$.

   By lemma 10.4.10 there exists e' such that $e \twoheadrightarrow_\beta e' = (\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.\ b)$ and $? \vdash^\omega e' : E$. Hence $?, x_1{:}A_1 \ldots, x_n{:}A_n \vdash^\omega b : B$.

   (a) If $A_i \rightarrow_\beta A_i'$ then by the weak subject reduction theorem $?, x_1 : A_1 \ldots x_i : A_i', \ldots, x_n{:}A_n \vdash^\omega b : B$.

   (b) If $B \rightarrow_\beta B'$ then by lemma 10.4.5 and the weak subject reduction theorem we have that $?, x_1{:}A_1 \ldots, x_n{:}A_n \vdash^\omega B' : s$.

   Applying the conversion rule we have that $?, x_1{:}A_1, \ldots, x_n{:}A_n \vdash^\omega b : B'$.

□

For singly sorted specifications, we give another characterisation of the set of toptypes. Using this characterisation we prove that for singly sorted specifications, the toptypes are closed under β-reduction and substitution.

Next we define the set $\mathcal{N}_\Gamma$ similar to the set $\mathcal{M}_\Gamma$. The elements in $\mathcal{N}_\Gamma$ are not typable and they may or may not be inhabited.

Definition 10.4.12. We define the set $\mathcal{N}_\Gamma$ as the smallest subset of $\mathcal{T}$ satisfying the following clauses.

1. if s is a topsort then $s \in \mathcal{N}_\Gamma$,
2. if there are sorts s_1, s_2 such that $? \vdash^\omega A : s_1$, that $?, x{:}A \vdash^\omega B : s_2$ and there is no s_3 such that $(s_1, s_2, s_3) \in \boldsymbol{R}$ then $(\Pi x{:}A.\ B) \in \mathcal{N}_\Gamma$.
3. if $B \in \mathcal{N}_{\Gamma, x:A}$ then $(\Pi x{:}A.\ B) \in \mathcal{N}_\Gamma$.

Note that definitions 10.4.3 and 10.4.12 differ only in the second clause. For the case of singly sorted specifications, to require that the product $(\Pi x{:}A.\ B)$ cannot be formed in $?$ is equivalent to require that there are sorts s_1, s_2 such that $? \vdash^\omega A : s_1$, that $?, x{:}A \vdash^\omega B : s_2$ and there is no sort s_3 such that $(s_1, s_2, s_3) \in \boldsymbol{R}$.

Note that the inclusion $\mathcal{M}_\Gamma \subset \mathcal{N}_\Gamma$ holds for any specification. If the specification is not singly sorted, $\mathcal{N}_\Gamma$ may have more elements than $\mathcal{M}_\Gamma$.

Example 10.4.13. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be the non-singly sorted specification such that $\boldsymbol{S} = \{0, 1\}$, $\boldsymbol{A} = \{(0, 1), (0, 2)\}$ and $\boldsymbol{R} = \{(2, 2, 2)\}$. We have that $(0 \to 0) \in \mathcal{N}_\Gamma$ but $(0 \to 0) \notin \mathcal{M}_\Gamma$.

In the following lemma, we prove that $\mathcal{M}_\Gamma = \mathcal{N}_\Gamma$ for singly sorted specifications.

Lemma 10.4.14. Let S be a singly sorted specification. Then $\mathcal{M}_\Gamma = \mathcal{N}_\Gamma$.

Proof: $[\mathcal{M}_\Gamma \subset \mathcal{N}_\Gamma]$. We proceed by induction on the definition of $D \in \mathcal{M}_\Gamma$. Only one case is considered. Suppose that the pseudoterm D is $(\Pi x{:}A.\ B)$, there are A and s_2 such that $?, x{:}A \vdash^\omega B : s_2$ and the product $(\Pi x{:}A.\ B)$ cannot be formed in $?, x{:}A$. By lemma 10.3.2 we have that there is some s_1 such that $? \vdash^\omega A : s_1$. There is no s such that $(s_1, s_2, s) \in \boldsymbol{R}$. Therefore $(\Pi x{:}A.\ B) \in \mathcal{N}_\Gamma$.

$[\mathcal{N}_\Gamma \subset \mathcal{M}_\Gamma]$. We proceed by induction on the definition of $D \in \mathcal{N}_\Gamma$. Only one case is considered. Suppose that the pseudoterm D is $(\Pi x{:}A.\ B)$ and that there are sorts s_1, s_2 such that $? \vdash^\omega A : s_1$, $?, x{:}A \vdash^\omega B : s_2$ and there is no sort s_3 such that $(s_1, s_2, s_3) \in \boldsymbol{R}$.

Suppose towards a contradiction that $(\Pi x{:}A.\ B)$ can be formed in $?$. Then there are sorts s_1', s_2' such that $? \vdash^\omega A : s_1'$, $?, x{:}A \vdash^\omega B : s_2'$. By uniqueness of types we have that $s_1 = s_1'$ and $s_2 = s_2'$. This is a contradiction. $\square$

As a consequence of the previous lemma, the set $\mathcal{N}_\Gamma$ has the same properties as the set $\mathcal{M}_\Gamma$ if the specification is singly sorted.

Lemma 10.4.15. Let S be a singly sorted specification.

$$\text{If } A \in \mathcal{N}_\Gamma \text{ then } ? \not\vdash^\omega A : -.$$

Proof: This follows from lemmas 10.4.6 and 10.4.14. □

The next theorem states that a type is in $\mathcal{N}_\Gamma$ if and only if this type is not typable. Hence A is a toptype in ? if and only if A is a type and it is in the set $\mathcal{N}_\Gamma$.

Theorem 10.4.16. Let S be a singly sorted specification and $? \vdash^\omega a : A$.
The following statements are equivalent:

1. $? \not\vdash^\omega A : -$.
2. $A \in \mathcal{N}_\Gamma$.

Proof: This follows from theorem 10.4.9 and lemma 10.4.14. □

In the following lemma we prove that the set $\mathcal{N}_\Gamma$ is closed under β-reduction.

Lemma 10.4.17. If $D \in \mathcal{N}_\Gamma$ and $D \twoheadrightarrow_\beta D'$ then $D' \in \mathcal{N}_\Gamma$.

Proof: The following two statements are proved by induction on the definition of $D \in \mathcal{N}_\Gamma$.

a) if $D \to_\beta D'$ then $D' \in \mathcal{N}_\Gamma$,

b) if $? \to_\beta ?'$ then $D \in \mathcal{N}_{\Gamma'}$.

We only prove statement **a)**.

1. Suppose the pseudoterm D is a topsort s. Note that s is in β-normal form.
2. Suppose that the pseudoterm D is $(\Pi x{:}A.\ B)$ and that there are sorts s_1, s_2 such that $? \vdash^\omega A : s_1$, $?, x{:}A \vdash B : s_2$ and there is no sort s_3 such that $(s_1, s_2, s_3) \in \boldsymbol{R}$.

   If $A \to_\beta A'$ then it follows from weak subject reduction theorem that $? \vdash^\omega A' : s_1$ and $?, x{:}A \vdash^\omega B : s_2$. Hence $(\Pi x{:}A'.\ B) \in \mathcal{N}_\Gamma$.

   Similarly if $B \to_\beta B'$ we have that $(\Pi x{:}A.\ B') \in \mathcal{N}_\Gamma$.
3. Suppose that the pseudoterm D is $(\Pi x{:}A.\ B)$ with $B \in \mathcal{N}_{\Gamma, x:A}$.

   If $A \to_\beta A'$ then it follows from induction that $B \in \mathcal{N}_{\Gamma, x:A'}$. Therefore $(\Pi x{:}A'.\ B) \in \mathcal{N}_\Gamma$.

   Similarly if $B \to_\beta B'$ we have that $(\Pi x{:}A.\ B') \in \mathcal{N}_\Gamma$.

□

As a consequence of the previous lemma, we have that the set of toptypes is closed under β-reduction.

Theorem 10.4.18. **(β-closure of toptypes)** Let S be a singly sorted specification.

If A is a toptype and $A \twoheadrightarrow_\beta A'$ then A' is a toptype.

Proof: If A is a toptype then there are $?$ and a such that $? \vdash^\omega a : A$ and $? \not\vdash^\omega A : -$.

By theorem 10.4.16 we have that $A \in \mathcal{N}_\Gamma$. By lemma 10.4.17 we have that $A' \in \mathcal{N}_\Gamma$. By lemma 10.4.15 we have that $? \not\vdash^\omega A' : -$. By the weak type reduction theorem we have that there exists a' such that $? \vdash a' : A'$. Hence A' is a toptype. $\Box$

Corollary 10.4.19. Let S be a singly sorted specification and $? \vdash^\omega b : B$.

If there exists a sort s such that $? \vdash^\omega B' : s$ and $B' \twoheadleftrightarrow_\beta B$ then $? \vdash^\omega B : s$.

Proof: By Church-Rosser theorem, there exists a common reduct D_0 of B and B'. By weak subject reduction theorem we have that $? \vdash^\omega D_0 : s$. By theorem 10.4.18 we have that $? \vdash^\omega B : s'$ for some s'. By weak subject reduction $? \vdash^\omega D_0 : s'$ and by uniqueness of types we have that $s = s'$. $\Box$

In the following, we prove that the set of toptypes is closed under substitution.

Lemma 10.4.20. Let S be a singly sorted specification. Let $? \vdash d : D$.

If $E \in \mathcal{N}_{\Gamma, y{:}D, \Gamma'}$ then $E[y := d] \in \mathcal{N}_{\Gamma, \Gamma'[y:=d]}$.

Proof: This is proved by induction on $E \in \mathcal{N}_{\Gamma, y{:}D, \Gamma'}$. Only one case is considered.

Suppose that E is $(\Pi x{:}A.\ B)$ and that there are sorts s_1, s_2 such that $?, y{:}D, ?' \vdash^\omega A : s_1$ and $?, y{:}D, ?', x{:}A \vdash B : s_2$. Moreover there is no sort s_3 such that $(s_1, s_2, s_3) \in \boldsymbol{R}$.

Substitution lemma yields $?, ?' \vdash^\omega A[y := d] : s_1$ and $?, ?'[y := d], x{:}A[y := d] \vdash B[y := d] : s_2$. Hence $(\Pi x{:}A.\ B)[y := d] \in \mathcal{N}_{\Gamma, \Gamma'[y:=d]}$. $\Box$

Theorem 10.4.21. **(Substitution on Toptypes)** Let S be a singly sorted specification. Suppose that $?, y{:}D, ?' \vdash e : E$ and that $? \vdash d : D$.

If $?, y{:}D, ?' \not\vdash^\omega E : -$ then $?, ?'[y := d] \not\vdash^\omega E[y := d] : -$.

Proof: By theorem 10.4.16 we have that $E \in \mathcal{N}_{\Gamma, y{:}D, \Gamma'}$. By lemma 10.4.20 we have that $E[y := d] \in \mathcal{N}_{\Gamma, \Gamma'[y:=d]}$. By lemma 10.4.15 we have that $?, ?'[y := d] \not\vdash^\omega E[y := d] : -$. $\Box$

Corollary 10.4.22. Let S be a singly sorted specification. Let $? \vdash^\omega F : (\Pi x{:}A.\ B)$, $? \vdash^\omega B[x := a] : s$ and $? \vdash^\omega a : A$. Then $?, x{:}A \vdash^\omega B : s$.

The next example shows that when the specification is not singly sorted, a toptype may become typable after substitution.

Example 10.4.23. The following specification is not singly sorted:

$$S \quad \boxed{\begin{array}{ll} \boldsymbol{S} & 0, 1, 2 \\ \boldsymbol{A} & 0:1, 0:2, 1:2 \\ \boldsymbol{R} & (2, 2) \end{array}}$$

Given the context $< x{:}1 >$, the term $(x \to x)$ is a toptype. Substituting x by 0 we obtain $(0 \to 0)$ which has type 2.

10.5 Normalisation for β-reduction

In this section we define the notion of illegal redex and a function $\varphi : \mathcal{C} \times \mathcal{T} \to \mathcal{T}$ that contracts the illegal redexes of a term (we write $\varphi_\Gamma(a)$ instead of $\varphi(?\,, a)$). We prove that φ_Γ is a strategy for the β-reduction. Moreover we prove that for singly sorted specifications, if a term b is typable in $\lambda^\omega(S)$ then $\varphi_\Gamma(b)$ does not contain illegal redexes. For each singly sorted specification, we define a converting morphism from $\lambda^\omega(S)$ to $\lambda(S)$. Finally we prove that weak normalisation of $\lambda(S)$ implies weak normalisation of $\lambda^\omega(S)$ if S is singly sorted.

Definition 10.5.1. Let S be a singly sorted specification.

We say that an abstraction $\lambda x{:}A.b$ is *illegal in the context* $?$ if there exists D such that $? \vdash^\omega \lambda x{:}A.b : D$ and $? \not\vdash^\omega D : -$.

An abstraction is *legal* if it is typable in $?$ and it is not illegal in $?$.

We say that $(\lambda x{:}A.b)a$ is an *illegal β-redex in* $?$ if the abstraction $\lambda x{:}A.b$ is illegal in the context $?$.

Lemma 10.5.2. Let S be a singly sorted specification. An abstraction $\lambda x{:}A.b$ is legal in the context $?$ if and only if there exists D such that $? \vdash^\omega \lambda x{:}A.b : D$ and $? \vdash^\omega D : s$.

Proof:

($\Rightarrow$) Obvious.

($\Leftarrow$) Since $? \vdash^\omega \lambda x{:}A.b : D$, the abstraction $\lambda x{:}A.b$ is typable in $?$. Suppose there exists D' such that $? \vdash^\omega \lambda x{:}A.b : D'$. By the uniqueness of types theorem we have that $D' \twoheadleftrightarrow_\beta D$. Corollary 10.4.19 yields $? \vdash^\omega D' : s$.

□

We define the mapping φ that contracts the illegal redexes of a term.

Definition 10.5.3. We define $\varphi : \mathcal{C} \times \mathcal{T} \to \mathcal{T}$ as follows.

$$
\begin{array}{rcl}
\varphi_\Gamma(x) & = & x \\
\varphi_\Gamma(s) & = & s \\
\varphi_\Gamma(a\ b) & = & \begin{cases} a_0[x := \varphi_\Gamma(b)] & \text{if } \varphi_\Gamma(a) = \lambda x{:}A.a_0 \text{ is an illegal abstraction in } ? \\ (\varphi_\Gamma(a)\ \varphi_\Gamma(b)) & \text{otherwise} \end{cases} \\
\varphi_\Gamma(\lambda x{:}A.\ a) & = & (\lambda x{:}\varphi_\Gamma(A).\ \varphi_{\Gamma,x:A}(a)) \\
\varphi_\Gamma(\Pi x{:}A.\ B) & = & (\Pi x{:}\varphi_\Gamma(A).\ \varphi_{\Gamma,x:A}(B))
\end{array}
$$

Sometimes we write $\varphi(a)$ instead of $\varphi_\Gamma(a)$.

Definition 10.5.4. We define $\varphi : \mathcal{C} \to \mathcal{C}$ as follows.

$$\begin{array}{rcl} \varphi(\epsilon) & = & \epsilon \\ \varphi(?\,, x{:}A) & = & \varphi(?\,), x{:}\varphi_\Gamma(A) \end{array}$$

The following lemma states that φ_Γ is the identity on $\lambda(S)$.

Lemma 10.5.5. Let S be a singly sorted specification. If $? \vdash a : A$ then $\varphi(?\,) = ?\,$, $\varphi_\Gamma(a) = a$ and $\varphi_\Gamma(A) = A$.

The following lemma states that φ_Γ is a strategy for β-reduction.

Lemma 10.5.6. For all $?\,, a \twoheadrightarrow_\beta \varphi_\Gamma(a)$.

Lemma 10.5.7. Let S be a singly sorted specification. If $?\,, x : A, ?' \vdash^\omega b : B$ and $? \vdash^\omega a : A$ then $\varphi_{\Gamma, x{:}A, \Gamma'}(b)[x := \varphi_\Gamma(a)] = \varphi_{\Gamma,\Gamma'}(b[x := a])$.

Proof: We prove the case of the application.

$$\varphi(F\ b) = \begin{cases} a_0[y := \varphi(b)] & \text{if } \varphi(F) = \lambda y{:}A.a_0 \text{ is illegal} \\ (\varphi(F)\ \varphi(b)) & \text{otherwise} \end{cases}$$

$$\varphi((F\ b)[x := a]) = \begin{cases} a_1[y := \varphi(b[x := a])] & \text{if } \varphi(F[x := a]) = \lambda y{:}A'.a_1 \\ (\varphi(F[x := a])\ \varphi(b[x := a])) & \text{otherwise} \end{cases}$$

By induction we have that $\varphi(F)[x := \varphi(a)] = \varphi(F[x := a])$ and $\varphi(b)[x := \varphi(a)] = \varphi(b[x := a])$.

Note that F and $\varphi(F)$ are typable in $?\,, x{:}A, ?'$ and $\varphi(F)[x := \varphi(a)]$ is typable in $?\,, ?'$. There are several cases:

1. Suppose $\varphi(F) = \lambda y{:}E.\ a_0$ is an illegal abstraction.

   Therefore $?\,, x : A, ?' \vdash^\omega \lambda y : E.a_0 : D$ and $?\,, x : A, ?' \not\vdash^\omega D : -$ for some D. It follows from the weak subject reduction theorem that $? \vdash^\omega \varphi(a) : A'$ with $A \twoheadrightarrow_\beta A'$. Since $? \vdash^\omega A : s$ we have that $? \vdash \varphi(a) : A$. It follows from lemma 10.3.7 that $?\,, ?'[x := \varphi(a)] \vdash^\omega \varphi(F)[x := \varphi(a)] : D[x := \varphi(a)]$. By theorem 10.4.21, we have that $?\,, ?'[x := \varphi(a)] \not\vdash^\omega D[x := \varphi(a)] : -$. Therefore $\varphi(F)[x := \varphi(a)]$ is an illegal abstraction too.

   Moreover we have that

$$\begin{array}{rcl} \varphi(F[x := a]) & = & \varphi(F)[x := \varphi(a)] \\ & = & \lambda y{:}E[x := \varphi(a)].a_0[x := \varphi(a)] \end{array}$$

   We conclude that $\varphi(F\ b)[x := \varphi(a)] = \varphi((F\ b)[x := a])$ as follows.

$$\begin{array}{rcl} \varphi(F\ b)[x := \varphi(a)] & = & a_0[y := \varphi(b)][x := \varphi(a)] \\ & = & a_0[x := \varphi(a)][y := \varphi(b)[x := \varphi(a)]] \\ & = & \varphi(F[x := a]\ b[x := a]) \end{array}$$

2. Suppose $\varphi(F)$ is not an illegal abstraction.

   (a) Suppose that $\varphi(F)$ is not an abstraction. If $\varphi(F)[x := \varphi(a)] = \varphi(F[x := a])$ were an abstraction we would have that $\varphi(F) = x$ and $\varphi(a) = \lambda y{:}E.\ a_0$. We have that $? \vdash^\omega \varphi(a) : A$ and $? \vdash^\omega A : s$. By lemma 10.5.2 we have that $\varphi(F)[x := \varphi(a)]$ is a legal abstraction. Hence the value of the application can be computed as follows.

$$\begin{array}{rcl} \varphi(F\ b)[x := \varphi(a)] & = & (\varphi(F)\ \varphi(b))[x := \varphi(a)] \\ & = & (\varphi(F)[x := \varphi(a)]\ \varphi(b)[x := \varphi(a)]) \\ & = & (\varphi(F[x := a])\ \varphi(b[x := a])) \\ & = & \varphi(F[x := a]\ b[x := a]) \end{array}$$

   Hence $\varphi(F\ b)[x := \varphi(a)] = \varphi((F\ b)[x := a])$.

   (b) Suppose that $\varphi(F)$ is an abstraction but legal. By lemma 10.5.2 we have that $\varphi(F[x := a])$ is a legal abstraction. The equality $\varphi(F\ b)[x := \varphi(a)] = \varphi((F\ b)[x := a])$ is proved as in case a).

The rest of the cases are easy to prove. □

The next example shows that if the specification is not singly sorted $\varphi(b[x := a])$ may not be syntactically equal to $\varphi(b)[x := \varphi(a)]$.

Example 10.5.8. The following specification is not singly sorted:

S	$\boldsymbol{S}$	$0, 1, 2$
	$\boldsymbol{A}$	$0 : 1, 0 : 2, 1 : 2$
	$\boldsymbol{R}$	$(2, 2)$

We take $< ? = x : 1, z : x >$ and $b = (\lambda y{:}x.y)z$. Note that b contains illegal abstractions but $b[x := 0]$ does not. Hence $\varphi(b[x := 0]) \neq \varphi(b)[x := \varphi(0)]$.

In the following theorem, we prove that if b is typable in $\lambda^\omega(S)$ then $\varphi(b)$ does not contain illegal redexes when S is a singly sorted specification. The value $\varphi(b)$ is of the form $\lambda x_1 : A_1.\ \ldots \lambda x_n : A_n.\ b$ with b typable in $\lambda(S)$ and the abstractions whose bound variables are $x_1, \ldots, x_n$ are illegal.

Theorem 10.5.9. (Preservation of the typing relation)
Let S be a singly sorted specification.
If $? \vdash^\omega a : A$, $n = depth(A)$ and $A = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B$ then

$$\varphi(?, x_1{:}A_1 \ldots x_n{:}A_n) \vdash a' : \varphi(B) \text{ with } \varphi(a) = \lambda x_1{:}\varphi(A_1) \ldots \lambda x_n{:}\varphi(A_n).a'.$$

Proof: This property is proved by induction on the derivation of $? \vdash^\omega a : A$. We consider the cases of the abstraction, the application and the conversion rule.

- (abstraction) $\dfrac{?\,, x : A \vdash^\omega b : B}{? \vdash^\omega (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$.

  There are two possibilities, either $(\Pi x{:}A.\ B)$ is a toptype or not.

  1. Suppose $? \vdash^\omega (\Pi x{:}A.\ B) : s$. By the generation lemma we have that $?\,, x{:}A \vdash^\omega B : s_2$, $? \vdash^\omega A : s_1$ and $(s_1, s_2, s) \in \boldsymbol{R}$.
     By induction we have that $\varphi(?\,, x : A) \vdash \varphi(b) : \varphi(B)$.
     Using weak subject reduction theorem we deduce that $\varphi(?\,, x{:}A) \vdash^\omega \varphi(B) : s_2$.
     By lemma 10.3.5 we have that $\varphi(?\,, x : A) \vdash \varphi(B) : s_2'$ for some sort s_2'.
     Besides we have that $\varphi(?\,) \vdash \varphi(A) : s_1'$ for some sort s_1'.
     Since the specification is singly sorted, we have that $s_1 = s_1'$ and $s_2 = s_2'$ and we know that $(s_1, s_2, s) \in \boldsymbol{R}$.
     We obtain the following derivation:

$$\dfrac{\varphi(?\,, x{:}A) \vdash \varphi(b) : \varphi(B) \quad \dfrac{\varphi(?\,) \vdash \varphi(A) : s_1 \quad \varphi(?\,, x{:}A) \vdash \varphi(B) : s_2}{\varphi(?\,) \vdash \varphi(\Pi x{:}A.\ B) : s}}{\varphi(?\,) \vdash \varphi(\lambda x{:}A.\ b) : \varphi(\Pi x{:}A.\ B)}$$

  2. Suppose $? \not\vdash^\omega (\Pi x{:}A.\ B) : -$. There are two possibilities, either B is a toptype or not.

     (a) Suppose $?\,, x{:}A \vdash^\omega B : s$. By induction we have that:
     $$\varphi(?\,, x{:}A) \vdash^\omega \varphi(b) : \varphi(B)$$
     where $\varphi(\lambda x{:}A.\ b) = \lambda x{:}\varphi(A).\ \varphi(b)$

     (b) Suppose $?\,, x{:}A \not\vdash^\omega B : -$. It follows from induction that
     $$\varphi(?\,, x{:}A, x_1{:}A_1 \ldots x_n{:}A_n) \vdash b' : \varphi(B') \quad \text{where} \quad \begin{array}{l} B = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B', \\ \varphi(b) = \lambda x_1{:}\varphi(A_1) \ldots \lambda x_n{:}\varphi(A_n).b' \end{array}$$

- (application) $\dfrac{? \vdash^\omega b : (\Pi x{:}A.\ B) \quad ? \vdash^\omega a : A}{? \vdash^\omega (b\ \ a) : B[x := a]}$.

  There are two possibilities: either $B[x := a]$ is a toptype in $?$ or not.

  1. Suppose $? \vdash^\omega B[x := a] : s$.

     (a) Suppose $? \vdash^\omega (\Pi x{:}A.\ B) : s'$. By induction we have that
     $$\varphi(?\,) \vdash \varphi(b) : \varphi(\Pi x{:}A.\ B)$$
     By the generation lemma we have that $? \vdash^\omega A : s_1$. By induction we have that $\varphi(?\,) \vdash \varphi(a) : \varphi(A)$.
     Using the weak subject reduction theorem, we deduce that $? \vdash^\omega \varphi(b) : D$ and $? \vdash^\omega D : s$. Hence $\varphi(b)$ is not an illegal abstraction in $?$ and $\varphi(b\ a) = (\varphi(b)\ \varphi(a))$.
     Hence we have the following derivation:

$$\frac{\varphi(?\,) \vdash \varphi(b) : \varphi(\Pi x{:}A.\ B) \quad \varphi(?\,) \vdash \varphi(a) : \varphi(A)}{\varphi(?\,) \vdash \varphi(b\ \ a) : \varphi(B)[x := \varphi(a)]}$$

By lemma 10.5.7 we have that $\varphi(B)[x := \varphi(a)] = \varphi(B[x := a])$.

(b) Suppose $? \not\vdash^{\omega} (\Pi x{:}A.\ B) : -$.
By lemma 10.3.8 we have that $? \vdash^{\omega} A : s_1$. By induction we have that $\varphi(?\,) \vdash \varphi(a) : \varphi(A)$.
By corollary 10.4.22 we have that $?\,, x{:}A \vdash^{\omega} B : s_2$. By induction we have that $\varphi(?\,, x{:}A) \vdash b' : \varphi(B)$ with $\varphi(b) = \lambda x{:}\varphi(A).b'$.
We have that $? \vdash^{\omega} \lambda x{:}\varphi(A).b' : (\Pi x{:}\varphi(A).\ \varphi(B))$. By theorem 10.4.18 we have that $? \not\vdash^{\omega} (\Pi x{:}\varphi(A).\ \varphi(B)) : -$. Therefore $\varphi(b) = \lambda x{:}\varphi(A).b'$ is an illegal abstraction in the context $?$.
Since $\varphi(b) = \lambda x{:}A'.b'$ is an illegal abstraction, we have that $\varphi(b\ \ a) = b'[x := \varphi(a)]$.
By the substitution lemma,

$$\varphi(?\,) \vdash b'[x := \varphi(a)] : \varphi(B)[x := \varphi(a)]$$

By lemma 10.5.7, we have that $\varphi(B)[x := \varphi(a)] = \varphi(B[x := a])$.

2. Suppose $? \not\vdash^{\omega} B[x := a] : -$.
It follows from lemma 10.4.7 that $? \not\vdash^{\omega} (\Pi x{:}A.\ B) : -$.
By lemma 10.3.8 we have that $? \vdash^{\omega} A : s_1$. By induction we have that $\varphi(?\,) \vdash \varphi(a) : \varphi(A)$ and that $\varphi(?\,, x{:}A, x_1{:}A_1, \ldots, x_n{:}A_n) \vdash b' : \varphi(B_0)$ with $\varphi(b) = \lambda x{:}\varphi(A).\ \lambda x_1{:}\varphi(A_1) \ldots \lambda x_n{:}\varphi(A_n).b'$ and $B = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ B_0$.
Weak subject reduction theorem yields $? \vdash^{\omega} \varphi(b) : (\Pi x : A'.\ B')$. By theorem 10.4.18 we have that $? \not\vdash^{\omega} (\Pi x : A'.\ B') : -$. Therefore $\varphi(b)$ is an illegal abstraction and then the value of $\varphi(b\ a)$ is computed as follows.

$$\varphi(b\ \ a) \quad = \quad \lambda x_1{:}\varphi(A_1)[x := \varphi(a)] \ldots \lambda x_n{:}\varphi(A_n)[x := \varphi(a)].b'[x := \varphi(a)]$$

By the substitution lemma we have that $b'[x := \varphi(a)]$ has type $\varphi(B_0)[x := \varphi(a)]$ in the context $\varphi(?\,), x_1{:}\varphi(A_1)[x := \varphi(a)], \ldots, x_n{:}\varphi(A_n)[x := \varphi(a)])$.
By lemma 10.5.7 we have that: $\varphi(B_0)[x := \varphi(a)] = \varphi(B_0[x := a])$.
By theorem 10.4.21, we have that $n = depth_{\Gamma}(B[x := a]) = depth_{\Gamma, x:A}(B)$.

- (conversion) $\dfrac{? \vdash^{\omega} b : B \quad ? \vdash^{\omega} A : s \quad B \twoheadleftarrow\!\twoheadrightarrow_{\beta} A}{? \vdash^{\omega} b : A}$

By corollary 10.4.19, we have that $? \vdash^{\omega} B : s'$. By induction we have that $\varphi(?\,) \vdash \varphi(b) : \varphi(B)$ and $\varphi(?\,) \vdash \varphi(A) : s$. After applying the conversion rule, we obtain $\varphi(?\,) \vdash \varphi(b) : \varphi(A)$.

The rest of the cases are easy to prove. □

Definition 10.5.10. Let S a singly sorted specification. We define $\varphi^\bullet_S : \mathcal{C} \times \mathcal{T} \to \mathcal{C} \times \mathcal{T}$ as follows.

$$\varphi^\bullet_S(?\,, a) \;=\; \begin{cases} (<\varphi(?\,), x_1{:}A_1 \ldots x_n{:}A_n>, b) & \text{if } ? \vdash^\omega a : A \text{ for some } A,\ n = \mathit{depth}(A) \text{ and} \\ & \varphi(a) = \lambda x_1{:}A_1 \ldots x_n{:}A_n.\ b \\ (<\varphi(?\,), x_1{:}A_1 \ldots x_n{:}A_n>, b) & \text{if } a \text{ is a toptype in } ?\,,\ n = \mathit{depth}(a) \text{ and} \\ & \varphi(a) = \Pi x_1{:}A_1 \ldots x_n{:}A_n.\ b \\ (?\,, a) & \text{otherwise} \end{cases}$$

Sometimes we write φ instead of $\varphi^\bullet$.

Corollary 10.5.11. (Converting morphism from $\lambda^\omega(S)$ to $\lambda(S)$)
Let S be a singly sorted specification. There is a weak converting morphism from $\lambda^\omega(S)$ to $\lambda(S)$.

Proof: By the previous theorem we have that $\varphi^\bullet_S$ preserves the typing relation. Since φ is a strategy, $\varphi^\bullet_S$ preserves conversion. Hence $\varphi^\bullet_S$ is a weak converting morphism from $\lambda^\omega(S)$ to $\lambda(S)$. $\Box$

Note that the previous theorem in fact shows that $\varphi^\bullet$ is a natural transformation from λ^ω to λ when these are considered as functors from the category of singly sorted specifications into a category analogous to $\mathbf{Carst}_\omega$ where morphisms only preserve β-conversion.

Corollary 10.5.12. (Weak Normalisation) Let S be a singly sorted specification.
If $\lambda(S)$ is weakly normalising then so is $\lambda^\omega(S)$.

10.6 Weak and Strong Normalisation for β_ι-reduction

The mapping φ contracts the illegal redexes in a given order. Now we contract the illegal redexes in any order by means of a reduction called β_ι-reduction. In this section we define the notion of β_ι-reduction and prove that for singly sorted specifications if $? \vdash^\omega a : A$ then a is β_ι-strongly normalising.

Definition 10.6.1. We define the *illegal β-reduction in* $?$ (or β_ι-reduction) by the following rules:

$$? \vdash (\lambda x{:}A.\ b)a \rightarrow_{\beta_\iota} b[x := a] \qquad \text{if } (\lambda x{:}A.\ b)a \text{ is an illegal redex in } ?\,.$$

$$\frac{?\,, x{:}A \vdash b \rightarrow_{\beta_\iota} b'}{? \vdash (\lambda x{:}A.\ b) \rightarrow_{\beta_\iota} (\lambda x{:}A.\ b')} \qquad \frac{? \vdash A \rightarrow_{\beta_\iota} A'}{? \vdash (\lambda x{:}A.\ b) \rightarrow_{\beta_\iota} (\lambda x{:}A'.\ b)}$$

$$\frac{?\,, x{:}A \vdash b \rightarrow_{\beta_\iota} b'}{? \vdash (\Pi x{:}A.\ b) \rightarrow_{\beta_\iota} (\Pi x{:}A.\ b')} \qquad \frac{? \vdash A \rightarrow_{\beta_\iota} A'}{? \vdash (\Pi x{:}A.\ b) \rightarrow_{\beta_\iota} (\Pi x{:}A'.\ b)}$$

$$\frac{? \vdash b \rightarrow_{\beta_\iota} b'}{? \vdash (b\ a) \rightarrow_{\beta_\iota} (b'\ a)} \qquad \frac{? \vdash a \rightarrow_{\beta_\iota} a'}{? \vdash (b\ a) \rightarrow_{\beta_\iota} (b\ a')}$$

In the following lemma, we prove that φ is a strategy for β_ι-reduction.

Lemma 10.6.2. Let S be a singly sorted specification. If $? \vdash^\omega a : A$ then $? \vdash a \twoheadrightarrow_{\beta_\iota} \varphi_\Gamma(a)$.

In the following lemma, we prove that φ is a forgetting morphism.

Lemma 10.6.3. Let S be a singly sorted specification. If $? \vdash^\omega a : A$ and $? \vdash a \rightarrow_{\beta_\iota} b$ then $\varphi_\Gamma(a) = \varphi_\Gamma(b)$.

Proof: We prove only one case. Suppose that $? \vdash (\lambda x{:}A.\ b)a \rightarrow_{\beta_\iota} b[x := a]$ and $(\lambda x{:}A.\ b)a$ is an illegal redex in $?$. Note that $\varphi(\lambda x{:}A.\ b)$ is illegal. We have that

$$\begin{array}{rcll} \varphi((\lambda x{:}A.\ b)a) & = & \varphi(b)[x := \varphi(a)] & \\ & = & \varphi(b[x := a]) & \text{by lemma 10.5.7.} \end{array}$$

□

Theorem 10.6.4. (Weak Normalisation for β_ι-reduction)
Let S be a singly sorted specification. If $? \vdash^\omega b : B$ then b is β_ι-weakly normalising.

Proof: Suppose $? \vdash^\omega a : A$. By lemma 10.6.2, $? \vdash a \twoheadrightarrow_{\beta_\iota} \varphi_\Gamma(a)$. By theorem 10.5.9, we have that $\varphi_\Gamma(a)$ does not contain illegal redexes and hence it is in β_ι-normal form in $?$. By lemma 10.6.3 the normal form is unique. □

For singly sorted specifications, the illegal abstractions of a term b constitute an initial labelling for a superdevelopment (see chapter 7).

We write a morphism,

$$\lambda^\omega(S) \xrightarrow{L} (\Lambda_l, \rightarrow_{\beta_l})$$

The function L puts labells in the illegal abstractions of a pseudoterm.

Definition 10.6.5. We define the mapping $L : \mathcal{C} \times \mathcal{T} \to \Lambda_l$ as follows (we write $L(b)$ instead of $L(?\,, b)$).

$$L(x) = x$$

$$L(\lambda x{:}A.\ b) = \begin{cases} @^{\{0\}}(\lambda_0 z.\lambda_i x.L(b), L(A)) & \text{if } \lambda x{:}A.\ b \text{ is illegal in } ?\,, \\ & \text{take some } i \text{ and } z \text{ fresh} \\ @^{\{0\}}(\lambda_0 z.\lambda_0 x.L(b), L(A)) & \text{otherwise} \end{cases}$$

$$L(b\ a) = \begin{cases} @^{\{i\}}(L(b), L(a)) & \text{if } nf_{\beta_l}(L(b)) = \lambda_i x.\ b_0 \text{ and } i > 0 \\ @^{\emptyset}(L(b), L(a)) & \text{otherwise} \end{cases}$$

In the case of the abstraction, the function L puts a fresh label (greater than 0) to an illegal abstraction and the label 0 to a legal one. The terms in Λ_l are untyped so the declaration $x{:}A$ of an abstraction $\lambda x{:}A.\ b$ in $\mathcal{T}$ is encoded as a β_l-redex whose argument is the type A.

In the case of the application, the function L puts the label $i > 0$ to the application whose operator reduces to $\lambda_i x.\ b_0$, an abstraction that corresponds to an illegal abstraction.

All the illegal abstractions should have different labels. In the case of the abstraction, we assume that all the labels of the illegal abstractions in $L(b)$ and in $L(A)$ are different. Similarly, in the case of the the application, we assume that all the labels of the illegal abstractions in $L(b)$ and in $L(a)$ are different.

Lemma 10.6.6. Let S be a singly sorted specification. If $? \vdash^\omega b : B$ then $L(b) \in \Lambda_l$.

Lemma 10.6.7. Let S be a singly sorted specification and $?\,, x{:}A \vdash^\omega b : B$ and $? \vdash^\omega a : A$. Suppose that all the abstractions in $L(b)$ and in $L(a)$ have different labels except for 0.

$$L(b[x := a]) = L(b)[x := L(a)]$$

Theorem 10.6.8. Let S be a singly sorted specification. If $? \vdash^\omega b : B$ and $? \vdash b \to_{\beta_\iota} b'$ then $L(b) \to^+_{\beta_l} L(b')$.

Corollary 10.6.9. Let S be a singly sorted specification. Then L is a morphism from $\lambda^\omega(S)$ equipped with β_ι-reduction to $(\Lambda_l, \to_{\beta_l})$.

As a consequence, a β_ι-rewrite sequence can be mapped into a superdevelopment and so it is strongly normalising.

Corollary 10.6.10. (Strong Normalisation for β_ι-reduction)
Let S be a singly sorted specification. If $? \vdash^\omega b : B$ then b is β_ι-strongly normalising.

10.7 Syntax Directed Rules

In this section we define a syntax directed set of rules for any singly sorted pure type system. This system will be used in the following section to define a function that infers the type.

When we try to infer the type of a term, we construct the derivation tree bottom-up, from the conclusion to the premises. This tree is constructed by means of an analysis of the term and of the context. According to the structure of the term we try to deduce which rule may fit as the last rule in the derivation tree. For a term like an abstraction, the last rule in the derivation tree should be the abstraction rule. However the weakening and the conversion rules of a pure type system can always be applied at any point in the derivation. This means that in the case of the abstraction, the last rule in the derivation tree might be either the abstraction, or the weakening or the conversion rules. The last rule is not determined by the shape of the term. If the term constructor together with the context determine the last rule to be applied then we can build the derivation tree of the term. A set of rules is called *syntax directed* if it has this property: the last rule in the derivation of the type of a term is determined by the structure of the term and of the context. The rules for pure type systems are not syntax directed since the last rule in the derivation can be the conversion or the weakening rules besides the corresponding structural rule. In order to make these sets of rules syntax directed, we should remove the non-structural rules (like the weakening and the conversion ones) and keep only the structural rules (the ones for term constructors). The system obtained by eliminating the non-structural rules should be equivalent to the original one. Therefore, the weakening rule is not removed but restricted to variables or constants and eventhough the conversion rule is removed, reduction or conversion is needed in the premises of almost all the rules. The equivalence (soundness and completeness) between the syntax directed set of rules and the original one for pure type systems is not easy to be proved. In order to be able to prove soundness and completeness we do not check the Π-condition in the same system but in a weaker one. The weaker system is the pure type system without the Π-condition.

First we define the weak head β-reduction.

Definition 10.7.1. The weak head β-reduction is defined by the following rules:

$$(\lambda x{:}A.\ b)a \rightarrow_\beta^{wh} b[x := a]$$

$$\frac{F \rightarrow_\beta^{wh} F'}{(F\ a) \rightarrow_\beta^{wh} (F'\ a)}$$

Note that this reduction is not closed under the compatibility rules, i.e. it is not true that $C[a] \rightarrow_\beta^{wh} C[b]$ if $a \rightarrow_\beta^{wh} b$.

We define a sytem $\lambda_{sd}^\omega(S)$ whose rules are syntax directed and is proved to be 'equivalent' to $\lambda^\omega(S)$. We denote that $? \vdash A : C$ and $C \twoheadrightarrow B$ by $? \vdash A :\twoheadrightarrow B$ for an arbitrary typing relation $\vdash$ and a rewrite relation $\rightarrow$.

Definition 10.7.2.

The functor $\lambda^{\omega}_{sd} : \mathbf{Spec} \to \mathbf{Carst}_{\nu\omega}$ is defined as $\lambda^{\omega}_{sd}(S) = (\mathcal{T}, \mathcal{C}, \to_{\beta}, \vdash^{\omega}_{sd})$ for $S \in \mathbf{Spec}$. The sets $\mathcal{T}$ and $\mathcal{C}$ and the relation $\to_{\beta}$ are as in definition 9.3.14. The typing relation $\vdash^{\omega}_{sd}$ is defined as the smallest relation closed under the following rules.

$$(axiom) \quad \epsilon \vdash^{\omega}_{sd} c : s \qquad \text{for } (c, s) \in \boldsymbol{A}$$

$$(start) \quad \frac{? \vdash^{\omega}_{sd} A :\twoheadrightarrow_{\beta} s}{?, x : A \vdash^{\omega}_{sd} x : A} \qquad \text{where } x \text{ is } ?\text{-fresh}$$

$$(weakening) \quad \frac{? \vdash^{\omega}_{sd} b : B \quad ? \vdash^{\omega}_{sd} A :\twoheadrightarrow_{\beta} s}{?, x : A \vdash^{\omega}_{sd} b : B} \qquad \text{where } x \text{ is } ?\text{-fresh and } b \in \mathbf{C} \cup V$$

$$(formation) \quad \frac{? \vdash^{\omega}_{sd} A :\twoheadrightarrow_{\beta} s_1 \quad ?, x : A \vdash^{\omega}_{sd} B :\twoheadrightarrow_{\beta} s_2}{? \vdash^{\omega}_{sd} (\Pi x{:}A.B) : s_3} \qquad \text{for } (s_1, s_2, s_3) \in \boldsymbol{R}$$

$$(abstraction) \quad \frac{?, x : A \vdash^{\omega}_{sd} b : B}{? \vdash^{\omega}_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

$$(application) \quad \frac{? \vdash^{\omega}_{sd} b :\twoheadrightarrow^{wh}_{\beta} (\Pi x{:}A.\ B) \quad ? \vdash^{\omega}_{sd} a : A'}{? \vdash^{\omega}_{sd} (b\ \ a) : B[x := a]} \qquad A \twoheadleftrightarrow_{\beta} A'$$

where $s \in \boldsymbol{S}$.

Since the conversion rule is removed, reduction or conversion is needed in the premises of almost all the rules. In the start, the weakening and the product rules, the types are reduced to some sort. In the application rule, the type of F is reduced to $(\Pi x{:}A.\ B)$ and the type A' of a should be convertible to A.

Note that these rules are syntax directed if the specification is singly sorted. If the specification is not singly sorted, given a context $?$ and a term b, the type of b may not be unique and the last rule to be applied in the derivation of the type of b is not determined by the shape of b and $?$.

Note that these systems verify only very weak subject reduction (see definition 4.3.1).

We prove that the system $\lambda^{\omega}_{sd}(S)$ is 'equivalent' to $\lambda^{\omega}(S)$ (soundness and completeness).

Lemma 10.7.3. If $? \vdash^{\omega} A : D$ and $D \twoheadrightarrow_{\beta} s$ then $? \vdash^{\omega} A : s$.

Proof: Suppose that $? \vdash^{\omega} D : s'$ for some s'. By the weak subject reduction theorem, $? \vdash s : s'$ and applying the conversion rule, we obtain $? \vdash A : s$.

If D is a toptype, we have that $D = s$. $\square$

Theorem 10.7.4. (Soundness)

If $? \vdash^{\omega}_{sd} a : A$ then $? \vdash^{\omega} a : A$.

Proof: The proof proceeds by induction on the derivation of $? \vdash^{\omega}_{sd} a : A$. Only two cases are considered.

- (product) $\dfrac{? \vdash^{\omega}_{sd} A :\twoheadrightarrow_{\beta} s_1 \quad ?, x : A \vdash^{\omega}_{sd} B :\twoheadrightarrow_{\beta} s_2}{? \vdash^{\omega}_{sd} (\Pi x{:}A.B) : s_3}$ for $(s_1, s_2, s_3) \in \boldsymbol{R}$.

  By induction hypothesis and lemma 10.7.3, we have $? \vdash^{\omega} A : s_1$ and $?, x : A \vdash^{\omega} B : s_2$. Hence $? \vdash^{\omega} (\Pi x{:}A.B) : s_3$.

- (application) $\dfrac{? \vdash^{\omega}_{sd} b :\twoheadrightarrow^{wh}_{\beta} (\Pi x{:}A.\ B) \quad ? \vdash^{\omega}_{sd} a : A'}{? \vdash^{\omega}_{sd} (b\ \ a) : B[x := a]}$ with $A \twoheadleftrightarrow_{\beta} A'$.

  By induction hypothesis, we have that $? \vdash^{\omega} b : D$ and $D \twoheadrightarrow^{wh}_{\beta} (\Pi x{:}A.\ B)$. Also $? \vdash^{\omega} a : A'$. There are two cases.

  1. Suppose that $? \vdash^{\omega} D : s$ for some s. By weak subject reduction, we have that $? \vdash^{\omega} (\Pi x{:}A.\ B) : s$. Generation lemma yields $? \vdash^{\omega} A : s'$ for some s'. Hence $? \vdash^{\omega} a : A$ and so $? \vdash^{\omega} (b\ \ a) : B[x := a]$.
  2. Suppose that D is a toptype in $?$. By the description of toptypes theorem, we have that $D \in \mathcal{M}_{\Gamma}$ and so $D = \Pi x{:}A''.\ B''$ for some A'' and B''. The weak head normal form of D is D itself, so $A = A''$ and $B'' = B$.

□

The proof of completeness is straightforward.

Theorem 10.7.5. (Completeness)
If $? \vdash^{\omega} a : A$ then there exists A' such that $? \vdash^{\omega}_{sd} a : A'$ and $A \twoheadleftrightarrow_{\beta} A'$.

Next we define a system $\lambda_{sd}(S)$ whose rules are syntax directed. We prove that if S is singly sorted then $\lambda_{sd}(S)$ is 'equivalent' to $\lambda(S)$. The systems $\lambda_{sd}(S)$ and $\lambda^{\omega}_{sd}(S)$ differ only in the abstraction rule. The abstraction rule in $\lambda^{\omega}_{sd}(S)$ does not contain the Π-condition and the one in $\lambda_{sd}(S)$ does. The Π-condition of $\lambda_{sd}(S)$ is not checked in the same system $\lambda_{sd}(S)$ but in $\lambda^{\omega}_{sd}(S)$. The idea is to use an auxiliary system to check for the Π-condition [BJMP93] so that we can prove the 'equivalence' between $\lambda_{sd}(S)$ and $\lambda(S)$. In our case, the auxiliary system is the corresponding pure type system without the Π-condition.

Definition 10.7.6.
The functor $\lambda_{sd} : \mathbf{Spec} \to \mathbf{Carst}_{\nu\omega}$ is defined by $\lambda_{sd}(S) = (\mathcal{T}, \mathcal{C}, \to_{\beta}, \vdash_{sd})$ for $S \in \mathbf{Spec}$. The sets $\mathcal{T}$ and $\mathcal{C}$ and the relation $\to_{\beta}$ are as in definition 9.3.14. The typing relation $\vdash_{sd}$ is the smallest relation closed under the same rules as in definition 10.7.2 except that the abstraction rule is replaced by the following one.

$$(abstraction)\quad \frac{?, x : A \vdash_{sd} b : B \quad ? \vdash^{\omega}_{sd} (\Pi x{:}A.\ B) :\twoheadrightarrow_{\beta} s}{? \vdash_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

where $s \in \boldsymbol{S}$.

Observe that in the abstraction rule we have the requirement that $(\Pi x{:}A.\ B)$ should be typable in $\lambda^{\omega}_{sd}(S)$.

Note that these systems verify only very weak subject reduction (see definition 4.3.1).

Theorem 10.7.7. (Soundness) Let S be a singly sorted specification.
If $? \vdash_{sd} a : A$ then $? \vdash a : A$.

Proof: We prove only the case of the abstraction rule.

$$(\text{abstraction})\ \frac{?, x : A \vdash_{sd} b : B \quad ? \vdash^{\omega}_{sd} (\Pi x{:}A.\ B) :\twoheadrightarrow s}{? \vdash_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

By induction hypothesis we have that $?, x{:}A \vdash b : B$. By lemma 10.5.5, we have that $\varphi(?) = ?$, $\varphi(A) = A$, $\varphi(b) = b$ and $\varphi(B) = B$. By theorem 10.7.4 and lemma 10.7.3 we have that $? \vdash^{\omega} (\Pi x{:}A.\ B) : s$. By theorem 10.5.9 we have that $? \vdash (\Pi x{:}A.\ B) : s$. □

Theorem 10.7.8. (Completeness) If $? \vdash a : A$ then there exists A' such that $? \vdash_{sd} a : A'$ and $A \twoheadleftrightarrow_{\beta} A'$.

Proof: We prove only the case of the abstraction rule.

$$(\text{abstraction})\ \frac{?, x : A \vdash b : B \quad ? \vdash (\Pi x{:}A.\ B) : s}{? \vdash (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

By induction hypothesis we have that $?, x{:}A \vdash_{sd} b : B'$ for some B' such that $B \twoheadleftrightarrow_{\beta} B'$. By the previous theorem (soundness) we have that $?, x{:}A \vdash b : B'$.

If $? \vdash (\Pi x{:}A.\ B) : s$ then there exists s_1 and s_2 such that $(s_1, s_2, s) \in \boldsymbol{R}$, $? \vdash A : s_1$ and $?, x{:}A \vdash B : s_2$. By correctness and unicity of types we have that $?, x{:}A \vdash B' : s_2$. Hence $? \vdash (\Pi x{:}A.\ B') : s$. By theorem 10.7.5 we have that $? \vdash^{\omega}_{sd} (\Pi x{:}A.\ B') : D$ and $D \twoheadrightarrow_{\beta} s$. □

10.8 Type Inference

In this section we define a semi-algorithm of type inference for the class of singly sorted pure type systems. A semi-algorithm of type inference is a partial function or program that terminates and yields the type of a term if the term is typable and it may not terminate otherwise.

We cannot expect to find a terminating algorithm for the class of singly sorted pure type systems since typability (and also conversion) for some non-normalising pure type systems is not decidable (see theorem 9.3.20). Hence we define a type inference semi-algorithm, for the class of singly sorted pure type systems, including the non-normalising ones.

In order to define such a semi-algorithm, we use the syntax directed set of rules presented in section 10.7. Although those rules are syntax directed when the specification is singly sorted, they are not yet deterministic. There are several conditions, called *side conditions*, that should be solved. For these conditions we have to verify if some element

belongs to some of the sets in a specification, or we have to perform β-reduction or check if two types are β-convertible.

If the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ of the specification are not recursively enumerable (recursive) then the conditions $s \in \boldsymbol{S}$, $(s_1, s_2) \in \boldsymbol{A}$ and $(s_1, s_2, s_3) \in \boldsymbol{R}$ are not semi-decidable (decidable). In order to have a semi-algorithm we have to assume that these sets are recursively enumerable.

There are several ways of β-reducing or checking β-conversion. We should specify in which way we reduce and how we check if two types are convertible. We write semi-algorithms or partial functions to compute the weak head normal form and to check for β-conversion. The first one computes the weak head normal form if it exists and it may not terminate otherwise. The second one is a common-reduct strategy. The termination of the semi-algorithm of type inference depends on the termination of these two functions. Therefore, in order to reduce the cases of non-termination of the type inference semi-algorithm, it is sufficient to reduce the cases of non-termination in the semi-algorithm that computes the weak head normal form and in the common-reduct strategy.

Reducing to a sort or to a product. In the cases of the start, weakening and product rules the types are reduced to a sort. In case the term is typable, we know that the weak head normal form exists and it is a sort. In the application rule, the type of the operator is a product $(\Pi x{:}A.\ B)$ also a weak head normal form.

A weak head normal form can be either an abstraction $(\lambda x{:}A.\ b)$, or a product $(\Pi x{:}A.\ B)$, or an application $(b\ a_1\ \ldots a_n)$ where b is a sort or a variable or a product.

Now we write a function that computes the weak head β-normal form if it exists.

Definition 10.8.1. A function $\mathbf{whnf} : \mathcal{T} \to \mathcal{T}$ is defined as follows .

$$\begin{aligned} \mathbf{whnf}(a) &= a \text{ if } a \text{ is in weak head normal form} \\ \mathbf{whnf}((\lambda x{:}A.\ b)\ a\ d_1\ \ldots d_n) &= \mathbf{whnf}(b[x := a]d_1 \ldots d_n) \end{aligned}$$

This function is a semi-algorithm, i.e. it may not terminate only in case the term is not weak head normalising.

Lemma 10.8.2. Let $a \in \mathcal{T}$ be weak head normalising. Then $\mathbf{whnf}(a)$ is the weak head normal form of a.

This lemma is proved by induction on the number of steps of the leftmost reduction to normal form.

Checking β-conversion. In the application rule we have to check if two types are β-convertible. It would be sufficient to find a computable common-reduct strategy $F : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ and then check if $F(a, b) \neq \emptyset$.

The strategy presented in [Coq91] (see also [Mag94]) reduces as less as possible by performing weak head reduction. The idea is to compute the weak head normal forms of

the terms and compare their heads. In spite of being efficient, it gives a set of common-reducts only if both terms are normalising. It might happen that $a \twoheadleftrightarrow_\beta b$ and $\mathbf{F^n}(a,b)$ does not terminate if a or b are not normalising.

Definition 10.8.3. We define $\mathbf{F^n} : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ as follows.

$$\begin{array}{rcl}
\mathbf{F^n}(d,d') & = & \{d\} \text{ if } d = d' \\[2ex]
\mathbf{F^n}((\lambda x{:}A.\ b),(\lambda x{:}A'.\ b')) & = & \{(\lambda x{:}A''.\ b'') \mid A'' \in \mathbf{F^n}(A,A') \ \ \& \ \ b'' \in \mathbf{F^n}(b,b')\} \\[2ex]
\mathbf{F^n}((\Pi x{:}A.\ B),(\Pi x{:}A'.\ B')) & = & \{(\Pi x{:}A''.\ B'') \mid A'' \in \mathbf{F^n}(A,A') \ \ \& \ \ B'' \in \mathbf{F^n}(B,B')\} \\[2ex]
\mathbf{F^n}((x\ a_1\ \ldots a_n),(x\ a'_1\ \ldots a'_n)) & = & \{((x\ a''_1\ \ldots a''_n) \mid a''_i \in \mathbf{F^n}(a_i,a'_i)\} \\[2ex]
\mathbf{F^n}(d,d') & = & \mathbf{F^n}(\mathbf{whnf}(d),\mathbf{whnf}(d')) \\
 & & \text{if } d \text{ or } d' \text{ are not in weak head normal form} \\[2ex]
\mathbf{F^n}(d,d') & = & \emptyset \qquad \text{otherwise}
\end{array}$$

The next lemma says that the function $\mathbf{F^n}$ is a common-reduct strategy only for normalising pseudoterms, i.e. for all $a, b \in \mathcal{T}$ that are normalising, for all $c \in \mathbf{F^n}(a,b)$ we have that $a \twoheadrightarrow_\beta c$ and $b \twoheadrightarrow_\beta c$.

Lemma 10.8.4. Let a and b be weakly normalising. Then

1. $\mathbf{F^n}(a,b)$ terminates.
2. For all $c \in \mathbf{F^n}(a,b)$, we have that $a \twoheadrightarrow_\beta c$ and $b \twoheadrightarrow_\beta c$.
3. $a \twoheadleftrightarrow_\beta b$ if and only if $\mathbf{F^n}(a,b) \neq \emptyset$.

All the parts of the previous lemma are proved by induction on $(\mathbf{l}(a)+\mathbf{l}(b), \mathbf{n}(a)+\mathbf{n}(b))$ where $\mathbf{l}$ computes the number of steps of the leftmost reduction to normal form and $\mathbf{n}$ computes the number of symbols.

This strategy may not terminate when one of the terms is not weakly normalising. For example, suppose that $\Omega = (\lambda x{:}A.\ x\ x)(\lambda x{:}A.\ x\ x)$. Then $\mathbf{F^n}(b,\Omega)$ does not terminate for any $b \in \mathcal{T}$.

This common-reduct strategy is not satisfactory because we do not want to restrict our type inference procedure to normalising pure type systems.

Now we define three common-reduct strategies that work also for non-normalising pseudoterms and hence for non-normalising pure type systems. The three compute the bounded graphs of the terms (see also section 2.5) step by step. We compare these three strategies according to their order and the number of cases they do not terminate. The third strategy is the best one for having a best order and least cases of non-termination.

The first strategy is called **F** and it always computes the bounded reduction graphs of the terms step by step. In each step, we have to check only if the 'new part' of the graph has a common element with the other set. This strategy terminates and gives at least one common reduct if both terms are β-convertible and it may not terminate otherwise.

Definition 10.8.5. We define $\mathbf{F} : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ as follows.

$$\begin{array}{lll} \mathbf{F}(a,b) & = & \\ & & \text{If} \quad a = b \ \text{ then} \\ & & | \quad \{\text{a}\} \\ & & \text{else} \\ & & | \quad \mathbf{H}(\{a\},\{b\}) \\ & & \text{end} \end{array}$$

Next we define the function $\mathbf{H} : \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$. We suppose that it is always applied to subsets X, Y of $\mathcal{T}$ that verify the following preconditions.

1. X and Y are the n and m-bounded reduction graphs of a and b, i.e. they are of the form
$$\mathcal{G}^{<n}_{\to_\beta}(a) = \{d \mid a \twoheadrightarrow_\beta d \text{ in less than } n \text{ steps}\}$$
and
$$\mathcal{G}^{<m}_{\to_\beta}(b) = \{d \mid a \twoheadrightarrow_\beta d \text{ in less than } m \text{ steps}\}$$
2. The intersection of X and Y is empty.
3. $0 \leq n - m \leq 1$.

We only have to check if $\mathcal{G}_{\to_\beta}(X)$ has elements in common with Y. If it does, $\mathcal{G}_{\to_\beta}(X) \cap Y$ are the common-reducts we are looking for.

$$\begin{array}{lll} \mathbf{H}(X,Y) & = & \\ & & \text{If} \quad \mathcal{G}_{\to_\beta}(X) \cap Y \neq \emptyset \ \text{ then} \\ & & | \quad \mathcal{G}_{\to_\beta}(X) \cap Y \\ & & \text{else} \\ & & | \quad \mathbf{H}(Y, X \cup \mathcal{G}_{\to_\beta}(X)) \\ & & \text{end} \end{array}$$

Note that Y takes the role of X in the else-part.
In the following lemma we prove that the function **F** is a common-reduct strategy.

Lemma 10.8.6.

1. If $a \twoheadleftrightarrow_\beta b$ then $\mathbf{F}(a,b)$ terminates and yields a non-empty set that verifies that for all $c \in \mathbf{F}(a,b)$ we have that $a \twoheadrightarrow_\beta c$ and $b \twoheadrightarrow_\beta c$.

2. If $\mathbf{F}(a,b) \neq \emptyset$ then $a \twoheadleftarrow\twoheadrightarrow_\beta b$.

Proof:

1. If $a \twoheadleftarrow\twoheadrightarrow_\beta b$ then there exists m and n such that $0 \leq m-n \leq 1$ and $\mathcal{G}^{\leq n}_{\rightarrow_\beta}(a) \cap \mathcal{G}^{\leq m}_{\rightarrow_\beta}(b) \neq \emptyset$. Hence $\mathbf{F}(a,b)$ terminates and yields a non-empty set of common-reducts.

2. Suppose $\mathbf{F}(a,b) \neq \emptyset$. Note that $\mathbf{F}(a,b)$ yields a set of common reducts for a and b. Hence $a \twoheadleftarrow\twoheadrightarrow_\beta b$.

□

This strategy presents some drawbacks, the order and the cases of non-termination. Firstly, its order is clearly exponential since we compute the bounded reduction graphs of the terms. Secondly, this function does not terminate in cases the terms are not convertible but normalising. For example, the terms $(K\ x\ \Omega)$ and y are not convertible and $\mathbf{F}((K\ x\ \Omega), y)$ does not terminate.

The next two common-reduct strategies we present try to improve these two features, order and cases of non-termination, by avoiding the computation of the bounded graphs and by reducing the cases of non-termination.

A first improvement in this direction can be done if we use a normalising strategy. In the case we find a normal form in one of the bounded graphs, we stop computing them. Supposing the terms were convertible and one of them normalising then the other term should also be normalising. Here we could apply a normalising strategy to the second term. We first write a function **nf** that finds the normal form (if it exists) by reducing the spine redexes (see [BKKS87]).

Definition 10.8.7. A function $\mathbf{nf} : \mathcal{T} \rightarrow \mathcal{T}$ is defined as follows.

$$\begin{aligned}
\mathbf{nf}(\lambda x{:}A.\ b) &= (\lambda x{:}\mathbf{nf}(A).\ \mathbf{nf}(b)) \\
\mathbf{nf}(\Pi x{:}A.\ B) &= (\Pi x{:}\mathbf{nf}(A).\ \mathbf{nf}(B)) \\
\mathbf{nf}(x\ a_1\ \ldots a_n) &= (x\ \mathbf{nf}(a_1)\ldots\mathbf{nf}(a_n)) \\
\mathbf{nf}((\lambda x{:}A.\ b)\ a_1\ \ldots a_n) &= \mathbf{nf}(b[x := a_1]\ldots a_n) \\
\mathbf{nf}((\Pi x{:}A.\ B)\ a_1\ \ldots a_n) &= (\Pi x{:}\mathbf{nf}(A).\ \mathbf{nf}(B))\mathbf{nf}(a_1)\ldots\mathbf{nf}(a_n)
\end{aligned}$$

The function **nf** reduce all the spine redexes at the same time. It is a semi-algorithm, i.e. it computes the normal form if the term is weakly normalising and it may not terminate otherwise.

Lemma 10.8.8. Let $a \in \mathcal{T}$ be weakly normalising. Then $\mathbf{nf}(a)$ is the normal form of a.

The second common-reduct strategy is called $\mathbf{F}^+$ and applies the function $\mathbf{nf}$ to one of the terms when it finds that the other is weakly normalising.

Definition 10.8.9. We define $\mathbf{F}^+ : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ as follows.

```
F+(a, b)  =
        If    a = b  then
        |     { a }
        else
              If    a or b are in normal form  then
                    If    nf(a) = nf(b)  then
                    |     {nf(a)}
              |     else
        |           |     ∅
                    end
              else
              |     H+({a}, {b})
              end
        end
```

Next we define the function $\mathbf{H}^+ : \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$. We suppose that it is always applied to subsets X, Y of $\mathcal{T}$ that verify the following preconditions.

1. X and Y are the n and m-bounded reduction graphs of a and b.
2. The intersection of X and Y is empty.
3. $0 \leq n - m \leq 1$.
4. X and Y do not contain any normal form.

If we find that $\mathcal{G}_{\to}(X)$ contains a normal form then we do not go on computing the bounded graph. We choose one element of Y and reduce it to normal form.

```
H+(X, Y)  =
        If    G→β(X) ∩ Y ≠ ∅  then
        |     G→β(X) ∩ Y
        else
              If    there exists a ∈ G→β(X) in normal form  then
                    If    for some b ∈ Y, nf(b) = a  then
                    |     {a}
              |     else
        |           |     ∅
                    end
              else
              |     H+(Y, X ∪ G→β(X))
              end
        end
```

In the following lemma we prove that $\mathbf{F}^+$ is a common-reduct strategy. The proof is similar to the one of lemma 10.8.6.

Lemma 10.8.10.

1. If $a \twoheadleftarrow\twoheadrightarrow_\beta b$ then $\mathbf{F}^+(a,b)$ terminates and yields a non-empty set that verifies that for all $c \in \mathbf{F}^+(a,b)$ we have that $a \twoheadrightarrow_\beta c$ and $b \twoheadrightarrow_\beta c$.

2. If $\mathbf{F}^+(a,b) \neq \emptyset$ then $a \twoheadleftarrow\twoheadrightarrow_\beta b$.

In a sense the strategy $\mathbf{F}^+$ is 'worse' than $\mathbf{F}$ since in $\mathbf{F}^+$ the operation **nf** that computes the normal form can consume a great amount of time and space. One can also argue that $\mathbf{F}^+$ is 'better' than $\mathbf{F}$ for having a better order and less cases of non-termination.

On one hand, in some cases the order of $\mathbf{F}^+$ is less than the order of $\mathbf{F}$. The function $\mathbf{F}$ always computes the bounded graph of the terms in a silly way. While $\mathbf{F}^+$ stops computing the bounded reduction graph if one of the terms is weakly normalising. On the other hand, there are cases in which $\mathbf{F}$ does not terminate while $\mathbf{F}^+$ does. If the terms are not convertible and normalising then $\mathbf{F}^+(a,b)$ terminates and yields the empty set. However $\mathbf{F}$ does not terminate in some of these cases, like we have shown before.

We define a third common-reduct strategy that uses weak head reduction. This strategy is the best amongst the ones we presented for having the best order and least cases of non-termination.

Definition 10.8.11. We define $\mathbf{F}^{++} : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ as follows.

$$
\begin{array}{lll}
\mathbf{F}^{++}(a,b) & = & \\
 & & \text{If} \quad a = b \quad \text{then} \\
 & & | \quad \{a\} \\
 & & \text{else} \\
 & & \quad \text{If} \quad a \text{ or } b \text{ are in weak head normal form} \quad \text{then} \\
 & & \quad | \quad \mathbf{L}(\mathbf{whnf}(a), \mathbf{whnf}(b)) \\
 & & | \quad \text{else} \\
 & & \quad | \quad \mathbf{H}^{++}(\{a\},\{b\}) \\
 & & \quad \text{end} \\
 & & \text{end}
\end{array}
$$

Note that $\mathbf{whnf}(a)$ and $\mathbf{whnf}(b)$ may not terminate. If they both terminate, the function $\mathbf{L}$ is applied only to weak head normal forms. We define the function $\mathbf{L} : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$ as follows. In case both terms are weak head normal forms and their heads are equal, we try to find a common-reduct for their subterms. In any other case, this function yields the empty set.

$$
\begin{array}{rcl}
\mathbf{L}((\lambda x{:}A_1.\ b_1),(\lambda x{:}A_2.\ b_2)) & = & \{(\lambda x{:}A_3.\ b_3) \mid A_3 \in \mathbf{F}^{++}(A_1,A_2)\ \ \&\ \ b_3 \in \mathbf{F}^{++}(b_1,b_2)\} \\
\mathbf{L}((\Pi x{:}A_1.\ b_1),(\Pi x{:}A_2.\ b_2)) & = & \{(\Pi x{:}A_3.\ b_3) \mid A_3 \in \mathbf{F}^{++}(A_1,A_2)\ \&\ b_3 \in \mathbf{F}^{++}(b_1,b_2)\} \\
\mathbf{L}((x\ a_1 \ldots a_n),(x\ b_1 \ldots b_n)) & = & \{((x\ c_1 \ldots c_n) \mid\ \forall\ i = 1,n\ c_i \in \mathbf{F}^{++}(a_i,b_i)\} \\
\mathbf{L}(a,b) & = & \emptyset \qquad \text{otherwise}
\end{array}
$$

The function $\mathbf{H}^{++} : \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$ is always applied to subsets X and Y of $\mathcal{T}$ that verify the following preconditions.

1. X and Y are the n and m-bounded reduction graphs of a and b.
2. The intersection of X and Y is empty.
3. $0 \leq n - m \leq 1$.
4. X and Y do not contain any weak head normal form.

If the bounded graphs X contains a weak head normal form, we choose a b of Y and reduce it to weak head normal form.

$$
\begin{array}{rl}
\mathbf{H}^{++}(X,Y) & = \\
& \text{If} \quad \mathcal{G}_{\to_\beta}(X) \cap Y \neq \emptyset \ \ \text{then} \\
& \mid \quad \mathcal{G}_{\to_\beta}(X) \cap Y \\
& \text{else} \\
& \qquad \text{If} \quad \text{there exists } a \in \mathcal{G}_{\to_\beta}(X) \text{ in weak head normal form} \quad \text{then} \\
& \qquad \mid \quad \text{Choose } b \in Y,\ \mathbf{L}(a, \mathbf{whnf}(b)) \\
& \mid \quad \text{else} \\
& \qquad \mid \quad \mathbf{H}^{++}(Y, X \cup \mathcal{G}_{\to_\beta}(X)) \\
& \qquad \text{end} \\
& \text{end}
\end{array}
$$

In the following lemma we prove that the function $\mathbf{F}^{++}$ is a common-reduct strategy.

Lemma 10.8.12.

1. If $a \twoheadleftrightarrow_\beta b$ then $\mathbf{F}^{++}(a,b)$ terminates and yields a non-empty set that verifies that for all $c \in \mathbf{F}^{++}(a,b)$ we have that $a \twoheadrightarrow_\beta c$ and $b \twoheadrightarrow_\beta c$.
2. If $\mathbf{F}^{++}(a,b) \neq \emptyset$ then $a \twoheadleftrightarrow_\beta b$.

The proof of lemma is similar to the proof of lemma 10.8.6.

The strategy $\mathbf{F}^{++}$ is better than $\mathbf{F}^{+}$ (and better than $\mathbf{F}$) again for having a better order and less cases of non-termination. On one hand, in some cases the order of $\mathbf{F}^{++}$ is less than the order of $\mathbf{F}$. The function $\mathbf{F}^{++}$ avoids constructing the bounded graphs in more cases than $\mathbf{F}^{+}$. This is very important since the bounded graphs grow exponentially. If we find that one of the terms is weak head normalising, we try to reduce the other to weak head normal form. On the other hand, the function $\mathbf{F}^{++}$ terminates in cases that $\mathbf{F}^{+}$ does not. More precisely, the strategy $\mathbf{F}^{++}$ terminates and yields the empty set when the terms are not normalising but they are both weak head normalising and their heads are different. The strategy $\mathbf{F}^{+}$ does not terminate in some of these cases like in for example $\mathbf{F}^{+}((\lambda x{:}A.\ \Omega), (\Pi x{:}B.\ x))$. Many of these cases can appear when we have recursion.

In table 10.1, we compare the strategies defined before by answering the question: does the strategy terminate?.

Reduction behaviour of the terms a and b			$\mathbf{F^n}$	$\mathbf{F}$	$\mathbf{F^+}$	$\mathbf{F^{++}}$
$a \twoheadleftarrow\!\twoheadrightarrow b$	a , b are WN		Yes	Yes	Yes	Yes
	Otherwise		May not	Yes	Yes	Yes
Otherwise	a , b are WN		Yes	May not	Yes	Yes
	Otherwise	a , b are WN with $\neq$ heads	Yes	May not	May not	Yes
		Otherwise	May not	May not	May Not	May Not

Table 10.1: Termination of the Strategies

Semi-algorithm of type inference. Now we define a function **type** that computes the type of a term (up to β-conversion) in a singly sorted pure type system. If a is typable in ? in a singly sorted pure type system then $\mathbf{type}(?, a)$ terminates and yields the type of a in ? (up to β-conversion), i.e. if $? \vdash a : A$ then $\mathbf{type}(?, a) \twoheadleftarrow\!\twoheadrightarrow_\beta A$. If the term a is not typable in ? then $\mathbf{type}(?, a)$ either yields $-$ or it does not terminate.

This function is obtained from the syntax directed set of rules defined in section 10.7 for pure type systems. For each rule, we write a case of 'pattern matching'. The first case in this definition is for the axiom rule. The following three cases correspond to the start , the weakening and the product rules. The conditions that appear in these rules that are of the form '$? \vdash A :\twoheadrightarrow_\beta s$' are replaced by '$\mathbf{whnf}(\mathbf{type}(?, A)) = s$'. The fifth case corresponds to the abstraction rule. Here we need to define an auxiliary function $\mathbf{type}^\omega$ to compute the type in a pure type system without the Π-condition. The condition that appears in this rule, '$? \vdash^\omega (\Pi x{:}A.\ B) :\twoheadrightarrow_\beta s$' is replaced by '$\mathbf{whnf}(\mathbf{type}^\omega(?, (\Pi x{:}A.\ B))) = s$'. The last case corresponds to the application rule. The condition '$? \vdash b :\twoheadrightarrow_\beta (\Pi x{:}A.\ B)$' that appears in this rule is replaced by '$\mathbf{whnf}(\mathbf{type}(?, b)) = (\Pi x{:}A.\ B)$'. The other condition in this same rule '$A \twoheadleftarrow\!\twoheadrightarrow_\beta A'$' is replaced by $\mathbf{F}^{++}(A, A') \neq \emptyset$.

Definition 10.8.13. The function $\mathbf{type}_S : \mathcal{C} \times \mathcal{T} \to \mathcal{T}_-$ (or just **type**) is defined as follows.

$$
\begin{array}{lcll}
\mathbf{type}(\epsilon, s) & = & s' & \text{if } (s, s') \in \boldsymbol{A} \\[2ex]
\mathbf{type}(<?, x : A >, x) & = & A & \text{if } \mathbf{whnf}(\mathbf{type}(?, A)) = s \in \boldsymbol{S} \\
 & & & \text{ and } x \text{ is } ?\text{-fresh} \\[2ex]
\mathbf{type}(<?, x : A >, b) & = & \mathbf{type}(?, b) & \text{if } b \in \mathbf{C} \cup V,\ b \neq x\ x \text{ is } ?\text{-fresh and} \\
 & & & \mathbf{whnf}(\mathbf{type}(?, A)) = s \in \boldsymbol{S} \\[2ex]
\mathbf{type}(?, (\Pi x{:}A.B)) & = & s_3 & \text{if } \mathbf{whnf}(\mathbf{type}(?, A)) = s_1, \\
 & & & \mathbf{whnf}(\mathbf{type}(<?, x : A >, B)) = s_2, \\
 & & & \text{and } (s_1, s_2, s_3) \in \boldsymbol{R} \\[2ex]
\mathbf{type}(?, (\lambda x{:}A.\ b)) & = & (\Pi x{:}A.\ B) & \text{if } \mathbf{type}(<?, x : A >, b) = B \text{ and} \\
 & & & \mathbf{whnf}(\mathbf{type}^{\omega}(?, \Pi x{:}A.\ B)) = s \in \boldsymbol{S} \\[2ex]
\mathbf{type}(?, (b\ \ a)) & = & B[x := a] & \text{if } \mathbf{whnf}(\mathbf{type}(?, b)) = (\Pi x{:}A.\ B), \\
 & & & \mathbf{type}(?, a) = A' \text{ and } \mathbf{F}^{++}(A, A') \neq \emptyset \\[2ex]
\mathbf{type}(?, a) & = & - & \text{otherwise}
\end{array}
$$

The function $\mathbf{type}^{\omega}_{S} : \mathcal{C} \times \mathcal{T} \to \mathcal{T}_{-}$ is defined exactly like **type** except for the case that corresponds to the abstraction rule. In that case, the condition $\mathbf{whnf}(\mathbf{type}^{\omega}(?, \Pi x{:}A.\ B)) = s \in \boldsymbol{S}$ is removed. The function $\mathbf{type}^{\omega}$ computes the type of a term in a pure type system without the Π-condition.

The first part of the following theorem says that if the term has a type in a singly sorted pure type system then this is computed by **type** (up to β-conversion). The second part says that the value $\mathbf{type}(?, a) \neq -$ is the type of a in $?$ in a singly sorted pure type system.

Theorem 10.8.14. (Correctness of 'type') Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursively enumerable.

1. If $? \vdash a : A$ then $\mathbf{type}(?, a)$ terminates and $\mathbf{type}(?, a) \twoheadleftrightarrow_{\beta} A$.
2. If $\mathbf{type}(?, a)$ terminates and yields A then $? \vdash a : A$.

Three functions are used in the definition of **type**: $\mathbf{type}^{\omega}$, **whnf** and $\mathbf{F}^{++}$. The function $\mathbf{type}^{\omega}$ computes the type of a term in a pure type system without the Π-condition. The other two functions are used to solve the side-conditions, **whnf** computes the weak head normal form of a term and the other $\mathbf{F}^{++}$ is a common-reduct strategy.

The termination of $\mathbf{type}^{\omega}$ and **type** depend on the termination of **whnf** and $\mathbf{F}^{++}$. If **whnf** or $\mathbf{F}^{++}$ do not terminate then neither do **type** and $\mathbf{type}^{\omega}$.

Type checking can be solved from type inference. Using the function **type** that infers the type of a term, we write another function **check** that checks if a term has certain type.

Definition 10.8.15. We define the function $\mathbf{check} : \mathcal{C} \times \mathcal{T} \times \mathcal{T} \to Bool$ as follows.

$$\mathbf{check}(?\,, a, A) \quad = \quad \begin{cases} true & \text{if } \mathbf{whnf}(\mathbf{type}(?\,, A)) = s \in \boldsymbol{S}, \\ & \mathbf{type}(?\,, a) \neq - \text{ and} \\ & \mathbf{F}^{++}(\mathbf{type}(?\,, a), A) \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

We have the following conclusion which has been proved before in [BJMP93] and [Pol96] (see theorem 9.3.19).

Theorem 10.8.16. (Decidability of Type Inference and Type Checking)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursive.
If $\lambda(S)$ is β-weakly normalising then type inference and type checking in $\lambda(S)$ are decidable.

Proof: Suppose that $\lambda(S)$ is normalising. Then **whnf** and $\mathbf{F}^{++}$ are applied to normalising terms and hence they terminate. Since the sets of the specification are recursive, we have that $\mathbf{type}^{\omega}$ terminates and so do **type** and **check**. Therefore type inference and type checking for $\lambda(S)$ are decidable. □

10.9 Conclusions and Related Work

Type Inference semi-algorithm. In order to solve the type inference problem for pure type systems we have first considered a syntax directed set of rules and then we have written a function that infers the type based on this syntax directed set of rules. In table 10.2 we illustrate our methods.

Π-condition	**Original type system**	**Type system with syntax directed rules**	**Type inference semi-algorithm**
included	$\lambda(S)$	$\lambda_{sd}(S)$	**type**
removed	$\lambda^{\omega}(S)$	$\lambda^{\omega}_{sd}(S)$	$\mathbf{type}^{\omega}$

Table 10.2: Type Inference Semi-algorithm

In the first column of the table we indicate if the Π-condition is included or removed, in the second one we give the names of the original type systems (with and without the Π-condition), in the third one we give the corresponding type systems with a syntax directed set of rules and finally the functions that infer the type in the original systems.

The definitions of **type** and $\mathbf{type}^{\omega}$ are based on the systems that appear next to them in the preceding column.

The Π-condition of $\lambda(S)$ is checked in the same system, whereas the one of $\lambda_{sd}(S)$ is checked in $\lambda^{\omega}_{sd}(S)$. Therefore in the definition of **type**, the Π-condition is checked using the function $\mathbf{type}^{\omega}$ and not **type**. In other words, we have to use $\mathbf{type}^{\omega}$ to define **type**.

The Π-condition of $\lambda_{sd}(S)$ is checked in $\lambda^{\omega}_{sd}(S)$ that is weaker than $\lambda_{sd}(S)$.

We have proved that if S is a singly sorted specification then $\lambda_{sd}(S)$ is 'equivalent' to $\lambda(S)$ by using the relations shown in the diagram below. (see theorems 10.7.7 and 10.7.8).

$$\begin{array}{cccc} & \lambda(S) & \cong & \lambda_{sd}(S) \\ \varphi & \cap & & \cap \\ & \lambda^{\omega}(S) & \cong & \lambda^{\omega}_{sd}(S) \end{array}$$

Related work. Several syntax directed sets of rules for pure type systems are studied in [BJMP93]. Our definition of a syntax directed set of rules follows the idea in [BJMP93] of using an auxiliary system to check for the Π-condition. In that paper, the auxiliary system is much weaker than $\lambda(S)$. In our case, the auxiliary system is $\lambda^{\omega}(S)$, which is very close to $\lambda(S)$. Moreover, $\lambda^{\omega}(S)$ preserves some properties of $\lambda(S)$ like normalisation.

In [Pol93a] a syntax directed set of rules for bijective pure type systems is presented. The class of bijective pure type systems includes all systems of the λ-cube and is a proper subclass of the class we study here, the class of singly sorted pure type systems. The class of bijective pure type systems does not include any of the systems of the family of AUTOMATH as described on page 216 and 217 in [Bar85].

Decidability of type checking for normalising pure type systems whose set of sorts is *finite* is proved in [BJ93]. In that paper, a type inference algorithm is defined that computes the normal form in all the rules. A discussion on the side-conditions can be found in [Pol96]. In this paper, decidability of type inference is proved for normalising pure type systems that are either singly sorted or semi-full under the assumption that the sets forming the specification are *recursive*. In theorem 10.8.16, we have given a new proof of the same result for the singly sorted but not for the semi-full pure type systems.

Concerning the problem with α-conversion, we define substitution for the set of pseudoterms as in [CF58]. Using this definition of substitution the variable convention in [Bar85] is not necessary. The typing rules for pure type systems do not allow to type terms unless their nested variables are all different. In the start and the weakening rules a variable is added to the context only if it is ?-fresh. The term $(\lambda x{:}A.\ \lambda x{:}B.\ x)$ is not typable because the second occurrence of x is not $< A : *, x{:}A >$-fresh. It is necessary to perform α-conversion to ensure subject reduction. In fact subject reduction holds up to α-conversion (see [Pol93b]). In the context $A : *$, the term $(\lambda y{:}(A \to A).\ \lambda x{:}A.\ y)(\lambda x{:}B.\ x)$ is typable in a pure type system. This term reduces to $(\lambda x : A.\ \lambda x : B.\ x)$ which is not typable. Subject reduction holds if we identify $(\lambda x{:}A.\ \lambda x{:}B.\ x)$ with $(\lambda z{:}A.\ \lambda x{:}B.\ x)$

A solution to the implementation of α-conversion appeared in [Bru72]. Here, reference numbers to the positions of the abstractions are used instead of name variables. Another solution can be found in [Coq96]. In this case, a semantical argument is used to prove the correctness of the type checking algorithm.

For pure type systems, the problem of unification does not arise when inferring the type. In typing à la Curry, since the types of the variables are unknown, we have to solve

the unification problem when checking and inferring the type. If we are checking whether an application $(f\ a)$ has a type B we do not know the type of the argument a. We could infer a type for a and a type for f. The terms f and a may have several types but we could infer the principal type, i.e. a type from which all others can be obtained by substitution. When we have inferred the principal types of f and a, we need to find a unifier for the type of f and a type whose domain is the type of a. The unification and type inference problems for normalising systems à la Curry are not always decidable.

Chapter 11

Pure Type Systems with Definitions

11.1 Introduction

A pure type system does not provide the possibility to introduce a definition, i.e. an abbreviation (name) for a larger term which can be used several times in a program or proof. A definition mechanism is essential for practical use, and indeed implementations of pure type systems such as Coq [Dow91], Lego [LP92] or Constructor [HA91] do provide such a facility, even though the formal definition of the systems they implement do not. In this chapter, we extend the pure type system to include (non-recursive) definitions.

The extension of a pure type system with definitions looks very harmless and this may not seem a topic worthy of investigation. However the local definitions complicate matters and it is an open problem whether extending an arbitrary pure type system with definitions preserves strong normalisation or not. Worse still, proving strong normalisation for particular pure type systems extended with definitions is already a problem. The strong normalisation proofs for particular type systems given in [Coq85], [Luo89], [GN91], [Bar92] cannot be extended in any obvious way.

In this chapter, we show how strong normalisation of a pure type system extended with definitions follows from strong normalisation of another (larger) pure type system. This enables us to prove that for all strongly normalising pure type systems that we know the extensions with definitions are also strongly normalising.

In the systems of the AUTOMATH family [NGdV94] definitions are considered as part of the formal language. The meta-theory of these systems -including strong normalisation - is treated in detail in [Daa80]. However, the proofs of strong normalisation apply only to the particular type system that they consider and do not extend to other type systems.

This chapter is organised as follows. In section 11.2 we define the pseudoterms and the pseudocontexts extended with definitions, the δ-reduction and the typing rules for definitions. In section 11.3 we prove properties for all pseudoterms like confluence for $\beta\delta$-reduction and strong normalisation for δ-reduction. In section 11.4 we prove properties for typable terms. We prove that weak normalisation is preserved by the extension. This

property is easy to prove. We first unfold all the definitions and then we perform β-reduction. The unfolding of definitions may be inefficient and we want to perform other strategies for $\beta\delta$. Hence we prove that strong normalisation is preserved by the extension for a class of pure type systems.

11.2 Pure Type Systems with Definitions

We define a functor λ^δ from the category of specifications to the category of environmental abstract rewriting systems with typing similar to λ. A pure type systems with definitions is a value $\lambda^\delta(S)$ for $S \in \mathbf{Spec}$ of λ^δ given by a 4-tuple:

1. a set $\mathcal{T}_\delta$ of pseudoterms
2. a set $\mathcal{C}_\delta$ of pseudocontexts,
3. two reduction relations on pseudoterms and pseudocontexts: one reduction is the β-reduction and the other relation is called δ-reduction,
4. a typing relation denoted by $\vdash^\delta$.

11.2.1 Pseudoterms

Definitions will be of the form $x{=}a{:}A$. A definition $x{=}a{:}A$ introduces x as an abbreviation of the term a of type A.

Definitions are allowed both in pseudocontexts, e.g. $?, x{=}a:A$, and in pseudoterms, e.g. $x{=}a{:}A\ in\ b$. Definitions in pseudocontexts are called *global* definitions and definitions in pseudoterms are called *local* definitions.

Next we extend the set of pseudoterms to include local definitions, expressions of the form $(x{=}a{:}A\ in\ b)$.

Definition 11.2.1. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The set $\mathcal{T}_\delta$ of *pseudoterms* is given by

$$\mathcal{T}_\delta \ ::= \ V \mid \boldsymbol{S} \mid (\mathcal{T}_\delta\ \mathcal{T}_\delta) \mid (\lambda V{:}\mathcal{T}_\delta.\ \mathcal{T}_\delta) \mid (\Pi V{:}\mathcal{T}_\delta.\ \mathcal{T}_\delta) \mid (V{=}\mathcal{T}_\delta{:}\mathcal{T}_\delta\ in\ \mathcal{T}_\delta)$$

where V is the set of variables and $\boldsymbol{S}$ is the set of sorts.

Definition 11.2.2. The mapping $FV : \mathcal{T}_\delta \to \mathcal{P}(V)$ is defined as in definition 9.3.2 by adding the following case.

$$FV(x{=}a{:}A\ in\ b) \ = \ FV(A) \cup FV(a) \cup (FV(b) - \{x\})$$

We say that x is *free* in a if $x \in FV(a)$.

Definition 11.2.3. The mapping $BV : \mathcal{T}_\delta \to \mathcal{P}(V)$ is defined as in definition 9.3.3 by adding the following case.

$$BV(x{=}a{:}A\ in\ b) \quad = \quad BV(A) \cup BV(a) \cup (BV(b) \cup \{x\})$$

We say that x is *bound* in a if $x \in BV(a)$.

Definition 11.2.4. The result of *substituting* d for (the free occurrences of) x in e is denoted as $e[x := d]$ and defined as in definition 9.3.4 by adding the following cases.

$$\begin{array}{lll}
(x{=}a{:}A\ in\ b)[x := d] & = (x{=}a{:}A\ in\ b) & \\
(y{=}a{:}A\ in\ b)[x := d] & = (y{=}a[x := d]{:}A[x := d]\ in\ b[x := d]) & \text{if } x \neq y \text{ and } y \notin FV(d) \\
(y{=}a{:}A\ in\ b)[x := d] & = (z{=}a[x := d]{:}A[x := d]\ in\ b[y := z][x := d]) & \text{if } x \neq y,\ y \in FV(d) \\
 & & \text{and } z \text{ is fresh}
\end{array}$$

The set of pseudoterms with holes in it is defined as follows.

Definition 11.2.5. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The set $\mathcal{H}_\delta$ is given by

$$\mathcal{H}_\delta \ ::= \ [\ ] \mid V \mid \ \boldsymbol{S} \mid \ (\mathcal{H}_\delta\ \mathcal{H}_\delta) \mid \ (\lambda V{:}\mathcal{H}_\delta.\ \mathcal{H}_\delta) \mid \ (\Pi V{:}\mathcal{H}_\delta.\ \mathcal{H}_\delta) \mid \ (V{=}\mathcal{H}_\delta{:}\mathcal{H}_\delta\ in\ \mathcal{H}_\delta)$$

where V is the set of variables and $\boldsymbol{S}$ is the set of sorts.

An element in $\mathcal{H}_\delta$ is denoted by $C[\ ]$.

Next we extend the set of pseudocontexts to include global definitions, expressions of the form $?, x{=}a{:}A, ?'$.

Definition 11.2.6. Let S be a specification. The set $\mathcal{C}_\delta$ of *pseudocontexts* is given by

i) $\epsilon \ \in \mathcal{C}_\delta$

ii) $< ?, x{:}A > \in \mathcal{C}_\delta$ if $? \ \in \mathcal{C}_\delta$, $x \in V$,$A \in \mathcal{T}_\delta$ and x is $?$-fresh

iii) $< ?, x = a : A > \in \mathcal{C}_\delta$ if $? \ \in \mathcal{C}_\delta$, $x \in V$ $a \in \mathcal{T}_\delta$, $A \in \mathcal{T}_\delta$, x is $?$-fresh and $x \notin FV(a) \cup FV(A)$

Note that the set of pseudocontexts is not given by

$$\mathcal{C}_\delta ::= \epsilon \mid \ \mathcal{C}_\delta, V : \mathcal{T}_\delta \mid \ \mathcal{C}_\delta, V = \mathcal{T}_\delta : \mathcal{T}_\delta \ .$$

We have additional requirements for the well-formation of the pseudocontext $?, x{=}a{:}A$. We require that x should be $?$-fresh in order to avoid capture of bound variables in the definition of δ-reduction given below. Moreover we require that $x \notin FV(a) \cup FV(A)$ in order that definitions are not recursive.

The expression $?, x{=}a{:}A$ stands for the pseudocontext $< ?, x{=}a{:}A >$.

Next we define a mapping *Dom* that gives the set of variables declared in a pseudocontext.

Definition 11.2.7. The mapping $Dom : \mathcal{C}_\delta \to \mathcal{P}(V)$ is defined as definition 9.3.7 by adding the case for definitions.

$$Dom(?\,, x{=}a{:}A) \;=\; Dom(?\,) \cup \{x\}$$

Next we define a mapping Def that gives the set of variables declared as definitions in a pseudocontext.

Definition 11.2.8. The mapping $Def : \mathcal{C}_\delta \to \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} Def(\epsilon) & = & \emptyset \\ Def(?\,, x{:}A) & = & Def(?\,) \\ Def(?\,, x{=}a{:}A) & = & Def(?\,) \cup \{x\} \end{array}$$

Definition 11.2.9. The result of *substituting* d for (the free occurrences of) a variable x in $?$ such that $x \notin Dom(?\,)$ is denoted as $?\,[x := d]$ and is defined as in definition 9.3.8 by adding the following case for definitions.

$$< ?\,, y{=}a{:}A > [x := d] \;=< ?\,[x := d], y{=}a[x := d]{:}A[x := d] >$$

Definition 11.2.10. Let $d \in \mathcal{T}_\delta$. A *change of a bound variable* in the term d is the replacement of a subterm $(x{=}a{:}A\ in\ b)$, $(\lambda x{:}A.\ b)$ or $(\Pi x{:}A.\ b)$ by $(y{=}a{:}A\ in\ b[x := y])$, $(\lambda y{:}A.\ b[x := y])$ or $(\Pi y{:}A.\ b[x := y])$, respectively, where $y \notin FV(b)$.

Definition 11.2.11. The pseudoterm b is *α-convertible* to b' if b' is the result of applying to b a series of changes of variables or vice versa.

Convention 11.2.12. Two terms are identified if they are α-convertible.

11.2.2 Reductions

In this section we define the β and the δ-reduction. The β-reduction is defined as usual.

Definition 11.2.13. The *β-reduction* is written as $a \to_\beta a'$ and is defined by the following rule.

$$C[(\lambda x{:}A.\ b)a] \to_\beta C[b[x := a]]$$

where $C[\ ] \in \mathcal{H}_\delta$ has only one occurrence of $[\ ]$.

The intended meaning of a definition $(x = a : A\ in\ b)$ is that the *definiendum* x can be substituted by the *definiens* a in the expression b. A definition $(x{=}a{:}A\ in\ b)$ can be considered as having a similar behaviour to $(\lambda x{:}A.\ b)a$, i.e. the substitution of the variable x by a in the expression b. In the β-reduction where $(\lambda x{:}A.\ b)a$ reduces to $b[x := a]$, the operation $b[x := a]$ is the substitution of **all the occurrences** of x by a in the expression b. In contrast to β-reduction, the expression $(x = a : A\ in\ b)$ reduces to the expression $(x{=}a{:}A\ in\ b')$ where b' is obtained from b by unfolding **one occurrence** of x by a. This is clearly illustrated in the example we presented in section 1.2.2. In order to perform the unfolding of a definition, we introduce a new relation called δ-reduction.

Definition 11.2.14. We define the *δ-reduction* (or $\to_\delta$) as the smallest relation on $\mathcal{C}_\delta \times \mathcal{T}_\delta \times \mathcal{T}_\delta$ closed under the following rules (we write $? \vdash d \to_\delta d'$ instead of $(?, d, d') \in \to_\delta$):

$$\frac{}{?_1, x{=}a{:}A, ?_2 \vdash x \to_\delta a}$$

$$\frac{}{? \vdash (x{=}a{:}A\ in\ b) \to_\delta b} \quad \text{if } x \notin FV(b)$$

$$\frac{?, x{=}a{:}A \vdash b \to_\delta b'}{? \vdash (x{=}a{:}A\ in\ b) \to_\delta (x{=}a{:}A\ in\ b')}$$

$$\frac{? \vdash a \to_\delta a'}{? \vdash (x{=}a{:}A\ in\ b) \to_\delta (x{=}a'{:}A\ in\ b)} \qquad \frac{? \vdash A \to_\delta A'}{? \vdash (x{=}a{:}A\ in\ b) \to_\delta (x{=}a{:}A'\ in\ b)}$$

$$\frac{? \vdash a \to_\delta a'}{? \vdash (a\ b) \to_\delta (a'\ b)} \qquad \frac{? \vdash b \to_\delta b'}{? \vdash (a\ b) \to_\delta (a\ b')}$$

$$\frac{?, x{:}A \vdash a \to_\delta a'}{? \vdash (\lambda x{:}A.\ a) \to_\delta (\lambda x{:}A.\ a')} \qquad \frac{? \vdash A \to_\delta A'}{? \vdash (\lambda x{:}A.\ a) \to_\delta (\lambda x{:}A'.\ a)}$$

$$\frac{?, x{:}A \vdash a \to_\delta a'}{? \vdash (\Pi x{:}A.\ a) \to_\delta (\Pi x{:}A.\ a')} \qquad \frac{? \vdash A \to_\delta A'}{? \vdash (\Pi x{:}A.\ a) \to_\delta (\Pi x{:}A'.\ a)}$$

When $?$ is the empty pseudocontext, $a \to_\delta a'$ is written instead of $? \vdash a \to_\delta a'$.

Unfolding of definitions. The first rule allows to unfold definitions. The definiens x reduces to its definiendum a. This rule together with the compatibility rules perform the unfolding of global definitions. In a pseudocontext $?, x{=}a{:}A, ?'$ a term d is δ-reduced to a term d' if d' is obtained replacing one occurrence of x by a in d.

The third rule allows to unfold local definitions. We consider a local definition as if it were global. The declaration $x{=}a{:}A$ passes from the pseudoterm to the pseudocontext. Besides, we keep the bound variable x in $(x{=}a{:}A\ in\ b)$ until we unfold all the occurrences of x in b.

After unfolding all the occurrences of x in b, we can remove the definition $x{=}a{:}A$ from the term. The second rule allows to remove definitions when the variable x does not occur in b.

Reductions depending on pseudocontexts. The δ-reduction is an example of a rewrite relation depending on a pseudocontext. We have that $(\mathcal{T}_\delta, \mathcal{C}_\delta, \to_\delta)$ and $(\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta})$ are environmental abstract rewriting systems. If $R \in \{\twoheadrightarrow_\delta, \twoheadleftrightarrow_\delta, \twoheadrightarrow_{\beta\delta}, \twoheadleftrightarrow_{\beta\delta}\}$ a pseudocontext has to be specified, i.e. $? \vdash a\ R\ b$ for $a, b \in \mathcal{T}_\delta$ and $? \in \mathcal{C}_\delta$.

Renaming of variables. The variables introduced as definitions are bound variables. Therefore α-conversion is necessary when rewriting terms. For example,

$$\begin{aligned} x{=}y{:}A \vdash\ & (y{=}z{:}u\ in\ x) \\ & \twoheadleftrightarrow_\alpha (y'{=}z{:}u\ in\ x) \\ & \to_\delta (y'{=}z{:}u\ in\ y) \end{aligned}$$

The variable y occurs in the term $(y{=}z{:}u\ in\ x)$ and in the pseudocontext $x{=}y{:}A$.

From now on, we assume that all the bound variables of the term and all the declared variables of the pseudocontext are different.

We extend the definition of δ-reduction to pseudocontexts.

Definition 11.2.15. We define the *δ-reduction on pseudocontexts* (or $\rightarrow_\delta$) as the smallest relation on $\mathcal{C}_\delta \times \mathcal{C}_\delta$ closed under the following rules:

$$\frac{? \vdash E \rightarrow_\delta E'}{?\,, y{:}E, ?' \rightarrow_\delta ?\,, y{:}E', ?'}$$

$$\frac{? \vdash E \rightarrow_\delta E'}{?\,, y{=}e{:}E, ?' \rightarrow_\delta ?\,, y{=}e{:}E', ?'}$$

$$\frac{? \vdash e \rightarrow_\delta e'}{?\,, y{=}e{:}E, ?' \rightarrow_\delta ?\,, y{=}e'{:}E, ?'}$$

We extend the definition of β-reduction to pseudocontexts.

Definition 11.2.16. We define the *β-reduction on pseudocontexts* (or $\rightarrow_\beta$) as the smallest relation on $\mathcal{C}_\delta \times \mathcal{C}_\delta$ closed under the following rules:

$$\frac{? \vdash E \rightarrow_\beta E'}{?\,, y{:}E, ?' \rightarrow_\beta ?\,, y{:}E', ?'}$$

$$\frac{? \vdash E \rightarrow_\beta E'}{?\,, y{=}e{:}E, ?' \rightarrow_\beta ?\,, y{=}e{:}E', ?'}$$

$$\frac{? \vdash e \rightarrow_\beta e'}{?\,, y{=}e{:}E, ?' \rightarrow_\beta ?\,, y{=}e'{:}E, ?'}$$

11.2.3 Types

We define the typing relation $\vdash^\delta$ which allows to type definitions.

Definition 11.2.17. The *typing relation* $\vdash^\delta_S$ (or $\vdash^\delta$ for short) is the smallest relation on $\mathcal{C} \times \mathcal{T} \times \mathcal{T}$ closed under the following rules and the rules of definition 9.3.14 (we write $? \vdash^\delta b : B$ instead of $(?\,, b, B) \in \vdash^\delta$) :

$$(\delta - start) \qquad \frac{? \vdash^\delta a : A}{?\,, x{=}a{:}A \vdash^\delta x : A} \qquad \text{where } x \text{ is } ?\text{-fresh}$$

$$(\delta{-}weakening) \qquad \frac{? \vdash^\delta b : B \quad ? \vdash^\delta a : A}{?\,, x{=}a{:}A \vdash^\delta b : B} \qquad \text{where } x \text{ is } ?\text{-fresh}$$

$$(\delta{-}formation) \qquad \frac{?\,, x{=}a{:}A \vdash^\delta B : s}{? \vdash^\delta (x{=}a{:}A\ in\ B) : s}$$

$$(\delta{-}introduction) \qquad \frac{?\,, x{=}a{:}A \vdash^\delta b : B \quad ? \vdash^\delta (x{=}a{:}A\ in\ B) : s}{? \vdash^\delta (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)}$$

$$(\delta{-}conversion) \qquad \frac{? \vdash^\delta b : B \quad ? \vdash^\delta B' : s \quad ? \vdash B \twoheadleftrightarrow_\delta B'}{? \vdash^\delta b : B'}$$

where s ranges over sorts, i.e. $s \in \boldsymbol{S}$.

Definition 11.2.18. The functor $\lambda^\delta : \mathbf{Spec} \to \mathbf{Carst}$ is defined for $S \in \mathbf{Spec}$ as follows.

$$\lambda^\delta(S) = (\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash^\delta).$$

The functor $\lambda^\delta : \mathbf{Spec} \to \mathbf{Carst}$ is defined for $f \in \mathbf{Spec}$ as the extension of f to the set of pseudoterms and pseudocontexts.

We have to verify that $\lambda^\delta(S) \in \mathbf{Carst}$ and that $\lambda^\delta(f) \in \mathbf{Carst}$. For $\lambda^\delta(S) \in \mathbf{Carst}$, we have to prove the subject and type reduction properties. which are proved in the following sections. For the second one, we have to verify that $\lambda^\delta(f)$ preserves the rewrite and the typing relations. This is very easy to prove.

Definition 11.2.19.

A pure type system with definitions (DPTS) is defined as an element of

$$\lambda^\delta(\mathbf{Spec}) = \{\lambda^\delta(S) \mid S \in \mathbf{Spec}\}$$

A singly sorted pure type system with definitions is defined as an element of

$$\{\lambda^\delta(S) \mid S \in \mathbf{Spec} \ \& \ \ S \text{ is singly sorted } \}$$

Observe that the system $\lambda^\delta(S)$ is an extension of $\lambda(S)$.

Explanation of the typing rules. The *δ-start and δ-weakening rules* allow the typing of global definitions. We cannot add $x{=}a{:}A$ to the context ? unless the term a has type A in ?. These rules ensure the correctness of what we abbreviate. Moreover they do not allow to abbreviate topsorts. For example in the systems of the λ-cube it is not possible to abbreviate $\Box$.

The *δ-formation and δ-introduction rules* allow the typing of local definitions. These rules are similar to the abstraction and the Π-formation rules. They differ in the fact that the δ-formation rule is not restricted by a set $\boldsymbol{R}$ of rules like the Π-formation rule.

We could have removed the δ-formation rule and the condition '$? \vdash^\delta (x{=}a{:}A \ in \ B) : s$' of the δ-introduction rule. In that case correctness of types and subject reduction would hold in a weaker form. The weaker form of correctness of types is stated as follows: if $? \vdash^\delta a : A$ then either $A \twoheadrightarrow_{\beta\delta} s$ or $? \vdash^\delta A : B$ and $B \twoheadrightarrow_{\beta\delta} s$. The weaker form of subject reduction is stated as follows: if $? \vdash^\delta a : A$ and $a \twoheadrightarrow_{\beta\delta} a'$ then there exists A' such that $? \vdash^\delta a' : A'$ and $A \twoheadrightarrow_{\beta\delta} A'$ (see also definition 4.3.1).

The *δ-conversion rule* plays an important role in typing definitions. This rule allows to use the fact that the definiens x and the definiendum a are (δ-)equal for typing a term.

Definition vs abstraction and application. A definition $(x{=}a{:}A\ in\ b)$ is not another way of writing $(\lambda x{:}A.\ b)a$. There are important differences between $(x{=}a{:}A\ in\ b)$ and $(\lambda x{:}A.\ b)a$, both regarding their reduction behaviour and their typing. One reason for considering $(x{=}a{:}A\ in\ b)$ and not $(\lambda x{:}A.\ b)a$ is that in some cases it is convenient to have the freedom of substituting only in **some of the occurrences** of an expression in a given formula.

Another reason for considering $(x{=}a{:}A\ in\ b)$ and not $(\lambda x{:}A.\ b)a$ is that the first may be typable when the second is not. There are two situations where this happens:

1. The fact that x is an abbreviation for a can be used to type b. This is shown in the following example.

   Example 11.2.20. The term $\lambda\alpha:*.\ (X=\alpha:*\ in\ \lambda y:X.\ \lambda f:\alpha\to\alpha.\ fy)$ is typable in the system $\lambda 2$ extended with definitions. But it is not possible to type the corresponding term expressed with an application and an abstraction in $\lambda 2$. As a matter of fact, the following term is not typable in any system of the λ-cube.

   $$\lambda\alpha{:}*.\ (\lambda X{:}*.\ \lambda y{:}X.\ \lambda f{:}\alpha\to\alpha.\ fy)\alpha$$

   In this term the application fy is not well-typed because the type X of the argument y does not match the type $\alpha\to\alpha$ of the function f. In the first term this application is well-typed because we know that X is an abbreviation of α. Note that here we apply the δ-conversion rule.

2. The abstraction $(\lambda x{:}A.\ b)$ may not be allowed in a given type system.

   Example 11.2.21. The term $(X{=}\alpha\to\alpha{:}*\ in\ \lambda y{:}X.\ \lambda f{:}X\to X.\ fy)$ is typable in the system $\lambda_{\to}$ extended with definitions. The corresponding term expressed with an application and an abstraction, i.e. $(\lambda X{:}*.\ \lambda y{:}X.\ \lambda f{:}X\to X.\ fy)\alpha\to\alpha$, is not typable in $\lambda_{\to}$ because in $\lambda_{\to}$ abstractions over type variables are not allowed.

Properties of pure type systems with definitions. In the following sections we will prove these properties:

- Confluence for $\to_\delta$ and for $\to_{\beta\delta}$.
- Strong normalisation for $\to_\delta$.
- $(\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash^\delta)$ verifies subject reduction, i.e. if $?\ \vdash b\to_{\beta\delta} b'$ and $?\ \vdash^\delta b:B$ then $?\ \vdash^\delta b':B$.
- $(\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash^\delta)$ verifies type reduction, i.e. if $?\ \vdash B\to_{\beta\delta} B'$ and $?\ \vdash^\delta b:B$ then $?\ \vdash^\delta b:B'$.

- Uniqueness of types for singly sorted specifications, i.e. if $? \vdash^{\delta} d : D$ and $? \vdash^{\delta} d : D'$ then $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} D'$ with S a singly sorted specification.
- Conservativity, i.e. for $A \in \mathcal{T}$, $? \in \mathcal{C}$ $\exists a$ $? \vdash a : A$ iff $\exists a$ $? \vdash^{\delta} a : A$.
- Strengthening, i.e. if $?_1, x{:}A, ?_2 \vdash^{\delta} b : B$ and $x \notin FV(?_2) \cup FV(b) \cup FV(B)$ then $?_1, ?_2 \vdash^{\delta} b : B$.
- If a pure type system is $\beta\delta$-weakly normalising then the corresponding pure type system with definitions is $\beta\delta$-weakly normalising.
- An extension of a pure type system is $\beta\delta$-strongly normalising if a 'slightly' larger pure type system is β-strongly normalising. In particular, the Calculus of Constructions extended with definitions is $\beta\delta$-strongly normalising.

11.3 Properties of Pseudoterms

In this section, we prove properties of β and δ-reductions for all pseudoterms. Amongst these properties, we will prove confluence for δ and for $\beta\delta$. Besides we will prove weak and strong normalisation for δ-reduction.

11.3.1 Basic Properties

In the following lemma we show that a δ-reduction step remains invariant if we enlarge the context. The proof is done by induction on the definition of $\rightarrow_{\delta}$.

Lemma 11.3.1. Let $< ?_1, ?_2, ?_3 > \in \mathcal{C}_{\delta}$ be such that $?_1, ?_3 \vdash b \rightarrow_{\delta} b'$. Then

$$?_1, ?_2, ?_3 \vdash b \rightarrow_{\delta} b'.$$

Both implications from left to right of the following lemma are a particular case of lemma 11.3.1. Both implications from right to left allow to make the context shorter. The first part states that declarations of the form $x{:}A$ can always be removed from the context. The second part states that declarations of the form $x = a : A$ can be removed from the context only if $x \notin FV(b)$. This allows to remove global definitions as the second rule in the definition of δ-reduction does for local definitions. Both parts are proved by induction on the definition of β and δ.

Lemma 11.3.2.

1. Let $< ?, x{:}A, ?' > \in \mathcal{C}_{\delta}$ and $b \in \mathcal{T}_{\delta}$.
   $?, ?' \vdash b \rightarrow_{\beta\delta} b'$if and only if $?, x{:}A, ?' \vdash b \rightarrow_{\beta\delta} b'$
2. Let $< ?, x{=}a{:}A, ?' > \in \mathcal{C}_{\delta}$ and $b \in \mathcal{T}_{\delta}$ be such that $x \notin FV(b)$.
   $?, ?' \vdash b \rightarrow_{\beta\delta} b'$ if and only if $?, x{=}a{:}A, ?' \vdash b \rightarrow_{\beta\delta} b'$.

In the following lemma we show that the compatibility rule for $(x{=}a{:}A\ in\ b)$ when we reduce inside b is a derivable rule.

Lemma 11.3.3. The following rule is derivable from the ones in the definition of $\rightarrow_\delta$.

$$\frac{? \vdash b \rightarrow_\delta b'}{? \vdash (x{=}a{:}A\ in\ b) \rightarrow_\delta (x{=}a{:}A\ in\ b')}$$

Proof: By lemma 11.3.1, it follows that $?, x{=}a{:}A \vdash b \rightarrow_\delta b'$. By definition of $\rightarrow_\delta$, it follows that $? \vdash (x{=}a{:}A\ in\ b) \rightarrow_\delta (x{=}a{:}A\ in\ b')$. $\Box$

We will prove that a definition $(x{=}a{:}A\ in\ b)$ has the same behaviour as $(\lambda x{:}A.\ b)a$ in the sense that the definition $(x{=}a{:}A\ in\ b)$ δ-reduces in *several* steps to $b[x := a]$. Then the unfolding of definitions that is achieved via the δ-reduction corresponds to the operation of substitution $b[x := a]$. The proof is done by induction on the structure of b.

Theorem 11.3.4. Let $? = < ?_1, x{=}a{:}A, ?_2 >$. Then

$$? \vdash b \twoheadrightarrow_\delta b[x := a].$$

Corollary 11.3.5. $? \vdash (x{=}a{:}A\ in\ b) \rightarrow_\delta^+ b[x := a]$.

Proof: By theorem 11.3.4, it follows that $?, x{=}a{:}A \vdash b \twoheadrightarrow_\delta b[x := a]$. Then

$$\begin{array}{lll} ? \vdash (x{=}a{:}A\ in\ b) & \twoheadrightarrow_\delta & (x{=}a{:}A\ in\ b[x := a]) \\ & \rightarrow_\delta & b[x := a] \end{array}$$

Note that $x \notin FV(b[x := a])$. $\Box$

The following lemma is proved by induction on the structure of a.

Lemma 11.3.6. (Substitution Lemma) Suppose $x \neq y$ and $x \notin FV(d)$. Then

$$a[x := b][y := d] = a[y := d][x := b[y := d]]$$

The following lemma shows that β is substitutive. It is proved by induction on the generation of $\rightarrow_\beta$.

Lemma 11.3.7. (Substitutivity for β) If $a \rightarrow_\beta a'$ then $a[x := b] \rightarrow_\beta a'[x := b]$.

The following example shows that δ is not substitutive for those variables which are definitions.

Example 11.3.8. Let $?$ be the context $< A : *, id{=}(\lambda y{:}A.\ y){:}(\Pi y{:}A.\ A) >$.

$$? \vdash id \rightarrow_\delta (\lambda y{:}A.\ y)$$

But it is not true that $? \vdash id[id := b] \rightarrow_\delta (\lambda y{:}A.\ y)[id := b]$ for all $b \in \mathcal{T}_\delta$. In this case the variable id is a definition, this means that it can be substituted only by the definiendum $(\lambda y{:}A.\ y)$.

Lemma 11.3.9. If $?, x{=}a{:}A, ?' \vdash b \rightarrow_{\beta\delta} b'$ then $?, ?'[x := a] \vdash b[x := a] \twoheadrightarrow_{\beta\delta} b'[x := a]$.

In the following lemma we reduce inside the pseudoterm a of $b[x := a]$. This lemma holds for any variable x, including those which are definitions. It is proved by induction on the structure of b.

Lemma 11.3.10. If $? \vdash a \rightarrow_{\beta\delta} a'$ then $? \vdash b[x := a] \twoheadrightarrow_{\beta\delta} b[x := a']$.

Lemma 11.3.11. If $? \vdash a \rightarrow_{\beta\delta} a'$ and $?, x{=}a{:}A, ?' \vdash b \rightarrow_\delta b'$ then $?, x{=}a'{:}A, ?' \vdash b \twoheadrightarrow_{\beta\delta} b'$.

The previous lemma is proved by induction on the definition of $?, x{=}a{:}A, ?' \vdash b \rightarrow_\delta b'$.

11.3.2 Confluence for β, δ and $\beta\delta$-reductions

The proof of confluence for β-reduction is very easy. It follows from the fact that the combinatorial reduction system $< \mathcal{T}_\delta, \rightarrow_\beta >$ is orthogonal(see [Klo90]).

Theorem 11.3.12. (Confluence for β-reduction) Let $a, a_1, a_2 \in \mathcal{T}_\delta$ such that $a \twoheadrightarrow_\beta a_1$ and $a \twoheadrightarrow_\beta a_2$. Then there exists a pseudoterm a_3 such that $a_1 \twoheadrightarrow_\beta a_3$ and $a_2 \twoheadrightarrow_\beta a_3$.

We prove confluence for δ and $\beta\delta$-reductions using the criteria proved in chapter 4. The idea of the proof of confluence for δ-reduction is as follows. We define a 'projection mapping', $|-|_{\_} : \mathcal{C}_\delta \times \mathcal{T}_\delta \rightarrow \mathcal{T}$. The 'projection' $|a|_\Gamma$ is a pseudoterm that is obtained from a by unfolding all the definitions occurring in $?$ and in a. First we prove that this mapping is a strategy for δ-reduction. Then we prove that this mapping is a forgetting morphism from $(\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_\delta)$ to $(\mathcal{T}, \rightarrow_\beta)$. Finally, the proof of confluence for $\beta\delta$-reduction follows from the previous considerations and the fact that the projection mapping is an implementing morphism from $(\mathcal{T}_\delta, \rightarrow_\beta)$ to $(\mathcal{T}, \rightarrow_\beta)$.

Next we define the mapping $|-|_{\_}$.

Definition 11.3.13. The mapping $|-|_{\_} : \mathcal{C}_\delta \times \mathcal{T}_\delta \rightarrow \mathcal{T}$ is defined as follows.

$$\begin{array}{rcl}
|x|_\Gamma & = & \begin{cases} |a|_{\Gamma_1} & \text{if } ? = ?_1, x{=}a{:}A, ?_2 \\ x & \text{otherwise} \end{cases} \\
|s|_\Gamma & = & s \\
|a\ b|_\Gamma & = & |a|_\Gamma |b|_\Gamma \\
|\lambda x{:}A.\ a|_\Gamma & = & (\lambda x{:}|A|_\Gamma.\ |a|_{\Gamma, x:A}) \\
|\Pi x{:}A.\ B|_\Gamma & = & (\Pi x{:}|A|_\Gamma.\ |B|_{\Gamma, x:A}) \\
|x{=}a{:}A\ in\ b|_\Gamma & = & |b|_\Gamma[x := |a|_\Gamma] \quad \text{where } x \text{ is } ?\text{-fresh.}
\end{array}$$

The value $|d|_\Gamma$ is obtained from d by unfolding all the global and local definitions. The unfolding of global definitions is performed in the first line $|x|_\Gamma = |a|_{\Gamma_1}$. The unfolding of local definitions is performed in the last line $|x{=}a{:}A \ in \ b|_\Gamma = |b|_\Gamma[x := |a|_\Gamma]$.

Note that $|d|_\Gamma$ does not contain either local or global definitions, i.e. $|d|_\Gamma \in \mathcal{T}$ and $Def(?) \cap FV(|d|_\Gamma) = \emptyset$. Conversely, if $d \in \mathcal{T}$ is such that $Def(?) \cap FV(d) = \emptyset$ then $|d|_\Gamma = d$. This means that for those pseudoterms which contain neither global nor local definitions, the mapping $|-|_-$ is the identity. Hence $|-|_-$ is a 'projection' from $\mathcal{T}_\delta$ and $\mathcal{C}_\delta$ to $\mathcal{T}$. Later, we will prove that $|d|_\Gamma$ is the δ-normal form of the pseudoterm d.

The function $FV : \mathcal{T}_\delta \to \mathcal{P}(V)$ is extended to $\mathcal{C}_\delta \times \mathcal{T}_\delta$. We say that $FV_\Gamma(b)$ is the set of *free variables of b with respect to $?$*.

Definition 11.3.14. The mapping $FV : \mathcal{C}_\delta \times \mathcal{T}_\delta \to \mathcal{P}(V)$ is defined as follows.

$$\begin{array}{rcl} FV_\epsilon(b) & = & FV(b) \\ FV_{x:A,\Gamma}(b) & = & FV(A) \cup (FV_\Gamma(b) - \{x\}) \\ FV_{x=a:A,\Gamma}(b) & = & FV_\Gamma(A) \cup FV(a) \cup (FV_\Gamma(b) - \{x\}) \end{array}$$

Lemma 11.3.15.

1. If $x \notin FV(b)$ and x is $?$-fresh then $x \notin FV(|b|_\Gamma)$.
2. Let $< ?_1, ?_2, ?_3 > \in \mathcal{C}_\delta$ and $b \in \mathcal{T}_\delta$ be such that $(FV_{\Gamma_3}(b)) \cap Def(?_2) = \emptyset$. Then
$$|b|_{\Gamma_1,\Gamma_2,\Gamma_3} = |b|_{\Gamma_1,\Gamma_3}.$$
3. Let $< ?_1, x{=}a{:}A, ?_2 > \in \mathcal{C}_\delta$ and $a \in \mathcal{T}_\delta$. Then $|a|_{\Gamma_1,x=a:A,\Gamma_2} = |a|_{\Gamma_1}$.

Proof:

1. It is proved by induction on the number of symbols occurring in $?$ and b.
2. It is proved by induction on the structure of b.
3. None of the variables in $Def(x{=}a{:}A, ?_2)$ can occur in a. Hence the result follows immediately from the previous part.

□

The following lemma states that $|-|_-$ preserves substitution. Also it shows that $|-|_-$ yields the same value for global and local definitions and this value is given by substitution. It is proved by induction on the structure of b.

Lemma 11.3.16. Let $< ?, x{=}a{:}A > \in \mathcal{C}_\delta$ and $b \in \mathcal{T}_\delta$. Then

$$|b|_\Gamma\,[x := |a|_\Gamma] \;=\; |b[x := a]|_\Gamma \;=\; |b|_{\Gamma,x=a:A}$$

The following lemma states that a pseudoterm reduces to its projection.

Lemma 11.3.17. The 'projection mapping' is a strategy for δ-reduction. In other words,

$$? \vdash d \twoheadrightarrow_\delta |d|_\Gamma \text{ for all } d \in \mathcal{T}_\delta,\ ? \in \mathcal{C}_\delta.$$

Proof: It is proved by induction on the number of symbols occurring in $?$ and in d. Only two cases are considered.

- Assume $d = x$. There are two possibilities, either $x \in Def(?)$ or not.

  If $x \in Def(?)$ then $? =< ?_1, x{=}b{:}B, ?_2 >$.

$$\begin{array}{lll} ? \vdash x & \rightarrow_\delta & b \\ & \twoheadrightarrow_\delta & |b|_{\Gamma_1} \quad \text{by induction hypothesis and lemma 11.3.1} \\ & = & |x|_\Gamma \end{array}$$

  If $x \notin Def(?)$ then $? \vdash x \twoheadrightarrow_\delta x = |x|_\Gamma$.

- Assume $d = (x{=}a{:}A\ in\ b)$. By induction hypothesis, it follows that $? \vdash a \twoheadrightarrow_\delta |a|_\Gamma$, $? \vdash A \twoheadrightarrow_\delta |A|_\Gamma$ and that $? \vdash b \twoheadrightarrow_\delta |b|_\Gamma$.

$$\begin{array}{lll} ? \vdash (x{=}a{:}A\ in\ b) & \twoheadrightarrow_\delta & (x{=}|a|_\Gamma{:}|A|_\Gamma\ in\ |b|_\Gamma) \quad \text{by definition of } \delta \text{ and lemma 11.3.3} \\ & \twoheadrightarrow_\delta & |b|_\Gamma[x := |a|_\Gamma] \quad \text{by corollary 11.3.5} \\ & = & |x{=}a{:}A\ in\ b|_\Gamma \end{array}$$

The rest of the cases are easy to prove. □

The following lemma states that the projections of two pseudoterms that are in $\rightarrow_\delta$ are equal.

Lemma 11.3.18. The projection mapping is a forgetting morphism from $(\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_\delta)$ to $(\mathcal{T}, \rightarrow_\beta)$, i.e. if $? \vdash c \rightarrow_\delta d$ then $|c|_\Gamma = |d|_\Gamma$ for all $c, d \in \mathcal{T}_\delta$ and $? \in \mathcal{C}_\delta$.

Proof: It is proved by induction on the structure of c. Only some cases are considered.

- Suppose that $c = x$. This means that $? = ?_1, x{=}a{:}A, ?_2$ and $? \vdash x \rightarrow_\delta a$.

$$\begin{array}{lll} |x|_\Gamma & = & |a|_{\Gamma_1} \\ & = & |a|_\Gamma \quad \text{by lemma 11.3.15} \end{array}$$

- Suppose that c is $(x{=}a{:}A\ in\ b)$ and $? \vdash (x{=}a{:}A\ in\ b) \rightarrow_\delta b$ with $x \notin FV(b)$.

$$\begin{array}{lll} |x{=}a{:}A\ in\ b|_\Gamma & = & |b|_\Gamma[x := |a|_\Gamma] \\ & = & |b|_\Gamma \quad \text{by lemma 11.3.15 part 1} \end{array}$$

The rest of the cases are easy to prove. □

Theorem 11.3.19. (Confluence for δ-reduction) If $? \vdash a \twoheadrightarrow_\delta a_1$ and $? \vdash a \twoheadrightarrow_\delta a_2$ then there exists a_3 such that $? \vdash a_1 \twoheadrightarrow_\delta a_3$ and $? \vdash a_2 \twoheadrightarrow_\delta a_3$.

Proof: By theorem 11.3.17, the projection mapping is a strategy for δ-reduction. Moreover it follows from lemma 11.3.18 that this mapping is a forgetting morphism. By lemma 2.6.1, $\rightarrow_\delta$ is confluent. $\Box$

The following lemma states that the projection preserves β-reduction.

Lemma 11.3.20. Let $? \in \mathcal{C}_\delta$. The projection mapping $|-|_\Gamma$ is an implementing morphism from $(\mathcal{T}_\delta, \rightarrow_\beta)$ to $(\mathcal{T}, \rightarrow_\beta)$, i.e. if $a \rightarrow_\beta a'$ then $|a|_\Gamma \twoheadrightarrow_\beta |a'|_\Gamma$, for all $a, a' \in \mathcal{T}_\delta$.

Proof: It is proved by induction on the structure of a. Only the case $a = (\lambda x{:}B.\ b)d$ and $(\lambda x{:}B.\ b)d \rightarrow_\beta b[x := d]$ is considered.

$$\begin{array}{rcll} |a|_\Gamma & = & |(\lambda x{:}B.\ b)d|_\Gamma & \\ & = & (\lambda x{:}|B|_\Gamma.\ |b|_\Gamma)|d|_\Gamma & \\ & \rightarrow_\beta & |b|_\Gamma[x := |d|_\Gamma] & \\ & = & |b[x := d]|_\Gamma & \text{by lemma 11.3.16} \end{array}$$

The rest of the cases are easy to prove. $\Box$

Theorem 11.3.21. (Confluence for $\beta\delta$-reduction) If $? \vdash a \twoheadrightarrow_{\beta\delta} b$ and $? \vdash a \twoheadrightarrow_{\beta\delta} c$ then there exists $d \in \mathcal{T}_\delta$ such that $? \vdash b \twoheadrightarrow_{\beta\delta} d$ and $? \vdash c \twoheadrightarrow_{\beta\delta} d$.

Proof: We apply lemma 2.6.2 to the abstract rewriting system $(\mathcal{T}, \rightarrow_\beta)$ and the environmental abstract rewriting system $(\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_{\beta\delta})$. We fix a pseudocontext, say $? \in \mathcal{C}_\delta$ and we consider the $\beta\delta$-reduction in $?$ (we write $\rightarrow_{\beta\delta}$ instead of $\rightarrow_{\beta\delta_\Gamma}$).

- The inclusion mapping is an implementing morphism from $(\mathcal{T}, \rightarrow_\beta)$ to $(\mathcal{T}_\delta, \rightarrow_{\beta\delta})$.
- By lemma 11.3.17, we have that the projection mapping $|-|_\Gamma$ is a strategy for δ and hence for $\beta\delta$-reduction.
- By lemmas 11.3.18 and 11.3.20, the projection mapping is an implementing morphism from $(\mathcal{T}_\delta, \rightarrow_{\beta\delta})$ to $(\mathcal{T}, \rightarrow_\beta)$.

Since $(\mathcal{T}, \rightarrow_\beta)$ is confluent so is $(\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_{\beta\delta})$. $\Box$

Note that if the contexts $?$ and $?'$ are different then $|a|_\Gamma$ may be different from $|a|_{\Gamma'}$.

Lemma 11.3.22. If $? \twoheadrightarrow_{\beta\delta} ?'$ then $|a|_\Gamma \twoheadleftarrow\!\!\twoheadrightarrow_\beta |a|_{\Gamma'}$.

This is proved by induction on the length of a and $?$.

11.3.3 Weak and Strong Normalisation for $\rightarrow_\delta$

In this section an illustrative and intuitive proof of weak normalisation for δ-reduction is presented. In order to prove strong normalisation for $\twoheadrightarrow_\delta$, the well-known method of defining a function $w_{\_}(-) : \mathcal{C}_\delta \times \mathcal{T}_\delta \rightarrow \mathbb{N}$ which decreases with δ-reduction is used. This function computes the length of a maximal δ-rewrite sequence from a term to the δ-normal form.

Note that a pseudoterm can be in δ-normal form but not in β-normal form, for example $(\lambda y{:}*.\ y)(\lambda y{:}A.\ y)$.

According to the following theorem, a pseudoterm a is in δ-normal form in a context $?$ if and only if a does not contain either global or local definitions. The pseudoterms that do not contain local definitions are in $\mathcal{T}$. The pseudoterms that do not contain global definitions are those pseudoterms whose free variables are not included in the set of definitions of the context. The proof follows easily by induction on the structure of a.

Theorem 11.3.23. Let $a \in \mathcal{T}_\delta$.

$$a \text{ is in } \delta\text{-normal form in } ? \text{ if and only if } a \in \mathcal{T} \text{ and } FV(a) \cap Def(?) = \emptyset.$$

The projection $|a|_\Gamma$ does not contain either global or local definitions. Therefore $|a|_\Gamma$ is in δ-normal form. Since δ-reduction is confluent we have that the δ-normal form is unique. As a consequence of this we have the result that follows.

Corollary 11.3.24. (Weak Normalisation for δ-reduction)
The pseudoterm $|a|_\Gamma$ is the δ-normal form of a in $?$.

Note that by corollary 11.3.24 the δ-normal form for an arbitrary pseudoterm exists, but it is not guaranteed that all δ-paths starting at the pseudoterm are finite.

Next we define a function $w_{\_}(-) : \mathcal{C}_\delta \times \mathcal{T}_\delta \rightarrow \mathbb{N}$ that decreases with δ-reduction. We use this function in order to prove strong normalisation for δ. We also prove that $w_\Gamma(b)$ computes the length of a maximal δ-rewrite sequences starting at b.

Definition 11.3.25. If $? \in \mathcal{C}_\delta$ and $b \in \mathcal{T}_\delta$, $w_\Gamma(b)$ is defined by induction on the number of symbols in $?$ and in b as follows.

$$\begin{array}{rcl}
w_{\Gamma_1, x=a:A, \Gamma_2}(x) & = & w_{\Gamma_1}(a) + 1 \\
w_\Gamma(x) & = & 0 \ \text{ if } x \notin Def(?) \\
w_\Gamma(s) & = & 0 \\
w_\Gamma(x{=}a{:}A\ in\ b) & = & w_\Gamma(a) + w_\Gamma(A) + w_{\Gamma, x=a:A}(b) + 1 \\
w_\Gamma(a\ b) & = & w_\Gamma(a) + w_\Gamma(b) \\
w_\Gamma(\Pi x{:}A.\ a) & = & w_{\Gamma, x:A}(a) + w_\Gamma(A) \\
w_\Gamma(\lambda x{:}A.\ a) & = & w_{\Gamma, x:A}(a) + w_\Gamma(A)
\end{array}$$

Lemma 11.3.26. If $FV(b) \cap Def(?_2) = \emptyset$ then $w_{\Gamma_1,\Gamma_2,\Gamma_3}(b) = w_{\Gamma_1,\Gamma_3}(b)$.

The previous lemma is proved by induction on the number of symbols in $< ?_1, ?_2, ?_3 >$ and in b.

Lemma 11.3.27. $w_{\Gamma_1,x=a:A,\Gamma_2}(b) \geq w_{\Gamma_1,\Gamma_2}(b)$.

The previous lemma is proved by induction on the number of symbols in $?_1, x{=}a{:}A, ?_2$ and in b.

Lemma 11.3.28. If $? \vdash d \rightarrow_\delta d'$ then $w_\Gamma(d) > w_\Gamma(d')$.

Proof: The following two properties are proved simultaneously by induction on the number of symbols in $?$ and in d.

1. If $? \vdash d \rightarrow_\delta d'$ then $w_\Gamma(d) > w_\Gamma(d')$.
2. If $? \rightarrow_\delta ?'$ then $w_\Gamma(d) \geq w_{\Gamma'}(d)$.

We only consider the proof of the first property for the case that $d = x$. We have that $? = ?_1, x = a : A, ?_2$ and $? \vdash x \rightarrow_\delta a$.

$$\begin{array}{rcll} w_\Gamma(x) & = & w_{\Gamma_1,x=a:A,\Gamma_2}(x) & \\ & = & w_{\Gamma_1}(a) + 1 & \\ & > & w_{\Gamma_1}(a) & \\ & = & w_\Gamma(a) & \text{by lemma 11.3.26} \end{array}$$

□

As an immediate consequence of lemma 11.3.28, we have the result that follows.

Theorem 11.3.29. (Strong Normalisation for δ)
The reduction δ is strongly normalising.

Finiteness of developments can be deduced from strong normalisation of δ-reduction. This is our third proof of finiteness of developments (see chapter 7).

We write a function that maps a marked redex $(\underline{\lambda}x{:}A.\ b)a$ into a definition $x{=}a{:}A\ in\ b$. This function maps one step of $\underline{\beta}$-rewrite step into one or more steps of δ-reduction.

We suppose that the terms in $\underline{\Lambda}$ are typed.

Definition 11.3.30. The mapping $\Upsilon : \underline{\Lambda} \rightarrow \mathcal{T}_\delta$ is defined as follows.

$$\begin{array}{rcl} \Upsilon(x) & = & x \\ \Upsilon(\lambda x{:}A.b) & = & \lambda x{:}\Upsilon(A).\ \Upsilon(b) \\ \Upsilon(b\ a) & = & \Upsilon(b)\ \Upsilon(a) \\ \Upsilon((\underline{\lambda}x{:}A.b)a) & = & (x{=}\Upsilon(a){:}\Upsilon(A).\ \Upsilon(b)) \end{array}$$

The last clause maps a marked redex $(\underline{\lambda}x{:}A.b)a$ into a definition $x{=}a{:}A\ in\ b$.

Lemma 11.3.31. $\Upsilon(b[x := a]) = \Upsilon(b)[x := \Upsilon(a)]$

This function is a refining morphism, i.e. it maps one step of β-reduction to one or more steps of δ-reduction.

Lemma 11.3.32. If $a \rightarrow_{\underline{\beta}} b$ then $\Upsilon(a) \rightarrow_{\delta}^{+} \Upsilon(b)$.

This lemma is proved by induction on the structure of a. In the case of a marked redex, $(\underline{\lambda}x{:}A.b)a$, we use corollary 11.3.5 and the previous lemma.

Theorem 11.3.33. (Finiteness of Developments)
The $\underline{\beta}$-reduction is strongly normalising.

Proof: By the criterion on strong normalisation 2.6.5, we have that the $\underline{\beta}$-reduction is strongly normalising. □

Note that $w_\Gamma(a)$ is an upper bound for the number of reductions steps in a δ-reduction sequence starting at a in ?, i.e. for all n, $a_1, \ldots, a_n$,

$$\text{if } ? \vdash a = a_1 \rightarrow_\delta a_2 \rightarrow_\delta \ldots \rightarrow_\delta a_{n-1} \rightarrow_\delta a_n \text{ then } w_\Gamma(a) \geq n.$$

We will show that $w_\Gamma(a)$ is the length of a maximal δ-rewrite sequence from a to its δ-normal form. In order to show that $w_\Gamma(a)$ is a maximum we will build a δ-reduction sequence of this length. We will define a strategy $F_\infty^\delta : \mathcal{C}_\delta \times \mathcal{T}_\delta \rightarrow \mathcal{T}_\delta$ for δ-reduction such that the F_∞^δ-reduction sequence of a has length $w_\Gamma(a)$.

The δ-reduction unfolds only one occurrence of a variable at a time. We need to give an order to these occurrences in order to define a strategy of reduction. We choose to unfold them from left to right. We will define $e[x^0 := d]$ as the substitution of the leftmost occurrence of the variable x for d in the expression e.

Definition 11.3.34. Let $x \in FV(e)$. The result of substituting d for the leftmost occurrence of x in e is denoted as $e[x^0 := d]$ and is defined as follows.

$$y[x^0 := d] = \begin{cases} d & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$(a\ \ b)[x^0 := d] = \begin{cases} (a[x^0 := d]\ \ b) & \text{if } x \in FV(a) \\ (a\ \ b[x^0 := d]) & \text{otherwise} \end{cases}$$

$$(y{=}a{:}A\ in\ b)[x^0 := d] = \begin{cases} y{=}a[x^0 := d]{:}A\ in\ b & \text{if } x \in FV(a) \\ y{=}a{:}A[x^0 := d]\ in\ b & \text{if } x \notin FV(a) \text{ and } x \in FV(A) \\ y{=}a{:}A\ in\ b[x^0 := d] & \text{if } x \notin FV(a) \cup FV(A) \end{cases}$$

$$(\Pi y{:}A.\ B)[x^0 := d] = \begin{cases} (\Pi y{:}A[x^0 := d].\ B) & \text{if } x \in FV(A) \\ (\Pi y{:}A.\ B[x^0 := d]) & \text{otherwise} \end{cases}$$

$$(\lambda y{:}A.\ a)[x^0 := d] = \begin{cases} (\lambda y{:}A[x^0 := d].\ a) & \text{if } x \in FV(A) \\ (\lambda y{:}A.\ a[x^0 := d]) & \text{otherwise} \end{cases}$$

The strategy we define for δ-reduction is similar to the perpetual strategy F_∞ on λ-terms and for β-reduction(see section 6). The leftmost δ-redex of a pseudoterm could be either a global or a local definition. If the leftmost redex is a variable x such that $x{=}a{:}A$ is in the context then the strategy gives the pseudoterm obtained by unfolding this occurrence of x by its definiendum a. If the leftmost redex is a definition ($x{=}a{:}A\ in\ b$), we check whether the variable x occurs as a free variable in b or not. If it does, then the strategy yields the pseudoterm obtained by unfolding the leftmost occurrence of x in b. If the variable x does not occur as a free variable in b then we do not remove the definition. We will apply the strategy first to the definiendum a and then to the type A. Only in the case that both a and A are in δ-normal form we remove the definition.

Definition 11.3.35. The mapping $F^\delta_\infty : \mathcal{C}_\delta \times \mathcal{T}_\delta \to \mathcal{T}_\delta$ is defined as follows (we write $F^\delta_\infty(b)$ instead of $F^\delta_\infty(?\,, b)$).

$$F^\delta_\infty(x) = \begin{cases} a & \text{if } ? = ?_1, x{=}a{:}A, ?_2 \\ x & \text{otherwise} \end{cases}$$

$$F^\delta_\infty(s) = s$$

$$F^\delta_\infty(x{=}a{:}A\ in\ b) = \begin{cases} x{=}a{:}A\ in\ b[x^0 := a] & \text{if } x \in FV(b) \\ x{=}F^\delta_\infty(a){:}A\ in\ b & \text{if } x \notin FV(b) \text{ and} \\ & a \text{ is not in } \delta\text{-normal form in } ? \\ x{=}a{:}F^\delta_\infty(A)\ in\ b & \text{if } x \notin FV(b),\ a \text{ is in } \delta\text{-normal form in } ? \\ & \text{and } A \text{ is not in } \delta\text{-normal form in } ? \\ b & \text{if } x \notin FV(b) \text{ and} \\ & a \text{ and } A \text{ are in } \delta\text{-normal form in } ? \end{cases}$$

$$F^\delta_\infty(a\ b) = \begin{cases} (F^\delta_\infty(a)\ b) & \text{if } a \text{ is not in } \delta\text{-normal form in } ? \\ (a\ F^\delta_\infty(b)) & \text{otherwise} \end{cases}$$

$$F^\delta_\infty(\Pi x{:}A.\ B) = \begin{cases} (\Pi x{:}F^\delta_\infty(A).\ B) & \text{if } A \text{ is not in } \delta\text{-normal form in } ? \\ (\Pi x{:}A.\ F^\delta_\infty(B)) & \text{otherwise} \end{cases}$$

$$F^\delta_\infty(\lambda x{:}A.\ a) = \begin{cases} (\lambda x{:}F^\delta_\infty(A).\ a) & \text{if } A \text{ is not in } \delta\text{-normal form in } ? \\ (\lambda x{:}A.\ F^\delta_\infty(a)) & \text{otherwise} \end{cases}$$

The following two lemmas are proved by induction on the structure of the term.

Lemma 11.3.36. Let $x \in FV(b)$. Then $w_{\Gamma_1, x=a:A, \Gamma_2}(b) = w_{\Gamma_1, x=a:A, \Gamma_2}(b[x^0 := a]) + 1$.

Lemma 11.3.37.

1. If b is in δ-normal form in $?$ then $F^\delta_\infty(b) = b$.

2. If b is not in δ-normal form in ? then $? \vdash b \rightarrow_\delta F^\delta_\infty(b)$ and $w_\Gamma(b) = w_\Gamma(F^\delta_\infty(b)) + 1$

The next theorem states that the F^δ_∞-reduction sequence of a has length $w_\Gamma(a)$. The proof follows immediately from the previous lemma.

Theorem 11.3.38. Let $n = w_\Gamma(a)$. Then

$$? \vdash a \rightarrow_\delta F^\delta_\infty(a) \rightarrow_\delta \ldots (F^\delta_\infty)^{n-1}(a) \rightarrow_\delta (F^\delta_\infty)^n(a) = |a|_\Gamma.$$

As a corollary we have that $w_\Gamma(a)$ is the length of a maximal δ-rewriting sequence starting at a and that the strategy F^δ_∞ is maximal. Here $maxred(a)$ denotes the length of the maximal δ-rewrite sequence starting at a.

Corollary 11.3.39. (Maximal Strategy for δ)

$$w_\Gamma(a) = maxred(a) \text{ and } F^\delta_\infty \text{ is maximal.}$$

Proof: By theorem 11.3.28, $w_\Gamma(a)$ is an upper bound for the number of reductions steps in a δ-reduction sequence starting at a in ?, i.e. for all n, $a_1, \ldots, a_n$,

$$\text{if } ? \vdash a = a_1 \rightarrow_\delta a_2 \rightarrow_\delta \ldots \rightarrow_\delta a_{n-1} \rightarrow_\delta a_n \text{ then } w_\Gamma(a) \geq n.$$

By the previous theorem, there is a δ-rewrite sequence of length $w_\Gamma(a)$ that is the F^δ_∞-rewrite sequence. Hence $w_\Gamma(a)$ is the length of the maximal δ-rewrite sequence and F^δ_∞ is a maximal strategy. □

11.4 Properties of Well-Typed Terms

The properties in this section are proved for all terms typable in a pure type system with definitions, i.e. for pseudoterms a such that $\exists A, ?[? \vdash^\delta a : A]$. Amongst these properties, we will prove that $\lambda(S)$ is weakly normalising if and only if $\lambda^\delta(S)$ is weakly normalising. Also we will prove strong normalisation of $\beta\delta$-reduction for a class of pure type systems with definitions.

11.4.1 Basic Properties

In the following lemma we will show that the structure of the term gives an idea of the shape of its type and its derivation. For example, the type of an abstraction will be a product (up to $\beta\delta$-conversion) and the last part of the derivation consists of the application of the abstraction rule and then 0 or more applications of the β or the δ-conversion rules.

The different cases of next lemma are all proved by induction on the derivation.

Lemma 11.4.1. (Generation Lemma)

1. If $? \vdash^\delta s : D$ then there exists a sort s' such that $? \vdash D \twoheadleftrightarrow_{\beta\delta} s'$ and $(s, s') \in \boldsymbol{A}$.

2. If $? \vdash^{\delta} x : D$ then there exists s such that $? \vdash B \twoheadleftarrow\twoheadrightarrow_{\beta\delta} D$, either $? \vdash^{\delta} B : s$ and $x : B \in ?$ or $? \vdash^{\delta} b : B$ and $x = b : B \in ?$ for some b.

3. If $? \vdash^{\delta} (\Pi x{:}A.B) : D$ then there are sorts such that $(s_1, s_2, s_3) \in \boldsymbol{R}$, that $? \vdash^{\delta} A : s_1$, that $?, x{:}A \vdash^{\delta} B : s_2$ and $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} s_3$.

4. If $? \vdash^{\delta} (\lambda x{:}A.\ b) : D$ then there are s and B such that $? \vdash^{\delta} (\Pi x{:}A.\ B) : s$, that $?, x : A \vdash^{\delta} b : B$ and $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} (\Pi x{:}A.\ B)$.

5. If $? \vdash^{\delta} (b\ \ a) : D$ then there are A and B such that $? \vdash^{\delta} b : (\Pi x{:}A.\ B)$, $? \vdash^{\delta} a : A$ and $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} B[x := a]$.

6. If $? \vdash^{\delta} (x{=}a{:}A\ in\ b) : D$ then there exists B such that either $?, x{=}a{:}A \vdash^{\delta} b : B$, $? \vdash^{\delta} (x{=}a{:}A\ in\ B) : s$ and $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} (x{=}a{:}A\ in\ B)$ or $?, x{=}a{:}A \vdash^{\delta} b : s$ and $? \vdash D \twoheadleftarrow\twoheadrightarrow_{\beta\delta} s$

Observe that in 6, the type of a term $(x{=}a{:}A\ in\ b)$ can be a sort s or an expression of the form $(x{=}a{:}A\ in\ B)$.

Lemma 11.4.2. (Correctness of Types) If $? \vdash^{\delta} A : B$ then there exists s such that either $B = s$ or $? \vdash^{\delta} B : s$.

The previous lemma is proved by induction on the derivation of $? \vdash^{\delta} A : B$.

Lemma 11.4.3. If $? \vdash^{\delta} b : (\Pi x : A.B)$ then there are sorts $(s_1, s_2, s_3) \in \boldsymbol{R}$ such that $? \vdash^{\delta} A : s_1$ and $?, x{:}A \vdash^{\delta} B : s_2$.

Proof: It follows from lemma 11.4.2 that $? \vdash^{\delta} (\Pi x{:}A.B) : s$. By the generation lemma part 3, it follows that there are sorts $(s_1, s_2, s_3) \in \boldsymbol{R}$ such that $? \vdash^{\delta} A : s_1$ and $?, x{:}A \vdash^{\delta} B : s_2$. □

Recall that $?$ is *a context* if there are b and B such that $? \vdash^{\delta} b : B$.

Lemma 11.4.4. (Correctness of Contexts)

1. If $?, x{:}A, ?'$ is a context then there exists a sort s such that $? \vdash A : s$.

2. If $?, x{=}a{:}A, ?'$ is a context then $? \vdash^{\delta} a : A$.

Both parts of the previous lemma are proved by induction on the derivation.

Lemma 11.4.5. (Thinning Lemma) Let $?'$ be a context.

$$\text{If } ? \vdash^{\delta} a : A \text{ and } ? \subseteq ?' \text{ then } ?' \vdash^{\delta} a : A.$$

The previous lemma is proved by induction on the derivation of $? \vdash^{\delta} a : A$.

Lemma 11.4.6. If $? \vdash^{\delta} a : A$ and $?, x{:}A, ?' \vdash^{\delta} b : B$ then $?, x{=}a{:}A, ?' \vdash^{\delta} b : B$.

The previous lemma is proved by induction on the derivation of $?, x{:}A, ?' \vdash^{\delta} b : B$.

Example 11.4.7. It is not true that if $?, x{=}a{:}A, ?' \vdash^{\delta} b : B$ then $?, x{:}A, ?' \vdash^{\delta} b : B$.
For instance, we can derive $\alpha{:}*, X{=}\alpha{:}*, y{:}X, f{:}(\alpha \to \alpha) \vdash^{\delta} (f\ \ y) : \alpha$.
But there is no term B such that $\alpha{:}*, X{:}*, y{:}X, f{:}(\alpha \to \alpha) \vdash^{\delta} (f\ \ y) : B$. This example is similar to example 11.2.20.

Lemma 11.4.8. (Substitution Lemma)

1. If $?, x{=}a{:}A, ?' \vdash^{\delta} b : B$ then $?, ?'[x := a] \vdash^{\delta} b[x := a] : B[x := a]$
2. If $? \vdash^{\delta} a : A$ and $?, x{:}A, ?' \vdash^{\delta} b : B$ then $?, ?'[x := a] \vdash^{\delta} b[x := a] : B[x := a]$.

Proof: The first part is proved by induction on the derivation of $?, x{=}a{:}A, ?' \vdash^{\delta} b : B$. The second part follows by lemma 11.4.6 and the previous part. □

Theorem 11.4.9. (Subject Reduction Theorem)

$$\text{If } ? \vdash^{\delta} d : D \text{ and } ? \vdash d \to_{\beta\delta} d' \text{ then } ? \vdash^{\delta} d' : D.$$

Proof: The following properties are proved simultaneously by induction on the derivation of $? \vdash^{\delta} d : D$.

1. If $? \vdash d \to_{\beta\delta} d'$ and $? \vdash^{\delta} d : D$ then $? \vdash^{\delta} d' : D$
2. If $? \to_{\beta\delta} ?'$ and $? \vdash^{\delta} d : D$ then $?' \vdash^{\delta} d : D$

We only give the proof for some cases of the first property. Suppose that the last rule in the derivation of $? \vdash^{\delta} d : D$ is:

- (application) $\dfrac{? \vdash^{\delta} b : (\Pi x{:}A.\ B) \quad ? \vdash^{\delta} a : A}{? \vdash^{\delta} (b\ \ a) : B[x := a]}$

  Only one case is considered.

  Assume $b = (\lambda x{:}A_1.\ b_1)$ and $? \vdash (\lambda x{:}A_1.\ b_1)a \to_{\beta} b_1[x := a]$. Generation lemma part 4 and confluence for $\beta\delta$, yield $A \twoheadleftarrow\twoheadrightarrow_{\beta\delta} A_1$ and by also using the conversion rule we have $?, x : A_1 \vdash^{\delta} b_1 : B$. By generation lemma part 3, it follows that $? \vdash^{\delta} A_1 : s$ and by using the conversion rule we have that $? \vdash^{\delta} a : A_1$. By substitution lemma 11.4.8 part 2, it follows that $? \vdash^{\delta} b_1[x := a] : B[x := a]$.

- (δ -introduction) $\dfrac{?\,, x{=}a{:}A \vdash^{\delta} b : B \quad ? \vdash^{\delta} (x{=}a{:}A\ in\ B) : s}{? \vdash^{\delta} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)}$

  Only one case is considered.

  Assume $? \vdash (x{=}a{:}A\ in\ b) \rightarrow_{\delta} b$ under the hypothesis $x \notin FV(b)$.

  By lemma 11.4.4, it follows that $? \vdash^{\delta} a : A$. By the substitution lemma 11.4.8 part 1, it follows that $? \vdash^{\delta} b[x := a] : B[x := a]$. Since $x \notin FV(b)$, the equality $b[x := a] = b$ holds.

  By corollary 11.3.5, it follows that $? \vdash (x{=}a{:}A\ in\ B) \twoheadleftarrow\!\!\twoheadrightarrow_{\delta} B[x := a]$.

  By conversion rule, $? \vdash^{\delta} b : (x{=}a{:}A\ in\ B)$.

The rest of the cases are easy to prove. □

Theorem 11.4.10. (Type Reduction Theorem)

$$\text{If } ? \vdash^{\delta} d : D \text{ and } ? \vdash D \rightarrow_{\beta\delta} D' \text{ then } ? \vdash^{\delta} d : D'.$$

The previous theorem is proved using correctness of types lemma and subject reduction theorem.

We extend the mapping $|-|_{-}$ to contexts.

Definition 11.4.11. The mapping $|-| : \mathcal{C}_{\delta} \rightarrow \mathcal{C}$ is defined as follows.

$$\begin{array}{rcl} |\epsilon| & = & \epsilon \\ |?\,, x{:}A| & = & |?\,|, x{:}|A|_{\Gamma} \\ |?\,, x{=}a{:}A| & = & |?\,| \end{array}$$

Note that if $? \in \mathcal{C}$ then $|?\,| = ?$. This means that for contexts that do not contain definitions the mapping $|-|$ is the identity. This mapping $|-|$ is the projection from $\mathcal{C}_{\delta}$ to $\mathcal{C}$.

In the following theorem we prove that the range of $|-|$ restricted to the set of typable terms in $\lambda^{\delta}(S)$ is the set of typable terms in $\lambda(S)$.

Theorem 11.4.12. If $? \vdash^{\delta} a : A$ then $|?\,| \vdash |a|_{\Gamma} : |A|_{\Gamma}$.

Proof: Suppose that the last rule in the derivation of $? \vdash^{\delta} a : A$ is:

- (δ- start) $\dfrac{? \vdash^{\delta} a : A}{?\,, x{=}a{:}A \vdash^{\delta} x : A}$ where x is $?$-fresh.

  By induction $|?\,| \vdash |a|_{\Gamma} : |A|_{\Gamma}$. By definition 11.3.13 we have that $|x|_{\Gamma, x=a:A} = |a|_{\Gamma}$. By lemma 11.3.15 part 2, it follows that $|A|_{\Gamma, x=a:A} = |A|_{\Gamma}$. Hence $|?\,, x{=}a{:}A| \vdash |x|_{\Gamma, x=a:A} : |A|_{\Gamma, x=a:A}$.

- (δ-introduction) $\dfrac{?\,, x{=}a{:}A \vdash^{\delta} b : B \quad ? \vdash^{\delta} (x{=}a{:}A\ in\ B) : s}{? \vdash^{\delta} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)}$.

  By induction, $|?\,, x{=}a{:}A| \vdash |b|_{\Gamma, x=a:A} : |B|_{\Gamma, x=a:A}$. By lemma 11.3.16 and definition 11.3.13 we have that

$$\begin{array}{rcl} |b|_{\Gamma, x=a:A} & = & |b|_{\Gamma}[x := |a|_{\Gamma}] \\ & = & |x{=}a{:}A\ in\ b|_{\Gamma} \\ |B|_{\Gamma, x=a:A} & = & |B|_{\Gamma}[x := |a|_{\Gamma}] \\ & = & |x{=}a{:}A\ in\ B|_{\Gamma}. \end{array}$$

  Hence $|?\,| \vdash |x{=}a{:}A\ in\ b|_{\Gamma} : |x{=}a{:}A\ in\ B|_{\Gamma}$.

The rest of the cases are easy to prove. □

The first part of the next corollary states that a term is typable in $\lambda(S)$ iff it is typable in $\lambda^{\delta}(S)$. The second part states that a type is inhabited in $\lambda(S)$ if and only if it is inhabited in $\lambda^{\delta}(S)$. If we interpret types as propositions and terms as proofs, this means that a proposition is provable in $\lambda(S)$ iff it is provable in $\lambda^{\delta}(S)$.

Corollary 11.4.13. (Conservativity) Let $a \in \mathcal{T}$ and $? \in \mathcal{C}$. Then

1. $\exists A \ \ ? \vdash a : A$ iff $\exists A \ \ ? \vdash^{\delta} a : A$.
2. Let $A \in \mathcal{T}$. Then $? \vdash a : A$ iff $\ ? \vdash^{\delta} a : A$.
3. $\exists A \ \ ? \vdash A : a$ iff $\exists A \ \ ? \vdash^{\delta} A : a$.

As a consequence of conservativity, we have the following result.

Corollary 11.4.14. (Undecidability)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The problems of type inference, type checking and inhabitation are undecidable in $\lambda^{\delta}(S)$ if they are undecidable in $\lambda(S)$.

Therefore, type inference and type checking is undecidable in inconsistent pure type system with definitions whose specification is impredicative and non-dependent (see theorem 9.3.20). The problem of inhabitation in the systems of the λ^{δ}-cube (the λ-cube extended with definitions) except for $\lambda^{\delta}_{\rightarrow}$ and $\lambda^{\delta}\underline{\omega}$ is undecidable.

Theorem 11.4.15. (Implementing morphism from $\lambda^{\delta}(S)$ to $\lambda(S)$) The pair $(|\_|, |\_|\_)$ of projections mappings is an implementing morphism from $\lambda^{\delta}(S)$ to $\lambda(S)$.

Proof: By theorem 11.4.12, the pair $(|\_|, |\_|\_)$ preserves the typing from $\lambda^{\delta}(S)$ to $\lambda(S)$. It follows from lemmas 11.3.18 and 11.3.20 that $(|\_|, |\_|\_)$ is an implementing morphism from $\lambda^{\delta}(S)$ to $\lambda(S)$. □

We write $|\ |$ instead of $(|\_|, |\_|\_)$.

Note that the previous theorem in fact shows that there is a natural transformation from λ^δ to λ when these are considered as functores from the category of specifications into a category analogous to **Carst** where morphisms are the implementing morphisms.

Corollary 11.4.16. (Semantics) A semantics for $\lambda(S)$ is a semantics for $\lambda^\delta(S)$.

Proof: Suppose there is an intepretation f from $\lambda(S)$ to some environmental abstract rewriting system with typing $\mathcal{A}$. The proof is shown in the following diagram.

$$\lambda^\delta(S) \xrightarrow{|\ |} \lambda(S) \xrightarrow{f} \mathcal{A}$$

□

We apply the uniqueness of types criteria of chapter 4 to prove the following theorem.

Theorem 11.4.17. (Uniqueness of Types)
Let S be a singly sorted specification. Then $\lambda^\delta(S)$ verifies uniqueness of types.

Proof: By lemma 11.3.17, we know that $|\ |$ is a strategy for δ-reduction. Moreover by lemma 11.4.12, this mapping preserves the typing. Since S is singly sorted, we have that $\lambda(S)$ verifies uniqueness of types. It follows from lemma 4.3.13 that if $\lambda(S)$ verifies uniqueness of types so does $\lambda^\delta(S)$. □

11.4.2 Strengthening

In this section we will prove strengthening. This is a property we expect to hold: if in a derivation a variable x occurs only in the declaration $x{:}A$ in the context then we should be able to construct a derivation omitting this variable from the context. When the variable has been declared as a definition $x{=}a{:}A$ in the context, this property is an immediate consequence of the substitution lemma. We need to prove it for the case of variables which are not definitions.

In order to prove strengthening, we use the corresponding result for pure type systems (see [BJ93]).

Lemma 11.4.18. Let $?'$ be a context. If $?' \twoheadrightarrow_{\beta\delta} ?$ and $? \vdash^\delta a : A$ then $?' \vdash^\delta a : A$.

Proof: By induction on the number of steps in the derivation of $? \vdash^\delta a : A$. Only some cases are considered.

- (formation) $\dfrac{? \vdash^\delta A : s_1 \quad ?, x : A \vdash^\delta B : s_2}{? \vdash^\delta (\Pi x{:}A.B) : s}$ for $(s_1, s_2, s) \in \boldsymbol{R}$.

  By induction we have that $?' \vdash^\delta A : s_1$. Therefore $?', x{:}A$ is a context. By induction we have that $?', x : A \vdash^\delta B : s_2$. Hence

$$\frac{?' \vdash^\delta A : s_1 \quad ?', x : A \vdash^\delta B : s_2}{?' \vdash^\delta (\Pi x{:}A.B) : s}$$

- (δ - formation) $\dfrac{?\,, x{=}a{:}A \vdash^{\delta} B : s}{? \vdash^{\delta} (x{=}a{:}A\ in\ B) : s}$.

  There exists a derivation of $? \vdash^{\delta} a : A$ that has less number of steps than this derivation of $?\,, x = a : A \vdash^{\delta} B : s$. By induction we have that $?' \vdash^{\delta} a : A$. Hence $?', x{=}a{:}A$ is a context. By induction we have that $?', x{=}a{:}A \vdash^{\delta} B : s$.

- (conversion) $\dfrac{? \vdash^{\delta} b : B \quad ? \vdash^{\delta} B' : s_0 \quad ? \vdash B \twoheadleftarrow\twoheadrightarrow_{\beta\delta} B'}{? \vdash^{\delta} b : B'}$

  By induction we have that $?' \vdash^{\delta} b : B$ and that $?' \vdash^{\delta} B' : s_0$. By lemma 11.3.22 we have that $?' \vdash B \twoheadleftarrow\twoheadrightarrow_{\beta\delta} B'$.

The rest of the cases are easy to prove. □

Lemma 11.4.19. If $?_1, z{:}D, ?_2 \vdash^{\delta} e : E$ and $z \notin FV(?_2) \cup FV(e)$ then there exists E' such that $?_1, ?_2 \vdash E \twoheadrightarrow_{\beta\delta} E'$ and $?_1, ?_2 \vdash^{\delta} e : E'$.

Proof: By induction on the number of steps in the derivation of $?_1, z{:}D, ?_2 \vdash^{\delta} e : E$. Only some cases are considered.

- (abstraction) Suppose that the last rule in the derivation is the abstraction:

$$\frac{?_1, z{:}D, ?_2, x : A \vdash^{\delta} b : B \quad ?_1, z{:}D, ?_2 \vdash^{\delta} (\Pi x{:}A.\ B) : s}{?_1, z{:}D, ?_2 \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

  By induction, there exists B' such that $?_1, ?_2 \vdash B \twoheadrightarrow_{\beta\delta} B'$ and that:

$$?_1, ?_2, x : A \ \vdash^{\delta}\ b : B' \tag{i}$$

  By generation lemma there exist s_1 and s_2 such that $(s_1, s_2, s) \in \boldsymbol{R}$.

$$?_1, z{:}D, ?_2 \ \vdash^{\delta}\ A : s_1 \tag{ii}$$
$$?_1, z{:}D, ?_2, x{:}A \ \vdash^{\delta}\ B : s_2 \tag{iii}$$

  There exists a derivation of $?_1, z{:}D, ?_2 \vdash^{\delta} A : s_1$ that has less number of steps than the derivation of $?_1, z{:}D, ?_2 \vdash^{\delta} (\Pi x{:}A.\ B) : s$. It follows by induction that

$$?_1, ?_2 \ \vdash^{\delta}\ A : s_1 \tag{iv}$$

  We can not apply the induction hypothesis to (iii) because the variable z may occur in B.
  Since the mapping $|\_|_{\_}$ preserves the typing, in particular for (iii) we have that:

$$|?_1, z{:}D, ?_2, x{:}A| \vdash |B|_{\Gamma_1, z{:}D, \Gamma_2, x{:}A} : s_2$$

  Note that $|B|_{\Gamma_1, z{:}D, \Gamma_2, x{:}A} = |B|_{\Gamma_1, \Gamma_2}$.

It follows from subject reduction theorem for pure type systems that

$$|?_1, z{:}D, ?_2, x{:}A| \vdash |B'|_{\Gamma_1,\Gamma_2,x:A} : s_2$$

The variable z does not occur in $|B'|_{\Gamma_1,\Gamma_2,x:A}$. By strengthening for pure type systems,

$$|?_1, ?_2, x{:}A| \quad \vdash \quad |B'|_{\Gamma_1,\Gamma_2} : s_2 \tag{v}$$

By lemma 11.4.18,

$$?_1, ?_2, x{:}A \vdash^{\delta} |B'|_{\Gamma_1,\Gamma_2} : s_2 \tag{vi}$$

This is a derivation of $?_1, ?_2 \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ |B'|_{\Gamma_1,\Gamma_2})$:

$$\dfrac{\dfrac{?_1, ?_2 \vdash B' \twoheadleftarrow\twoheadrightarrow_{\beta\delta} |B'|_{\Gamma_1,\Gamma_2} \quad \text{(i)} \quad \text{(vi)}}{?_1, ?_2, x : A \vdash^{\delta} b : |B'|_{\Gamma_1,\Gamma_2}} \quad \dfrac{\text{(iv)} \quad \text{(vi)}}{?_1, ?_2 \vdash^{\delta} (\Pi x{:}A.\ |B'|_{\Gamma_1,\Gamma_2}) : s}}{?_1, ?_2 \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ |B'|_{\Gamma_1,\Gamma_2})}$$

- (δ -introduction) $\dfrac{?_1, z{:}D, ?_2, x{=}a{:}A \vdash^{\delta} b : B \quad ?_1, z{:}D, ?_2 \vdash^{\delta} (x{=}a{:}A\ in\ B) : s}{?_1, z{:}D, ?_2 \vdash^{\delta} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)}$.

  By induction, there exists B' such that $?_1, ?_2, x{=}a{:}A \vdash B \twoheadrightarrow_{\beta\delta} B'$ and that:

  $$?_1, ?_2, x{=}a{:}A \quad \vdash^{\delta} \quad b : B' \tag{i}$$

  We can not apply the induction hypothesis to $?_1, z : D, ?_2 \vdash^{\delta} (x = a : A\ in\ B) : s$ because the variable z may occur in B.
  We apply correctness of types to (i) and we have that either $?_1, ?_2, x{=}a{:}A \vdash^{\delta} B' : s'$ or $B' = s'$.

  1. Suppose $?_1, ?_2, x{=}a{:}A \vdash^{\delta} B' : s'$. This is a derivation of $?_1, ?_2 \vdash^{\delta} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B')$ with $?_1, ?_2, x{=}a{:}A \vdash B \twoheadrightarrow_{\beta\delta} B'$.

     $$\dfrac{?_1, ?_2, x{=}a{:}A \vdash^{\delta} b : B' \quad \dfrac{?_1, ?_2, x{=}a{:}A \vdash^{\delta} B' : s'}{?_1, ?_2 \vdash^{\delta} (x{=}a{:}A\ in\ B') : s'}}{?_1, ?_2 \vdash^{\delta} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B')}$$

     Note that this case is not difficult like the case of the abstraction rule. The proof of the case for the abstraction rule is complicated because the Π-formation rule has more restrictions than the δ-formation rule.

  2. Suppose $B' = s'$. By generation lemma $?_1, z{:}D, ?_2, x{=}a{:}A \vdash^{\delta} B : s$. By subject reduction theorem we have that $?_1, z{:}D, ?_2, x{=}a{:}A \vdash^{\delta} B' : s$. Then (s', s) is an axiom. Since $?_1, ?_2$ is a context we have that

     $$?_1, ?_2, x{:}A \vdash^{\delta} s' : s \tag{ii}$$

This is a derivation of $?_1, ?_2 \vdash^\delta (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ s')$.

$$\frac{?_1, ?_2, x{=}a{:}A \vdash^\delta b : s' \qquad \dfrac{?_1, ?_2, x{=}a{:}A \vdash^\delta s' : s}{?_1, ?_2 \vdash^\delta (x{=}a{:}A\ in\ s') : s}}{?_1, ?_2 \vdash^\delta (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ s')}$$

The rest of the cases are easy to prove. □

Strengthening for arbitrary specifications follows immediately from the previous lemma.

Theorem 11.4.20. (Strengthening for arbitrary specifications)
If $?_1, x{:}A, ?_2 \vdash^\delta b : B$ and $x \notin FV(?_2) \cup FV(b) \cup FV(B)$ then $?_1, ?_2 \vdash^\delta b : B$.

11.4.3 Weak and Strong Normalisation for $\beta\delta$-reduction

In this section we prove that if $\lambda(S)$ is weakly normalising so is $\lambda^\delta(S)$. Also we prove that a pure type system with definitions $\lambda^\delta(S)$ is $\beta\delta$-strongly normalising if a slightly larger pure type system $\lambda(S')$ is β-strongly normalising. The idea of the proof of strong normalisation is as follows.

- We define a mapping $\{\_\}\_ : \mathcal{T}_\delta \times \mathcal{C}_\delta \to \mathcal{T}$ similar to the projection $|\ \_\ |\_$. The value $\{a\}_\Gamma$ is a term that is obtained from a by unfolding all the global and local definitions. However $\{\_\}\_$ differs from $| - |\_$ in the value given to $(x = a : A\ in\ b)$. Instead of removing the local definition it is translated to a β-redex, i.e. an application and an abstraction.

- This function $\{\_\}\_$ maps an infinite $\beta\delta$ reduction sequence to an infinite β reduction sequence.

- The function $\{\_\}\_$ maps terms that are typable in a DPTS $\lambda^\delta(S)$ to terms that are typable in a PTS slightly larger than $\lambda(S)$.

Theorem 11.4.21. (Weak Normalisation for $\beta\delta$)
Let S be a specification.

$\lambda(S)$ is β-weakly normalising if and only if $\lambda^\delta(S)$ is $\beta\delta$-weakly normalising.

Proof: By theorem 11.3.23, the mapping $|\ |$ computes the δ-normal form. By 11.4.15 this mapping is an implementing morphism from $\lambda^\delta(S)$ to $\lambda(S)$. It follows from lemma 4.3.9 that if $\lambda(S)$ is β-weakly normalising then $\lambda^\delta(S)$ is $\beta\delta$-weakly normalising. □

The function $|\ \_\ |\_$ is not a refining morphism. It may not map infinite β-reduction sequences to infinite β-reduction sequences as the following example shows.

Example 11.4.22. Suppose there is an infinite β-reduction sequence starting at a and hence at $(x{=}a{:}A\ in\ b)$. If $x \notin FV(b)$ and b is a $\beta\delta$ normal form then there is no β-reduction sequence starting at $\mid x{=}a{:}A\ in\ b \mid_\epsilon = b$.

The mapping $\mid \_ \mid_\_$ erases the term a which could contain an infinite reduction sequence. We define a new function $\{\_\}_\_$ that is a refining morphism.

Definition 11.4.23. The mapping $\{\_\}_\_ : \mathcal{C}_\delta \times \mathcal{T}_\delta \to \mathcal{T}$ is defined as follows.

$$
\begin{array}{rcl}
\{x\}_\Gamma & = & \begin{cases} \{a\}_{\Gamma_1} & \text{if } ? = ?_1, x{=}a{:}A, ?_2 \\ x & \text{otherwise} \end{cases} \\
\{s\}_\Gamma & = & s \\
\{a\ b\}_\Gamma & = & \{a\}_\Gamma \{b\}_\Gamma \\
\{\lambda x{:}A.\ b\}_\Gamma & = & \lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma, x:A} \\
\{\Pi x{:}A.\ b\}_\Gamma & = & \Pi x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma, x:A} \\
\{x{=}a{:}A\ in\ b\}_\Gamma & = & (\lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma, x=a:A})\{a\}_\Gamma
\end{array}
$$

Like $\mid \_ \mid_\_$, the value $\{a\}_\Gamma$ is the unfolding of all the definitions occurring in the context $?$ and in the term a. Global definitions are unfolded in the first line $\{x\}_\Gamma = \{a\}_{\Gamma_1}$. Local definitions are unfolded in the last line since $\{b\}_{\Gamma, x=a:A} = \{b\}_\Gamma[x := \{a\}_\Gamma]$. However $\{\_\}_\_$ differs from $\mid - \mid_\_$ in the value given to $(x = a : A\ in\ b)$. Instead of removing the local definition, it is translated to a β-redex. The bound variable x of the local definition is transformed into the bound variable of an abstraction. The abbreviation a is transformed into the argument of an application.

Example 11.4.24. Recall that in example 11.2.20 we show that $(\lambda x{:}A.\ b)a$ may not be typable when $(x{=}a{:}A\ in\ b)$ is. Let $e = \lambda\alpha{:}*.\ (x{=}\alpha{:}*\ in\ \lambda y{:}x.\ \lambda f{:}\alpha \to \alpha.\ fy)$ be the term used in example 11.2.20. The corresponding term expressed as an application and an abstraction is not typable in any system of the λ-cube. But

$$\{e\}_\Gamma = \lambda\alpha{:}*.\ (\lambda x{:}*.\ \lambda y{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ fy)\alpha$$

is typable in $\lambda 2$. This is because the definition of x is unfolded by α and then x does not occur in the expression $\lambda y{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ fy$.

The mapping $\{\_\}_\_$ is extended to contexts.

Definition 11.4.25. The mapping $\{-\} : \mathcal{C}_\delta \to \mathcal{C}$ is defined as follows.

$$
\begin{array}{rcl}
\{\epsilon\} & = & \epsilon \\
\{?, x : A\} & = & \{?\}, x : \{A\}_\Gamma \\
\{?, x{=}a{:}A\} & = & \{?\}, x : \{A\}_\Gamma
\end{array}
$$

Similar properties proved for the projection $|\_|\_$ hold for the function $\{\_\}\_$.

Lemma 11.4.26.

1. If x is ?-fresh and $x \notin FV(b)$ then $x \notin FV(\{b\}_\Gamma)$.
2. Let $< ?_1, ?_2, ?_3 > \in \mathcal{C}_\delta$ and $b \in \mathcal{T}_\delta$ be such that $(FV_{\Gamma_3}(b)) \cap Def(?_2) = \emptyset$. Then
$$\{b\}_{\Gamma_1,\Gamma_2\Gamma_3} = \{b\}_{\Gamma_1,\Gamma_3}.$$
3. Let $< ?_1, y{=}a{:}A, ?_2 > \in \mathcal{C}_\delta$. Then $\{a\}_{\Gamma_1,y=a:A,\Gamma_2} = \{a\}_{\Gamma_1}$.

The following lemma is proved by induction on the structure of the term b.

Lemma 11.4.27. $\{b\}_\Gamma[x := \{a\}_\Gamma] \;=\; \{b[x := a]\}_\Gamma \;=\; \{b\}_{\Gamma,x=a:A}$

Lemma 11.4.28. The mapping $\{\_\}$ is an implementing morphism from $(\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_\delta)$ to $(\mathcal{T}, \rightarrow_\beta)$. More precisely, if $? \vdash c \rightarrow_\delta d$ then $\{c\}_\Gamma \rightarrow_\beta^= \{d\}_\Gamma$ for all $c, d \in \mathcal{T}_\delta$ and $? \in \mathcal{C}_\delta$.

Proof: This is proved by induction on the structure of c. Only some cases are considered.

- Suppose $c = x$. This means that $? = ?_1, x = d : D, ?_2$ and $? \vdash x \rightarrow_\delta d$.
$$\begin{array}{rcll} \{x\}_{\Gamma_1,x=d:D,\Gamma_2} & = & \{d\}_{\Gamma_1} & \\ & = & \{d\}_{\Gamma_1,x=d:D,\Gamma_2} & \text{by lemma 11.4.26 part 3} \end{array}$$
- Suppose $c = (x{=}a{:}A\ in\ b)$ and $? \vdash (x{=}a{:}A\ in\ b) \rightarrow_\delta b$ with $x \notin FV(b)$.
$$\begin{array}{rcll} \{x{=}a{:}A\ in\ b\} & = & (\lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma,x=a:A})\{a\}_\Gamma & \\ & \rightarrow_\beta & \{b\}_{\Gamma,x=a:A}[x := \{a\}_\Gamma] & \\ & = & \{b\}_{\Gamma,x=a:A} & \text{by lemma 11.4.26} \\ & = & \{b\}_\Gamma & \end{array}$$

The rest of the cases are easy to prove. □

By the following lemma, $\{\_\}\_$ maps an infinite $\beta\delta$-reduction sequence to an infinite β-reduction sequence.

Lemma 11.4.29. The mapping $\{\_\}$ is a refining morphism from $(\mathcal{T}_\delta, \rightarrow_\beta)$ to $(\mathcal{T}, \rightarrow_\beta)$, i.e. if $c \rightarrow_\beta d$ then $\{c\}_\Gamma \rightarrow_\beta^+ \{d\}_\Gamma$.

Proof: This is proved by induction on the structure of c. Only the case $c = (\lambda x{:}A.\ b)a$ with $(\lambda x{:}A.\ b)a \rightarrow_\beta b[x := a]$ is considered.

$$\begin{array}{rcll} \{c\}_\Gamma & = & \{\lambda x{:}A.\ b\}_\Gamma\{a\}_\Gamma & \\ & = & (\lambda x{:}\{A\}_\Gamma.\ \{b\}_\Gamma)\{a\}_\Gamma & \\ & \rightarrow_\beta & \{b\}_\Gamma[x := \{a\}_\Gamma] & \\ & = & \{b[x := a]\}_\Gamma & \text{by lemma 11.4.27} \end{array}$$

□

Definition 11.4.30. The specification $S = (\mathbf{S}, \mathbf{A}, \mathbf{R})$ is called *quasi-full* if for all s_1, $s_2 \in \mathbf{S}$ there exists $s_3 \in \mathbf{S}$ such that $(s_1, s_2, s_3) \in \mathbf{R}$.

Note that if a specification is full then it is quasi-full. But the converse is not true.

Definition 11.4.31. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $S' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that

1. $\boldsymbol{S} \subseteq \boldsymbol{S}'$, $\boldsymbol{A} \subseteq \boldsymbol{A}'$, and $\boldsymbol{R} \subseteq \boldsymbol{R}'$
2. S' is quasi-full
3. for all $s \in \boldsymbol{S}$ there is a sort $s' \in \boldsymbol{S}'$ such that $(s : s') \in \boldsymbol{A}'$ (i.e. the sorts of S are not topsorts in S').

Then the specification S' is called *a completion* of S.

Example 11.4.32. The specification C_∞ is a completion of C, HOL and itself.

This definition is necessary in order to prove that $\{\_\}\_$ maps terms that are typable in a pure type system with definitions $\lambda^\delta(S)$ to terms that are typable in a slightly larger pure type system $\lambda S'$ with S' a completion of S.
Remember that $\{\_\}\_$ translates a local definition to a λ-abstraction with an argument:

$$\{x{=}a{:}A\ in\ b\}_\Gamma = (\lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma, x=a:A})\{a\}_\Gamma$$

Condition 2 is necessary to ensure that all these λ-abstractions introduced by $\{\_\}\_$ are allowed in S'. The typing of the abstraction is restricted by the set $\boldsymbol{R}$ of rules whereas the typing of $(x = a : A\ in\ b)$ is not. Let $e = (x = \alpha : *\ in\ \lambda y : x.\ \lambda f : x \to x.\ fy)$ be the term in the example 11.2.21. This term is typable in the system $\lambda^\delta_\to$ but the term $\{e\}_\epsilon = (\lambda x{:}*.\ \lambda y{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ fy)\alpha$ is not typable in $\lambda_\to$.
Condition 3 is necessary because we can not type these abstractions introduced by $\{\_\}\_$ if A is a topsort. For example the term $(x = * : \Box\ in\ x)$ is typable in $\lambda^\delta C$ but

$$\{x = * : \Box\ in\ x\}_\epsilon\ =\ (\lambda x{:}\Box.\ *)*$$

is not typable in λC.

The next lemma is proved by induction on the derivation.

Lemma 11.4.33. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. If s occurs either in A or in a or in ? and ? $\vdash^\delta_S a : A$ then $s \in \boldsymbol{S}$.

The next lemma states that a $\lambda S'$ type that is in the range of $\{\_\}\_$ cannot be one of the topsorts of $\lambda S'$.

Lemma 11.4.34. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $S' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that S' is a completion of S. Then If $\Delta \vdash^\delta_S a : A$ and $\Delta \vdash^\delta_{S'} \{a\}_\Gamma : \{A\}_\Gamma$ then $\Delta \vdash^\delta_{S'} \{A\}_\Gamma : s$.

Proof: Assume $\Delta \vdash^{\delta}_{S'} \{a\}_\Gamma : \{A\}_\Gamma$. By correctness of types $\Delta \vdash^{\delta}_{S'} \{A\}_\Gamma : s'$ or $\{A\}_\Gamma = s$. Suppose $\{A\}_\Gamma = s$. Since $\Delta \vdash^{\delta}_{S} a : A$ and by lemma 11.4.33 we have that $s \in \boldsymbol{S}$. By the condition (3) of the definition of completion, there is a sort $s' \in \boldsymbol{S'}$ such that $(s, s') \in \boldsymbol{A'}$ and hence $\Delta \vdash^{\delta}_{S'} s : s'$, i.e. $\Delta \vdash^{\delta}_{S'} \{A\}_\Gamma : s'$. $\square$

The next theorem states that $\{\_\}_\_$ maps terms that are typable in $\lambda^\delta(S)$ to terms that are typable in the pure type system $\lambda S'$ with S' a completion of S.

Theorem 11.4.35. Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $S' = (\boldsymbol{S'}, \boldsymbol{A'}, \boldsymbol{R'})$ be such that S' is a completion of S. Then $? \vdash^{\delta}_{S} a : A \;\Rightarrow \{?\} \vdash^{\delta}_{S'} \{a\}_\Gamma : \{A\}_\Gamma$.

Proof: By induction on the derivation of $? \vdash^{\delta}_{S} a : A$. Only some cases are considered.

- (δ-start) $\dfrac{? \vdash^{\delta}_{S} a : A}{?, x = a : A \vdash^{\delta}_{S} x : A}$

  By induction $\{?\} \vdash^{\delta}_{S'} \{a\}_\Gamma : \{A\}_\Gamma$. By definition we have that $\{x\}_{\Gamma, x=a:A} = \{a\}_\Gamma$. By lemma 11.4.26 part 2 we have that $\{A\}_{\Gamma, x=a:A} = \{A\}_\Gamma$.

  It follows from lemma 11.4.34 that $\{?\} \vdash^{\delta}_{S'} \{A\}_\Gamma : s$. By weakening rule, we have that $\{?\}, x : \{A\}_\Gamma \vdash^{\delta}_{S'} \{x\}_{\Gamma, x=a:A} : \{A\}_\Gamma$.

- (δ-weakening) $\dfrac{? \vdash^{\delta}_{S} b : B \quad ? \vdash^{\delta}_{S} a : A}{?, x = a : A \vdash^{\delta}_{S} b : B}$

  By induction, $\{?\} \vdash^{\delta}_{S'} \{b\}_\Gamma : \{B\}_\Gamma$ and $\{?\} \vdash^{\delta}_{S'} \{a\}_\Gamma : \{A\}_\Gamma$. Then by lemma 11.4.34, we have that $\{?\} \vdash^{\delta}_{S'} \{A\}_\Gamma : s$. By weakening rule we have that $\{?\}, x : \{A\}_\Gamma \vdash^{\delta}_{S'} \{b\}_\Gamma : \{B\}_\Gamma$.

  We have that $x \notin FV(b)$ and $x \notin FV(B)$. By lemma 11.4.26, we have that $\{b\}_{\Gamma, x=a:A} = \{b\}_\Gamma$ and that $\{B\}_{\Gamma, x=a:A} = \{B\}_\Gamma$.

- (δ-formation) $\dfrac{?, x{=}a{:}A \vdash^{\delta}_{S} B : s}{? \vdash^{\delta}_{S} (x{=}a{:}A \; in \; B) : s}$

  By induction

  $$\{?, x{=}a{:}A\} \;\vdash^{\delta}_{S'}\; \{B\}_{\Gamma, x=a:A} : s \tag{i}$$

  The derivation of $?, x{=}a{:}A \vdash^{\delta}_{S} B : s$ contains a (shorter) derivation of $? \vdash^{\delta}_{S} a : A$, so also by induction

  $$\{?\} \;\vdash^{\delta}_{S'}\; \{a\}_\Gamma : \{A\}_\Gamma \tag{ii}$$

  By lemma 11.4.34 it follows from (i) and (ii) that there are $s_1, s_2 \in \boldsymbol{S'}$ such that

  $$\{?, x{=}a{:}A\} \;\vdash^{\delta}_{S'}\; s : s_2 \tag{iii}$$
  $$\{?\} \;\vdash^{\delta}_{S'}\; \{A\}_\Gamma : s_1 \tag{iv}$$

The following is a derivation of $\{?\} \vdash^{\delta}_{S'} \{x{=}a{:}A\ in\ B\}_\Gamma : s$.

$$\dfrac{\dfrac{\dfrac{\text{(iii)} \quad \text{(iv)}}{\{?\} \vdash^{\delta}_{S'} (\Pi x{:}\{A\}_\Gamma.\ s) : s_3}(prod) \quad \text{(i)}}{\{?\} \vdash^{\delta}_{S'} (\lambda x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A}) : (\Pi x{:}\{A\}_\Gamma.\ s)}(abs) \quad \text{(ii)}}{\{?\} \vdash^{\delta}_{S'} (\lambda x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A})\{a\}_\Gamma : s[x := \{a\}_\Gamma]}(app)$$

and $(\lambda x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A})\{a\}_\Gamma = \{x{=}a{:}A\ in\ B\}_\Gamma$.

- (δ-introduction) $\dfrac{?\,, x{=}a{:}A \vdash^{\delta}_{S} b : B \quad ? \vdash^{\delta}_{S} (x{=}a{:}A\ in\ B) : s}{? \vdash^{\delta}_{S} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)}$

  By induction

$$\begin{aligned} \{?\,, x{=}a{:}A\} \quad &\vdash^{\delta}_{S'} \quad \{b\}_{\Gamma,x=a:A} : \{B\}_{\Gamma,x=a:A} && \text{(i)} \\ \{?\} \quad &\vdash^{\delta}_{S'} \quad \{x{=}a{:}A\ in\ B\}_\Gamma : s && \text{(ii)} \end{aligned}$$

  The derivation of $?\,, x{=}a{:}A \vdash^{\delta}_{S} b : B$ contains a (shorter) derivation of $? \vdash^{\delta}_{S} a : A$, so also by the induction

$$\{?\} \quad \vdash^{\delta}_{S'} \quad \{a\}_\Gamma : \{A\}_\Gamma \qquad \text{(iii)}$$

  By lemma 11.4.34 it follows from (i) and (iii) that there are $s_1, s_2 \in \boldsymbol{S'}$ such that

$$\begin{aligned} \{?\,, x{=}a{:}A\} \quad &\vdash^{\delta}_{S'} \quad \{B\}_{\Gamma,x=a:A} : s_2 && \text{(iv)} \\ \{?\} \quad &\vdash^{\delta}_{S'} \quad \{A\}_\Gamma : s_1 && \text{(v)} \end{aligned}$$

  Then

$$\dfrac{\dfrac{\dfrac{\text{(iv)} \quad \text{(v)}}{\{?\} \vdash^{\delta}_{S'} (\Pi x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A}) : s_3}(prod) \quad \text{(i)}}{\{?\} \vdash^{\delta}_{S'} (\lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma,x=a:A}) : (\Pi x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A})}(abs) \quad \text{(iii)}}{\{?\} \vdash^{\delta}_{S'} (\lambda x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A})\{a\}_\Gamma : \{B\}_{\Gamma,x=a:A}[x := \{a\}_\Gamma]}(app)$$

  $\{x{=}a{:}A\ in\ b\}_\Gamma = (\lambda x{:}\{A\}_\Gamma.\ \{b\}_{\Gamma,x=a:A})\{a\}_\Gamma$ and

$$\begin{aligned} \{x{=}a{:}A\ in\ B\}_\Gamma \quad &= \quad (\lambda x{:}\{A\}_\Gamma.\ \{B\}_{\Gamma,x=a:A})\{a\}_\Gamma \\ &\twoheadleftrightarrow_\beta \quad \{B\}_{\Gamma,x=a:A}[x := \{a\}_\Gamma] && \text{(vi)} \end{aligned}$$

  so using the conversion rule

$$\dfrac{\{?\} \vdash^{\delta}_{S'} \{x{=}a{:}A\ in\ b\}_\Gamma : \{B\}_{\Gamma,x=a:A}[x := \{a\}_\Gamma] \quad \text{(ii)} \quad \text{(vi)}}{\{?\} \vdash^{\delta}_{S'} \{x{=}a{:}A\ in\ b\}_\Gamma : \{x{=}a{:}A\ in\ B\}_\Gamma}(\beta - conv)$$

- (δ-conversion) $\dfrac{? \vdash^{\delta}_{S} b : B \quad ? \vdash^{\delta}_{S} B' : s \quad ? \vdash B \twoheadleftrightarrow_\delta B'}{? \vdash^{\delta}_{S} b : B}$

  It follows from induction that $\{?\} \vdash^{\delta}_{S'} \{b\}_\Gamma : \{B\}_\Gamma$ and $\{?\} \vdash^{\delta}_{S'} \{B'\}_\Gamma : s$. By lemma 11.4.28 it follows from $? \vdash B \twoheadleftrightarrow_\delta B'$ that $\{B\}_\Gamma \twoheadleftrightarrow_\beta \{B'\}_\Gamma$. Then using the β-conversion rule $\{?\} \vdash^{\delta}_{S'} \{b\}_\Gamma : \{B'\}_\Gamma$.

The rest of the cases are easy to prove. □

Theorem 11.4.36. (Implementing morphism from $\lambda^\delta(S)$ to $\lambda(S')$)
Let S' be a completion of S. The pair $(\{\_\}, \{\_\}\_)$ is an implementing morphism from $\lambda^\delta(S)$ to $\lambda(S')$.

Proof: By theorem 11.4.35, the pair $(\{\_\}, \{\_\}\_)$ preserves the typing from $\lambda^\delta(S)$ to $\lambda(S')$. It follows from lemmas 11.4.28 and 11.4.29 that this pair is an implementing morphism from $\lambda^\delta(S')$ to $\lambda(S)$. □

We write $\{\ \}$ instead of $(\{\_\}, \{\_\}\_)$.

Theorem 11.4.37. (Strong Normalisation for $\beta\delta$)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $S' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that S' is a completion of S.
If $\lambda(S')$ is β-strongly normalising, then $\lambda^\delta(S)$ is $\beta\delta$-strongly normalising.

Proof: By theorem 11.3.29, the δ-reduction is strongly normalising. It follows from lemma 11.4.29 and theorem 11.4.35 that $\{\ \}$ is a refining morphism from $\lambda^\delta(S')$ with only β-reduction to $\lambda(S)$. Moreover it follows from lemma 11.4.28 and theorem 11.4.35 that $\{\ \}$ is an implementing morphism from $\lambda^\delta(S')$ with only δ-reduction to $\lambda(S)$. By lemma 4.3.11, if $\lambda(S')$ is β-strongly normalising, then $\lambda^\delta(S)$ is $\beta\delta$-strongly normalising. □

Corollary 11.4.38. The following systems are strongly normalising:

1. The system $\lambda(C_\infty)$ extended with definitions, i.e. $\lambda^\delta(C_\infty)$.
2. The calculus of constructions extended with definitions, i.e. $\lambda^\delta C$.
3. The system of higher order logic extended with definitions, i.e. $\lambda^\delta(HOL)$ is strongly normalising.

Proof: The system $\lambda(C_\infty)$ is strongly normalising. This specification C_∞ is a completion of itself. Hence it follows from the previous theorem that $\lambda^\delta(C_\infty)$ is strongly normalising. Since $\lambda^\delta(C_\infty)$ contains $\lambda^\delta C$ and $\lambda^\delta(HOL)$, the parts 2 and 3 follow from part 1. □

Theorem 11.4.37 is somewhat unsatisfactory. It would be nicer to prove a stronger property, namely that $\lambda^\delta(S)$ is $\beta\delta$-strongly normalising if $\lambda(S)$ is β-strongly normalising. On the other hand, we do not know any strongly normalising pure type system λS for which theorem 11.4.37 cannot be used to prove strong normalisation of $\lambda^\delta(S)$. In particular, all strongly normalising pure type systems given in [Bar92] have a completion that is $\lambda(C_\infty)$.

11.5 Conclusions and Related Work

In this chapter we have considered definitions as part of the formal language. In our opinion, this extension has been done in a neat and general way (for pure type systems). The inclusion of definitions in the formal language and its study have not been considered before except for the systems of the AUTOMATH family [NGdV94].

Definitions vs local β-reduction. In the systems of the AUTOMATH family, definitions are written as β-redexes [Ned73] [Daa80]. A reduction called *local β-reduction* to unfold one occurrence of a variable at a time is introduced in [Ned92]. In some later versions of AUTOMATH, definitions are connected with local β-reduction. We prefer, as in the original AUTOMATH systems, to have a special constructor for definition and a reduction for the unfolding of definitions to make a clear distinction between definitions and β-redexes.

Global and Local Definitions. Our extension provides global and local definitions, i.e. definitions in the context and in the term. Coq [Dow91] provides only global definitions. In our opinion, it is important to have local definitions as well as global ones for practical use. The study of the meta-theory is rather simple if we have only global definitions and in this case, it is easy to prove that strong normalisation is preserved by the extension.

Definitions vs Abstraction and Application: the third proof of finiteness of developments. As we said before, a definition and a β-redex are very similar. Intuitively, we know that a β-redex $(\lambda x{:}A.\ b)a$ is 'like' a definition $x{=}a{:}A\ in\ b$. However the δ-reduction is *finer* than the β-reduction since one step of β-reduction corresponds to several steps of δ-reduction.

The relation between definitions and β-redexes is formalised by the morphism Υ. This morphism maps a marked redex $(\underline{\lambda} x{:}A.\ b)a$ into a definition $x{=}a{:}A\ in\ b$. Moreover this function maps one step of $\underline{\beta}$-reduction into one or more steps of δ-reduction, i.e. it is a *refining morphism*.

A development is then mapped into a δ-rewrite sequence via the morphism Υ. Since δ is strongly normalising, we conclude that all the developments are finite. This is our third proof of finiteness of developments (see chapter 7) and it is similar to the proof of the same result in [Klo80] using β-reduction with memory.

The length of a development is always smaller than the length of the corresponding δ-rewrite sequence. This can also be formalised. The function $w_{\_}(-)$ used to prove strong normalisation of δ computes the length of a maximal δ-rewrite sequence to the normal form. In [Vri85], a mapping h is defined that computes the length of a maximal development, i.e. the length of a maximal $\underline{\beta}$-rewrite sequence. The mapping $w_{\_}(-)$ composed with Υ is an upper bound for the number of steps in a development. It is easy to verify that $w_{\emptyset}(\Upsilon(a))$ is greater than $h(a)$. It is clear that the difference between $w_{\emptyset}(\Upsilon(a))$ and $h(a)$ is the number of extra-steps needed in the δ-rewrite sequence.

Definitions vs Explicit Substitution. There is a common reason to introduce definitions $x{=}a{:}A\ in\ b$ and explicit sustitution is $b\{x := a\}$: to delay the global substitution of x for a in b and perform the substitution for one occurrence of the variable x at a time because the substitution for all the occurrences at once may not be desirable.

The substitution is said to be *explicit* if it is not a meta-operation on terms but part of the formal language with a special constructor $b\{x := a\}$ and a special rewrite relation β_x

to define 'the substitution behaviour' [ACCL91] [KN93] [BG96] [Tas93].

The β_x-reduction and the δ-reduction are very similar, they both unfold one occurrence of a variable at a time. They differ in the way they perform this unfolding.

In a β_x-rewrite step, the explicit substitution '$\{x := a\}$' is *pushed* inside the structure of the term b until a variable is found. For example in the case of the application

$$(b\ c)\{x := a\} \rightarrow_{\beta_x} (b\{x := a\}\ c\{x := a\})$$

In a δ-rewrite step, the definition $x{=}a{:}A$ is always kept *outside* the term b (in a context).

In the moment of unfolding the variable, the explicit susbtitution '$\{x := a\}$' is inside the term and the unfolding is performed as follows.

$$x\{x := a\} \rightarrow_{\beta_x} a$$

In the moment of unfolding the variable, the definition $x{=}a{:}A$ is taken from the context $?\,, x{=}a{:}A, ?'$ and the unfolding is performed as follows.

$$?\,, x{=}a{:}A, ?' \vdash x \rightarrow_{\delta} a$$

The length of a δ-rewrite sequence depends on the number of occurrences of the variable x in the term b. The β_x-reduction needs more steps than the δ-reduction to perform the unfolding of **one** occurrence of a variable. The extra-steps that β_x needs depend on the structure of the term b. We do not see any reason to have these extra-steps if we just want to perform the unfolding of one occurrence at a time. Definitions achieve this purpose very well besides having all the good properties. On the other hand, it is not clear how to define a type system for explicit substitutions [Blo97] that verifies all the good properties like subject reduction.

The main difference, however, between definitions and explicit substitution is that the explicit substitution is defined together with a rule that creates a β_x-redex from a β-redex.

$$(\lambda x{:}A.\ b)a \rightarrow_{\beta'_x} b\{x := a\}.$$

The crucial problem for explicit substitution is called *preservation of strong normalisation* and is the following.

If a term b is β-strongly normalising, is the term b also $\beta_x\beta'_x$-strongly normalising?.

So β_x-reduction is in fact a refinement of β-reduction whereas δ-reduction is not. One still has to consider the combination of the δ with the β-reduction. The problem analogous to this for definitions is the following: if $b \in \mathcal{T}_\delta$ is β-strongly normalising then b is $\beta\delta$-strongly normalising. This is not true since there may be pseudoterms that are β-strongly normalising but not $\beta\delta$-strongly normalising. For example,

$$\omega = (\lambda x{:}A.\ x\ x){:}B\ in\ (\omega\omega) \twoheadrightarrow_{\delta} (\lambda x{:}A.\ x\ x)(\lambda x{:}A.\ x\ x)$$

The pseudoterm $\omega = (\lambda x{:}A.\ x\ x){:}B\ in\ (\omega\omega)$ is in β-normal form but it is not $\beta\delta$-strongly normalising. Therefore in the case of definitions, we need to restrict the set of terms by using types. The problem for definitions is then the following.

If a pure type system $\lambda(S)$ is β-strongly normalising, is its extension with definitions $\lambda^\delta(S)$ $\beta\delta$-strongly normalising?.

Normalisation for $\beta\delta$. In order to prove that weak and strong normalisation are preserved by the extension we have written two morphisms that are illustrated in the following diagram.

$$\begin{tikzcd} & \lambda^\delta(S) \arrow[dl, "|\,|"'] \arrow[dr, "\{\ \}"] & \\ \lambda(S) & & \lambda(S') \end{tikzcd}$$

The first one is $|\,|$ and computes the δ-normal form of a term. This is an implementing morphism from a pure type system with definitions to the corresponding pure type system without definitions. By the results of chapter 4, this morphism allows us to prove that weak normalisation is preserved by the extension.

The second one is $\{\ \}$ and transforms a δ-redex into a β-redex. This is a refining morphism from $\lambda^\delta(S)$ to $\lambda(S')$ where S' is a completion of S. By the results of chapter 4, this morphism allows us to prove that for a class of pure type systems, strong normalisation is preserved by the extension.

It is still an open problem whether extending an arbitrary pure type system with definitions preserves strong normalisation or not.

Definitions with parameters. In [LSZ96], we consider definitions with parameters like in the systems of the AUTOMATH family [NGdV94]. We write $x(y_1 : B_1, \ldots, y_n : B_n) = a : A$ to denote that x is an abbreviation for a and the variables $y_1, \ldots y_n$ may occur free in a. Then $x(b_1, \ldots b_n)$ reduces to $a[y_1 := b_1, \ldots y_n := b_n]$. We are presently investigating the pure type systems extended with this kind of definitions.

Chapter 12

Type Inference for Definitions

12.1 Introduction

In this chapter we write a partial function that infers the type of a term in a singly sorted pure type system with definitions.

As we said in chapter 10, the type can be easily inferred if the last rule to be applied is determined by the shape of the term and the context. In other words, we have to define a set of typing rules that are syntax directed in order to write a function that infers the type. We present a syntax directed set of rules for singly sorted pure type systems with definitions similar to the one presented in chapter 10.

This chapter is organised as follows. In section 12.2 we modify the δ-start and the δ-weakening rules in the definition of pure type systems with definitions. In section 12.3, we define a syntax directed set of rules for singly sorted pure type systems with definitions. In section 12.4, we define a function that infers the type for a term in a singly sorted pure type system with definitions.

12.2 The δ-start and δ-weakening rules

In this section, we change the rules of pure type systems with definitions. These new rules define the same typing relation as before.

We split the the δ-start rule into two rules, δ-start$_1$ and δ-start$_2$. The δ-start rule is equivalent to these two rules. Similarly, we split the δ-weakening rule into two rules, the δ-weakening$_1$ and δ-weakening$_2$ rules.

We give an intuitive explanation of why the split rules are equivalent to the original one. The δ-start rule has the premise $? \vdash^{\delta} a : A$. We know that $? \vdash^{\delta} A : s$ or $A = s$ by correctness of types. We could add the superfluous condition '$? \vdash^{\delta} A : s$ or $A = s$' to the δ-start rule. This condition is an 'or' and hence we can split the rule in two as follows.

$$(\delta\text{–}start_1) \quad \frac{? \vdash^{\delta'} a : A \quad ? \vdash^{\delta'} A : s}{?\,, x{=}a{:}A \vdash^{\delta'} x : A} \quad x \text{ is } ?\text{-fresh}$$

$$(\delta\text{–}start_2) \quad \frac{? \vdash^{\delta'} a : s \quad s \text{ is a topsort}}{?\,, x{=}a{:}s \vdash^{\delta'} x : s} \quad x \text{ is } ?\text{-fresh}$$

Next we define the notion of a pure type systems with definitions whose δ-start and δ-weakening rules are split.

Definition 12.2.1.
The functor $\lambda^{\delta'}$: **Spec** $\rightarrow$ **Carst**$_{\nu\omega}$ is defined as $\lambda^{\delta'}(S) = (\mathcal{T}_\delta, \mathcal{C}_\delta, \rightarrow_{\beta\delta}, \vdash^{\delta'})$ for $S \in$ **Spec**. The sets $\mathcal{T}_\delta$ and $\mathcal{C}_\delta$ and the relation $\rightarrow_{\beta\delta}$ are as in definition 11.2.18. The typing relation $\vdash^{\delta'}$ is defined as the smallest relation closed under the same rules as in definition 11.2.18 except for the δ-start and δ-weakening rules.

$$(\delta\text{–}start_1) \quad \frac{? \vdash^{\delta'} a : A \quad ? \vdash^{\delta'} A : s}{?\,, x{=}a{:}A \vdash^{\delta'} x : A} \quad x \text{ is } ?\text{-fresh}$$

$$(\delta\text{–}start_2) \quad \frac{? \vdash^{\delta'} a : s \quad s \text{ is a topsort}}{?\,, x{=}a{:}s \vdash^{\delta'} x : s} \quad x \text{ is } ?\text{-fresh}$$

$$(\delta\text{–}weakening_1) \quad \frac{? \vdash^{\delta'} b : B \quad ? \vdash^{\delta'} a : A \quad ? \vdash^{\delta'} A : s}{?\,, x{=}a{:}A \vdash^{\delta'} b : B} \quad x \text{ is } ?\text{-fresh}$$

$$(\delta\text{–}weakening_2) \quad \frac{? \vdash^{\delta'} b : B \quad ? \vdash^{\delta'} a : s \quad s \text{ is a topsort}}{?\,, x{=}a{:}s \vdash^{\delta'} b : B} \quad x \text{ is } ?\text{-fresh}$$

In the following theorem we prove that the typing rules of definitions 11.2.18 and 12.2.1 generate the same typing relation.

Theorem 12.2.2. $? \vdash^{\delta'} a : A$ if and only if $? \vdash^{\delta} a : A$.

From now on, we consider the rules for pure type systems with definitions as presented in this section.

12.3 Syntax Directed Rules for Definitions

In this section we define a syntax directed set of rules for any singly sorted pure type system with definitions. As for pure type systems, the main features are that the weakening rule is restricted to variables and constants and the conversion rule has been removed. Moreover in the abstraction rule the Π-condition is checked in a weaker system. The weaker system is the pure type system with definitions and without the Π-condition.

Now we remove the Π-condition from the rules for pure type system with definitions. In the introduction rule, the premise that the type $(x{=}a{:}A\ in\ B)$ of a definition is well-typed, is replaced by the condition 'B is not a sort'.

Definition 12.3.1.
The functor $\lambda^{\delta\omega} : \mathbf{Spec} \to \mathbf{Carst}_{\nu\omega}$ is defined as $\lambda^{\delta\omega}(S) = (\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash^{\delta\omega})$ for $S \in \mathbf{Spec}$. The sets $\mathcal{T}_\delta$ and $\mathcal{C}_\delta$ and the relation $\to_{\beta\delta}$ are as in definition 11.2.18. The typing relation $\vdash^{\delta\omega}$ is defined as the smallest relation closed under the same rules as in definition 12.2.1 except for the following ones.

$$(abstraction) \qquad \frac{?\,, x : A \vdash^{\delta\omega} b : B}{? \vdash^{\delta\omega} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

$$(\delta\text{–}introduction) \qquad \frac{?\,, x{=}a{:}A \vdash^{\delta\omega} b : B}{? \vdash^{\delta\omega} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)} \quad B \text{ is not a sort}$$

Definition 12.3.2. (Pure type systems with definitions and without the Π-condition) *A pure type system with definitions and without the* Π*-condition* is an element of the set

$$\lambda^{\delta\omega}(\mathbf{Spec}) = \{\lambda^{\delta\omega}(S) \mid S \in \mathbf{Spec}\}.$$

Note that $\lambda^{\delta\omega}(S)$ is an extension of $\lambda^{\delta}(S)$ and $\lambda^{\omega}(S)$. Diagramatically,

$$\begin{array}{ccc} \lambda(S) & \subset & \lambda^{\omega}(S) \\ \cap & & \cap \\ \lambda^{\delta}(S) & \subset & \lambda^{\delta\omega}(S) \end{array}$$

We define the syntax directed set of rules for the pure type systems with definitions and without the Π-condition.

First we define the weak head $\beta\delta$-reduction.

Definition 12.3.3. The weak head $\beta\delta$-reduction is defined by the following rules:

$$?\,, x{=}a{:}A, ?' \vdash x \to^{wh}_{\beta\delta} a \qquad ? \vdash (x{=}a{:}A\ in\ b) \to^{wh}_{\beta\delta} b[x := a]$$

$$? \vdash (\lambda x{:}A.\ b)a \to^{wh}_{\beta\delta} b[x := a] \qquad ? \vdash \frac{F \to^{wh}_{\beta\delta} F'}{(F\ a) \to^{wh}_{\beta\delta} (F'\ a)}$$

Definition 12.3.4.
The functor $\lambda^{\delta\omega}_{sd} : \mathbf{Spec} \to \mathbf{Carst}_{\nu\omega}$ is defined as $\lambda^{\delta\omega}_{sd}(S) = (\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash^{\delta\omega}_{sd})$ for $S \in \mathbf{Spec}$. The sets $\mathcal{T}_\delta$ and $\mathcal{C}_\delta$ and the relation $\to_{\beta\delta}$ are as in definition 11.2.18. The typing relation $\vdash^{\delta\omega}_{sd}$ is defined as the smallest relation closed under the following rules.

$$(axiom) \qquad \epsilon \vdash^{\delta\omega}_{sd} s : s' \qquad \text{for } (s, s') \in \boldsymbol{A}$$

$$(start)\quad \frac{?\ \vdash^{\delta\omega}_{sd} A :\twoheadrightarrow_{\beta\delta} s}{?\,, x : A \vdash^{\delta\omega}_{sd} x : A} \qquad \text{where } x \text{ is } ?\text{-fresh}$$

$$(weakening)\quad \frac{?\ \vdash^{\delta\omega}_{sd} b : B \quad ?\ \vdash^{\delta\omega}_{sd} A :\twoheadrightarrow_{\beta\delta} s}{?\,, x : A \vdash^{\delta\omega}_{sd} b : B} \qquad x \text{ is } ?\text{-fresh and } b \in \mathbf{C} \cup V$$

$$(formation)\quad \frac{?\ \vdash^{\delta\omega}_{sd} A :\twoheadrightarrow_{\beta\delta} s_1 \quad ?\,, x : A \vdash^{\delta\omega}_{sd} B :\twoheadrightarrow_{\beta\delta} s_2}{?\ \vdash^{\delta\omega}_{sd} (\Pi x{:}A.B) : s_3} \qquad \text{for } (s_1, s_2, s_3) \in \boldsymbol{R}$$

$$(abstraction)\quad \frac{?\,, x : A \vdash^{\delta\omega}_{sd} b : B}{?\ \vdash^{\delta\omega}_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

$$(application)\quad \frac{?\ \vdash^{\delta\omega}_{sd} b :\twoheadrightarrow^{wh}_{\beta\delta} (\Pi x{:}A.\ B) \quad ?\ \vdash^{\delta\omega}_{sd} a : A'}{?\ \vdash^{\delta\omega}_{sd} (b\ \ a) : B[x := a]} \qquad A \twoheadleftrightarrow_{\beta\delta} A'$$

$$(\delta{-}start_1)\quad \frac{?\ \vdash^{\delta\omega}_{sd} a : A' \quad ?\ \vdash^{\delta\omega}_{sd} A :\twoheadrightarrow_{\beta\delta} s}{?\,, x{=}a{:}A \vdash^{\delta\omega}_{sd} x : A'} \qquad x \text{ is } ?\text{-fresh and } A \twoheadleftrightarrow_{\beta\delta} A'$$

$$(\delta{-}start_2)\quad \frac{?\ \vdash^{\delta\omega}_{sd} a :\twoheadrightarrow_{\beta\delta} s}{?\,, x{=}a{:}s \vdash^{\delta\omega}_{sd} x : s} \qquad s \text{ is a topsort and } x \text{ is } ?\text{-fresh}$$

$$(\delta{-}weakening_1)\quad \frac{?\ \vdash^{\delta\omega}_{sd} b : B \quad ?\ \vdash^{\delta\omega}_{sd} a : A' \quad ?\ \vdash^{\delta\omega}_{sd} A :\twoheadrightarrow_{\beta\delta} s}{?\,, x{=}a{:}A \vdash^{\delta\omega}_{sd} b : B} \qquad x \text{ is } ?\text{-fresh, } b \in \mathbf{C} \cup V \text{ and } A \twoheadleftrightarrow_{\beta\delta} A'$$

$$(\delta{-}weakening_2)\quad \frac{?\ \vdash^{\delta\omega}_{sd} b : B \quad ?\ \vdash^{\delta\omega}_{sd} a :\twoheadrightarrow_{\beta\delta} s}{?\,, x{=}a{:}s \vdash^{\delta\omega}_{sd} b : B} \qquad s \text{ is a topsort, } b \in \mathbf{C} \cup V \text{ and } x \text{ is } ?\text{-fresh}$$

$$(\delta{-}formation)\quad \frac{?\,, x{=}a{:}A \vdash^{\delta\omega}_{sd} B : s}{?\ \vdash^{\delta\omega}_{sd} (x{=}a{:}A\ in\ B) : s}$$

$$(\delta{-}introduction)\quad \frac{?\,, x{=}a{:}A \vdash^{\delta\omega}_{sd} b : B}{?\ \vdash^{\delta\omega}_{sd} (x{=}a{:}A\ in\ b) : (x{=}a{:}A\ in\ B)} \qquad B \text{ is not a sort}$$

where $s \in \boldsymbol{S}$.

Note that these systems verify only very weak subject reduction (see definition 4.3.1).

Lemma 12.3.5. If $?\ \vdash^{\delta\omega}_{sd} a : A$ then $|?\,| \vdash^{\omega}_{sd} |a|_\Gamma : |A|_\Gamma$.

This is proved by induction on the derivation of $?\ \vdash^{\delta\omega}_{sd} a : A$.

Theorem 12.3.6. (Completeness) If $?\ \vdash^{\delta\omega} a : A$ then $?\ \vdash^{\delta\omega}_{sd} a : A'$ with $A \twoheadleftrightarrow_{\beta\delta} A'$.

This is proved by induction on the derivation of $?\ \vdash^{\delta\omega}_{sd} a : A$.

Next we define a syntax directed set of rules for any singly sorted pure type system with definitions. These rules are exactly the ones in definition 12.3.4 except for the abstraction rule that contains the Π-condition.

Definition 12.3.7.

The functor $\lambda^{\delta}_{sd} : \mathbf{Spec} \to \mathbf{Carst}_{\nu\omega}$ is defined as $\lambda^{\delta}_{sd}(S) = (\mathcal{T}_\delta, \mathcal{C}_\delta, \to_{\beta\delta}, \vdash_{sd})$ for $S \in \mathbf{Spec}$. The sets $\mathcal{T}_\delta$ and $\mathcal{C}_\delta$ and the relation $\to_{\beta\delta}$ are as in definition 11.2.18. The typing

relation $\vdash^{\delta}_{sd}$ is the smallest relation closed under the same rules as in definition 12.3.4 except that the abstraction rule is replaced by the following ones.

$$(abstraction)\quad \frac{?\,, x : A \vdash^{\delta}_{sd} b : B \quad ? \vdash^{\delta\omega}_{sd} (\Pi x{:}A.\ B) :\twoheadrightarrow_{\beta\delta} s}{? \vdash^{\delta}_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

where $s \in \boldsymbol{S}$.

The relations between the systems with a syntax directed set of rules are shown in the following diagram.

$$\begin{array}{ccc} \lambda_{sd}(S) & \subset & \lambda^{\omega}_{sd}(S) \\ \cap & & \cap \\ \lambda^{\delta}_{sd}(S) & \subset & \lambda^{\delta\omega}_{sd}(S) \end{array}$$

Note that these systems verify only very weak subject reduction (see definition 4.3.1).

Lemma 12.3.8. If $? \vdash^{\delta} A :\twoheadrightarrow_{\beta\delta} s$ then $? \vdash^{\delta} A : s$.

Theorem 12.3.9. (Soundness) Let S be a singly sorted specification.
If $? \vdash^{\delta}_{sd} a : A$ then $? \vdash^{\delta} a : A$.

Proof: This is proved by induction on the derivation of $? \vdash^{\delta}_{sd} a : A$. Suppose that the last rule in this derivation is the abstraction rule.

$$(\text{abstraction})\ \frac{?\,, x : A \vdash^{\delta}_{sd} b : B \quad ? \vdash^{\delta\omega}_{sd} (\Pi x{:}A.\ B) :\twoheadrightarrow s}{? \vdash^{\delta}_{sd} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

By induction hypothesis we have that $?\,, x{:}A \vdash^{\delta} b : B$.

We have that $? \vdash^{\delta\omega}_{sd} (\Pi x{:}A.\ B) :\twoheadrightarrow s$. By lemma 12.3.5, $|?\,| \vdash^{\omega}_{sd} |\Pi x{:}A.\ B|_{\Gamma} :\twoheadrightarrow s$. By soundness for $\lambda^{\omega}_{sd}(S)$, we have that $|?\,| \vdash^{\omega} |\Pi x{:}A.\ B|_{\Gamma} :\twoheadrightarrow s$. Hence $|?\,| \vdash^{\omega} |\Pi x{:}A.\ B|_{\Gamma} : s$. By generation lemma, $|?\,, x{:}A| \vdash^{\omega} |B|_{\Gamma} : s_2$, $|?\,| \vdash^{\omega} |A| : s_1$ and $(s_1, s_2, s) \in \boldsymbol{R}$.

By correctness of types we have that either $?\,, x{:}A \vdash^{\delta} B : s'$ or $B = s'$ is a topsort. The second possibility is impossible. Hence $?\,, x{:}A \vdash^{\delta} B : s'$. We have that $s' = s_2$ because the specification is singly sorted.

By correctness of contexts $? \vdash^{\delta} A : s''$. Since the specification is singly sorted we have that $s'' = s_1$.

This is a derivation of $? \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)$.

$$\frac{?\,, x : A \vdash^{\delta} b : B \quad \dfrac{? \vdash^{\delta} A : s_1 \quad ?\,, x{:}A \vdash^{\delta} B : s_2}{? \vdash^{\delta} (\Pi x{:}A.\ B) : s}}{? \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

□

Theorem 12.3.10. (Completeness) Let S be a singly sorted specification.
If $? \vdash^{\delta} a : A$ then there exists A' such that $? \vdash^{\delta}_{sd} a : A'$ and $A \twoheadleftrightarrow_{\beta} A'$.

Proof: We prove only the case of the abstraction rule.

$$(abstraction) \quad \frac{?\,, x : A \vdash^{\delta} b : B \quad ? \vdash^{\delta} (\Pi x{:}A.\ B) : s}{? \vdash^{\delta} (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

By induction hypothesis we have that $?\,, x{:}A \vdash^{\delta}_{sd} b : B'$ for some B' such that $B \twoheadleftrightarrow_{\beta\delta} B'$. By theorem 12.3.9 we have that $?\,, x{:}A \vdash^{\delta} b : B'$.

If $? \vdash^{\delta} (\Pi x{:}A.\ B) : s$ then there exists s_1 and s_2 such that $(s_1, s_2, s) \in \boldsymbol{R}$, $? \vdash^{\delta} A : s_1$ and $?\,, x{:}A \vdash^{\delta} B : s_2$. By correctness and unicity of types we have that $?\,, x{:}A \vdash^{\delta} B' : s_2$. Hence $? \vdash^{\delta} (\Pi x{:}A.\ B') : s$. By theorem 12.3.6 we have that $? \vdash^{\delta\omega}_{sd} (\Pi x{:}A.\ B') : D$ and $D \twoheadleftrightarrow_{\beta\delta} s$. $\square$

12.4 Type Inference for Definitions

In this section we write a semi-algorithm of type inference for all the class of singly sorted pure type systems with definitions.

As for pure type systems, we cannot expect to find a terminating algorithm for the class of singly sorted pure type systems since typability for some non-normalising pure type systems with definitions is not decidable (see corollary 11.4.14). Hence we define a type inference semi-algorithm for all the class of singly sorted pure type systems with definitions, including the non-normalising ones.

In order to define such a semi-algorithm, we use the syntax directed set of rules presented in section 12.3. Although those rules are syntax directed when the specification is singly sorted, they are not yet deterministic. We have to solve the side conditions as in chapter 10. In this case we have to perform $\beta\delta$-weak head reduction and check $\beta\delta$-conversion.

First we write a function that computes the weak head $\beta\delta$-normal form if it exists.

Definition 12.4.1. A function $\mathbf{whnf}_{\beta\delta} : \mathcal{C}_{\delta} \times \mathcal{T}_{\delta} \to \mathcal{T}_{\delta}$ is defined as follows.

$$\begin{aligned}
\mathbf{whnf}_{\beta\delta}(?\,, a) &= a \quad \text{if } a \text{ is in weak head normal form in } ? \\
\mathbf{whnf}_{\beta\delta}(?\,, (x\ d_1\ \ldots d_n)) &= \mathbf{whnf}_{\beta\delta}(?\,, a d_1 \ldots d_n) \quad \text{if } x{=}a{:}A \in ? \\
\mathbf{whnf}_{\beta\delta}(?\,, (x{=}a{:}A\ in\ b)\ d_1\ \ldots d_n) &= \mathbf{whnf}_{\beta\delta}(?\,, b[x := a] d_1 \ldots d_n) \\
\mathbf{whnf}_{\beta\delta}(?\,, (\lambda x{:}A.\ b) a\ d_1\ \ldots d_n) &= \mathbf{whnf}_{\beta\delta}(?\,, b[x := a] d_1 \ldots d_n)
\end{aligned}$$

Lemma 12.4.2. Let $a \in \mathcal{T}$ be weak head normalising. Then $\mathbf{whnf}_{\beta\delta}(a)$ is the weak head $\beta\delta$-normal form of a.

This lemma is proved by induction on the number of steps of the leftmost reduction to the normal form.

In order to check conversion in the semi-algorithm we define a common-reduct strategy for $\beta\delta$-reduction. We define two common-reduct strategies for $\beta\delta$-reduction.

First we define the strategy $\mathbf{F}^{\mathbf{n}}_{\beta\delta}$ similar to $\mathbf{F}^{\mathbf{n}}$.

Definition 12.4.3. We define $\mathbf{F}^{\mathbf{n}}_{\beta\delta} : \mathcal{C}_\delta \times \mathcal{T}_\delta \times \mathcal{T}_\delta \to \mathcal{P}(\mathcal{T}_\delta)$ as follows.

$$
\begin{array}{rcl}
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, d, d) & = & \{d\} \text{ if } d = d' \\[2ex]
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, (\lambda x{:}A.\ b), (\lambda x{:}A'.\ b')) & = & \{(\lambda x{:}A''.\ b'') \mid A'' \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, A, A') \ \ \& \\
 & & \qquad b'' \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, b, b')\} \\[2ex]
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, (\Pi x{:}A.\ B), (\Pi x{:}A'.\ B')) & = & \{(\Pi x{:}A''.\ B'') \mid A'' \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, A, A') \ \ \& \\
 & & \qquad B'' \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, B, B')\} \\[2ex]
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, (x\ a_1\ \ldots a_n), (x\ a'_1\ \ldots a'_n)) & = & \{((x\ a''_1\ \ldots a''_n) \mid a''_i \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, a_i, a'_i)\} \\
 & & \qquad \text{if } x \notin Def(?\,) \\[2ex]
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(?\,, d, d') & = & \mathbf{F}^{\mathbf{n}}_{\beta\delta}(\mathbf{whnf}_{\beta\delta}(d), \mathbf{whnf}_{\beta\delta}(d')) \\
 & & \qquad \text{if } d \text{ or } d' \text{ are not in weak head normal form} \\[2ex]
\mathbf{F}^{\mathbf{n}}_{\beta\delta}(d, d') & = & \emptyset \qquad \text{otherwise}
\end{array}
$$

The next lemma says that the function $\mathbf{F}^{\mathbf{n}}_{\beta\delta}$ is a common-reduct strategy only for normalising pseudoterms.

Lemma 12.4.4. Let a and b be $\beta\delta$-weakly normalising. Then

1. $\mathbf{F}^{\mathbf{n}}_{\beta\delta}(a, b)$ terminates.

2. For all $c \in \mathbf{F}^{\mathbf{n}}_{\beta\delta}(a, b)$, we have that $a \twoheadrightarrow_{\beta\delta} c$ and $b \twoheadrightarrow_{\beta\delta} c$.

3. $a \twoheadleftarrow\!\!\twoheadrightarrow_{\beta\delta} b$ if and only if $\mathbf{F}^{\mathbf{n}}_{\beta\delta}(a, b) \neq \emptyset$.

We define a common-reduct strategy $\mathbf{F}^{++}_{\beta\delta}$ similar to the strategy $\mathbf{F}^{++}$. This strategy terminates if the terms are convertible, even if they are not normalising.

Definition 12.4.5. We define $\mathbf{F}^{++}_{\beta\delta} : \mathcal{C}_\delta \times \mathcal{T}_\delta \times \mathcal{T}_\delta \to \mathcal{P}(\mathcal{T}_\delta)$ as follows.

$$
\begin{array}{lll}
\mathbf{F}^{++}_{\beta\delta}(?\,,a,b) & = & \\
 & & \text{If} \quad a = b \;\; \text{then} \\
 & & | \quad \{a\} \\
 & & \text{else} \\
 & & \quad \text{If} \quad a \text{ or } b \text{ are in weak head } \beta\delta\text{-normal form in } ? \quad \text{then} \\
 & & \quad | \quad \mathbf{L}(?\,,\mathbf{whnf}_{\beta\delta}(?\,,a),\mathbf{whnf}_{\beta\delta}(?\,,b)) \\
 & & | \quad \text{else} \\
 & & \quad | \quad \mathbf{H}^{++}_{\beta\delta}(?\,,\{a\},\{b\}) \\
 & & \quad \text{end} \\
 & & \text{end}
\end{array}
$$

We define the function $\mathbf{L} : \mathcal{C}_\delta \times \mathcal{T}_\delta \times \mathcal{T}_\delta \to \mathcal{P}(\mathcal{T}_\delta)$ as follows.

$$
\begin{array}{rcl}
\mathbf{L}(?\,,(\lambda x{:}A_1.\ b_1),(\lambda x{:}A_2.\ b_2)) & = & \{(\lambda x{:}A_3.\ b_3) \mid A_3 \in \mathbf{F}^{++}_{\beta\delta}(?\,,A_1,A_2)\ \ \& \\
 & & \qquad b_3 \in \mathbf{F}^{++}_{\beta\delta}(?\,,b_1,b_2)\} \\
\mathbf{L}(?\,,(\Pi x{:}A_1.\ b_1),(\Pi x{:}A_2.\ b_2)) & = & \{(\Pi x{:}A_3.\ b_3) \mid A_3 \in \mathbf{F}^{++}_{\beta\delta}(?\,,A_1,A_2)\ \& \\
 & & \qquad b_3 \in \mathbf{F}^{++}_{\beta\delta}(?\,,b_1,b_2)\} \\
\mathbf{L}(?\,,(x\ a_1 \ldots a_n),(x\ b_1 \ldots b_n)) & = & \{((x\ c_1 \ldots c_n) \mid\ \forall\ i = 1,n \quad c_i \in \mathbf{F}^{++}_{\beta\delta}(?\,,a_i,b_i)\} \\
\mathbf{L}(?\,,a,b) & = & \emptyset \qquad \text{otherwise}
\end{array}
$$

The function $\mathbf{H}^{++}_{\beta\delta} : \mathcal{C}_\delta \times \mathcal{P}(\mathcal{T}_\delta) \times \mathcal{P}(\mathcal{T}_\delta) \to \mathcal{P}(\mathcal{T}_\delta)$ is always applied to a context $?$ and subsets X and Y of $\mathcal{T}_\delta$ that verify the following preconditions.

1. X and Y are the n and m-bounded reduction graphs of a and b in $?$.

$$X = \mathcal{G}^{\leq n}_{\to_{\beta\delta}}(?\,,a) = \{d \mid ? \vdash a \twoheadrightarrow_{\beta\delta} d \text{ in less than } n \text{ steps}\}$$

$$Y = \mathcal{G}^{\leq m}_{\to_{\beta\delta}}(?\,,b) = \{d \mid ? \vdash b \twoheadrightarrow_{\beta\delta} d \text{ in less than } m \text{ steps}\}$$

2. The intersection of X and Y is empty.

3. $0 \leq n - m \leq 1$.

4. X and Y do not contain any weak head $\beta\delta$-normal form in $?$.

If the bounded graphs X contains a weak head normal form, we choose a b of Y and reduce it to weak head normal form.

$$
\begin{array}{lll}
\mathbf{H}^{++}_{\beta\delta}(?\,,X,Y) & = & \\
& & \text{If} \quad \mathcal{G}_{\rightarrow_{\beta\delta}}(?\,,X) \cap Y \neq \emptyset \ \text{ then} \\
& & | \quad \mathcal{G}_{\rightarrow_{\beta\delta}}(?\,,X) \cap Y \\
& & \text{else} \\
& & \qquad \text{If} \quad \exists a,\, a \in \mathcal{G}_{\rightarrow_{\beta\delta}}(?\,,X) \text{ in weak head } \beta\delta\text{-normal form then} \\
& & \qquad | \quad \text{Choose } b \in Y,\, \mathbf{L}(?\,,a,\mathbf{whnf}_{\beta\delta}(?\,,b)) \\
& & | \quad \text{else} \\
& & \qquad | \quad \mathbf{H}^{++}_{\beta\delta}(?\,,Y,X \cup \mathcal{G}_{\rightarrow_{\beta\delta}}(?\,,X)) \\
& & \qquad \text{end} \\
& & \text{end}
\end{array}
$$

In the following lemma we prove that the function $\mathbf{F}^{++}_{\beta\delta}$ is a common-reduct strategy.

Lemma 12.4.6.

1. If $? \vdash a \twoheadleftrightarrow_{\beta\delta} b$ then $\mathbf{F}^{++}_{\beta\delta}(?\,,a,b)$ terminates and yields a non-empty set that verifies that for all $c \in \mathbf{F}^{++}_{\beta\delta}(?\,,a,b)$ we have that $a \twoheadrightarrow_{\beta\delta} c$ and $b \twoheadrightarrow_{\beta\delta} c$.

2. If $\mathbf{F}^{++}_{\beta\delta}(?\,,a,b) \neq \emptyset$ then $? \vdash a \twoheadleftrightarrow_{\beta\delta} b$.

Semi-algorithm of type inference. We define a function $\mathbf{type}^{\delta}$ that computes the type of a term (up to $\beta\delta$-conversion) in a singly sorted pure type system with definitions. If a is typable in $?$ in a singly sorted pure type system with definitions then $\mathbf{type}^{\delta}(?\,,a)$ terminates and yields the type of a in $?$ (up to $\beta\delta$-conversion), i.e. if $? \vdash^{\delta} a : A$ then $\mathbf{type}^{\delta}(?\,,a) \twoheadleftrightarrow_{\beta\delta} A$. If the term a is not typable in $?$ then $\mathbf{type}^{\delta}(?\,,a)$ either yields $-$ or it does not terminate.

This function is obtained from the syntax directed set of rules defined in section 12.3 for pure type systems with definitions. For each rule, we write a case of 'pattern matching'.

The conditions that appear in these rules that are of the form '$? \vdash^{\delta} A :\twoheadrightarrow_{\beta\delta} s$' are replaced by '$\mathbf{whnf}_{\beta\delta}(\mathbf{type}^{\delta}(?\,,A)) = s$'.

The condition '$? \vdash^{\delta} b :\twoheadrightarrow_{\beta\delta} (\Pi x{:}A.\ B)$' is replaced by

$$\text{`}\mathbf{whnf}_{\beta\delta}(\mathbf{type}^{\delta}(?\,,b)) = (\Pi x{:}A.\ B)\text{'}.$$

The other condition '$A \twoheadleftrightarrow_{\beta\delta} A'$' is replaced by $\mathbf{F}^{++}_{\beta\delta}(A,A') \neq \emptyset$.

The condition '$? \vdash^{\delta\omega} (\Pi x{:}A.\ B) :\twoheadrightarrow_{\beta\delta} s$' is replaced by

$$\text{`}\mathbf{whnf}_{\beta\delta}(\mathbf{type}^{\delta\omega}(?\,,(\Pi x{:}A.\ B))) = s\text{'}.$$

We need to define an auxiliary function $\mathbf{type}^{\delta\omega}$ to compute the type in a pure type system without the Π-condition.

The function $\mathbf{type}^{\delta\omega} : \mathcal{C}_{\delta} \times \mathcal{T}_{\delta} \rightarrow \mathcal{T}_{\delta\_}$ is defined as $\mathbf{type}^{\delta}$ except that we remove the condition $\mathbf{whnf}_{\beta\delta}(?\,,\mathbf{type}^{\delta\omega}(?\,,\Pi x{:}A.\ B)) = s \in \boldsymbol{S}$ that corresponds to the Π-condition.

Definition 12.4.7. The function $\mathbf{type}^{\delta}_{S} : \mathcal{C}_{\delta} \times \mathcal{T}_{\delta} \rightarrow \mathcal{T}_{\delta\_}$ (or just $\mathbf{type}^{\delta}$) is defined as follows.

$\mathbf{type}^{\delta}(\epsilon, s)$	$=$	s'	if $(s, s') \in \boldsymbol{A}$
$\mathbf{type}^{\delta}(< ?, x : A >, x)$	$=$	A	if $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, A)) = s \in \boldsymbol{S}$ and x is $?$-fresh
$\mathbf{type}^{\delta}(< ?, x : A >, b)$	$=$	$\mathbf{type}^{\delta}(?, b)$	if $b \in \mathbf{C} \cup V$, x is $?$-fresh, $b \neq x$ and $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, A)) = s \in \boldsymbol{S}$
$\mathbf{type}^{\delta}(?, (\Pi x{:}A.B))$	$=$	s_3	if $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, A)) = s_1$, $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(< ?, x : A >, B)) = s_2$, and $(s_1, s_2, s_3) \in \boldsymbol{R}$
$\mathbf{type}^{\delta}(?, (\lambda x{:}A.\ b))$	$=$	$(\Pi x{:}A.\ B)$	if $\mathbf{type}^{\delta}(< ?, x : A >, b) = B$ and $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta\omega}(?, \Pi x{:}A.\ B)) = s \in \boldsymbol{S}$
$\mathbf{type}^{\delta}(?, (b\ \ a))$	$=$	$B[x := a]$	if $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, b)) = (\Pi x{:}A.\ B)$, $\mathbf{type}^{\delta}(?, a) = A'$ and $\mathbf{F}^{++}_{\beta\delta}(?, A, A') \neq \emptyset$
$\mathbf{type}^{\delta}(< ?, x{=}a{:}A >, x)$	$=$	A'	if x is $?$-fresh and $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, A)) = s$, $\mathbf{type}^{\delta}(?, a) = A'$ and $\mathbf{F}^{++}_{\beta\delta}(?, A, A') \neq \emptyset$
$\mathbf{type}^{\delta}(< ?, x{=}a{:}s >, x)$	$=$	s	if s is a topsort, x is $?$-fresh and $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, a)) = s$
$\mathbf{type}^{\delta}(< ?, x{=}a{:}A >, b)$	$=$	B	if $b \in \mathbf{C} \cup V$, x is $?$-fresh, $b \neq x$ $\mathbf{type}^{\delta}(?, b) = B$, $\mathbf{type}^{\delta}(?, a) = A'$, $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, A)) = s$ and $\mathbf{F}^{++}_{\beta\delta}(?, A, A') \neq \emptyset$
$\mathbf{type}^{\delta}(< ?, x{=}a{:}s >, b)$	$=$	B	if $b \in \mathbf{C} \cup V$, s is a topsort, x is $?$-fresh, $\mathbf{type}^{\delta}(?, b) = B$ and $\mathbf{whnf}_{\beta\delta}(?, \mathbf{type}^{\delta}(?, a)) = s$
$\mathbf{type}^{\delta}(?, (x{=}a{:}A\ in\ B))$	$=$	s	if $\mathbf{type}^{\delta}(< ?, x{=}a{:}A >, B) = s$
$\mathbf{type}^{\delta}(?, (x{=}a{:}A\ in\ b))$	$=$	$(x{=}a{:}A\ in\ B)$	if $\mathbf{type}^{\delta}(< ?, x{=}a{:}A >, b) = B$ and B is not a sort
$\mathbf{type}^{\delta}(?, a)$	$=$	$-$	otherwise

Theorem 12.4.8. (Correctness of 'type$^\delta$') Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursively enumerable.

1. If $? \vdash^\delta a : A$ then $\mathbf{type}^\delta(?\,, a)$ terminates and $\mathbf{type}^\delta(?\,, a) \twoheadleftrightarrow_{\beta\delta} A$.

2. If $\mathbf{type}^\delta(?\,, a)$ terminates and yields A then $? \vdash^\delta a : A$.

Like for pure type systems without definitions, type checking for pure type systems with definitions can be solved from type inference (see definition 10.8.15). A function $\mathbf{check}^\delta$ is defined that checks if a term has a given type in a singly sorted pure type system with definitions.

Definition 12.4.9. We define the function $\mathbf{check}^\delta : \mathcal{C}_\delta \times \mathcal{T}_\delta \times \mathcal{T}_\delta \to \mathit{Bool}$ as follows.

$$\mathbf{check}^\delta(?\,, a, A) \quad = \quad \begin{cases} \mathit{true} & \text{if } \mathbf{whnf}_{\beta\delta}(\mathbf{type}^\delta(?\,, A)) = s \in \boldsymbol{S}, \\ & \mathbf{type}^\delta(?\,, a) \neq - \text{ and} \\ & \mathbf{F}^{++}_{\beta\delta}(\mathbf{type}^\delta(?\,, a), A) \neq \emptyset \\ \mathit{false} & \text{otherwise} \end{cases}$$

Theorem 12.4.10. (Decidability of Type Inference and Type Checking)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursive.
If $\lambda^\delta(S)$ is $\beta\delta$-weakly normalising then type inference and type checking in $\lambda^\delta(S)$ are decidable.

Proof: Since the sets of the specification are recursive, we have that $\mathbf{type}^{\delta\omega}$ always terminates and so do $\mathbf{type}^\delta$ and $\mathbf{check}^\delta$. Therefore type inference and type checking for $\lambda^\delta(S)$ are decidable. □

Theorem 12.4.11. (Decidability of Inhabitation)
Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursive. Suppose that $\lambda(S)$ is β-weakly normalising.
If the problem of inhabitation is decidable in $\lambda(S)$ then it is decidable in $\lambda^\delta(S)$.

Proof: If $\lambda(S)$ is β-normalising then $\lambda^\delta(S)$ is $\beta\delta$-normalising. Let $? \in \mathcal{C}_\delta$ and $A \in \mathcal{T}_\delta$. By theorem 12.4.10, we have that it is decidable whether $? \vdash^\delta A : s$ or not. In case $? \vdash^\delta A : s$, we find a such that $|?\,| \vdash a : |A|_\Gamma$. By applying the conversion rule and lemma 11.4.18, we obtain $? \vdash^\delta a : A$. If $|A|_\Gamma$ has no inhabitant then A cannot have any inhabitant. □

As a consequence of this, we have that the problems of inhabitation in $\lambda^\delta_\to$ and in $\lambda^\delta\underline{\omega}$ are decidable.

Π-condition	Original type system	Type system with syntax directed rules	Type inference semi-algorithm
included	$\lambda^{\delta}(S)$	$\lambda^{\delta}_{sd}(S)$	$\mathbf{type}^{\delta}$
removed	$\lambda^{\delta\omega}(S)$	$\lambda^{\delta\omega}_{sd}(S)$	$\mathbf{type}^{\delta\omega}$

Table 12.1: Type Inference Semi-algorithm

12.5 Conclusions and Related Work

Type Inference Semi-algorithm. In order to solve the type inference problem for pure type systems with definitions we have first considered a syntax directed set of rules and then we have written a function that infers the type based on this syntax directed set of rules. Table 12.1 illustrates our methods.

In the first column of the table we indicate if the Π-condition is included or removed, in the second one the original type systems with definitions (with and without the Π-condition), in the third one, the corresponding type systems with a syntax directed set of rules and finally the functions that infer the type in the original systems.

The definitions of $\mathbf{type}^{\delta}$ and $\mathbf{type}^{\delta\omega}$ are based on the system that appear next to them in their preceding columns.

The Π-condition of $\lambda^{\delta}(S)$ is checked in the same system, whereas the one of $\lambda^{\delta}_{sd}(S)$ is checked in $\lambda^{\delta\omega}_{sd}(S)$. Therefore in the definition of $\mathbf{type}^{\delta}$, the condition that corresponds to the Π-condition is checked using the function $\mathbf{type}^{\delta\omega}$ and not $\mathbf{type}^{\delta}$. In other words, we have to use $\mathbf{type}^{\delta\omega}$ to define $\mathbf{type}^{\delta}$.

The Π-condition of $\lambda^{\delta}_{sd}(S)$ is checked in $\lambda^{\delta\omega}_{sd}(S)$ that is weaker than $\lambda^{\delta}_{sd}(S)$.

We have proved that if S is a singly sorted specification then $\lambda^{\delta}_{sd}(S)$ is 'equivalent' to and $\lambda^{\delta}(S)$ by using the relation shown in the diagram (see theorems 12.3.9 and 12.3.10).

$$\begin{array}{ccccc}
\lambda^{\delta}(S) & \subset & \lambda^{\delta\omega}(S) & \subset & \lambda^{\delta\omega}_{sd}(S) \\
\Big\| & & & & \Big\| \\
\lambda(S) & \subset & \lambda^{\omega}(S) & \cong & \lambda^{\omega}_{sd}(S)
\end{array}$$

Decidability. Table 12.2 summarises some results concerning decidability discussed in this chapter. In the first column, we write the conditions that a pure type system with definitions should verify. In the second and third columns, we say whether type inference and type checking are decidable or undecidable.

We deduce from the table that the problems of type inference and type checking in the systems of the λ^{δ}-cube (the λ-cube extended with definitions) are decidable.

For the systems of the cube, the problem of inhabitation is decidable in $\lambda(S)$ if and only if it is decidable in $\lambda^{\delta}(S)$. The problems of inhabitation in $\lambda^{\delta}_{\rightarrow}$ and in $\lambda^{\delta}\underline{\omega}$ are decidable and in the rest of the systems of the cube extended with definitions it is undecidable.

Pure type systems with definitions	**Type inference**	**Type checking**
$S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ singly sorted $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ recursive $\lambda(S)$ is β-normalising	decidable	decidable
S is singly sorted, impredicative and non-dependent $\lambda^{\delta}(S)$ is inconsistent	undecidable	undecidable

Table 12.2: Decidability of type inference and type checking

In our opinion, the algorithm for normalising pure type systems whose set of sorts is finite defined in [BJ93] and the syntax directed sets of rules defined in [BJMP93] and in [Pol93a] can be adapted to include definitions by using the function $|\ |$ to prove the equivalence between the syntax directed set of rules and the original system.

Chapter 13

Conclusions

In this chapter we discuss the results of this thesis. We mark what we believe are the main contributions to the field, give a global overview and show the interconnections of the subjects.

The discussion is divided in the same parts as the thesis: 1) an abstract presentation of rewriting and typing, 2) lambda calculus and 3) pure type systems with definitions.

13.1 Abstract Presentation of Rewriting and Typing

In this part, the concepts of computation and typing have been formalised in an abstract way, as binary relations on a set.

The chapters 2 and 4 on abstract rewriting and typing systems give uniformity and clarity to the exposition of these subjects.

The main concepts are defined only once in this abstract setting. It is clear that this avoids repetitions and provides us with a common definition that covers all the particular cases. Moreover it enables us to state properties in a very general way. All these definitions and properties are used in the chapters that follow.

13.2 Lambda Calculus

We have given a characterisation of the set of strongly normalising λ-terms that permits us to give new and simple proofs of classical results about λ-calculus.

All the proofs of this part follow a common line since they all use the definition of the set $\mathcal{SN}$. Even though we did not prove any new result, the methods for proving them are new.

In most cases the new proofs are essentially simpler than already existing ones and they help us to understand not only *the mechanics* of the proofs of the results but also *the reasons* for their validity.

13.3 Pure Type Systems with Definitions

In this part we have studied the meta-theory of pure type systems with definitions. Also we have written semi-algorithms of type inference for pure type systems with (and without) definitions.

Pure Type Systems. We have presented the definition of pure type systems in a slightly different way (see chapter 9) from the usual one. The typing rules for pure type systems are parametric in the specifications. Pure type systems are written as a functor λ that, given a specification S, produces a particular pure type system $\lambda(S)$. The codomain of this functor is the category of environmental abstract rewriting systems with typing defined in chapter 4.

Definitions. We have considered definitions as part of the formal language, the language of pure type systems. In our opinion, this extension has been done in a neat and general way (for pure type systems). The inclusion of definitions in the formal language and its study have not been considered before except for the systems of the AUTOMATH family [NGdV94], that can be seen as particular pure type systems.

13.3.1 Normalisation

The two extension of pure type systems, without the Π-condition and with definitions have similar properties that are compared in table 13.1. WN is an abbreviation for weak normalisation and SN is an abbreviation for strong normalisation. In the first column, we write the properties of a pure type system without the Π-condition and in the second one, the properties of a pure type system with definitions.

We have written two functions φ and $|\ |$, one is a weak converting morphism from $\lambda^{\omega}(S)$ to $\lambda(S)$ and the other is an implementing morphism from $\lambda^{\delta}(S)$ to $\lambda(S)$. Diagramatically,

$$\begin{tikzcd}
\lambda^{\omega}(S) \arrow[dr, "\varphi"] & \\
& \lambda(S) \\
\lambda^{\delta}(S) \arrow[ur, "|\ |"'] &
\end{tikzcd}$$

The function φ computes the illegal β-normal form of a term and $|\ |$ computes the δ-normal form of a term.

In the case of pure type systems without the Π-condition, an illegal β-rewrite sequence is a superdevelopment. Since all superdevelopments are finite, we have that the illegal β-reduction is strongly normalising. In the case of pure type systems with definitions

$\lambda^\omega(S)$	$\lambda^\delta(S)$
φ computes the normal form of the illegal β-reduction	$\|\ \|$ computes the normal form of the δ-reduction
φ is a weak converting morphism (S is singly sorted)	$\|\ \|$ is an implementing morphism
An illegal β-rewrite sequence is a superdevelopment	A development is a δ-rewrite sequence
The illegal β-reduction is SN	The δ-reduction is SN
WN($\lambda(S)$) implies WN($\lambda^\omega(S)$) (S is singly sorted)	WN($\lambda(S)$) implies WN ($\lambda^\delta(S)$)
SN($\lambda(S')$) implies SN($\lambda^\omega(S)$) (S' is a completion of S)	SN($\lambda(S')$) implies SN($\lambda^\delta(S)$) (S' is a completion of S)

Table 13.1: $\lambda^\omega(S)$ vs. $\lambda^\delta(S)$

a δ-rewrite sequence is a development. A definition $x = a : A\ in\ b$ is like a marked redex $(\underline{\lambda} x{:}A.\ b)a$. However a δ-rewrite sequence is not a particular case of a development and we cannot deduce strong normalisation of the δ-reduction from finiteness of developments. Instead we deduce finiteness of developments from the strong normalisation of the δ-reduction.

In both extensions we have that weak normalisation is preserved by the extension. For pure type systems without the Π-condition, we used φ to prove that weak normalisation is preserved by the extension and for definitions, we used the mapping $|\ |$.

In both cases we have that strong normalisation of the extension follows from the strong normalisation of another (larger) pure type system. In order to prove that strong normalisation is preserved by the extension with definitions, we have written a refinining morphism $\{\ \}$ from $\lambda^\delta(S)$ to $\lambda(S')$, S' being a completion of S. The identity is a morphism from $\lambda^\omega(S)$ to $\lambda(S')$ and it can be used to prove that if $\lambda(S')$ is strongly normalising then so is $\lambda^\omega(S)$ (this proof has not been included in this thesis). In the following diagram, we show how these morphisms are related.

$$\begin{tikzcd}
\lambda^{\omega}(S) \arrow[dr, "id"] & & \\
& \lambda(S') \supset \lambda(S) & \\
\lambda^{\delta}(S) \arrow[uu, "\{\ \}"] \arrow[ur] & &
\end{tikzcd}$$

Strong normalisation is the most important property we have proved for definitions. The $\beta\delta$-strong normalisation of $\lambda(S)$ follows from the β-strong normalisation of $\lambda(S')$ with S' a completion of S. This enables us to prove that for all pure type systems that are known to be β-strongly normalising, their extensions with definitions are also $\beta\delta$-strongly normalising.

Systems that are $\beta\delta$-strongly normalising are, for example, the calculus of constructions extended with definitions and the system of higher order logic extended with definitions.

The question 'Given an arbitrary pure type system, is its extension with definitions strongly normalising?' remains open.

13.3.2 Type Inference

In order to solve the type inference problem for pure type systems with (and without) definitions we have first considered a syntax directed set of rules and then we have written a function that infers the type.

Syntax directed sets of rules. The definitions of the semi-algorithms of type inference are based upon the definitions of a syntax directed set of rules.

Table 13.2 illustrates the type systems we considered.

	Original System	**Syntax-directed System**
PTS	$\lambda(S)$	$\lambda_{sd}(S)$
PTS without Π-condition	$\lambda^{\omega}(S)$	$\lambda^{\omega}_{sd}(S)$
PTS with definitions	$\lambda^{\delta}(S)$	$\lambda^{\delta}_{sd}(S)$
PTS with defs. without Π-cond.	$\lambda^{\delta\omega}(S)$	$\lambda^{\delta\omega}_{sd}(S)$

Table 13.2: Type Systems Considered

In the first column of table 13.2 we find the original systems whose rules are not syntax directed. In the second column, we find the equivalent systems whose rules are syntax directed. Two systems that appear on the same line are equivalent.

In the second and fourth lines, the systems do not contain the Π-condition. The systems of the first and third line do contain the Π-condition. The Π-condition of the systems $\lambda(S)$ and $\lambda^{\delta}(S)$ is checked inside the system, whereas the Π-condition of $\lambda_{sd}(S)$ and $\lambda^{\delta}_{sd}(S)$ is

checked outside them. The auxiliary systems used to check the Π-condition of $\lambda_{sd}(S)$ and $\lambda^{\delta}_{sd}(S)$ appear on the line below in the table 13.2.

The following diagram shows some relations between these type systems. We suppose that S is a singly sorted specification.

$$\begin{tikzcd}
\lambda^{\delta}_{sd}(S) \arrow[d, "|\,|"'] & \cong & \lambda^{\delta}(S) \arrow[r, phantom, "\subset"] & \lambda^{\delta\omega}(S) \arrow[r, phantom, "\subset"] & \lambda^{\delta\omega}_{sd}(S) \arrow[d, "|\,|"] \\
\lambda_{sd}(S) & \cong & \lambda(S) \arrow[r, phantom, "\subset"] \arrow[rr, bend left, "\varphi"] & \lambda^{\omega}(S) & \lambda^{\omega}_{sd}(S)
\end{tikzcd}$$

In order to prove the equivalences $\lambda^{\delta}(S) \cong \lambda^{\delta}_{sd}(S)$ and $\lambda(S) \cong \lambda_{sd}(S)$, we have used the other relations that are depicted in the diagram (see theorems 10.7.7, 10.7.8, 12.3.9 and 12.3.10).

Semi-algorithms. In chapter 10.2 we have defined the semi-algorithm '**type**' of type inference for singly sorted pure type systems and in chapter 12 the other semi-algorithm '$\mathbf{type}^{\delta}$' of type inference for singly sorted pure type systems with definitions. These semi-algorithms terminate if the term is typable and otherwise they may not terminate.

The semi-algorithm **type** is based on the system $\lambda_{sd}(S)$ and $\mathbf{type}^{\delta}$ is based on $\lambda^{\delta}_{sd}(S)$. In order to define both semi-algorithms we have followed the same method. The method consists of considering the corresponding type system without the Π-condition to check for the Π-condition.

13.3.3 Normalisation versus Type Inference

The normalisation property and the type inference problem are related in two places: in the proof of the correctness of the semi-algorithms of type inference and in the results on decidability of type inference and type checking for pure type systems with and without definitions.

The proof of correctness. We have proved the correctness of the type inference semi-algorithms for singly sorted pure type systems with and without definitions. The correctness of the semi-algorithms is proved using the equivalence between the syntax directed sets of rules and the original systems.

The proof of the equivalence between the syntax directed sets of rules and the original ones, use the relations shown in the diagram above. The morphisms that appear in this diagram are $|\ |$ and φ which compute the δ-normal form and the illegal β-normal form, respectively.

Decidability of Type Inference, Type Checking Let $S = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be singly sorted such that the sets $\boldsymbol{S}$, $\boldsymbol{A}$ and $\boldsymbol{R}$ are recursive. If $\lambda(S)$ is β-weakly normalising then the following statements on decidability hold.

- Type inference and type checking in $\lambda(S)$ are decidable (see theorems 9.3.19 and 10.8.16).
- Type inference and type checking in $\lambda^{\delta}(S)$ are decidable (see theorem 12.4.10).

Undecidability of Type Inference, Type Checking and Inhabitation. Let S be a singly sorted, impredicative and non-dependent specification. If $\lambda(S)$ is inconsistent then the following statements hold.

- Type inference and type checking in $\lambda(S)$ are undecidable (see theorem 9.3.20).
- Type inference and type checking in $\lambda^{\delta}(S)$ are undecidable (see theorem 11.4.14).

Hence, for the systems of the λ-cube extended with definitions, we have that type inference and type checking are decidable.

Bibliography

[ACCL91] M. Abadi, L. Cardelli, P. L. Curien, and J.J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks*, pages 39–42, 1990.

[Acz95] P. Aczel. Schematic consequence. Preprint, 1995.

[Avr92] A. Avron. Axiomatic Systems, Deduction and Implication. *Journal of Logic Computation*, 2(1):51–98, 1992.

[Bar74] K. J. Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic*, 7:221–265, 1974.

[Bar85] H. Barendregt. *The Lambda Calculus. Its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North Holland. Amsterdam, 1985.

[Bar92] H. Barendregt. Lambda Calculi with Types. In D. M. Gabbai, S. Abramsky, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 117–309. Oxford University Press, 1992.

[BBKV76] H. Barendregt, J. Bergstra, J. W. Klop, and H. Volken. Degrees, reductions and representability in the lambda calculus. Technical Report 22, University of Utrecht, 1976.

[Ber88] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and Other Systems in Barendregt's Cube. Technical report, Carnegie Mellon University and Universitá di Torino, 1988.

[Ber90] S. Berardi. *Type Dependency and Constructive Mathematics*. PhD thesis, Carnegie Mellon University and Universitá di Torino, 1990.

[BG96] R. Bloo and H. Geuvers. Explicit substitution: On the edge of strong normalisation. Technical Report CS-9610, Eindhoven University of Technology, 1996.

[BJ93] B. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.

[BJMP93] B. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In H. Barendregt and T. Nipkov, editors, *Types for Proofs and Programs*, volume 806, 1993.

[BK82] J. A. Bergstra and J. W. Klop. Strong normalization and perpetual reductions in the lambda calculus. *Journal of Information Processing and Cybernetics*, 18:403–417, 1982.

[BKKS87] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[Blo97] R. Bloo. *In preparation.* PhD thesis, Eindhoven University of Technology, 1997.

[Bru70] N. G. de Bruijn. The mathematical language of AUTOMATH, its usage and some of its extensions. In *Symposium on automatic demonstration*, volume 125 of *LNCS*, pages 27–61. Springer, 1970.

[Bru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, pages 381–392, 1972.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* North Holland. Amsterdam, 1958.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CH94] T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4(1):77–88, 1994.

[Chu40] A. Church. A formulation of the simply theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press. Princeton., 1941.

[Con86] R. L. Constable et al. *Implementing mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Coq85] T. Coquand. *Une Théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[Coq91] T. Coquand. An algorithm for testing conversion in type theory. In *Logical Frameworks*, 1991.

[Coq96] T. Coquand. An Algorithm for Type-Checking Dependent Types. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory*. Göteborg University and Chalmers University of Technology, 1996.

[CP90] T. Coquand and C. Paulin. Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the Americal Society*, 39:472–482, 1936.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proc. nat. Acad. Science USA*, 20:584–590, 1934.

[Cur69] H. B. Curry. Modified based functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[Daa80] D. T. van Daalen. *The Language Theory of Automath*. PhD thesis, Eindhoven University of Technology, 1980.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.

[Dow91] G. Dowek et al. The Coq proof assistant version 5.6, users guide. Rapport de Recherche 134, INRIA, 1991.

[FS93] W. Ferrer and P. Severi. Abstract reduction and topology. Technical Report CS-9335, Eindhoven University of Technology, 1993.

[Gal90] J. H. Gallier. On Girard's 'candidats de reductibilité'. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC*, pages 123–203. Academic Press, London, New York, 1990.

[Gan80] R.O. Gandy. An Early Proof of Normalisation by A. M. Turing. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 453–455. Academic Press, 1980.

[Geu93] H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.

[Ghi94] S. Ghilezan. Applications of typed lambda calculi in the untyped lambda calculus. In A. Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science: Proceedings of the Third International Symposium*, volume 813, pages 129–139. Springer-Verlag, 1994.

[Gir72] J. Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GLM92] G. Gonthier, J. Lévy, and P. A. Melliès. An abstract standardisation theorem. In *Seventh Annual IEEE Symposium on Logic In Computer Science*, pages 72–81. IEEE, 1992.

[GLSH92] E. R. Griffor, I. Lindstroem, and V. Stoltenberg-Hansen. *Mathematical Theory of Domains.* Cambridge University Press, 1992.

[GN91] H. Geuvers and M. J. Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[HA91] L. Helmink and R. M. C. Ahn. Goal directed proof construction in type theory. In *Procs. of the first Workshop on Logical Frameworks.* Cambridge University Press, 1991.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, LFCS-University of Edinburgh, March 1986.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HST89] R. Harper, D. Sanella, and A. Tarlecki. Structure and Representation in LF. In *IEEE Fourth Annual Symposium On Logic in Computer Science*, pages 226–237, 1989.

[HW88] P. Hudak and P.L. Wadler. Report of the functional programming language Haskell. Technical Report YALEU/DCS/RR656, Yale University, 1988.

[Hyl73] J. M. E. Hyland. A simple proof of the Church-Rosser theorem. Technical report, Oxford University, 1973.

[Kel55] J. Kelley. *General Topology.* Van Nordstram Company, Inc., 1955.

[KKS95] R. Kennaway, J. W. Klop, and R. Sleep. Infinitary lambda calculus. Technical report, CWI, Amsterdam, 1995.

[KKSdV91] R. Kennaway, J. W. Klop, R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. Technical report, Centre for Mathematics and Computer Science Amsterdam, 1991.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, Rijksuniversiteit Utrecht, 1980.

[Klo90] J. W. Klop. Term rewriting systems. Technical Report CS-R9073, Centre for Mathematics and Computer Science Amsterdam, 1990.

[KN93] F. Kamareddine and R. Nederpelt. On stepwise explicit substitution. *Internation Journal of Foundation of Computer Science*, 4(3):197–240, 1993.

[KN95] F. Kamareddine and R. Nederpelt. Refining reduction in the lambda calculus. *J. Functional Programming*, 5(4):637–651, 1995.

[Kri93] J. L. Krivine. *Lambda-Calculus, Types and Models*. Masson, France, 1993.

[Lev78] J. J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris VII, 1978.

[Loa95] R. Loader. Normalisation by translation. Note distributed on the 'types' mailing list, 1995.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS-University of Edinburgh, 1992.

[LSZ96] T. Laan, P. Severi, and J. Zwanenburg. Pure type systems with parameters. Technical Report To appear, Eindhoven University of Technology, 1996.

[Luo89] Z. Luo. ECC, the Extended Calculus of Constructions. In *The fourth Annual Symposium on Logic in Computer Science*, pages 386–395. IEEE, 1989.

[Luo90] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[Mag94] L. Magnusson. *The Implementation of Alf-a proof editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology, 1994.

[Mel96] P. A. Melliès. *Description abstraite de Systémes de Réécriture*. PhD thesis, Université de Paris VII, 1996.

[Mes89] J. Meseguer. Relating models of polymorphism. In *Principles of Programming Languages*, pages 228–241. ACM, 1989.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit86] J. C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, pages 308–319. ACM, 1986.

[MR86] A. R. Meyer and M. B. Reinhold. 'type' is not a type: Preliminary report. In *Principles of Programming Languages*, pages 287–295. ACM, 1986.

[Ned73] R. P. Nederpelt. *Strong Normalisation in a Typed Lambda-Calculus with Lambda-Structured Typed.* PhD thesis, Eindhoven University of Technology, 1973.

[Ned92] R. P. Nederpelt. The fine-structure of lambda calculus. Computer science notes CS-9207, TU, 1992.

[New42] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath. Studies in Logic and the Foundation of Mathematics*, volume 133. North Holland Publ. Company, 1994.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An introduction.* Oxford Science Publications, 1990.

[Oos94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, 1994.

[Par90] M. Parigot. Internal labelling in lambda calculus. In *MFCS'90*, number 452 in LNCS, pages 439–445. Springer, 1990.

[Pol93a] E. Poll. A type checker for bijective pure type systems. Computing Science Note CS-9322, Eindhoven University of Technology, 1993.

[Pol93b] R. Pollack. Closure under Alpha Conversion. In H. Barendregt and T. Nipkov, editors, *In Types for Proofs and Programs*, LNCS, pages 313–332. Springer-Verlag, 1993.

[Pol94] E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1994.

[Pol96] R. Pollack. A Verified Type Checker. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory*, 86. Göteborg University and Chalmers University of Technology, 1996.

[Raa93] F. van Raamsdonk. Confluence and superdevelopment. In Claude Kirchner, editor, *Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 168–182. Springer-Verlag, 1993.

[Raa96] F. van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting.* PhD thesis, VUA, University of Amsterdam, 1996.

[Reg94] L. Régnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126:281–292, 1994.

[RS95] F. van Raamsdonk and P. Severi. On normalisation. Technical Report CS-R9545 and CS-9520, CWI-Amsterdam and TU-Eindhoven, 1995.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium: Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer, 1974.

[RHP87] F. Honsell R. Harper and G. Plotkin. A Framework For Defining Logics. In *Proc. Second Symposium of Logic in Computer Science*, pages 194–204, 1987.

[Sch65] D. E. Schroer. *The Church-Rosser Theorem*. PhD thesis, Cornell University, 1965.

[Sev96] P. Severi. Pure type systems without the Π-condition. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory*, 86. Göteborg University and Chalmers University of Technology, 1996.

[Sor94] M. H. Sørensen. Effective longest reductions paths in untyped λ-calculus. Draft, Katholieke Universiteit Nijmegen, 1994.

[SP93] P. Severi and E. Poll. Pure type systems with definitions. Computing Science Note CS-9324, Eindhoven University of Technology, 1993.

[SP94] P. Severi and E. Poll. Pure type systems with definitions. In A. Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science: Proceedings of the Third International Symposium*, 813, pages 316–328. LFCS'94, St. Petersburg Russia, Springer-Verlag, Berlin, New York, 1994.

[Spr95] J. Springintveld. *Algorithms for Type Theory*. PhD thesis, Department of Philosophy - Utrecht University, 1995.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Tai75] W. W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Proceedings of the Logic Colloquim*, volume 453 of *LNCS*, pages 240–251. Springer, Berlin, 1975.

[Tas93] A. Tasistro. Formulation of Martin-Löf's Theory of Types with Explicit Substitution. Computing science note, Chalmers University of Technology and University of Göteborg, 1993.

[Ter89] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Manuscript, 1989.

[Tro73] A. S. Troelstra. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, volume 344 of *LNM*. Springer, 1973.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *IFIP Int'l Conf. Funct. Program. Comput. Arch.*, volume 201 of *LNCS*. Springer, 1985.

[Urz93a] P. Urzyczyn. The emptiness problem for intersection types. Technical report, Institute of Informatics University of Warsaw, 1993.

[Urz93b] P. Urzyczyn. Type Reconstruction in F ω is undecidable. In *Int'l Conf. Typed Lambda Calculus and its Applications*, volume 664 of *LNCS*, pages 418–432. Springer, 1993.

[Vri85] R. de Vrijer. A Direct Proof of the Finiteness of Developments Theorem. *The Journal of Symbolic Logic*, 50, 1985.

[Vri87] R. de Vrijer. Exactly estimating functionals and strong normalization. In *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, volume 90, pages 479–493, 1987.

[Wel94] J. B. Wells. Typablity and Type Checking in the Second Order λ-calculus are Equivalent and Undecidable. In *Ninth Anual IEEE Symposium on Logic in Computer Science*, pages 176–185. IEEE, 1994.

Index