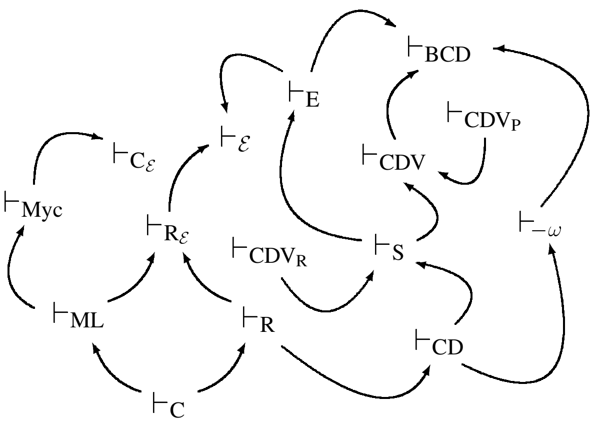


ISBN 90-9005752-8

Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems

een wetenschappelijke proeve op het gebied van de Wiskunde en Informatica.

Proefschrift

ter verkrijging van de graad van doctor aan de Katholieke Universiteit Nijmegen, volgens besluit van het College van Decanen in het openbaar te verdedigen op

woensdag 3 februari 1993 des namiddags te 1.30 uur precies

door

Stephanus Johannes van Bakel

geboren op 15 februari 1958 te Doetinchem.

Mathematisch Centrum, Amsterdam

Promotores

Prof. M. Dezani-Ciancaglini
Dipartimento di Informatica, Università degli Studi di Torino, Italia.

Prof. H.P. Barendregt.

Co-promotor

Dr. Ir. M.J. Plasmeijer.

‘*Castigo te non quod odio habeam, sed quod AMEM.*
I chastize thee not out of hatred, but out of love.’

– Henry Fielding, *Tom Jones, Book III* (1749)

Voor mijn moeder

Manuscriptcommissie

Prof. S. Ronchi della Rocca
Dipartimento di Informatica, Università degli Studi di Torino, Italia.

Dr. Ir. C. Hemerik
Vakgroep Theoretische Informatica, Technische Universiteit Eindhoven, Nederland.

Contents

Preface

The work on intersection type assignment systems presented in this thesis grew between 1988 and 1992. I first encountered intersection types in 1986 in an introductory course on Lambda Calculus by professor Henk Barendregt at the University of Nijmegen. Under his supervision I wrote a master's thesis [van Bakel '88], in which a first version of the Strict Type Assignment System was presented. After my MSc-graduation in 1988 I worked with professors Mariangiola Dezani-Ciancaglini, Mario Coppo and Simonetta Ronchi della Rocca in Turin, Italy, from February 15 until July 15 1988. This resulted in a paper [van Bakel '92a] that appeared in the June 1992 issue of *Theoretical Computer Science*. The results presented in this paper can be found in the chapters four and seven of this thesis.

In the first half of 1989 I implemented an intersection type inference algorithm for lambda terms, reported on in [van Bakel '90]. It led to the proof that the Strict Type Assignment System has the principal type property. This proof was presented in [van Bakel '91], which is submitted for publication to the journal *Logic and Computation*. The results presented in that paper can be found in chapter six.

In March 1991 I visited Turin. In a discussion with Mariangiola Dezani-Ciancaglini and Mario Coppo the observation was made that in Term Rewriting Systems, types are not preserved under rewriting. This problem was successfully tackled and reported on in the paper [van Bakel *et al.* '92] which appeared in the February 1992 proceedings of CAAP '92, *Colloquium on Trees in Algebra and Programming*, Rennes, France. Part of the results of that paper can be found in the chapters nine and eleven. The design of the Applicative Term Rewriting Systems and the notion of type assignment as presented in that paper was the result of a cooperation between Sjaak Smetsers of the University of Nijmegen, Simon Brock of the University of East Anglia, Norwich, U.K., and myself. In the second half of 1991 I generalized this notion of type assignment to the system that uses intersection types of Rank 2, which resulted in [van Bakel '92c], that is submitted for publication to the *Journal of Functional Programming*. The results of that paper can be found in chapters eight and twelve.

When writing this thesis I defined the Essential Type Assignment System, both for Lambda Calculus and Applicative Term Rewriting Systems. These two notions of type assignment were also presented in [van Bakel '92b]. The former has been submitted to LICS '93, *Logic In Computer Science*, Montreal, Canada, and can be found in chapter five, the latter will appear as [van Bakel '93] in the March 1993 proceedings of TLCA '93, *Typed Lambda Calculi and Applications*, Utrecht, the Netherlands, and can be found in chapter ten.

The research presented in this thesis was supported by:

- The Dutch Organisation for the Advancement of Pure Research (N.W.O.), grant NF-68.
- The Esprit Basic Research Action 3074 'Semantics and Pragmatics of Generalized Graph Rewriting (Semagraph)'.
- N.W.O.-project 'Design and implementation of a type system based on intersection types for the functional graph rewriting language Concurrent Clean', grant 612-316-027.

Prerequisites

I assume the reader to be familiar with the lambda calculus, including notions like normal forms, reduction strategies, λ-models, etc. For definitions and notions used here, see [Barendregt '84].

Notations

This section is given as a point of return while reading the thesis; its contents is not intended as something to read before studying the chapters of the thesis.

In this thesis, the symbol φ (often indexed, like in φ_i) will be a type-variable; when writing a type-variable φ_i, I will sometimes use the index i only, so as to obtain more readable types. Greek symbols like α, β, γ, μ, ν, η, ρ, σ and τ (often indexed) will range over types, and π will be used for principal types. To avoid parentheses in the notation of types, I will assume that '$\rightarrow$' associates to the right – so right-most, outer-most brackets will be omitted – and I will assume that, as in logic, '$\cap$' binds stronger than '$\rightarrow$'. So $\sigma\cap\tau\rightarrow\rho\cap\mu\rightarrow\gamma$ means $((\sigma\cap\tau)\rightarrow((\rho\cap\mu)\rightarrow\gamma))$. I use the word 'subtype' to express that one type is a syntactic component of another or the type itself.

I use M, N for lambda terms, x, y, z for term-variables and A for terms in $\lambda\bot$-normal form. F, G, H, I, etc. are used for function symbols, Q for constants and T for terms in rewriting systems.

I use B for bases, $B\backslash x$ for the basis obtained from B by erasing the statements that have x as subject, and P for principal bases. Two types (bases, pairs of basis and type) are *disjoint* if and only if they have no type-variables in common.

Notions of type assignment are defined as ternary relations on bases, terms and types, that are denoted by $\vdash$. I use $B \vdash_I M{:}\sigma$ for the statement 'M can be typed with the type σ starting from a basis B using the set of derivation rules that is indicated by I'. If in a notion of type assignment for M there are basis B and type σ such that $B \vdash M{:}\sigma$, or a node or edge in the tree-representation of terms is labelled with a type σ, then the term (node, edge) is *typed* with σ, and σ is *assigned* to it.

I use *O* for operations on types, bases and pairs of basis and type; I use *D* for duplications, *E* for expansions, *L* for liftings, *S* for substitutions, and *W* for weakenings.

I use the symbol $\square$ to mark the end of a proof.

Acknowledgements

The first person I would like to thank is Mariangiola Dezani, for the encouragement, support and many helpful suggestions on the various papers (and the various versions of them) I wrote. Visiting Turin has always been stimulating because of her: without her, this thesis would not have been written. I would also like to thank Henk Barendregt for the enlightening discussions, and Rinus Plasmeijer for creating the perfect working environment.

I thank my parents Gon van Bakel-van Heumen and Jan van Bakel for teaching me that the most important things in life are tolerance towards other people and to follow one's own ideas; in research, both attitudes proved valuable.

Introduction

Over the last decade the popularity of functional programming has increased. A large and still growing number of people is becoming interested in functional programming languages: computer scientists and logicians on the scientific side as well as hard- and software manufacturers on the applied side. Unfortunately for the major part of people working in computer science or companies, functional programming languages are but a toy, since programs written in the average functional language take a long time to execute. Nevertheless impressive improvements are achieved in implementing these languages (reaching nfib-numbers of over one million for the functional programming language Clean on an Apple Macintosch IIFX) and it seems just a matter of time before they will become a practical tool in program development.

In recent years several paradigms were investigated for the implementation of functional programming languages. Not only the Lambda Calculus [Barendregt '84], but also Term Rewriting Systems [Klop '90] and Term Graph Rewriting Systems [Barendregt *et al.* '87] were topics of research. Lambda Calculus (or rather combinator systems) constitutes the underlying model for the functional programming language Miranda [Turner '85]. Term Rewriting Systems were used in the underlying model for the language OBJ [Futatsugi *et al.* '85], and Term Graph Rewriting Systems were the model for the language Clean [Brus *et al.* '87, Nöcker *et al.* '91].

The extension from Lambda Calculus to Term Rewriting Systems is done via combinator systems. Term rewrite rules are written very much like the definitions of combinators, the difference being that a formal parameter can be a pattern: it is allowed to have structure and it need not be a term-variable. Term Graph Rewriting Systems are obtained from Term Rewriting Systems by writing terms as trees, and allowing cyclic structures as well as sharing of nodes.

The Lambda Calculus, Term Rewriting Systems and Graph Rewriting Systems themselves are type free, whereas in programming the notion of types plays an important role. Type assignment to programs and objects is in fact a way of performing abstract interpretation that provides necessary information for both compilers and programmers. Types are essential to obtain efficient machine code when compiling a program and are also used to make sure that the programmer has a clearer understanding of the programs that are written. Since Lambda Calculus is a fundamental basis for many functional programming languages, a type assignment system for pure untyped Lambda Calculus capable of deducing meaningful and expressive types has been a topic of research for many years.

There are various ways to deal with the problem of handling types in programming languages. These can roughly be divided in the 'typed' and 'untyped' approaches. The 'typed' approach can be found in programming languages that have explicit typing: objects in a program have types provided by the programmer, and the type-algorithm incorporated in the compiler for the language checks if these are used consistently. The 'untyped' approach is used in languages that allow programmers to write programs without any type-specification at all, and it is the task of the type-algorithm to infer types for objects and to check consistency.

There exists a well-understood and well-defined notion of type assignment on lambda terms, known as the Curry Type Assignment System [Curry & Feys '58, Curry '34]), which expressed abstraction and application. Many of the now existing type assignment systems for functional programming languages that use the 'untyped' approach are based on (extensions of) Curry's system. For example, the functional programming language ML [Milner '78] is in fact an extended lambda calculus and its type system is based on Curry's system.

It is well known that in Curry's system, the problem of typeability

Given a term M, are there basis B and type σ such that $B \vdash M{:}\sigma$?

is decidable. This system also has the principal type property: M is typeable if and only if there are a basis P, and type π, such that

$P \vdash M{:}\pi$, and for every pair $<B, \sigma>$ such that $B \vdash M{:}\sigma$, there exists an operation O (from a specified set of operations) such that $O(<P, \pi>) = <B, \sigma>$.

The type π is then called a 'principal type for M'. In general it is undecidable whether such an operation exists in the specified collection, except for certain special cases. For Curry's system the operation O consists entirely of substitutions, i.e. operations that replace type-variables by types. Principal type schemes for Curry's system were defined in [Hindley '69].

The existence of a principal type within a type assignment system for a typeable lambda term M shows an internal coherence between all types that can be assigned to M. Since substitution is an easy operation, the set $\{ <B, \sigma> \mid B \vdash M{:}\sigma \}$ can be computed in Curry's system easily from the principal pair for M. The principal type property plays an important role in the 'untyped' approach, since, in an implementation, it allows of the use of the principal type of M to find the right types for the various occurrences of M without deriving the type for M over and over again; this property constitutes the basis for the notion of 'polymorphic functions' in programming languages like ML.

In some type assignment systems it is uncertain whether or not this property holds. For example, for the Polymorphic Type Discipline [Girard '86] there is no natural way to obtain the types $(\forall\varphi.\varphi)\rightarrow(\forall\varphi.\varphi)$ and $(\forall\varphi.\varphi\rightarrow\varphi)\rightarrow(\forall\varphi.\varphi\rightarrow\varphi)$, both types for the lambda term $(\lambda x.xx)$ from a single type (see [Giannini & Ronchi della Rocca '88]). In any case, there exists no type σ derivable for $\lambda x.xx$ such that both types can be obtained from σ by substitution.

In [Milner '78] an extension of Curry's system is presented, which can be seen as a restriction of the Polymorphic Type Discipline. Type assignment in this system is decidable and it has the principal type property. It is designed to formalize the notion of polymorphic functions as used in functional programming languages like ML and Miranda. (In [Mycroft '84] a slightly more general extension of this system was presented in which type assignment is no longer decidable, but that is nevertheless used for type checking in Miranda.)

Although frequently used in functional programming languages, the Curry Type Assignment System has some drawbacks. In this system it is, for example, not possible to assign a type to the term $(\lambda x.xx)$. Moreover, although the lambda terms $(\lambda cd.d)$ and $((\lambda xyz.xz(yz))(\lambda ab.a))$ are β-equal, the principal type schemes for these terms are different.

The Intersection Type Discipline as presented in [Coppo *et al.* '81, Barendregt *et al.* '83] is an extension of Curry's system that does not have these drawbacks. The extension to Curry's system is essentially that term-variables are allowed to have more than one type. Intersection types are constructed by adding, next to the type constructor '$\rightarrow$', the type constructor '$\cap$' and the type constant 'ω'. This yields a type assignment system that is very powerful: it is closed under β-equality:

If $B \vdash M{:}\sigma$ and $M =_\beta N$, then $B \vdash N{:}\sigma$.

Because of this power, type assignment in this system is undecidable.

As stated above, if a type assignment system with intersection types is desired – instead of Curry's system – for the construction of a type inference system for an untyped functional programming language, then such a system should at least have the principal pair property. (Of course decidability of type assignment is also convenient.) There are two intersection type assignment systems for which this property is proved.

In [Coppo *et al.* '80] principal type schemes were defined for a type assignment system that is a restriction of the one presented in [Coppo *et al.* '81]. This system has as a disadvantage that it is not an extension of Curry's system: if $B \vdash M{:}\tau$, and the variable x does not occur in B, then for $\lambda x.M$ only the type $\omega\rightarrow\tau$ can be derived. Therefore, it is impossible to derive $\varphi_0\rightarrow\varphi_1\rightarrow\varphi_0$ for the lambda term $\lambda ab.a$. This type is derivable for that term in Curry's system.

For the BCD-system defined in [Barendregt *et al.* '83], principal type schemes can be defined as in [Ronchi della Rocca & Venneri '84]. In [Ronchi della Rocca '88] a unification semi-algorithm for intersection types was presented, together with a semi-algorithm that finds the principal type for every strongly normalizable lambda term. These algorithms are called 'semi-algorithms', because they do not terminate for every possible input.

The BCD-system has as a disadvantage that it is too general: in this system there are several ways to deduce a desired result, due to the presence of the derivation rules ($\cap$I), ($\cap$E) and ($\leq$). These rules not only allow of superfluous steps in derivations, but they also make it possible to give essentially different derivations for the same result. Moreover, in [Barendregt *et al.* '83] the relation $\leq$ induced an equivalence relation $\sim$ on types. Equivalence classes are big (for example: $\omega \sim \sigma\rightarrow\omega$, for all types σ) and type assignment is closed for $\sim$. And although the set $\{ <B, \sigma> \mid B \vdash M{:}\sigma \}$ can be generated using the operations specified in [Ronchi della Rocca & Venneri '84], the problem of type-checking

Given a term M and type σ, is there a B such that $B \vdash M{:}\sigma$?

is complicated.

Although the BCD-system has the principal type property, it is not used in type checkers for functional languages at the present time, as type assignment is undecidable in this system. In order to obtain a terminating type checker based on this system, some restrictions have to be made. There are of course limitations of the BCD-system that provide decidable type assignment and efficient implementation (the trivial one being the restriction to Curry's system). Another possibility is the one suggested in [Leivant '83]: limiting the set of types to intersection types of Rank 2.

Because of the similarity between this system and the one for ML, this system got little attention from the functional programming world in the past. This is surprising, considering the several advantages of the Rank 2 system over the one for ML. Not only the class of typeable terms is significantly extended when intersection types of Rank 2 are used, but also more accurate types can be deduced for terms. Moreover, when using the ML-type checker it is possible that a program (correct in the programmers mind) is rejected because of occurring type conflicts, while on the other hand it could be accepted after the programmer has rewritten the specification. Such a rewrite would not be necessary if Rank 2 types were used.

Most functional programming languages, for instance Miranda, allow programmers to specify an algorithm (function) as a set of rewrite rules. It is remarkable that little is known – apart from the algebraic approach [Dershowitz & Jouannaud '90] – about type assignment in

Term Rewriting Systems. The type assignment systems incorporated in most term rewriting languages are in fact extensions of type assignment systems for a(n extended) lambda calculus, and although it seems straightforward to generalize those systems to the (significantly larger) world of Term Rewriting Systems, it is at first glance not evident that those borrowed systems still have all the properties they possessed in the world of Lambda Calculus. For example, type assignment in Term Rewriting Systems in general does not satisfy the subject reduction property: i.e. types are not preserved under rewriting. In order to be able to study the details of type assignment for Term Rewriting Systems, a formal notion of type assignment on Term Rewriting Systems is needed, that is closer to the approach of type assignment in Lambda Calculus than the algebraic one.

The aim of the research presented in this thesis is develop to type assignment systems with intersection types for the Lambda Calculus and for Term Rewriting Systems, in order to investigate and understand their structure and properties and to see whether the definition of a type-checker for a functional programming language with intersection types is feasible. Intersection types are examined because they are a good means to perform abstract interpretation: better than Curry types, also even better than the kind of types used in languages like ML. Also, the presented formalisms could be extended to the world of Term Graph Rewriting Systems, which is favourable as in that world intersection types are the natural tool to type nodes that are shared. Moreover, intersection types seem to be promising for use in functional programming languages, since they seem to provide a good formalism to express overloading. The results of this research can be used as a guideline to develop type-inferencing (and type-checking) algorithms using intersection types in functional programming languages.

There is a great number of notions of type assignment presented in this thesis (fifteen in total), each defined as a ternary relation on bases, terms and types and denoted by $\vdash$, indexed with information to be able to distinguish them.

In chapters one through three a short overview of various type disciplines will be given. It starts with the presentation of the Curry Type Assignment System ($\vdash_{\mathrm{C}}$) in chapter one. Chapter two will contain the development of the Intersection Type Discipline, by presenting the several systems that were published in the past. In section 2.1 the Coppo-Dezani System ($\vdash_{\mathrm{CD}}$) will be discussed, and in section 2.2 three Coppo-Dezani-Venneri Systems ($\vdash_{\mathrm{CDV}}$, $\vdash_{\mathrm{CDV_R}}$, $\vdash_{\mathrm{CDV_P}}$). In section 2.3 the Barendregt-Coppo-Dezani System ($\vdash_{\mathrm{BCD}}$) will be discussed. In chapter three a short overview of the Milner Type Assignment System [Milner '78] ($\vdash_{\mathrm{ML}}$) will be given – that will be compared to the Rank 2 Intersection Type Assignment System in chapter eight – and the Mycroft Type Assignment System [Mycroft '84] ($\vdash_{\mathrm{Myc}}$).

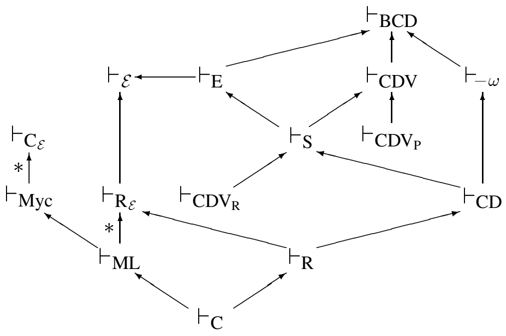
In chapters four through eight intersection type disciplines for the Lambda Calculus will be studied. In chapter four the Strict Type Assignment System ($\vdash_{\mathrm{S}}$) will be presented, a restriction of the BCD-system that satisfies all major properties of that full system, for which in chapter six will be proved that the principal type property holds. In chapter five the Essential Intersection Type Assignment System ($\vdash_{\mathrm{E}}$) will be defined, a slight generalization of the strict one and also a restriction of the BCD-system. The intersection type assignment system without the type constant ω ($\vdash_{-\omega}$) will be presented in chapter seven. It will be shown that all three restrictions yield undecidable systems. So these attempts to restrict the system in preparation for the construction of a type checker will fail. In chapter eight the Rank 2 Intersection Type Assignment System ($\vdash_{\mathrm{R}}$) will be defined, the smallest restriction of the Intersection Type Discipline that contains intersection types. It will be shown that in this system type assignment is decidable and that it has the principal type property.

In chapters nine through twelve intersection type disciplines will be studied for Applicative Term Rewriting Systems, that will be defined in chapter nine. In chapter ten a formal notion of type assignment ($\vdash_{\mathcal{E}}$) will be presented that uses strict intersection types. Chapters eleven and twelve will be presented in such a way that they can be read independently from the other chapters in this thesis, although it is advisable to study chapter eight before reading chapter twelve. Those chapters aim to present type assignment systems that can be used in functional programming languages, so not only will be shown that the presented systems have the principal type property, but also will be shown that type assignment is decidable in those systems by presenting (terminating) unification algorithms that should be implemented when such a system is used.

In chapter eleven a formal notion of type assignment on left linear Applicative Term Rewriting Systems will be presented ($\vdash_{\mathrm{C}_{\mathcal{E}}}$) that uses Curry types, and the way of dealing with recursion of the extension defined by Mycroft of Curry's system. This chapter aims to give formal motivation for the type system of Miranda, and to provide a formal type system for all languages that use pattern matching and have type systems based on Mycroft's extension of Curry's system.

In chapter twelve a type assignment system for Applicative Term Rewriting Systems will be defined that uses intersection types of Rank 2 ($\vdash_{\mathrm{R}_{\mathcal{E}}}$). The Rank 2 system is very close to the Milner Type Assignment System, as discussed in chapter eight, so should show the advantages of intersection types over Curry types (or ML types).

In a picture the relation between the several notions of type assignment looks like this:



An arrow is drawn from one turnstyle to another if the second (the one the arrow will eventually point at) is a natural extension (or generalization) of the first (the one the arrow will point from). This means that ideas and definitions of the system at the start of the arrow are used to define the system at the end of it, or the system at the start is a restriction of the system at the end. When the arrow does not stand for a real extension, it is marked by $*$. Otherwise if $B \vdash M{:}\sigma$ holds in the system at the start of the arrow, then $B \vdash M{:}\sigma$ holds in the system at the end; the latter M is sometimes obtained from the former one by bracket-abstraction.

Chapter 1 *The Curry Type Assignment System*

Type assignment for the Lambda Calculus was first studied in [Curry '34]. (See also [Curry & Feys '58].) Curry's system – the first and most primitive one – expresses abstraction and application and has as its major advantage that the problem of type assignment (given a term M, are there B, σ such that $B \vdash_{\mathrm{C}} M{:}\sigma$) is decidable. In this chapter we will present the main definitions and results for this system, using a notation that will be used throughout this thesis.

Definition 1.1 (cf. [Curry '34, Curry & Feys '58]) i) $\mathcal{T}_{\mathrm{C}}$, the set of *Curry-types* is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{\mathrm{C}}$.

b) If σ and $\tau \in \mathcal{T}_{\mathrm{C}}$, then $\sigma \rightarrow \tau \in \mathcal{T}_{\mathrm{C}}$.

ii) A *Curry statement* is an expression of the form $M{:}\sigma$, where $M \in \Lambda$ and $\sigma \in \mathcal{T}_{\mathrm{C}}$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

iii) A *Curry basis* is a set of Curry statements with only distinct variables as subjects.

Definition 1.2 (cf. [Curry '34, Curry & Feys '58]) i) *Curry-type assignment* and *Curry-derivations* are defined by the following natural deduction system.

$$(\rightarrow\mathrm{I}){:}\quad \frac{\begin{array}{c}[x{:}\sigma]\\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\rightarrow\tau}\ (a) \qquad\qquad (\rightarrow\mathrm{E}){:}\quad \frac{M{:}\sigma\rightarrow\tau \qquad N{:}\sigma}{MN{:}\tau}$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

ii) If $M{:}\sigma$ is derivable from B using a Curry-derivation, we write $B \vdash_{\mathrm{C}} M{:}\sigma$.

The main properties of this system are:

- The principal type property.
- Decidability of type assignment.
- Strongly normalizability of all typeable terms.

Because of the decidability of type assignment in this system, many of the now existing type assignment systems for functional programming languages are based on Curry's system.

Principal type schemes for Curry's system were defined in [Hindley '69]. In this paper Hindley actually proved the existence of principal types for an object in Combinatory Logic, but the same construction can be used for a proof of the principal type property for terms in the Lambda Calculus. We will discuss this construction briefly.

First we define the operation of Curry-substitution on types as the operation that replaces type-variables by types in a consistent way. There are various ways to formally define such

an operation; the one we choose here will be used throughout this thesis. The operation is called a Curry-substitution in order to distinguish it from other substitutions, very similar to Curry-substitution, as defined in definitions 2.2.2.2, 2.3.2.1, 3.1.2, 6.2.1.1, and 8.3.1.1.

Definition 1.3 i) a) The *Curry-substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_{\mathrm{C}} \rightarrow \mathcal{T}_{\mathrm{C}}$ where φ is a type-variable and $\alpha \in \mathcal{T}_{\mathrm{C}}$, is inductively defined by:

1) $(\varphi \mapsto \alpha)(\varphi) = \alpha$.

2) $(\varphi \mapsto \alpha)(\varphi') = \varphi'$, if $\varphi \neq \varphi'$.

3) $(\varphi \mapsto \alpha)(\sigma\rightarrow\tau) = (\varphi \mapsto \alpha)(\sigma) \rightarrow (\varphi \mapsto \alpha)(\tau)$.

b) If S_1 and S_2 are Curry-substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2(\sigma) = S_1(S_2(\sigma))$.

c) $S(B) = \{\, x{:}S(\alpha) \mid x{:}\alpha \in B \,\}$.

d) $S(<B, \sigma>) = <S(B), S(\sigma)>$.

ii) If for σ, τ there is a Curry-substitution S such that $S(\sigma) = \tau$, then τ is called a *(substitution) instance* of σ.

iii) If σ is an instance of τ, and τ is an instance of σ, then σ is called a *trivial variant* of τ. We identify types that are trivial variants of each other.

Curry-substitution is a sound operation on pairs:

Property 1.4 If $B \vdash_{\mathrm{C}} M{:}\sigma$, then for every Curry-substitution S: if $S(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_{\mathrm{C}} M{:}\sigma'$. $\square$

Principal types for lambda terms are defined using the notion of unification of types that was defined in [Robinson '65]. Robinson's unification is a procedure on Curry-types which, given two arguments, returns a Curry-substitution that maps the arguments to a common instance. It can be defined as follows:

Definition 1.5 Let $\mathcal{B}$ the collection of all Curry bases, $\mathcal{S}$ be the set of all Curry-substitutions, and $Id_{\mathcal{S}}$ be the Curry-substitution that replaces all type-variables by themselves.

i) *Robinson's unification algorithm.* The operation $\mathit{unify}_{\mathrm{R}} : \mathcal{T}_{\mathrm{C}} \times \mathcal{T}_{\mathrm{C}} \rightarrow \mathcal{S}$ is defined by:

$$\begin{array}{ll}
\mathit{unify}_{\mathrm{R}}(\varphi, \varphi') & = (\varphi := \varphi'). \\
\mathit{unify}_{\mathrm{R}}(\varphi, \mu) & = (\varphi := \mu), \text{ if } \varphi \text{ does not occur in } \mu \text{ and } \mu \text{ is not a type-variable.} \\
\mathit{unify}_{\mathrm{R}}(\sigma, \varphi) & = \mathit{unify}_{\mathrm{R}}(\varphi, \sigma). \\
\mathit{unify}_{\mathrm{R}}(\sigma\rightarrow\tau, \rho\rightarrow\mu) & = S_2 \circ S_1, \\
\multicolumn{2}{l}{\qquad \text{where } S_1 = \mathit{unify}_{\mathrm{R}}(\sigma, \rho),} \\
\multicolumn{2}{l}{\qquad\qquad S_2 = \mathit{unify}_{\mathrm{R}}(S_1(\tau), S_1(\mu)).}
\end{array}$$

ii) By defining the operation $\mathit{UnifyBases} : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{S}$ the operation $\mathit{unify}_{\mathrm{R}}$ can be generalized to bases:

$$\begin{array}{ll}
\mathit{UnifyBases}(B_0 \cup \{x{:}\sigma\}, B_1) & = S_2 \circ S_1, \qquad \text{if } x{:}\tau \in B_1 \\
\multicolumn{2}{l}{\qquad \text{where } S_1 = \mathit{unify}_{\mathrm{R}}(\sigma, \tau),} \\
\multicolumn{2}{l}{\qquad\qquad S_2 = \mathit{UnifyBases}(S_1(B_0), S_1(B_1)).} \\
\mathit{UnifyBases}(B_0 \cup \{x{:}\sigma\}, B_1) & = \mathit{UnifyBases}(B_0, B_1), \text{ if } x \text{ does not occur in } B_1. \\
\mathit{UnifyBases}(\emptyset, B_1) & = Id_{\mathcal{S}}.
\end{array}$$

Notice that *unify*$_R$ can fail only in the second alternative, when φ occurs in μ.

The following property of Robinson's unification is very important for all systems that depend on it, and formulates that *unify*$_R$ returns the most general unifier of two Curry-types. This means that if two Curry-types σ and τ have a common instance, then they have a least common instance γ and all their common instances can be obtained from γ by Curry-substitution.

Property 1.6 [Robinson '65] If two types have an instance in common, they have a *highest common instance* which is returned by *unify*$_R$, so for all σ, τ: if S_1 is a substitution such that $S_1(\sigma) = S_1(\tau)$, then there are substitutions S_2 and S_3 such that

$$S_2 = \mathit{unify}_R(\sigma, \tau) \text{ and } S_1(\sigma) = S_3 \circ S_2(\sigma) = S_3 \circ S_2(\tau) = S_1(\tau).$$

Since the substitution returned by *unify*$_R$ is only defined on type-variables occurring in σ and τ, it is even possible to show that $S_1 = S_3 \circ S_2$. □

The definition of principal pairs for lambda terms in Curry's system then looks like:

Definition 1.7 We define for every term M the *Curry principal pair* by defining the notion $PP_C(M) = <P, \pi>$ inductively by:

i) For all x, φ: $PP_C(x) = <\{x{:}\varphi\}, \varphi>$.

ii) If $PP_C(M) = <P, \pi>$, then:

  a) If x occurs free in M and $x{:}\sigma \in P$, then $PP_C(\lambda x.M) = <P\backslash x, \sigma \rightarrow \pi>$.

  b) otherwise $PP_C(\lambda x.M) = <P, \varphi \rightarrow \pi>$, where φ does not occur in $<P, \pi>$.

iii) If $PP_C(M_1) = <P_1, \pi_1>$ and $PP_C(M_2) = <P_2, \pi_2>$ (we choose, if necessary, trivial variants such that the $<P_i, \pi_i>$ are disjoint in pairs), φ is a type-variable that does not occur in any of the pairs $<P_i, \pi_i>$, and

$$S_1 = \mathit{unify}_R(\pi_1 \rightarrow \varphi, \pi_2)$$
$$S_2 = \mathit{UnifyBases}(S_1(P_1), S_1(P_2)),$$

then $PP_C(M_1 M_2) = S_2 \circ S_1(<P_1 \cup P_2, \varphi>)$.

The proof that these are indeed the principal pairs is given by showing that all possible pairs for a typeable term M can be obtained from the principal one by applying Curry-substitutions (similar to the proof of theorem 11.2.2). In this proof, property 1.6 is needed.

Chapter 2 *Intersection Type Assignment Systems*

Although type assignment in Curry's system is decidable and has the principal type property, it has drawbacks. It is for example not possible to assign a type to the lambda term $\lambda x.xx$, and although the lambda terms $\lambda cd.d$ and $(\lambda xyz.xz(yz))\lambda ab.a$ are β-equal, the principal type schemes for these terms are different, $\varphi_0 \rightarrow \varphi_1 \rightarrow \varphi_1$ and $(\varphi_1 \rightarrow \varphi_0) \rightarrow \varphi_1 \rightarrow \varphi_1$ respectively. The Intersection Type Discipline is an extension of Curry's Type Assignment System for the pure Lambda Calculus that does not have these drawbacks. It has developed over a period of several years; we will not just give the final presentation, but show the various systems as they appeared between 1980 and 1983, since they all play a role in this thesis.

2.1 *The Coppo-Dezani Type Assignment System*

The first paper by M. Coppo and M. Dezani-Ciancaglini from the University of Turin, Italy that introduced intersection types is [Coppo & Dezani-Ciancaglini '80] (in this paper the word 'intersection' was not used; instead it used the word 'sequence'). The system presented in this paper is a true extension of Curry's system: the extension made is to allow more than one type for term-variables in the ($\rightarrow$I)-derivation rule, and therefore to allow of, also, more than one type for the right hand term in the ($\rightarrow$E)-derivation rule.

We will present the definition and main properties of the Coppo-Dezani system in the notation used in this thesis.

Definition 2.1.1 [Coppo & Dezani-Ciancaglini '80] The set of Coppo-Dezani types is inductively defined by:

i) All type-variables $\varphi_0, \varphi_1, \ldots$ are types.

ii) If $\sigma_1, \ldots, \sigma_n$ are types ($n \geq 1$), then $\sigma_1 \cap \cdots \cap \sigma_n$ is a sequence.

iii) If $\sigma_1 \cap \cdots \cap \sigma_n$ is a sequence and τ is a type, then $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau$ is a type.

Definition 2.1.2 [Coppo & Dezani-Ciancaglini '80] i) *CD-type assignment* and *CD-derivations* are defined by the following natural deduction system.

$$(\rightarrow I): \quad \frac{\begin{array}{c}[x{:}\sigma_1 \cap \cdots \cap \sigma_n] \\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau} \ (a) \qquad (\rightarrow E): \quad \frac{M{:}\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau \quad N{:}\sigma_1 \cap \cdots \cap \sigma_n}{MN{:}\tau}$$

$$(\cap I): \quad \frac{M{:}\sigma_1 \quad \cdots \quad M{:}\sigma_n}{M{:}\sigma_1 \cap \cdots \cap \sigma_n} \qquad (\cap E): \quad \frac{x{:}\sigma_1 \cap \cdots \cap \sigma_n}{x{:}\sigma_i}$$

(a) If $x{:}\sigma_1 \cap \cdots \cap \sigma_n$ is the only statement about x on which $M{:}\tau$ depends.

ii) If $M{:}\sigma$ is derivable from B using a CD-derivation, we write $B \vdash_{\text{CD}} M{:}\sigma$.

The main properties of that system proved in [Coppo & Dezani-Ciancaglini '80] are:

- Subject reduction: If $B \vdash_{\text{CD}} M{:}\sigma$, and $M \rightarrow_\beta N$, then $B \vdash_{\text{CD}} N{:}\sigma$.
- Normalizability of typeable terms: If $B \vdash_{\text{CD}} M{:}\sigma$, then M has a normal form.
- Typeability of all terms in normal form.
- Closure for β-equality in the λI-calculus: if $B \vdash_{\text{CD}} M{:}\sigma$, and $M =_\beta N$, then $B \vdash_{\text{CD}} N{:}\sigma$.
- In the λI-calculus: $B \vdash_{\text{CD}} M{:}\sigma$ if and only if M has a normal form.

That subject reduction holds is not difficult to see (we give an intuitive argument). Suppose there exists a type assignment for the redex $(\lambda x.M)N$, so there are a basis B and a type σ such that there is a derivation for $B \vdash_{\text{CD}} (\lambda x.M)N{:}\sigma$. Then by ($\rightarrow$E) there is a sequence $\tau_1 \cap \cdots \cap \tau_n$ such that there are derivations $B \vdash_{\text{CD}} \lambda x.M{:}\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma$ and $B \vdash_{\text{CD}} N{:}\tau_1 \cap \cdots \cap \tau_n$. Since ($\rightarrow$I) should be the last step performed in the derivation for $B \vdash_{\text{CD}} \lambda x.M{:}\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma$, there is also a derivation for $B \cup \{x{:}\tau_1 \cap \cdots \cap \tau_n\} \vdash_{\text{CD}} M{:}\sigma$, and then a derivation for $B \vdash_{\text{CD}} M[x := N]{:}\sigma$ can be obtained from this one by replacing for $1 \leq i \leq n$ the statement $x{:}\tau_i$ by the CD-derivation for $N{:}\tau_i$.

The problem to solve in a proof for closure under β-equality is then that of β-expansion: suppose we have derived $B \vdash_{\text{CD}} M[x := N]{:}\sigma$ and also want to derive $B \vdash_{\text{CD}} (\lambda x.M)N{:}\sigma$. When restricting to λI-terms, the term-variable x occurs in M, and the term N is a subterm of $M[x := N]$, so N is typed in the derivation for $B \vdash_{\text{CD}} M[x := N]{:}\sigma$. It may be that in this derivation the subterm N is typed with several different types. Then in Curry's system the redex $(\lambda x.M)N$ can not be typed using the same types, since then the basis would contain more than one type for x, which is not allowed. In the CD-system this problem is solved by the introduction of sequences. The assignment of a sequence to a term-variable allows of that variable to have different types within a derivation, so in this formalism the redex can be typed.

From this initial system several others emerged. The best known is the one presented in [Barendregt *et al.* '83], but there are two earlier papers ([Coppo *et al.* '81] and [Coppo *et al.* '80]) that investigate interesting systems which can be regarded as in-between the ones in [Coppo & Dezani-Ciancaglini '80] and [Barendregt *et al.* '83].

2.2 *The Coppo-Dezani-Venneri Type Assignment Systems*

In this section two papers by M. Coppo, M. Dezani-Ciancaglini and B. Venneri will be discussed. The paper [Coppo *et al.* '81] presented a notion of type assignment that is an extension of the CD-system; the paper [Coppo *et al.* '80] contained the proof of the principal type property for a restricted version of the system in [Coppo *et al.* '81].

2.2.1 *The systems of [Coppo et al. '81]*

In [Coppo *et al.* '81] two type assignment systems are presented, that, in approach, are more general than the Coppo-Dezani Type Assignment System: in addition to the type constructors

'$\rightarrow$' and '$\cap$', they also contain the type constant 'ω'. The first type discipline as presented in [Coppo *et al.* '81] is a true extension of the one presented in [Coppo & Dezani-Ciancaglini '80] (a similar system was presented in [Sallé '78]); the second one limits the use of intersection types in bases. By introducing the type constant ω next to the intersection types, a system is obtained that is closed under β-equality for the full λK-calculus: if $M =_\beta N$, where M and N are lambda terms, then $B \vdash_{\text{CDV}} M{:}\sigma \Leftrightarrow B \vdash_{\text{CDV}} N{:}\sigma$.

Definition 2.2.1.1 [Coppo *et al.* '81] i) $\mathcal{T}_{\text{CDV}}$, the set of Coppo-Dezani-Venneri types is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{\text{CDV}}$.

b) $\omega \in \mathcal{T}_{\text{CDV}}$.

c) If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_{\text{CDV}}$, $(n \geq 1)$, then $\sigma_1 \cap \cdots \cap \sigma_n$ is a sequence.

d) If $\sigma_1 \cap \cdots \cap \sigma_n$ is a sequence and $\tau \in \mathcal{T}_{\text{CDV}}$, then $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau \in \mathcal{T}_{\text{CDV}}$.

ii) Every $\tau \in \mathcal{T}_{\text{CDV}}$ can be written as $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$, where $\sigma_1, \ldots, \sigma_n$ are sequences, and σ is a type-variable or ω. The type τ is called *tail-proper* if $\sigma \neq \omega$.

Definition 2.2.1.2 [Coppo *et al.* '81] i) *CDV-type assignment* and *CDV-derivations* are defined by:

$$(\rightarrow\text{I}): \quad \frac{\begin{array}{c}[x{:}\sigma_1] \cdots [x{:}\sigma_n] \\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma \rightarrow \tau} \ (a) \qquad (\rightarrow\text{E}): \quad \frac{M{:}\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau \quad N{:}\sigma_1 \cap \cdots \cap \sigma_n}{MN{:}\tau}$$

$$(\omega): \quad \frac{}{M{:}\omega} \qquad (\cap\text{I}): \quad \frac{M{:}\sigma_1 \quad \cdots \quad M{:}\sigma_n}{M{:}\sigma_1 \cap \cdots \cap \sigma_n}$$

(a) If $x{:}\sigma_1, \ldots, x{:}\sigma_n$ are all and nothing but the statements about x on which $M{:}\tau$ depends, and σ is a sequence that at least contains all $\sigma_1, \ldots, \sigma_n$.

ii) If $M{:}\sigma$ is derivable from B using a CDV-derivation, we write $B \vdash_{\text{CDV}} M{:}\sigma$.

The main properties of this system proved in [Coppo *et al.* '81] are:

- If $B \vdash_{\text{CDV}} M{:}\sigma$ and $M =_\beta N$, then $B \vdash_{\text{CDV}} N{:}\sigma$.
- $B \vdash_{\text{CDV}} M{:}\sigma$ and σ is tail-proper, if and only if M has a head normal form.
- $B \vdash_{\text{CDV}} M{:}\sigma$ and ω does not occur in B and σ, if and only if M has a normal form.

To illustrate the difference between this system and the Coppo-Dezani system, we again look at β-expansion: suppose we have a derivation for $B \vdash_{\text{CDV}} M[x := N]{:}\sigma$ and also want to derive $B \vdash_{\text{CDV}} (\lambda x.M)N{:}\sigma$. For the full λK-calculus this problem is solved by the introduction of the type constant ω and the sequences. The type constant ω is the universal type, i.e. each term can be typed by ω. It can be used in the expansion to type N if N does not occur in $M[x := N]$, and there is no other type ρ such that $B \vdash_{\text{CDV}} N{:}\rho$. If N does not occur in $M[x := N]$, x does not occur in M, and therefore there is a derivation for $B \vdash_{\text{CDV}} \lambda x.M{:}\omega \rightarrow \sigma$. Since N is typeable by ω, by derivation rule ($\rightarrow$E) we have that $(\lambda x.M)N$ is typeable by σ.

As before, the sequences allow a term-variable to have different types within a derivation; they are used for the cases that N occurs more than once in $M[x := N]$, and these occurrences were typed in the derivation for $B \vdash_{\text{CDV}} M[x := N]{:}\sigma$ with different types.

The second type assignment system presented in [Coppo *et al.* '81] is a restricted version of the first. Since it limits the possible bases that can be used in a derivation, it is not a proper extension of Curry's system.

As mentioned in [Coppo *et al.* '81]:

> " *(...) notice that a type $\sigma{\rightarrow}\omega$ represents a property of terms which applied to a particular class of terms (i.e. terms in σ), give a term. But this property characterizes all terms and, so, it is natural to identify with ω all types of the shape $\sigma{\rightarrow}\omega$ and, in general, all non-tail-proper types. By a similar argument we (...) identify a sequence $\sigma_1{\cap}\cdots{\cap}\sigma_n$ with a sequence $\sigma_1'{\cap}\cdots{\cap}\sigma_m'$ obtained by eliminating from $\sigma_1{\cap}\cdots{\cap}\sigma_n$ all occurrences of non-tail-proper types σ_i (unless, obviously, $\sigma_1{\cap}\cdots{\cap}\sigma_n \equiv \omega$). These arguments justify the elimination of all non-tail-proper types (except ω) and of all sequences that contain non-tail-proper types (except ω).* "

A good justification for this identification of types can be found in the definition of type-semantics (definition 2.3.1.4). This leads to the following definition.

Definition 2.2.1.3 [Coppo *et al.* '81] i) The set of *normalized types* is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots$ are normalized types.

b) ω is a normalized sequence.

c) If $\sigma_1, \ldots, \sigma_n$ are normalized types ($n \geq 1$) and for $1 \leq i \leq n$ $\sigma_i \neq \omega$, then $\sigma_1{\cap}\cdots{\cap}\sigma_n$ is a normalized sequence.

d) If σ is a normalized sequence and τ is a normalized type, where $\tau \neq \omega$, then $\sigma{\rightarrow}\tau$ is a normalized type.

Observe that the only normalized non-tail-proper type is ω.

ii) On the set of types, the relation $\sim_{\text{CDV}}$ is defined by:

a) $\varphi \sim_{\text{CDV}} \varphi$.

b) $\sigma_1{\cap}\cdots{\cap}\sigma_i{\cap}\sigma_{i+1}{\cap}\cdots{\cap}\sigma_n \sim_{\text{CDV}} \sigma_1'{\cap}\cdots{\cap}\sigma_{i+1}'{\cap}\sigma_i'{\cap}\cdots{\cap}\sigma_n' \iff \forall\, 1 \leq i \leq n\ [\, \sigma_i \sim_{\text{CDV}} \sigma_i'\,]$.

c) $\sigma{\rightarrow}\tau \sim_{\text{CDV}} \sigma'{\rightarrow}\tau' \iff \sigma \sim_{\text{CDV}} \sigma'\ \&\ \tau \sim_{\text{CDV}} \tau'$.

Types are considered modulo $\sim_{\text{CDV}}$.

Definition 2.2.1.4 [Coppo *et al.* '81] i) *Restricted CDV-type assignment* and *restricted CDV-derivations* are defined by:

$$(\rightarrow\text{I}):\ \frac{\begin{array}{c}[x{:}\sigma_1]\cdots[x{:}\sigma_n]\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma_1{\cap}\cdots{\cap}\sigma_n{\rightarrow}\tau}\ \text{(a)} \qquad (\rightarrow\text{E}):\ \frac{M{:}\sigma_1{\cap}\cdots{\cap}\sigma_n{\rightarrow}\tau \quad N{:}\sigma_1{\cap}\cdots{\cap}\sigma_n}{MN{:}\tau}$$

$$(\omega):\ \frac{}{M{:}\omega} \qquad ({\cap}\text{I}):\ \frac{M{:}\sigma_1 \quad \cdots \quad M{:}\sigma_n}{M{:}\sigma_1{\cap}\cdots{\cap}\sigma_n}\ \text{(b)}$$

(a) If $\tau \neq \omega$ and $x{:}\sigma_1, \ldots, x{:}\sigma_n$ are all and nothing but the statements about x on which $M{:}\tau$ depends. If $n = 0$, so in the derivation for $M{:}\tau$ there is no premise whose subject is x, then $\sigma_1{\cap}\cdots{\cap}\sigma_n = \omega$.

(b) If for $1 \leq i \leq n$, $\sigma_i \neq \omega$.

ii) If $M{:}\sigma$ is derivable from B using a restricted CDV-derivation, we write $B \vdash_{\text{CDV}_\text{R}} M{:}\sigma$.

It is obvious that $B \vdash_{\text{CDV}_\text{R}} M{:}\sigma$ implies $B \vdash_{\text{CDV}} M{:}\sigma$. The converse does not hold, due to the fact that the derivation rule ($\rightarrow$I) in the unrestricted system allows for the construction of sequences that contain 'redundant' types, which is not possible in the restricted system.

In both systems, types are not invariant by η-expansion, since for example $\vdash \lambda x.x{:}\varphi{\rightarrow}\varphi$, but not $\vdash \lambda xy.xy{:}\varphi{\rightarrow}\varphi$. Moreover, type assignment in the unrestricted system is not invariant under η-reduction: for example

$$\vdash_{\text{CDV}} \lambda xy.xy{:}(\sigma{\rightarrow}\tau){\rightarrow}\sigma{\cap}\rho{\rightarrow}\tau, \text{ but not } \vdash_{\text{CDV}} \lambda x.x{:}(\sigma{\rightarrow}\tau){\rightarrow}\sigma{\cap}\rho{\rightarrow}\tau.$$

This property however holds for the restricted system; properties of this restricted system proved in [Coppo *et al.* '81] are:

- If $B \vdash_{\text{CDV}_\text{R}} M{:}\sigma$, then σ is a normalized type.
- If $B \vdash_{\text{CDV}_\text{R}} M{:}\sigma$ and $M \rightarrow_\eta N$, then $B \vdash_{\text{CDV}_\text{R}} N{:}\sigma$.

2.2.2 *The system of [Coppo et al. '80]*

The type assignment system studied in [Coppo *et al.* '80] is the restricted one from [Coppo *et al.* '81], but with the set of types of the unrestricted system.

Definition 2.2.2.1 We will write $B \vdash_{\text{CDV}_\text{P}} M{:}\sigma$ for statements derivable in the system with the derivation rules from definition 2.2.1.4 and types in $\mathcal{T}_{\text{CDV}}$.

The two main results of that paper are the proof of existence of principal types and that it is possible to define a λ-calculus model the domain of which is the powerset of the set of type schemes. In this model a term is interpreted by the set of types assignable to it.

The principal type property is achieved by defining principal pairs of basis and type for terms in $\lambda\bot$-normal form, specifying the operations of expansion and substitution, proved sufficient to generate all possible pairs for those terms from their principal pair, and generalizing this result to arbitrary lambda terms. This technique is the same as for example used in [Ronchi della Rocca & Venneri '84] (discussed in subsection 2.3.2) and in chapter six of this thesis. We will briefly discuss this construction, using the notations of this thesis.

In [Coppo *et al.* '80] two operations on types were specified, substitution and expansion. The definition of substitution is similar to the one for Curry-substitutions. Expansion is defined as an operation that introduces extra types in a sequence, by replacing a subderivation by more than one subderivation with the same structure.

Definition 2.2.2.2 [Coppo *et al.* '80] i) The *CDV-substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_{\mathrm{CDV}} \rightarrow \mathcal{T}_{\mathrm{CDV}}$, where φ is a type-variable and α is a normalized type, is defined by:

a) $(\varphi \mapsto \alpha)(\varphi) = \alpha$.

b) $(\varphi \mapsto \alpha)(\varphi') = \varphi'$, if $\varphi \not\equiv \varphi'$.

c) $(\varphi \mapsto \alpha)(\sigma \rightarrow \tau) = (\varphi \mapsto \alpha)(\sigma) \rightarrow (\varphi \mapsto \alpha)(\tau)$.

d) $(\varphi \mapsto \alpha)(\sigma_1 \cap \cdots \cap \sigma_n) = (\varphi \mapsto \alpha)(\sigma_1) \cap \cdots \cap (\varphi \mapsto \alpha)(\sigma_n)$.

ii) If S_1 and S_2 are CDV-substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2(\sigma) = S_1(S_2(\sigma))$.

Notice that although α is normalized, if $\sigma \rightarrow \tau$ is normalized then the type $(\varphi \mapsto \alpha)(\sigma \rightarrow \tau)$ need not be normalized, since for example α can be ω and τ the type-variable φ.

CDV-substitution is, like Curry-substitution, extended to bases in a natural way and it is a sound operation on pairs:

Property 2.2.2.3 [Coppo *et al.* '80] If $B \vdash_{\mathrm{CDV_P}} M{:}\sigma$, then for every CDV-substitution S: $S(B) \vdash_{\mathrm{CDV_P}} M{:}S(\sigma)$. $\Box$

As mentioned in the introduction of [Coppo *et al.* '80]:

> *" (...) different types can be assigned to the same component of a given term. Then the structure of deductions does not simply correspond, as in Curry's theory, to the structure of terms but it is more 'ramified' and there is no limit to its complexity. With the only operation of substitution we cannot obtain, for a given term, all types whose deduction are more complex than the one of the type on which we are doing the substitution. (...) (*This*) difficulty (...) can be overcome if we introduce, besides substitution, the new (context-dependent) operation of expansion. "*

The definition of expansion is very complicated. It is an operation on types that deals with the replacement of (sub)types by a number of copies of that type. An expansion indicates not only the type to be expanded, but also the number of copies that has to be generated. It is a complex operation, possibly affecting more types than just the one to be expanded occurs in.

To clarify this, consider the following: suppose that μ is a subtype of σ that is to be expanded into n copies. If $\tau \rightarrow \mu$ is also a subtype of σ, then there are several ways to create the expansion of $\tau \rightarrow \mu$: just replacing μ by an intersection of copies of μ would generate $\tau \rightarrow \mu_1 \cap \cdots \cap \mu_n$, which is not a type in $\mathcal{T}_{\mathrm{CDV}}$. It could be replaced by $(\tau \rightarrow \mu_1) \cap \cdots \cap (\tau \rightarrow \mu_n)$. But this last approach is not sufficient: it would not be closed for ground pairs (definition 2.2.2.8), a property that is needed in the proof that the specified operations are complete.

The subtype $\tau \rightarrow \mu$ will therefore be expanded into $(\tau_1 \rightarrow \mu_1) \cap \cdots \cap (\tau_n \rightarrow \mu_n)$, where the $\tau_1, \ldots, \tau_n$ are copies of τ. Then τ is *affected* by the expansion of μ; all other occurrences of τ should be expanded into $\tau_1 \cap \cdots \cap \tau_n$, with possibly the same effect on other types.

Before we give the definition of expansion, the notion of nucleus of a multiset of type schemes is defined. A multiset is a like a set, but with possibly multiple occurrences of elements. A nucleus is defined on a collection of types, and is the set of all types that are affected by the expansion.

Definition 2.2.2.4 [Coppo *et al.* '80] Let C = $\{\tau_1, \ldots, \tau_n\}$ be a multiset of type schemes. Let $\rho_1, \ldots, \rho_m$ be *occurrences* of subtypes of $\tau_1, \ldots, \tau_n$ and let E denote the *multiset* $\{\rho_1, \ldots, \rho_m\}$. E is a *nucleus* of C if and only if:

i) $\rho_i \not\equiv \omega$ for $1 \leq i \leq m$.

ii) for all $1 \leq i \neq j \leq m$ the occurrence ρ_i is not a part of the occurrence ρ_j in C.

iii) all type-variables that occur in $\rho_1, \ldots, \rho_m$ do not occur in C outside $\rho_1, \ldots, \rho_m$.

iv) for all $1 \leq i \leq m$, either $\rho_i \equiv \tau_j$ or ρ_i is contained in some sequence which occurs in τ_j, for some $1 \leq j \leq n$.

These notions of multiset and nucleus are then used to define expansions.

Definition 2.2.2.5 [Coppo *et al.* '80] i) A multiset of type schemes C′ is an *immediate expansion* of a multiset C if and only if there are a nucleus E of C and integer $n \geq 1$ such that C′ can be obtained from C in the following way:

a) Suppose $\{\varphi_1, \ldots, \varphi_m\}$ is the set of all type-variables that occur in E. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i does not occur in C, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Let S_i be the substitution that replaces every φ_j by φ_j^i.

b) For each occurrence of $\tau \in$ E, replace τ in C by $S_1(\tau) \cap \cdots \cap S_n(\tau)$.

ii) The multiset C′ is a *CDV-expansion* of another multiset C if and only if we can obtain C′ from C by means of a finite number (possible none) of successive immediate expansions.

Notice that if C $(<B, \sigma>) = <B', \sigma'>$ and B, σ are normalized, then B', σ' are normalized too.

As remarked in the introduction of [Coppo *et al.* '80]:

> *" (...) in our theory also terms without normal form or, in particular, terms that have an 'infinite' normal form (...) have types. (...) the set of all functional characters of these terms must carry an infinite amount of information and it cannot be represented in a finite way. (...) we shall consider the existence of principal type schemes for arbitrary terms. We shall do this through the notion of approximant of a term. (...) It will turn out that terms with a finite number of approximants have finite principal type schemes while terms with a infinite number of approximants have 'infinite' principal type schemes. "*

The notion of approximant is defined using the notion of terms in $\lambda\bot$-normal form (like in [Barendregt '84], we use the symbol $\bot$ instead of Ω).

Definition 2.2.2.6 [Barendregt '84] i) The set of $\Lambda\bot$-*terms* is defined as the set Λ of lambda terms, extended by: $\bot \in \Lambda\bot$.

ii) The notion of reduction $\rightarrow_{\beta\bot}$ is defined as $\rightarrow_{\beta}$, extended by:

a) $\lambda x.\bot \rightarrow_{\beta\bot} \bot$.

b) $\bot M \rightarrow_{\beta\bot} \bot$.

iii) The set of *normal forms for elements of* $\Lambda\bot$ *with respect to* $\rightarrow_{\beta\bot}$ is the set $\mathcal{N}$ of $\lambda\bot$-*normal forms* or *approximate normal forms* and is inductively defined by:

a) All term-variables are elements of $\mathcal{N}$, $\bot \in \mathcal{N}$.
b) If $A \in \mathcal{N}$, $A \not\equiv \bot$, then $\lambda x.A \in \mathcal{N}$.
c) If $A_1, \ldots, A_n \in \mathcal{N}$, then $xA_1 \ldots A_n \in \mathcal{N}$.

iv) $A \in \mathcal{N}$ is a *direct approximant* of $M \in \Lambda$, if A matches M except for occurrences of $\bot$.

v) $A \in \mathcal{N}$ is an *approximant* of $M \in \Lambda$ (notation: $A \sqsubseteq M$) if there is an $M' =_\beta M$ such that A is a direct approximant of M'.

vi) $\mathcal{A}(M) = \{A \in \mathcal{N} \mid A \sqsubseteq M\}$.

The type assignment rules are generalized by allowing for the terms to be elements of $\Lambda\bot$.

Definition 2.2.2.7 [Coppo *et al.* '80] A set of pairs *V* is called *complete* for a term M if and only if for all pairs $<B, \sigma>$: $B \vdash_{\mathrm{CDV_P}} M{:}\sigma$ if and only if there is a pair $<B', \sigma'> \in V$ and an operation *O* such that $O(<B', \sigma'>) = <B, \sigma>$.

Then for $A \in \mathcal{N}$ the notion of ground pairs for A is introduced. The set of ground pairs for a term $A \in \mathcal{N}$ is proved to be complete for A. Ground pairs are those that express the essential structure of a derivation, and types in it are as general as possible with respect to CDV-substitutions. Ground pairs are defined as follows:

Definition 2.2.2.8 [Coppo *et al.* '80] The pair $<B, \sigma_1\cap\cdots\cap\sigma_n>$ is a *CDV-ground pair* for $A \in \mathcal{N}$ if and only if:

i) If $n > 1$, then $\sigma_i \not\equiv \omega$ for $1 \leq i \leq n$, there are $B_1, \ldots, B_n$ such that $B = B_1\cup\cdots\cup B_n$, the $<B_i, \sigma_i>$ are disjoint in pairs and for $1 \leq i \leq n$ $<B_i, \sigma_i>$ is a CDV-ground pair for A.

ii) If $n = 1$, then:
a) If $\sigma = \omega$, then $B = \emptyset$ and $A \equiv \bot$.
b) If $A \equiv x$, then $B = \{x{:}\varphi\}$, and $\sigma = \varphi$.
c) If $A \equiv \lambda x.A'$, then:
1) If $x \in \mathrm{FV}(A')$, then $\sigma = \alpha\rightarrow\beta$, and $<B \cup \{x{:}\alpha\}, \beta>$ is a CDV-ground pair for A'.
2) If $x \notin \mathrm{FV}(A')$, then $\sigma = \omega\rightarrow\beta$, and $<B, \beta>$ is a CDV-ground pair for A'.
d) If $A \equiv xA_1 \ldots A_m$, then $\sigma = \varphi$, and there are $B_1, \ldots, B_m, \tau_1, \ldots, \tau_m$ such that $B = B_1\cup\cdots\cup B_m \cup \{x{:}\tau_1\rightarrow\cdots\rightarrow\tau_m\rightarrow\varphi\}$, the $<B_j, \tau_j>$ are disjoint in pairs, and for every $1 \leq j \leq m$, φ does not occur in $<B_j, \tau_j>$, which is a CDV-ground pair for A_j.

The proof of the principal type property is obtained by first proving the following (proofs in [Coppo *et al.* '80] are obscured by the fact that, between steps, types have to be normalized; we will ignore these details here):

Property 2.2.2.9 [Coppo *et al.* '80] i) If $B \vdash_{\mathrm{CDV_P}} A{:}\sigma$ with $A \in \mathcal{N}$, then there is a substitution *S* and a CDV-ground pair $<B', \sigma'>$ for A such that $S(<B, \sigma>) = <B', \sigma'>$.

ii) If $<B, \sigma>$ is a CDV-ground pair for $A \in \mathcal{N}$, and $<B', \sigma'>$ can be obtained from $<B, \sigma>$ by an immediate expansion, then $<B', \sigma'>$ is a CDV-ground pair for A. So expansion is closed on CDV-ground pairs.

iii) For all $A \in \mathcal{N}$: every CDV-ground pair for A is complete for A. □

In the construction of principal pairs for lambda terms, first for every $A \in \mathcal{N}$ a particular pair $<P, \pi>$ is chosen of basis P and type π, called the *principal basis scheme* and *principal type scheme* of A respectively. This pair will be called the *principal pair* of A.

Definition 2.2.2.10 [Coppo *et al.* '80] i) Let $A \in \mathcal{N}$. *PP*$_{\mathrm{CDV}}$ (A), the *CDV-principal pair* of A, is defined by:

a) *PP*$_{\mathrm{CDV}}$ $(\bot) = <\emptyset, \omega>$.
b) *PP*$_{\mathrm{CDV}}$ $(x) = <\{x{:}\varphi\}, \varphi>$.
c) If $A \not\equiv \bot$, and *PP*$_{\mathrm{CDV}}$ $(A) = <P, \pi>$, then:
1) If x occurs free in A, and $x{:}\sigma \in P$, then *PP*$_{\mathrm{CDV}}$ $(\lambda x.A) = <P\backslash x, \sigma\rightarrow\pi>$.
2) otherwise *PP*$_{\mathrm{CDV}}$ $(\lambda x.A) = <P, \omega\rightarrow\pi>$.
d) If *PP*$_{\mathrm{CDV}}$ $(A_i) = <P_i, \pi_i>$ for $1 \leq i \leq n$ (we choose trivial variants that are disjoint in pairs), then *PP*$_{\mathrm{CDV}}$ $(xA_1 \ldots A_n) = <P_1\cup\cdots\cup P_n \cup \{x{:}\pi_1\rightarrow\cdots\rightarrow\pi_n\rightarrow\varphi\}, \varphi>$, where φ is a type-variable that does not occur in *PP*$_{\mathrm{CDV}}$ (A_i) for $1 \leq i \leq n$.

ii) For all terms M define $\Pi_{\mathrm{CDV}}(M) = \{$ *PP*$_{\mathrm{CDV}}$ $(A) \mid A \in \mathcal{A}(M)$ $\}$, the set of all principal pairs for all approximants of M.

Principal pairs are not completely well defined, since the type-variables mentioned are not unique. However, types that only differ in the names of type-variables can be considered identical.

The proof of the principal type property is completed by proving:

- The principal pair for A is a CDV-ground pair for A.
- $B \vdash_{\mathrm{CDV_P}} M{:}\sigma$ if and only if there exists $A \in \mathcal{A}(M)$ such that $B \vdash_{\mathrm{CDV_P}} A{:}\sigma$.
- $\Pi_{\mathrm{CDV}}(M)$ is complete for M.

2.3 *The Barendregt-Coppo-Dezani Type Assignment System*

The type assignment system presented in [Barendregt *et al.* '83] by H. Barendregt, M. Coppo and M. Dezani-Ciancaglini is based on the system as presented in [Coppo *et al.* '81]. This system was strengthened further by extending the set of types to $\mathcal{T}_{\mathrm{BCD}}$ and introducing a partial order relation '$\leq$' on types, as well as adding the type assignment rule $(\leq)$, and a more general form of the rules concerning intersection. The rule $(\leq)$ is mainly introduced to prove completeness of type assignment.

In this paper, it was shown that the set of types derivable for a lambda term in the extended system is a filter, i.e. a set closed under intersection and right-closed for $\leq$ (if $\sigma \leq \tau$ and $\sigma \in d$ where d is a filter, then $\tau \in d$.) The interpretation of a lambda term by the set of types derivable for it – $[\![M]\!]_\xi$ – is defined in the standard way, and gives a filter lambda model $\mathcal{F}$. The main result of that paper is that, using this model, completeness is proved by proving the statement: $\vdash_{\mathrm{BCD}} M{:}\sigma \iff [\![M]\!] \in \upsilon(\sigma)$, where $\upsilon : \mathcal{T}_{\mathrm{BCD}} \rightarrow \mathcal{F}$ is a simple type interpretation as defined in [Hindley '83]. In order to prove the $\Leftarrow$-part of this statement (completeness), the relation $\leq$ is needed. Other interesting use of filter lambda models can be found in [Coppo *et al.* '84], [Coppo *et al.* '87], [Dezani-Ciancaglini & Margaria '84], and [Dezani-Ciancaglini & Margaria '86].

In this subsection we give the definition of the Intersection Type Discipline as presented in [Barendregt *et al.* '83], together with its major features.

Definition 2.3.1 [Barendregt *et al.* '83] i) $\mathcal{T}_{\mathrm{BCD}}$, the set of *BCD-types* is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{\mathrm{BCD}}$.

b) $\omega \in \mathcal{T}_{\mathrm{BCD}}$.

c) If σ and $\tau \in \mathcal{T}_{\mathrm{BCD}}$, then $\sigma{\to}\tau$ and $\sigma{\cap}\tau \in \mathcal{T}_{\mathrm{BCD}}$.

ii) On $\mathcal{T}_{\mathrm{BCD}}$ the type inclusion relation $\leq$ is inductively defined by:

a) $\sigma \leq \sigma$.

b) $\sigma \leq \omega$.

c) $\omega \leq \omega{\to}\omega$.

d) $\sigma{\cap}\tau \leq \sigma$.

e) $\sigma{\cap}\tau \leq \tau$.

f) $(\sigma{\to}\tau){\cap}(\sigma{\to}\rho) \leq \sigma{\to}\tau{\cap}\rho$.

g) $\sigma \leq \tau \leq \rho \;\Rightarrow\; \sigma \leq \rho$.

h) $\sigma \leq \tau \;\&\; \sigma \leq \rho \;\Rightarrow\; \sigma \leq \tau{\cap}\rho$.

i) $\rho \leq \sigma \;\&\; \tau \leq \mu \;\Rightarrow\; \sigma{\to}\tau \leq \rho{\to}\mu$.

iii) $\sigma \sim \tau \;\Leftrightarrow\; \sigma \leq \tau \leq \sigma$.

iv) A *BCD-statement* is an expression of the form $M{:}\sigma$ where $M \in \Lambda$ and $\sigma \in \mathcal{T}_{\mathrm{BCD}}$. M is the *subject* and σ is the *predicate* of $M{:}\sigma$.

v) A *BCD-basis* is a set of statements with only distinct variables as subjects.

$\mathcal{T}_{\mathrm{BCD}}$ may be considered modulo $\sim$. Then $\leq$ becomes a partial order.

Notice that in the original paper [Barendregt *et al.* '83] the type inclusion relation was defined in a slightly different way. Instead of rule 2.3.1 (ii.h) the rules

h.1) $\sigma \leq \tau \;\&\; \mu \leq \rho \;\Rightarrow\; \sigma{\cap}\mu \leq \tau{\cap}\rho$.

h.2) $\sigma \leq \sigma{\cap}\sigma$.

are given. It is not difficult to show that these definitions are equivalent.

Notice, moreover, that in [Barendregt *et al.* '83] the subjects of statements in a basis need not be distinct. However, the presence of intersections makes these two approaches similar; for reasons of clarity we present bases as maps from term-variables to types.

Definition 2.3.2 i) The relation $\leq$ is extended to bases by: $B \leq B'$ if and only if for every $x{:}\sigma' \in B'$ there is an $x{:}\sigma \in B$ such that $\sigma \leq \sigma'$.

ii) $B \sim B' \;\Leftrightarrow\; B \leq B' \leq B$.

Definition 2.3.3 [Barendregt *et al.* '83] i) *BCD-type assignment* and *BCD-derivations* are defined by the following natural deduction system.

$$(\to\mathrm{I}):\ \frac{\begin{array}{c}[x{:}\sigma]\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma{\to}\tau}\ \text{(a)} \qquad (\to\mathrm{E}):\ \frac{M{:}\sigma{\to}\tau \qquad N{:}\sigma}{MN{:}\tau}$$

$$(\cap\mathrm{I}):\ \frac{M{:}\sigma \qquad M{:}\tau}{M{:}\sigma{\cap}\tau} \qquad (\cap\mathrm{E}):\ \frac{M{:}\sigma{\cap}\tau}{M{:}\sigma} \quad \frac{M{:}\sigma{\cap}\tau}{M{:}\tau}$$

$$(\leq):\ \frac{M{:}\sigma \qquad \sigma \leq \tau}{M{:}\tau} \qquad (\omega):\ \frac{}{M{:}\omega}$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

ii) If $M{:}\sigma$ is derivable from a basis B using a BCD-derivation, we write $B \vdash_{\mathrm{BCD}} M{:}\sigma$.

In the BCD-system there are several ways to deduce a desired result, due to the presence of the derivation rules ($\cap$I), ($\cap$E) and ($\leq$), which allow superfluous steps in derivations. In the CDV-system these rules are not present and there is a one-one relationship between terms and skeletons of derivations. In other words: that system is syntax directed. The BCD-type discipline has the same expressive power as the previous unrestricted CDV-system: all solvable terms have types other than ω, and a term has a normal form if and only if it has a type without ω occurrences.

- The set of normalizable terms can be characterized in the following way:

  $\exists\, B, \sigma\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma \ \&\ B, \sigma\ \omega\text{-free}\ ] \;\Leftrightarrow\; M$ has a normal form.

- The set of terms having a head normal form can be characterized in the following way:

  $\exists\, B, \sigma\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma \ \&\ \sigma \neq \omega\ ] \;\Leftrightarrow\; M$ has a head normal form.

The following properties of the BCD-system, used in this thesis, are listed here so as to refer to them easily:

Property 2.3.4 i) [Barendregt *et al.* '83].2.8 (i): $B \vdash_{\mathrm{BCD}} MN{:}\tau \;\Rightarrow$ $\exists\, \sigma \in \mathcal{T}_{\mathrm{BCD}}\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma{\to}\tau \ \&\ B \vdash_{\mathrm{BCD}} N{:}\sigma\ ]$.

ii) [Barendregt *et al.* '83].2.8 (iii): $B \vdash_{\mathrm{BCD}} \lambda x.M{:}\sigma{\to}\tau \;\Leftrightarrow\; B\backslash x \cup \{x{:}\sigma\} \vdash_{\mathrm{BCD}} M{:}\tau$.

iii) [Barendregt *et al.* '83].4.13 (i): $\exists\, B, \sigma\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma \ \&\ \sigma \neq \omega\ ] \;\Leftrightarrow$ M has a head normal form.

iv) [Barendregt *et al.* '83].4.13 (ii): $\exists\, B, \sigma\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma \ \&\ B, \sigma\ \omega\text{-free}\ ] \;\Leftrightarrow$ M has a normal form.

v) [Barendregt *et al.* '83].2.7 (ii): $B \vdash_{\mathrm{BCD}} x{:}\tau \;\Rightarrow\; \exists\, x{:}\sigma \in B\ [\ \sigma \leq \tau\ ]$.

vi) [Dezani-Ciancaglini & Margaria '86].5.6: $\rho \leq \tau_1{\cap}\cdots{\cap}\tau_n{\to}\sigma \;\Rightarrow$ $\rho = (\tau_1^1{\to}\cdots{\to}\tau_n^s{\to}\sigma_1){\cap}\cdots{\cap}(\tau_1^s{\to}\cdots{\to}\tau_n^s{\to}\sigma_s)\,{\cap}\rho'$, for some $\tau_1^j, \ldots, \tau_n^j, \sigma_j, \rho'$ such that $\tau_i^j \geq \tau_i$ with $1 \leq i \leq n$, $1 \leq j \leq s$ and $\sigma_1{\cap}\cdots{\cap}\sigma_s \leq \sigma$. $\square$

2.3.1 *Completeness of type assignment*

The main result of [Barendregt *et al.* '83] is the proof for completeness of type assignment. This is achieved by showing that the set of types derivable for a lambda term is a filter, i.e. a set closed under intersection and right closed for $\leq$. The construction of a filter lambda model and the definition of a map from types to elements of this model (a *simple type interpretation*) make the proof of completeness possible: if the interpretation of the term M is an element of the interpretation of the type σ, then M is typeable with σ.

Filters and the filter λ-model $\mathcal{F}$ are defined by:

Definition 2.3.1.1 [Barendregt *et al.* '83] i) A *BCD-filter* is a subset $d \subseteq \mathcal{T}_{\mathrm{BCD}}$ such that:

a) $\omega \in d$.

b) $\sigma, \tau \in d \Rightarrow \sigma \cap \tau \in d$.

c) $\sigma \geq \tau \in d \Rightarrow \sigma \in d$

ii) $\mathcal{F} = \{ d \mid d$ is a BCD-filter $\}$.

iii) For $d_1, d_2 \in \mathcal{F}$ define $d_1 \cdot d_2 = \{ \tau \in \mathcal{T}_{\mathrm{BCD}} \mid \exists\, \sigma \in d_2\ [\ \sigma \to \tau \in d_1\ ]\ \}$.

The following properties are proved in [Barendregt *et al.* '83]:

- $\forall\, M \in \Lambda\ [\ \{ \sigma \mid \exists\, B\ [\ B \vdash_{\mathrm{BCD}} M{:}\sigma\ ]\ \} \in \mathcal{F}\ ]$.
- Let ξ be a valuation of term-variables in $\mathcal{F}$, and $B_\xi = \{ x{:}\sigma \mid \sigma \in \xi\,(x) \}$.
  For $M \in \Lambda$ define $[\![M]\!]_\xi = \{ \sigma \mid B_\xi \vdash_{\mathrm{BCD}} M{:}\sigma \}$. Using the method of Hindley and Longo [Hindley & Longo '80] it is shown that $<\mathcal{F}, \cdot, [\![\ ]\!]>$ is a λ-model.

The following two definitions were absent [Barendregt *et al.* '83]. They are presented here so as to compare the construction of the completeness proof in [Barendregt *et al.* '83] with that of chapter four.

In constructing a complete system, the semantics of types play a crucial role. As in [Dezani-Ciancaglini & Margaria '86], [Mitchell '88] and essentially following [Hindley '82], a distinction can be made between several notions of type interpretations and semantic satisfiability. There are, roughly, three notions of type semantics that differ in the meaning of an arrow type scheme: inference type interpretations, simple type interpretations and *F* type interpretations. These different notions of type interpretations induce of course different notions of semantic satisfiability.

Following essentially [Mitchell '88], we distinguish several kinds of type interpretations.

Definition 2.3.1.2 i) Let $<\mathcal{D}, \cdot, \varepsilon>$ be a continuous lambda model.
A mapping υ: $\mathcal{T}_{\mathrm{BCD}} \to \wp(\mathcal{D}) = \{X \mid X \subseteq \mathcal{D}\}$ is a *type interpretation* if and only if:

a) $\{ \varepsilon \cdot d \mid \forall\, e\ [\ e \in \upsilon(\sigma) \Rightarrow d \cdot e \in \upsilon(\tau)\ ]\ \} \subseteq \upsilon(\sigma \to \tau)$.

b) $\upsilon(\sigma \to \tau) \subseteq \{ d \mid \forall\, e\ [\ e \in \upsilon(\sigma) \Rightarrow d \cdot e \in \upsilon(\tau)\ ]\ \}$.

c) $\upsilon(\sigma \cap \tau) = \upsilon(\sigma) \cap \upsilon(\tau)$.

ii) Following [Hindley '83] we say that a type interpretation is *simple* if and only if:
$\upsilon(\sigma \to \tau) = \{ d \mid \forall\, e\ [\ e \in \upsilon(\sigma) \Rightarrow d \cdot e \in \upsilon(\tau)\ ]\ \}$.

iii) On the other hand, a type interpretation is called an *F type interpretation* if it satisfies:
$\upsilon(\sigma \to \tau) = \{ \varepsilon \cdot d \mid \forall\, e\ [\ e \in \upsilon(\sigma) \Rightarrow d \cdot e \in \upsilon(\tau)\ ]\ \}$.

Notice that, in part (ii), the containment relation $\subseteq$ of part (i.b) is replaced by =, and that in part (iii) the same is done with regard to part (i.a).

These notions of type interpretation lead, naturally, to the following definitions for semantic satisfiability (called *inference-*, *simple-* and *F-semantics*, respectively).

Definition 2.3.1.3 i) Let $\mathcal{M} = <\mathcal{D}, \cdot, [\![\ ]\!]>$ be a λ-model, and ξ a valuation.
Then $[\![M]\!]^{\mathcal{M}}_\xi \in \mathcal{D}$ is the interpretation of M in $\mathcal{M}$ via ξ.

ii) We define $\models$ by: (where $\mathcal{M}$ is a lambda model, ξ a valuation and υ a type interpretation)

a) $\mathcal{M}, \xi, \upsilon \models M{:}\sigma \Leftrightarrow [\![M]\!]^{\mathcal{M}}_\xi \in \upsilon(\sigma)$.

b) $\mathcal{M}, \xi, \upsilon \models B \Leftrightarrow \mathcal{M}, \xi, \upsilon \models x{:}\sigma$ for every $x{:}\sigma \in B$.

c) 1) $B \models M{:}\sigma \Leftrightarrow \forall\, \mathcal{M}, \xi, \upsilon\ [\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

2) $B \models_{\mathrm{s}} M{:}\sigma \Leftrightarrow$
$\forall\, \mathcal{M}, \xi$, simple type interpretations $\upsilon\ [\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

3) $B \models_{\mathrm{F}} M{:}\sigma \Leftrightarrow$
$\forall\, \mathcal{M}, \xi$, *F* type interpretations $\upsilon\ [\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

If no confusion is possible, we will omit the superscript on $[\![\ ]\!]$.

The method followed in [Barendregt *et al.* '83] is to define a simple type interpretation υ and to use it for the proof of completeness.

Definition 2.3.1.4 (cf. [Barendregt *et al.* '83]) The type interpretation $\upsilon : \mathcal{T}_{\mathrm{BCD}} \to \wp(\mathcal{F})$ is defined as follows:

i) $\upsilon\,(\omega) = \mathcal{F}$.

ii) $\upsilon\,(\varphi) = \{ d \in \mathcal{F} \mid \varphi \in d \}$.

iii) $\upsilon\,(\sigma \to \tau) = \{ d \in \mathcal{F} \mid \forall\, e \in \upsilon\,(\sigma)\ [\ d \cdot e \in \upsilon\,(\tau)\ ]\ \}$.

iv) $\upsilon\,(\sigma \cap \tau) = \upsilon\,(\sigma) \cap \upsilon\,(\tau)$.

Notice that because of part (iii), υ is a simple type interpretation. For υ, the following properties are proved:

- For all types σ: $\upsilon\,(\sigma) = \{ d \in \mathcal{F} \mid \sigma \in d \}$.
- If $[\![M]\!]_{\xi_B} \in \upsilon\,(\sigma)$, then $\sigma \in [\![M]\!]_{\xi_B}$, where $\xi_B\,(x) = \{ \sigma \in \mathcal{T}_{\mathrm{BCD}} \mid B \vdash_{\mathrm{BCD}} x{:}\sigma \}$.

The main result of [Barendregt *et al.* '83] is obtained by proving, using the following properties:

Property 2.3.1.5 [Barendregt *et al.* '83] i) *Soundness*. $B \vdash_{\mathrm{BCD}} M{:}\sigma \Rightarrow B \models_{\mathrm{s}} M{:}\sigma$.

ii) *Completeness*. $B \models_{\mathrm{s}} M{:}\sigma \Rightarrow B \vdash_{\mathrm{BCD}} M{:}\sigma$. $\square$

The proof of completeness is obtained in a way very similar to the one of theorem 4.4.5. Since the type interpretation υ is simple, the results of [Barendregt *et al.* '83] in fact show that type assignment in the BCD-system is complete with respect to simple type semantics.

2.3.2 *Principal type schemes*

For the system as defined in [Barendregt *et al.* '83], principal type schemes can be defined as done by S. Ronchi della Rocca and B. Venneri in [Ronchi della Rocca & Venneri '84]. In this paper, three operations are provided: substitution, expansion, and rise. These are sound and sufficient to generate all suitable pairs for a term M from its principal pair. This result is achieved in a way similar to that of [Coppo *et al.* '80], as discussed in subsection 2.2.2. (Like in the CDV-system the type assignment rules of the BCD-system are generalized by allowing for the terms to be elements of $\Lambda\bot$.)

We will briefly discuss the construction of [Ronchi della Rocca & Venneri '84]. In this paper, all constructions and definitions are made modulo the equivalence relation $\sim$.

The first operation defined is substitution; it is a natural extension of Curry-substitution and CDV-substitution.

Definition 2.3.2.1 [Ronchi della Rocca & Venneri '84] i) The *RV-substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_{\text{BCD}} \to \mathcal{T}_{\text{BCD}}$, where φ is a type-variable and $\alpha \in \mathcal{T}_{\text{BCD}}$, is defined by:

a) $(\varphi \mapsto \alpha)(\varphi) \sim \alpha$.

b) $(\varphi \mapsto \alpha)(\varphi') \sim \varphi'$, if $\varphi \not\equiv \varphi'$.

c) $(\varphi \mapsto \alpha)(\sigma\to\tau) \sim (\varphi \mapsto \alpha)(\sigma) \to (\varphi \mapsto \alpha)(\tau)$.

d) $(\varphi \mapsto \alpha)(\sigma_1\cap\cdots\cap\sigma_n) \sim (\varphi \mapsto \alpha)(\sigma_1)\cap\cdots\cap(\varphi \mapsto \alpha)(\sigma_n)$.

ii) If S_1 and S_2 are RV-substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2(\sigma) = S_1(S_2(\sigma))$.

Notice that a RV-substitution is defined without restriction: the type α that is to be substituted can be every element of $\mathcal{T}_{\text{BCD}}$.

Next, the operation of expansion is defined, which is a generalization of the CDV-expansion.

Definition 2.3.2.2 [Ronchi della Rocca & Venneri '84] For every $\mu \in \mathcal{T}_{\text{BCD}}$, $n \geq 2$, the pair $<\mu, n>$ determines an *RV-expansion* $E_{<\mu,n>}$ that is constructed as follows:

i) Let $\mathcal{L}^e(B,\tau)$ be the set of type schemes defined as follows:

a) $\mu \in \mathcal{L}^e(B,\tau)$.

b) If $\sigma \in \mathcal{L}^e(B,\tau)$, any proper subtype of σ belongs to $\mathcal{L}^e(B,\tau)$.

c) For each type scheme σ, such that σ is a subtype of either τ or a predicate in B:

1) If either $\sigma \sim \alpha\to\beta$ or $\sigma \sim \alpha\to\beta\cap\gamma$ and $\beta \in \mathcal{L}^e(B,\tau)$, then $\sigma \in \mathcal{L}^e(B,\tau)$.

2) If $\sigma \sim \alpha\cap\beta$ and $\alpha, \beta \in \mathcal{L}^e(B,\tau)$, then $\sigma \in \mathcal{L}^e(B,\tau)$.

ii) Suppose $L^e(B,\tau)$ is the list obtained by ordering the elements of $\mathcal{L}^e(B,\tau)$ in decreasing order according to the number of symbols (if two type schemes have the same number of symbols, their mutual order is unimportant).

iii) Let $V = \{\varphi_1, \ldots, \varphi_m\}$ be all the variables occurring in $L^e(B,\tau)$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i does not occur in $<B,\tau>$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Let S_i be the substitution that replaces every φ_j by φ_j^i.

iv) For every $\sigma \in \mathcal{T}_{\text{BCD}}$, $E_{<\mu,n>}(\sigma)$ is obtained from σ by examining in order each element of $L^e(B,\tau)$, and each time an element α is found that is a subtype of σ, by replacing α, in σ, by $S_1(\alpha)\cap\cdots\cap S_n(\alpha)$.

The set $\mathcal{L}^e(B,\tau)$ is the equivalent of the nucleus in definition 2.2.2.4.

Substitution and expansions are in the natural way extended to operations on bases and pairs. The third operation defined is the operation of rise: it consists of adding applications of the derivation rule $(\leq)$ to a derivation.

Definition 2.3.2.3 [Ronchi della Rocca & Venneri '84] A *rise R* is an operation denoted by a pair of pairs $<<B_0, \tau_0>, <B_1, \tau_1>>$ such that $\tau_0 \leq \tau_1$ and $B_1 \leq B_0$, and is defined by:

i) a) $R(\sigma) \sim \tau_1$, if $\sigma \sim \tau_0$.

b) $R(\sigma) \sim \sigma$, otherwise.

ii) a) $R(B) \sim B_1$, if $B \sim B_0$.

b) $R(B) \sim B$, otherwise.

iii) $R(<B,\sigma>) \sim <R(B), R(\sigma)>$.

The following property is proved; it shows that all defined operations are sound:

Property 2.3.2.4 [Ronchi della Rocca & Venneri '84] Let $A \in \mathcal{N}$, $<B,\sigma>$ be such that $B \vdash_{\text{BCD}} A{:}\sigma$, and let O be an operation of substitution, expansion or rise.
Then $O(B) \vdash_{\text{BCD}} A{:}O(\sigma)$. □

As in [Coppo *et al.* '80], principal types are defined for terms in $\lambda\bot$-normal form.

Definition 2.3.2.5 [Ronchi della Rocca & Venneri '84] i) Let $A \in \mathcal{N}$. $PP_{\text{RV}}(A)$, the *RV-principal pair* of A, is defined by:

a) $PP_{\text{RV}}(\bot) \sim <\emptyset, \omega>$.

b) $PP_{\text{RV}}(x) \sim <\{x{:}\varphi\}, \varphi>$.

c) If $A \not\equiv \bot$, and $PP_{\text{RV}}(A) \sim <P, \pi>$, then:

1) If x occurs free in A, and $x{:}\sigma \in P$, then $PP_{\text{RV}}(\lambda x.A) \sim <P\backslash x, \sigma\to\pi>$.

2) otherwise $PP_{\text{RV}}(\lambda x.A) \sim <P, \omega\to\pi>$.

d) If $PP_{\text{RV}}(A_i) \sim <P_i, \pi_i>$ for $1 \leq i \leq n$ (we choose trivial variants that are disjoint in pairs), then $PP_{\text{RV}}(xA_1 \ldots A_n) \sim <P_1\cup\cdots\cup P_n\cup\{x{:}\pi_1\to\cdots\to\pi_n\to\varphi\}, \varphi>$, where φ is a type-variable that does not occur in $PP_{\text{RV}}(A_i)$ for $1 \leq i \leq n$.

ii) $\mathcal{P}_{\text{RV}} = \{<P,\pi> \mid \exists\, A \in \mathcal{N}\ [\ PP_{\text{RV}}(A) \sim <P,\pi>\ ]\}$.

Notice that this definition is almost the same as definition 2.2.2.10 (apart from the notion of sequence and the relation $\sim$).

Definition 2.3.2.6 (cf. [Ronchi della Rocca & Venneri '84]) i) *Linear chains* of operations are defined as sequences of operations that start with a number of expansions, followed by a number of substitutions, and that end with *one* rise.

ii) Let M be a term. As in [Coppo *et al.* '80] let $\Pi_{\text{RV}}(M)$ be the set of all RV-principal pairs for all approximants of M: $\Pi_{\text{RV}}(M) = \{\ PP_{\text{RV}}(A) \mid A \in \mathcal{A}(M)\ \}$.

iii) On $\mathcal{P}_{\text{RV}}$ is it possible to to define the preorder relation $\sqsubseteq_\omega$ by:

$$<P, \pi> \sqsubseteq_\omega <P', \pi'> \Leftrightarrow \exists\, \varphi_1, \ldots, \varphi_n \, [\, <P, \pi> = (\varphi_1 := \omega)\circ\cdots\circ(\varphi_n := \omega)\,(<P', \pi'>)\,],$$

and $\mathcal{P}_{\text{RV}}$, $\sqsubseteq_\omega$ is a meet semilattice isomorphic to $\mathcal{N}$, $\leq$.

iv) $\Pi_{\text{RV}}(M)$ is an ideal in $\mathcal{P}_{\text{RV}}$ and therefore:

a) If $\Pi_{\text{RV}}(M)$ is finite, there exists a pair $<P, \pi> = \sqcup\Pi_{\text{RV}}(M)$, where $<P, \pi> \in \mathcal{P}_{\text{RV}}$. This pair is then called the principal pair of M.

b) If $\Pi_{\text{RV}}(M)$ is infinite, $\sqcup\Pi_{\text{RV}}(M)$ does not exist in $\mathcal{P}_{\text{RV}}$. The principal pair of M is then the infinite set of pairs $\Pi_{\text{RV}}(M)$.

In [Ronchi della Rocca & Venneri '84], linear chains are defined as those sequences of operations that start with a number of expansions, followed by a number of substitutions or rises. Both are allowed. This definition is not complete: that the chain ends with one rise is essential in the proof for completeness.

Property 2.3.2.7 $B \vdash_{\text{BCD}} M{:}\sigma$ if and only if there exists $A \in \mathcal{A}(M)$ such that $B \vdash_{\text{BCD}} A{:}\sigma$. □

The proof of the principal type property is completed by proving:

- Let $A \in \mathcal{N}$ and $<P, \pi>$ be the RV-principal pair for A. For any pair $<B, \sigma>$ such that $B \vdash_{\text{BCD}} A{:}\sigma$ there exists a linear chain C such that $C(<P, \pi>) = <B, \sigma>$.
- – $\mathcal{A}(M)$ is finite. Let $<P, \pi>$ be the RV-principal pair of M. Then there exists a chain C such that $C(<P, \pi>) = <B, \sigma>$.
  – $\mathcal{A}(M)$ is infinite. Then there exist a pair $<P, \pi> \in \Pi_{\text{RV}}(M)$ and a chain C such that $C(<P, \pi>) = <B, \sigma>$.

Chapter 3 *The Milner - and Mycroft Type Assignment Systems*

In this chapter we focus on some aspects of two type assignment systems that were defined for a simple applicative language called Exp that is in fact an extended lambda calculus.

The first, defined by R. Milner, has become very famous and is implemented in a type checker that is embedded in the functional programming language ML. The second one is a generalization of Milner's system presented by A. Mycroft, which is defined by allowing a more permissive rule for recursion. Both systems are present in the implementation of the functional programming language Miranda. Milner's system is used when the Miranda type assignment algorithm infers a type for an object; Mycroft's system is used when the type assignment algorithm does type checking, i.e. when the programmer has specified a type for an object. These two systems will play an important role in the third part of this thesis.

3.1 *The Milner Type Assignment System*

In [Milner '78], a formal type discipline was presented for polymorphic procedures in a simple programming language called Exp, designed to express that certain procedures work well on objects of a wide variety. This kind of procedures is called *(shallow) polymorphic*, and they are essential to obtain enough flexibility in programming. To illustrate the need for polymorphic procedures, consider the following example.

Suppose we have a programming language in which we can write the following program:

```
I x = x
I I
```

The definition of I is of course a definition for the identity function. If the type assignment system for this language were not able to express that I works well on objects of a different type, then the term I I cannot be typed. Milner's Type Assignment System makes it possible to express that various occurrences of I can have different types, as long as these types are related (by Curry-substitution) to the type derived for the definition of I.

Moreover, Milner presented a compile type-checking algorithm $\mathcal{W}$ that is semantically sound (based on a formal semantics for the language) (so typed programs cannot go wrong), and syntactically sound, so if $\mathcal{W}$ accepts a program, then it is well typed.

In this subsection, we present Milner's Type Assignment System as was done in [Damas & Milner '82], and not as in [Milner '78], because the former presentation is clearer.

Definition 3.1.1 [Milner '78] The language Exp of expressions M is defined by:

i) All term-variables $x_1, x_2, x_3, \ldots \in$ Exp.

ii) If $M, N \in$ Exp, then $(MN) \in$ Exp.

iii) If x is a term-variable and $M \in$ Exp, then $(\lambda x.M) \in$ Exp.

iv) If x is a term-variable and $M \in \mathsf{Exp}$, then $(\mathsf{Fix}\, x.M) \in \mathsf{Exp}$.

v) If x is a term-variable and $M, N \in \mathsf{Exp}$, then $(\mathsf{let}\ x = N\ \mathsf{in}\ M) \in \mathsf{Exp}$.

The language defined in [Milner '78] also contains a conditional-structure (if-then-else). It is not present in the definition of Exp in [Damas & Milner '82], and neither does it play a role in this thesis, so we have omitted it from the definition. The language constructor Fix is introduced to model recursion, a very useful tool in a programming language. It is present in the definition of Exp in [Milner '78], but not in [Damas & Milner '82]. Since it plays a part in the extension defined by Mycroft of this system, we have inserted it here. Notice that Fix is a language constructor, not a combinator.

The example program given before would be expressed in Exp by: $(\mathsf{let}\ x = (\lambda y.y)\ \mathsf{in}\ (xx))$.

Definition 3.1.2 [Milner '78] i) a) The set of *ML-types* is inductively defined by:

1) All type-variables $\varphi_0, \varphi_1, \ldots$ are ML-types.
2) All type constants $c_0, c_1, \ldots$ are ML-types.
3) If σ, τ are ML-types, then $\sigma\to\tau$ is an ML-type.

Notice that the set of ML-types is the set of Curry-types, extended with type constants.

b) If σ is an ML-type, and $\varphi_1, \ldots, \varphi_n$ are type-variables, then $\forall\varphi_1 \cdots \forall\varphi_n.\sigma$ is called a *type-scheme*, and $\varphi_1, \ldots, \varphi_n$ are called its *generic* type-variables. Type-schemes are denoted by $\overline{\sigma}, \overline{\tau}$, etc.

Notice that if $\overline{\sigma} = \forall\varphi_1 \cdots \forall\varphi_n.\sigma$, then the set of type-variables occurring in σ is not necessarily equal to $\{\varphi_1, \ldots, \varphi_n\}$.

ii) An *ML-substitution* on types is defined like a Curry-substitution as the replacement of type-variables by types, extended with:

d) $(\varphi \mapsto \alpha)(c_i) = c_i$.

ML-substitution on a type-scheme $\overline{\sigma}$ is defined as the replacement of *free* type-variables by renaming the generic type-variables of $\overline{\sigma}$ if necessary.

iii) A type obtained from another by mere substitution is called an *instance*.

iv) If $\overline{\sigma}, \overline{\tau}$ are type-schemes, and $\overline{\sigma} = \forall\varphi_1 \cdots \forall\varphi_n.\mu$, $\overline{\tau} = \forall\varphi_{n+1} \cdots \forall\varphi_{n+m}.\rho$, and for ρ there is a Curry-substitution S such that $S(\mu) = \rho$, and none of the $\varphi_{n+1}, \ldots, \varphi_{n+m}$ occur free in $\overline{\sigma}$, then $\overline{\tau}$ is called a *generic instance* of $\overline{\sigma}$ and we write $\overline{\sigma} > \overline{\tau}$.

In the following definition, we show derivation rules for Milner's system as presented in [Damas & Milner '82]. (In [Milner '78] there are no derivation rules; instead, a rather complicated definition of 'well typed prefixed expressions' is given.) In this definition we deviate from the standard natural deduction style presentation, as the basis used to derive a statement plays a more important role in the definition of the rules; in the sequent style the basis is explicitly given.

Definition 3.1.3 [Damas & Milner '82] *ML-type assignment* and *ML-derivations* are defined by the following sequent style deduction system.

$$(\text{TAUT}):\quad B\cup\{x{:}\overline{\sigma}\} \vdash_{\text{ML}} x{:}\overline{\sigma} \qquad\qquad (\text{FIX}):\quad \frac{B\cup\{x{:}\sigma\} \vdash_{\text{ML}} M{:}\sigma}{B \vdash_{\text{ML}} (\mathsf{Fix}\, x.M){:}\sigma}$$

$$(\text{ABS}):\quad \frac{B\cup\{x{:}\sigma\} \vdash_{\text{ML}} M{:}\tau}{B \vdash_{\text{ML}} (\lambda x.M){:}\sigma\to\tau}\ (a) \qquad (\text{COMB}):\quad \frac{B \vdash_{\text{ML}} M{:}\sigma\to\tau \quad B \vdash_{\text{ML}} N{:}\sigma}{B \vdash_{\text{ML}} (MN){:}\tau}$$

$$(\text{INST}):\quad \frac{B \vdash_{\text{ML}} M{:}\overline{\sigma} \quad \overline{\sigma} > \overline{\tau}}{B \vdash_{\text{ML}} M{:}\overline{\tau}} \qquad (\text{GEN}):\quad \frac{B \vdash_{\text{ML}} M{:}\overline{\sigma}}{B \vdash_{\text{ML}} M{:}\forall\varphi.\overline{\sigma}}\ (b)$$

$$(\text{LET}):\quad \frac{B\cup\{x{:}\overline{\sigma}\} \vdash_{\text{ML}} M{:}\tau \qquad B \vdash_{\text{ML}} N{:}\overline{\sigma}}{B \vdash_{\text{ML}} (\mathsf{let}\ x = N\ \mathsf{in}\ M){:}\tau}$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

(b) If φ not free in any assumption in the basis B.

In understanding the (LET)-rule, notice that the statement $x{:}\overline{\sigma}$ is used. Assume that $\overline{\sigma} = \forall\varphi_1 \cdots \forall\varphi_n.\sigma$; suppose that in building the derivation for the statement $M{:}\tau$, $\overline{\sigma}$ is instantiated (otherwise the rules (ABS) and (COMB) cannot be used) into the types $\sigma_1, \ldots, \sigma_n$. So, for every σ_i there is a substitution S_i such that $S_i(\sigma) = \sigma_i$. Assume without loss of generality that $N{:}\overline{\sigma}$ is obtained from $N{:}\sigma$ by applying the rule (GEN). Notice that the types actually used for x in the derivation for $M{:}\tau$ are, therefore, instances of the type derived for N.

In a sense, the terms $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$ and $((\lambda x.M)N)$ are both denotations for a redex; however, the semantic interpretation of these terms is different (for details of this semantics, see [Milner '78]). The term $((\lambda x.M)N)$ is interpreted as a function with an operand, whereas the term $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$ is interpreted as the term $M[x := N]$. This difference is reflected in the way the type assignment system treats these terms: in assigning a type to $((\lambda x.M)N)$, the term-variable x can only be typed with *one* Curry-type. This is not required for x in $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$, because the type-scheme $\overline{\sigma}$ can be instantiated into several, different Curry-types. Also, as suggested by the semantics of the let-construct, in finding a type for the expression $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$, the ML-type inference system in fact looks for a type for the term $M[x := N]$.

As was also remarked by A. Mycroft in [Mycroft '84], instead of inserting Fix as a language constructor, a fixed-point combinator FIX can be inserted that has the type $\forall\varphi.(\varphi\to\varphi)\to\varphi$ (as implicitly done in [Damas & Milner '82]).

This system has several important properties:

- Because of the presence of type-schemes and the rules (GEN) and (INST) in this system, polymorphism can be modelled.
- The system has the principal pair property.
- Type assignment is decidable. Milner presents an algorithm (called $\mathcal{W}$) that takes as input a pair of (*basis*, *term*) and returns a pair of (*substitution*, *type*) such that:
  - *Completeness of* $\mathcal{W}$. If for a term M there are bases B and B' and type σ, such that B' is an instance of B and $B' \vdash_{\text{ML}} M{:}\sigma$, then $\mathcal{W}(B, M)$ succeeds and returns the

pair (S, τ), and there is a substitution S' such that $B' = S'(S(B))$ and $S'(S(\tau)) > \sigma$.
- *Soundness of* $\mathcal{W}$. For every term M: if $\mathcal{W}(B, M)$ returns the pair (S, σ), then $S(B) \vdash_{\text{ML}} M{:}\sigma$.

- By defining an adequate semantics for the language Exp, soundness of inference will hold: If $B \vdash_{\text{ML}} M{:}\overline{\sigma}$, then $B \models M{:}\overline{\sigma}$.

This notion of type assignment, when restricted to the pure Lambda Calculus, is also a restriction of the Polymorphic Type Discipline, $\Lambda 2$, as presented in [Girard '86]. This system is obtained from Curry's system by adding the type constructor '$\forall$': if φ is a type-variable and σ is a type, then $\forall\varphi.\sigma$ is a type. A difference between the types created in this way and the types (or type-schemes) of Milner's system is that in Milner's type-schemes the $\forall$-symbol can occur only at the outside of a type, not inside. In section 8.1 we will show that the restriction of the ML-type assignment system to the pure Lambda Calculus is equivalent to a restricted intersection system.

3.2 *The Mycroft Type Assignment System*

In [Mycroft '84] (and, independently, in [Kfoury *et al.* '88]), a generalization of Milner's Type Assignment System is presented. This generalization is made to obtain more permissive types for recursively defined objects.

The example that Mycroft gives to justify his generalization is the following (using a here not defined syntax):

$$\begin{array}{ll} \textit{map } f\ l & = \text{if } (\textit{null } l) \text{ then } \textit{nil} \text{ else } (\textit{cons } (f\ (\textit{hd } l))\ (\textit{map } f\ (\textit{tl } l))) \\ \textit{squarelist } l & = \textit{map } (\lambda x.x * x)\ l \\ \textit{squarelist } (\textit{cons } 2\ \textit{nil}) & \end{array}$$

where *hd*, *tl*, *null*, *nil*, *cons*, and 2 are assumed to be familiar, and $*$ is a multiplication.

In the implementation of ML, there is no check if functions are independent or are mutually recursive, so all definitions are dealt with in one step. For this purpose, the language Exp is formally extended with a pairing function '< , >', and the translation of the above expression into Exp will be:

$$\begin{array}{l} \text{let } <\textit{map}, \textit{squarelist}> = \text{Fix } <m, s>. \\ \quad <\lambda f.\ \lambda l.\ \text{if } (\textit{null } l) \text{ then } \textit{nil} \text{ else } (\textit{cons } (f\ (\textit{hd } l))\ (m\ f\ (\textit{tl } l))),\ \lambda l.\ (m\ (\lambda x.x * x)\ l)> \\ \text{in } (\textit{squarelist } (\textit{cons } 2\ \textit{nil})) \end{array}$$

Within Milner's Type Assignment System these definitions (when defined simultaneously in ML) would get the types:

$$\begin{array}{ll} \textit{map} & : (\text{INT}\rightarrow\text{INT}) \rightarrow (\text{INT-list}) \rightarrow (\text{INT-list}) \\ \textit{squarelist} & : (\text{INT-list}) \rightarrow (\text{INT-list}) \end{array}$$

while the definition of *map* alone would yield the type:

$$\textit{map} \quad : \forall\varphi_1\forall\varphi_2.(\varphi_1\rightarrow\varphi_2) \rightarrow (\varphi_1\text{-list})\rightarrow(\varphi_2\text{-list}).$$

Since the definition of *map* does not depend on the definition of *squarelist*, one would expect the type inferer to find the second type for *map*. That such is not the case is caused by the fact

that all occurrences of a recursively defined function on the right hand side within the definition must have the same type as in the left hand side.

There is more than one way to overcome this problem. One is to recognize mutual recursive rules, and treat them as one definition. (Easy to implement, but difficult to formalize, a problem we run into in chapters ten and twelve). Then, the translation of the above program could be:

$$\begin{array}{l} \text{let } \textit{map} = (\text{Fix } m.\ \lambda f.\ \lambda l.\ \text{if } (\textit{null } l) \text{ then } \textit{nil} \text{ else } (\textit{cons } (f\ (\textit{hd } l))\ (m\ f\ (\textit{tl } l)))) \\ \qquad \text{in } (\text{let } \textit{squarelist} = (\lambda l.\ (\textit{map } (\lambda x.x * x)\ l)) \\ \qquad\qquad \text{in } (\textit{squarelist } (\textit{cons } 2\ \textit{nil}))) \end{array}$$

The other solution, chosen by Mycroft, is to allow of a more general rule for recursion than Milner's (FIX)-rule. Mycroft, like Milner, defined a type assignment system for a language Exp that is the same as Milner's. Also the set of types used by Mycroft is the same as defined by Milner. In the following definition we show the derivation rules for Mycroft's system.

Definition 3.2.1 [Mycroft '84] *Mycroft type assignment* is defined by the following sequent style deduction system.

$$\text{(TAUT)}: \quad B \cup \{x{:}\overline{\sigma}\} \vdash_{\text{Myc}} x{:}\overline{\sigma} \qquad \text{(FIX)}: \quad \frac{B \cup \{x{:}\overline{\sigma}\} \vdash_{\text{Myc}} M{:}\overline{\sigma}}{B \vdash_{\text{Myc}} (\text{Fix } x.M){:}\overline{\sigma}}$$

$$\text{(ABS)}: \quad \frac{B \cup \{x{:}\sigma\} \vdash_{\text{Myc}} M{:}\tau}{B \vdash_{\text{Myc}} (\lambda x.M){:}\sigma\rightarrow\tau}\ \text{(a)} \qquad \text{(COMB)}: \quad \frac{B \vdash_{\text{Myc}} M{:}\sigma\rightarrow\tau \quad B \vdash_{\text{Myc}} N{:}\sigma}{B \vdash_{\text{Myc}} (MN){:}\tau}$$

$$\text{(INST)}: \quad \frac{B \vdash_{\text{Myc}} M{:}\overline{\sigma} \quad \overline{\sigma} \leq \overline{\tau}}{B \vdash_{\text{Myc}} M{:}\overline{\tau}} \qquad \text{(GEN)}: \quad \frac{B \vdash_{\text{Myc}} M{:}\overline{\sigma}}{B \vdash_{\text{Myc}} M{:}\forall\varphi.\overline{\sigma}}\ \text{(b)}$$

$$\text{(LET)}: \quad \frac{B \cup \{x{:}\overline{\sigma}\} \vdash_{\text{Myc}} M{:}\tau \qquad B \vdash_{\text{Myc}} N{:}\overline{\sigma}}{B \vdash_{\text{Myc}} (\text{let } x = N \text{ in } M){:}\tau}$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

(b) If φ not free in any assumption in the basis.

Thus, the only difference lies in the fact that, in this system, the derivation rule (FIX) allows for type-schemes instead of types, so the various occurrences of x in M can be typed with different Curry-types.

Mycroft's system has the following properties:

- Like in Milner's system, in this system polymorphism can be modelled.
- Type assignment in this system is undecidable, as shown by A.J. Kfoury, J. Tiuryn and P. Urzyczyn in [Kfoury *et al.* '90].
- By defining an adequate semantics for the language Exp, soundness of inference holds: If $B \vdash_{\text{Myc}} M{:}\overline{\sigma}$, then $B \models M{:}\overline{\sigma}$.

3.3 *The difference between Milner's and Mycroft's system*

Since Mycroft's system is a true extension of Milner's, there are terms typeable in Mycroft's system that are not typeable in Milner's. For example,

$$\mathsf{Fix}\, g.\lambda x.((\lambda ab.a)\,(g\,(\lambda c.c))\,(g\,(\lambda de.d))) : \forall\varphi_1\forall\varphi_2.(\varphi_1\rightarrow\varphi_2).$$

is a derivable statement in Mycroft's system. It is easy to see that this term is not typeable using Milner's system, because the types needed for g in the body of the term cannot be unified.

But, the generalization allows for more than was aimed at by Mycroft: in contrast to what Mycroft suggests, type assignment in this system is undecidable. And not only is the set of terms that can be typed in Mycroft's system larger than in Milner's, it is also possible to assign more general types to terms that are typeable in Milner's system. Take, for example,

$$R = \mathsf{Fix}\, r.\lambda xy.(r\,(r\,y\,(\lambda ab.a))\,x),$$

then the statement $R : \forall\varphi_1\forall\varphi_2\forall\varphi_3.(\varphi_1\rightarrow\varphi_2\rightarrow\varphi_3)$ is derivable in Mycroft's system. R is also typeable in Milner's system, where its principal type is:

$$\forall\varphi_4\forall\varphi_5.((\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow(\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4).$$

In [Milner '78] type assignment was not defined by presenting derivation rules, but by defining well-typed prefixed expressions. The derivation rules (GEN) and (INST) presented in [Damas & Milner '82] are motivated by the proof in [Milner '78] for the following statement: if $B \vdash_{\mathrm{ML}} M{:}\sigma$ and S is a substitution, then $S\,(B) \vdash_{\mathrm{ML}} M{:}S\,(\sigma)$ can be shown using a derivation with exactly the same height. This property is used in [Damas & Milner '82] for the proof of soundness of the algorithm $\mathcal{W}$.

For Mycroft's system, this cannot be proved: to prove that if $B \vdash_{\mathrm{Myc}} M{:}\sigma$ and S is a substitution, then $S\,(B) \vdash_{\mathrm{Myc}} M{:}S\,(\sigma)$, for some terms the derivation rules (GEN) and (INST) are needed: substitution is *only by definition* a sound operation.

For example, to show in Milner's system that any substitution instance of the type

$$\alpha = (\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow(\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4)\rightarrow\varphi_4\rightarrow\varphi_5\rightarrow\varphi_4$$

is a correct type for R, it is sufficient to take the derivation for

$$\{r{:}\alpha\} \vdash_{\mathrm{ML}} \lambda xy.(r\,(r\,y\,(\lambda ab.a))\,x) : \alpha,$$

just substitute the types in this derivation, and apply the (FIX)-derivation rule.

For Mycroft's system this construction does not yield correct derivations. If the derivation for the statement

$$\{r : \forall\varphi_1\forall\varphi_2\forall\varphi_3.(\varphi_1\rightarrow\varphi_2\rightarrow\varphi_3)\} \vdash_{\mathrm{Myc}} \lambda xy.(r\,(r\,y\,(\lambda ab.a))\,x) : \varphi_1\rightarrow\varphi_2\rightarrow\varphi_3$$

were taken and, for example, the type-variable φ_2 were replaced by φ_1, then the (FIX)-derivation rule cannot be applied because the type in the conclusion of the derivation is affected by the substitution, but the type $\forall\varphi_1\forall\varphi_2\forall\varphi_3.(\varphi_1\rightarrow\varphi_2\rightarrow\varphi_3)$ in the basis is not. The only way to obtain $\vdash_{\mathrm{Myc}} R : \varphi_1\rightarrow\varphi_1\rightarrow\varphi_3$ is by using the derivation rule (INST) on the derivation for $\vdash_{\mathrm{Myc}} R : \forall\varphi_1\forall\varphi_2\forall\varphi_3.(\varphi_1\rightarrow\varphi_2\rightarrow\varphi_3)$.

Chapter 4 *The Strict Type Assignment System*

In this chapter we present the Strict Type Assignment System, a restricted version of the BCD-system. Compared to the BCD-system, the major feature of this restricted system is the absence of the derivation rule ($\leq$). It is based on a set of strict types, which correspond to the normalized types of the CDV-system. We will show that these two together give rise to a strict filter lambda model that is essentially different from $\mathcal{F}$. We will show that the Strict Type Assignment System is the nucleus of the BCD-system, i.e. for every derivation in the BCD-type discipline there is a derivation in which ($\leq$) is used only at the very end. Finally we will prove that strict type assignment is complete for inference semantics.

4.1 *Strict type assignment*

In this section we will present the Strict Type Assignment System, together with the set of strict types. These two together will yield a lambda model $\mathcal{F}_{\mathrm{S}}$, with which we prove completeness of type assignment without the derivation rule ($\leq$).

The elimination of $\leq$ induces a set of strict types, a restriction of the set of types used in the BCD-system. Strict types are the types that are strictly needed to assign a type to a term in the BCD-system. The set of strict types is a true subset of set $\mathcal{T}_{\mathrm{BCD}}$; intersection type schemes and the type constant ω play a limited role in the Strict Type Assignment System. We will assume that ω is the same as an intersection over zero elements: if $n = 0$, then $\sigma_1\cap\cdots\cap\sigma_n = \omega$, so ω does not occur in an intersection subtype. Moreover, intersection type schemes (so also ω) occur in strict types only as subtypes at the left hand side of an arrow type scheme. We could have omitted the type constant ω completely from the presentation of the system, because we can always assume that $n = 0$ in $\sigma_1\cap\cdots\cap\sigma_n$, but some of the definitions and the results we obtain are more clear when ω is dealt with explicitly.

Definition 4.1.1 i) $\mathcal{T}_{\mathrm{s}}$, the set of *strict types*, is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{\mathrm{s}}$.

b) If $\tau, \sigma_1, \ldots, \sigma_n \in \mathcal{T}_{\mathrm{s}}$ $(n \geq 0)$, then $\sigma_1\cap\cdots\cap\sigma_n\rightarrow\tau \in \mathcal{T}_{\mathrm{s}}$.

ii) $\mathcal{T}_{\mathrm{S}}$, the set of strict intersection types is defined by: If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_{\mathrm{s}}$ $(n \geq 0)$, then $\sigma_1\cap\cdots\cap\sigma_n \in \mathcal{T}_{\mathrm{S}}$.

iii) On $\mathcal{T}_{\mathrm{S}}$, the relation $\leq_{\mathrm{S}}$ is defined by:

a) $\sigma \leq_{\mathrm{S}} \sigma$.

b) $\sigma \leq_{\mathrm{S}} \omega$.

c) $\sigma\cap\tau \leq_{\mathrm{S}} \sigma$ & $\sigma\cap\tau \leq_{\mathrm{S}} \tau$.

d) $\sigma \leq_{\mathrm{S}} \tau \leq_{\mathrm{S}} \rho \Rightarrow \sigma \leq_{\mathrm{S}} \rho$.

e) $\sigma \leq_{\mathrm{S}} \rho$ & $\sigma \leq_{\mathrm{S}} \tau \Rightarrow \sigma \leq_{\mathrm{S}} \rho\cap\tau$.

iv) On $\mathcal{T}_{\mathrm{S}}$, the relation $\sim_{\mathrm{S}}$ is defined by:

a) $\sigma \leq_S \tau \leq_S \sigma \Rightarrow \sigma \sim_S \tau$.
b) $\sigma \sim_S \rho$ & $\tau \sim_S \mu \Rightarrow \sigma \rightarrow \tau \sim_S \rho \rightarrow \mu$.

v) A *strict statement* is an expression of the form $M{:}\sigma$, where $M \in \Lambda$ and $\sigma \in \mathcal{T}_S$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

vi) A *strict basis* is a set of strict statements with only distinct variables as subjects. If $\sigma_1 \cap \cdots \cap \sigma_n$ is a predicate in a basis, then $n \geq 1$.

$\mathcal{T}_S$ may be considered modulo $\sim_S$. Then $\leq_S$ becomes a partial order.

The definition of $\sim_S$ as in [van Bakel '92a] did not contain part 4.1.1 (iv.b), but was defined by: $\sigma \leq_S \tau \leq_S \sigma \Leftrightarrow \sigma \sim_S \tau$. As was remarked by Professor G. Plotkin of the University of Edinburgh, Schotland (private communication), defining the equivalence relation on types in that way causes an anomaly in the definition of type-interpretation as in definition 4.4.2, since, then, the interpretation of an arrow type $\sigma \rightarrow \tau$ is no longer a map from the interpretation of σ onto the interpretation of τ. See also the remark made after theorem 4.4.3.

Unless stated otherwise, if $\sigma_1 \cap \cdots \cap \sigma_n$ is used to denote a type, then all $\sigma_1, \ldots, \sigma_n$ are assumed to be strict. Notice that $\mathcal{T}_s$ is a proper subset of $\mathcal{T}_S$, that the set $\mathcal{T}_s$ coincides with the set of normalized tail-proper types of the CDV-system, and that the set $\mathcal{T}_S$ coincides with the set of normalized tail-proper sequences.

It is an easy exercise to show that the definition of $\leq_S$ is equivalent to:

$$\{\sigma_1, \ldots, \sigma_n\} \subseteq \{\tau_1, \ldots, \tau_m\} \subseteq \mathcal{T}_s \, (n \geq 0, m \geq 0) \Leftrightarrow \tau_1 \cap \cdots \cap \tau_m \leq_S \sigma_1 \cap \cdots \cap \sigma_n.$$

It is also easy to show that, if $\sigma \leq_S \tau$, then either $\tau = \omega$ or $\tau = \sigma$ or σ is an intersection type scheme in which τ occurs. Notice also that, if $\sigma \sim_S \tau$, then τ can be obtained from σ by permuting the strict components in an intersection subtype, e.g. $\rho \cap (\sigma \cap \tau) \sim_S (\rho \cap \sigma) \cap \tau$. The differences affect none of our proofs and $\sigma = \tau$ means $\sigma \sim_S \tau$, so we consider types modulo $\sim_S$. Notice also that $\{ \sigma \mid \sigma \sim_S \omega \} = \{\omega\}$.

As in definition 2.3.2, with the relation $\leq$ we extend the relation $\leq_S$ to bases. The following definition introduces some terminology and notations for bases.

Definition 4.1.2 i) If $B \cup \{x{:}\sigma\}$ is a basis, then B is a basis, and x does not occur in B.

ii) If $B_1, \ldots, B_n$ are bases, then $\Pi\{B_1, \ldots, B_n\}$ is the basis defined as follows: $x{:}\sigma_1 \cap \cdots \cap \sigma_m \in \Pi\{B_1, \ldots, B_n\}$ if and only if $\{x{:}\sigma_1, \ldots, x{:}\sigma_m\}$ is the set of all statements whose subject is x that occur in $B_1 \cup \ldots \cup B_n$.

Notice that if $n = 0$, then $\Pi\{B_1, \ldots, B_n\} = \emptyset$. By abuse of notation, we sometimes write a basis as $B \cup \{x{:}\sigma\}$, where $\sigma = \omega$. We will then assume that $B \cup \{x{:}\sigma\} = B$.

The Strict Type Assignment System is constructed from the set of strict types and a minor extension of the derivation rules of the CD-system. This way, a syntax directed system is obtained that satisfies the main properties of the BCD-system: type assignment is closed under β-equality, the set of terms typeable with type σ from a basis B such that ω does not occur in B and σ is the set of normalizable terms, and the set of terms typeable with type $\sigma \neq \omega$ is the set of terms having a head normal form.

Strict types and strict derivations are closely related. Strict derivations are syntax directed and yield strict types.

Definition 4.1.3 i) *Strict type assignment* and *strict derivations* are defined by the following natural deduction system (where all types displayed are strict, except σ in the derivation rule $(\rightarrow I)$):

$$(\rightarrow I): \quad \frac{\begin{matrix}[x{:}\sigma] \\ \vdots \\ M{:}\tau\end{matrix}}{\lambda x.M{:}\sigma\rightarrow\tau} \ (a) \quad (\sigma \in \mathcal{T}_S) \qquad (\cap E): \quad \frac{x{:}\sigma_1\cap\cdots\cap\sigma_n}{x{:}\sigma_i} \ (n \geq 2)$$

$$(\rightarrow E): \quad \frac{M{:}\sigma_1\cap\cdots\cap\sigma_n\rightarrow\tau \qquad N{:}\sigma_1 \ \ldots \ N{:}\sigma_n}{MN{:}\tau} \ (n \geq 0)$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

If $M{:}\sigma$ is derivable from B using a strict derivation, we write $B \vdash_s M{:}\sigma$.

ii) We define $\vdash_S$ by: $B \vdash_S M{:}\sigma$ if and only if: there are $\sigma_1, \ldots, \sigma_n$ $(n \geq 0)$ such that $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$ and for every $1 \leq i \leq n$ $B \vdash_s M{:}\sigma_i$.

If $B \vdash_S M{:}\sigma$, and $B = \emptyset$, we write $\vdash_S M{:}\sigma$. Notice that in $B \vdash_s M{:}\sigma$ the basis can contain types that are not strict, and that $B \vdash_S M{:}\sigma$ is only defined for $\sigma \in \mathcal{T}_S$.

Example 4.1.4 In this notion of type assignment we cannot – unlike e.g. in the BCD-system and the CDV-systems – build *one* derivation for $\{x{:}\sigma\cap\tau\} \vdash_S x{:}\sigma\cap\tau$. By definition 4.1.3 (ii) we should show that there are derivations for *both* $\{x{:}\sigma\cap\tau\} \vdash_s x{:}\sigma$ and $\{x{:}\sigma\cap\tau\} \vdash_s x{:}\tau$.

$$\frac{x{:}\sigma\cap\tau}{x{:}\sigma} \quad (\cap E) \quad \frac{x{:}\sigma\cap\tau}{x{:}\tau}$$

The difference between the strict system and the BCD-system is essentially this: in the BCD-system, the derivation rule $(\cap E)$ is allowed of on all terms, whereas in the strict system it is only performed on variables, as in the CD-system. Also, the BCD-system has the derivation rules (ω) and $(\cap I)$ – also allowed of on all terms – that are implicitly present in the derivation rule $(\rightarrow E)$ of the strict system. Moreover, we cannot compose a derivation in the $\vdash_S$ system with conclusion $M{:}\omega$ with any other derivation.

The introduction of two different notions of derivability seems somewhat superfluous. Notice that we could limit ourselves to one, by stating:

We define $\vdash_s$ by: $B \vdash_s M{:}\sigma$ if and only if there are $\sigma_1, \ldots, \sigma_n$ $(n \geq 0)$ such that $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$ and for every $1 \leq i \leq n$ $M{:}\sigma_i$ is derivable from B using a strict derivation.

This definition would cause a lot of words in the proofs, and perhaps also a lot of confusion as well. We therefore prefer two different notions of derivability.

Apart from the presence of ω, the type assignment defined by $\vdash_S$ is in fact the same as the CD-system. Also, the derivation rules (not the set of types) for the one defined by $\vdash_s$ are in fact the same as for the $\vdash_{CDV}$ -system; the difference between these systems is nothing more than, in the $\vdash_{CDV}$ -system, ω is a type, whereas it is a sequence in the strict system. The type assignment defined by $\vdash_S$ is in fact an extension of $\vdash_{CDV_R}$, the main difference is that in the derivation rule $(\rightarrow I)$ more than just the used types can be cancelled.

For these notions of type assignment, the following properties hold:

Lemma 4.1.5 i) $B \vdash_s MN{:}\sigma \Leftrightarrow \exists \tau \, [B \vdash_s M{:}\tau\rightarrow\sigma \ \& \ B \vdash_S N{:}\tau]$.

ii) $B \vdash_{\mathrm{s}} M{:}\sigma \;\Leftrightarrow\; B \vdash_{\mathrm{S}} M{:}\sigma \;\&\; \sigma \in \mathcal{T}_{\mathrm{s}}$.

iii) $B \vdash_{\mathrm{s}} \lambda x.M{:}\sigma \;\Leftrightarrow\; \exists\, \rho \in \mathcal{T}_{\mathrm{S}}, \mu \in \mathcal{T}_{\mathrm{s}}\ [\ \sigma = \rho{\to}\mu \;\&\; B\backslash x \cup \{x{:}\rho\} \vdash_{\mathrm{s}} M{:}\mu\ ]$.

iv) $B \vdash_{\mathrm{S}} M{:}\sigma \;\Leftrightarrow\; \exists\, \sigma_1, \ldots, \sigma_n\ (n \geq 0)\ [\ \sigma = \sigma_1 \cap \cdots \cap \sigma_n \;\&\; \forall\, 1 \leq i \leq n\ [\ B \vdash_{\mathrm{s}} M{:}\sigma_i\ ]\ ]$.

v) $B \vdash_{\mathrm{S}} M{:}\sigma \;\Leftrightarrow\; \{\ x{:}\tau \in B \mid x \in \mathrm{FV}(M)\ \} \vdash_{\mathrm{S}} M{:}\sigma$.

vi) $\forall\, \sigma, \tau \in \mathcal{T}_{\mathrm{S}}\ [\ B \cup \{x{:}\sigma\} \vdash_{\mathrm{S}} M{:}\tau \;\Rightarrow\; B \cup \{x{:}\sigma\} \vdash_{\mathrm{S}} N{:}\tau\ ] \;\Rightarrow$
$\forall\, \rho \in \mathcal{T}_{\mathrm{S}}\ [\ B \vdash_{\mathrm{S}} \lambda x.M{:}\rho \;\Rightarrow\; B \vdash_{\mathrm{S}} \lambda x.N{:}\rho\ ]$.

vii) $\{x{:}\sigma\} \vdash_{\mathrm{S}} x{:}\tau \;\Leftrightarrow\; \sigma \leq_{\mathrm{S}} \tau$.

Proof: Easy. □

Lemma 4.1.6 If $B \vdash_{\mathrm{S}} M{:}\sigma$ and $B' \leq_{\mathrm{S}} B$, then $B' \vdash_{\mathrm{S}} M{:}\sigma$.

Proof: Since for every $x{:}\tau \in B$: if $B' \leq_{\mathrm{S}} B$, then by lemma 4.1.5 (vii) $B' \vdash_{\mathrm{S}} x{:}\tau$. □

4.2 *The strict filter lambda model*

As in [Barendregt *et al.* '83] we aim to construct a filter lambda model. By use of names we will distinguish between the definition of filters in the paper, and the ones given here.

Definition 4.2.1 i) A subset d of $\mathcal{T}_{\mathrm{S}}$ is called a *strict filter* if and only if:

a) $\sigma_1, \ldots, \sigma_n \in d\ (n \geq 0) \;\Rightarrow\; \sigma_1 \cap \cdots \cap \sigma_n \in d$.

b) $\tau \in d \;\&\; \tau \leq_{\mathrm{S}} \sigma \;\Rightarrow\; \sigma \in d$.

ii) If V is a subset of $\mathcal{T}_{\mathrm{S}}$, then $\uparrow_{\mathrm{S}} V$ is the smallest strict filter that contains V, and $\uparrow_{\mathrm{S}} \sigma = \uparrow_{\mathrm{S}} \{\sigma\}$. If no confusion is possible, we will omit the subscript on $\uparrow$.

iii) $\mathcal{F}_{\mathrm{S}} = \{\ d \subseteq \mathcal{T}_{\mathrm{S}} \mid d$ is a strict filter $\}$. We define application on $\mathcal{F}_{\mathrm{S}}$, $\cdot : \mathcal{F}_{\mathrm{S}} \times \mathcal{F}_{\mathrm{S}} \to \mathcal{F}_{\mathrm{S}}$ by:

$$d \cdot e = \uparrow\{\ \tau \mid \exists\, \sigma \in e\ [\ \sigma{\to}\tau \in d\ ]\ \}.$$

Notice that if types are not considered modulo $\sim_{\mathrm{S}}$, then part (i.b) should also contain: $\tau \in d \;\&\; \tau \sim_{\mathrm{S}} \sigma \;\Rightarrow\; \sigma \in d$. Notice also that $\omega \in d$, for every strict filter d.

The application on BCD-filters as defined in definition 2.3.1.1 would not be useful in our approach, since it would not be well defined. We must force the application to yield filters, since in each arrow type scheme $\sigma{\to}\tau \in \mathcal{T}_{\mathrm{S}}$, τ is strict. $<\mathcal{F}_{\mathrm{S}}, \subseteq>$ is a cpo and henceforward we will consider it with the corresponding Scott topology. Because of the remark made after definition 4.1.1, part 4.2.1 (i) can be replaced by:

i) $\sigma_1 \cap \cdots \cap \sigma_n \in d\ (n \geq 0) \;\Leftrightarrow\; \forall\, 1 \leq i \leq n\ [\ \sigma_i \in d\ ]$.

Notice that a strict filter generated by a finite number of types is finite. Let, for example, σ be a strict type, then $\uparrow\sigma = \{\sigma, \omega\}$ (where by $\sim_{\mathrm{S}}$ we identify σ and $\sigma \cap \sigma$). If σ is an intersection of n strict types, $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$, then $\uparrow\sigma$ contains 2^n elements, namely:

$$\{\ \sigma_1, \ldots, \sigma_n, \sigma_1 \cap \sigma_2, \sigma_1 \cap \sigma_3, \ldots, \sigma_{n-1} \cap \sigma_n, \sigma_1 \cap \sigma_2 \cap \sigma_3, \ldots, \sigma_1 \cap \cdots \cap \sigma_n, \omega\ \}.$$

Of course $\mathcal{F}_{\mathrm{S}}$ contains also infinite elements.

Lemma 4.2.2 For strict filters the following properties hold:

i) $\sigma \in \uparrow V \;\&\; V \subseteq \mathcal{T}_{\mathrm{s}} \;\Leftrightarrow\; \exists\, \sigma_1, \ldots, \sigma_n\ (n \geq 0)\ [\ \sigma = \sigma_1 \cap \cdots \cap \sigma_n \;\&\; \forall\, 1 \leq i \leq n\ [\sigma_i \in V]\ ]$.

ii) $\sigma \in \mathcal{T}_{\mathrm{s}} \;\&\; \sigma \in \uparrow V \;\&\; V \subseteq \mathcal{T}_{\mathrm{s}} \;\Rightarrow\; \sigma \in V$.

iii) $\sigma \in \uparrow\tau \;\Leftrightarrow\; \tau \leq_{\mathrm{S}} \sigma$.

iv) $\sigma \in \uparrow\{\ \tau \mid B \vdash_{\mathrm{s}} M{:}\tau\ \} \;\Leftrightarrow\; \sigma \in \{\ \tau \mid B \vdash_{\mathrm{S}} M{:}\tau\ \}$.

Proof: Easy. □

Theorem 4.2.3 i) If $B \vdash_{\mathrm{S}} M{:}\sigma$ and $\sigma \leq_{\mathrm{S}} \tau$, then $B \vdash_{\mathrm{S}} M{:}\tau$.

ii) $\{\ \sigma \in \mathcal{T}_{\mathrm{S}} \mid B \vdash_{\mathrm{S}} M{:}\sigma\ \} \in \mathcal{F}_{\mathrm{S}}$.

Proof: i) By induction on $\leq_{\mathrm{S}}$.

ii) By lemma 4.2.2 (iv). □

Unlike [Barendregt *et al.* '83], we will not use the method of Hindley and Longo to show that $<\mathcal{F}_{\mathrm{S}}, \cdot>$ is a lambda model, but instead we will specify two maps F and G and show that these give rise to a lambda model.

Definition 4.2.4 We define F: $\mathcal{F}_{\mathrm{S}} \to [\mathcal{F}_{\mathrm{S}} \to \mathcal{F}_{\mathrm{S}}]$ and G: $[\mathcal{F}_{\mathrm{S}} \to \mathcal{F}_{\mathrm{S}}] \to \mathcal{F}_{\mathrm{S}}$ by:

i) F $d\ e = d \cdot e$.

ii) G $f = \uparrow\{\ \sigma{\to}\tau \in \mathcal{T}_{\mathrm{S}} \mid \tau \in f(\uparrow\sigma)\ \}$.

It is easy to check that F and G are continuous.

Theorem 4.2.5 $<\mathcal{F}_{\mathrm{S}}, \cdot>$ with F and G as defined in definition 4.2.4 is a lambda model.

Proof: By [Barendregt '84].5.4.1 it is sufficient to prove that F∘G = $id_{[\mathcal{F}_{\mathrm{S}} \to \mathcal{F}_{\mathrm{S}}]}$.

F∘G $f\ d = \uparrow\{\ \mu \mid \exists\, \rho \in d\ [\ \rho{\to}\mu \in \uparrow\{\ \sigma{\to}\tau \mid \tau \in f(\uparrow\sigma)\ \}\ ]\ \} =$ (4.2.2 (ii))

$\uparrow\{\ \mu \mid \exists\, \rho \in d\ [\ \mu \in f(\uparrow\rho)\ ]\ \} = f(d)$. □

Since the definition of $\sim_{\mathrm{S}}$ as in [van Bakel '92a] did not contain part 4.1.1 (iv.b), the filter model $\mathcal{F}_{\mathrm{S}}$ as defined in this chapter is, as domain, not equivalent to the one defined in the paper. With the definition we gave here, it is straightforward to show that $\mathcal{F}_{\mathrm{S}}$ is equivalent to Engeler's model $\mathcal{D}_A$ [Engeler '81].

Notice that $\mathcal{F}_{\mathrm{S}}$ and the filter lambda model $\mathcal{F}$ defined in [Barendregt *et al.* '83] are not isomorphic as complete lattices, since, for example, in $\mathcal{F}$ the BCD-filter $\uparrow\sigma \cap \tau{\to}\sigma$ is contained in $\uparrow\sigma{\to}\sigma$, whereas in $\mathcal{F}_{\mathrm{S}}$ the strict filter $\uparrow_{\mathrm{S}}\sigma \cap \tau{\to}\sigma$ is not contained in $\uparrow_{\mathrm{S}}\sigma{\to}\sigma$. Moreover, they are not isomorphic as lambda models, since, in $\mathcal{F}$, the meaning of $\lambda xy.xy$ is contained in the meaning of $\lambda x.x$, whereas this does not hold in $\mathcal{F}_{\mathrm{S}}$ (see the examples in 4.2.9).

Definition 4.2.6 Let ξ be a valuation of term-variables in $\mathcal{F}_{\mathrm{S}}$.

i) $[\![M]\!]_\xi$, the *interpretation of terms in* $\mathcal{F}_{\mathrm{S}}$ *via* ξ is inductively defined by:

a) $[\![x]\!]_\xi = \xi(x)$.

b) $[\![MN]\!]_\xi =$ F $[\![M]\!]_\xi [\![N]\!]_\xi$.

c) $[\![\lambda x.M]\!]_\xi =$ G $(\lambda\!\!\!\lambda\, v \in \mathcal{F}_{\mathrm{S}}.[\![M]\!]_{\xi(v/x)})$.

ii) $B_\xi = \{\ x{:}\sigma \mid \sigma \in \xi(x)\ \}$.

Theorem 4.2.7 For all M, ξ: $[\![M]\!]_\xi = \{ \sigma \in \mathcal{T}_S \mid B_\xi \vdash_S M{:}\sigma \}$.

Proof: By induction on the structure of lambda terms.

i) $[\![x]\!]_\xi = \xi(x)$. Since $\{ y{:}\rho \mid \rho \in \xi(y) \} \vdash_S x{:}\sigma \iff \sigma \in \xi(x)$.

ii) $[\![MN]\!]_\xi = \uparrow\{ \tau \mid \exists\, \sigma\, [\, B_\xi \vdash_s M{:}\sigma{\rightarrow}\tau \;\&\; B_\xi \vdash_S N{:}\sigma \,] \} =$ (4.1.5 (i) & (ii))
$\uparrow\{ \tau \mid B_\xi \vdash_s MN{:}\tau \} =$ (4.2.3 (ii))
$\{ \tau \mid B_\xi \vdash_S MN{:}\tau \}$.

iii) $[\![\lambda x.M]\!]_\xi = \uparrow\{ \sigma{\rightarrow}\tau \mid B_{\xi(\uparrow\sigma/x)} \vdash_S M{:}\tau \} =$ (4.1.5 (ii))
$\uparrow\{ \sigma{\rightarrow}\tau \mid B_{\xi(\uparrow\sigma/x)} \vdash_s M{:}\tau \} =$ ($B_{\xi'} = B_\xi \backslash x$)
$\uparrow\{ \sigma{\rightarrow}\tau \mid B_{\xi'} \cup \{x{:}\mu \mid \mu \in \uparrow\sigma\} \vdash_s M{:}\tau \} =$
$\uparrow\{ \sigma{\rightarrow}\tau \mid B_{\xi'} \cup \{x{:}\sigma\} \vdash_s M{:}\tau \} =$ (4.1.5 (iii))
$\uparrow\{ \sigma{\rightarrow}\tau \mid B_{\xi'} \vdash_s \lambda x.M{:}\sigma{\rightarrow}\tau \} =$ (4.1.5 (v))
$\uparrow\{ \sigma{\rightarrow}\tau \mid B_\xi \vdash_s \lambda x.M{:}\sigma{\rightarrow}\tau \} =$ (4.1.5 (iii) & 4.2.2 (iv))
$\{ \rho \mid B_\xi \vdash_S \lambda x.M{:}\rho \}$. □

Corollary 4.2.8 If $M =_\beta N$ and $B \vdash_S M{:}\sigma$, then $B \vdash_S N{:}\sigma$, so the following rule is a derived rule in $\vdash_S$:

$$(=_\beta):\quad \frac{M{:}\sigma \qquad M =_\beta N}{N{:}\sigma}$$

Proof: Since $\mathcal{F}_S$ is a lambda model, we know that if $M =_\beta N$, then $[\![M]\!]_\xi = [\![N]\!]_\xi$; so $\{ \sigma \in \mathcal{T}_S \mid B_\xi \vdash_S M{:}\sigma \} = \{ \sigma \in \mathcal{T}_S \mid B_\xi \vdash_S N{:}\sigma \}$. □

Notice that, because of the way $\vdash_S$ is defined, corollary 4.2.8 will also hold if $\vdash_S$ is replaced by $\vdash_s$.

Example 4.2.9 By using lemmas 4.1.5 and 4.2.2 we can show the following:

i) If M is a closed term, then for all ξ, $[\![M]\!]_\xi = \{ \sigma \in \mathcal{T}_S \mid\ \vdash_S M{:}\sigma \}$. So for closed terms we can omit the subscript ξ.

ii) $[\![\lambda xy.xy]\!] = \uparrow\{ \rho{\rightarrow}\sigma{\rightarrow}\tau \mid \exists\, \sigma' \,[\, \rho \leq_S \sigma'{\rightarrow}\tau \;\&\; \sigma \leq_S \sigma' \,] \}$.

iii) $[\![\lambda x.x]\!] = \uparrow\{ \sigma{\rightarrow}\tau \mid \sigma \leq_S \tau \}$.

iv) $[\![(\lambda x.x)y]\!]_\xi = \xi(y)$.

If we take, for example, $\mu = (\sigma{\rightarrow}\tau){\rightarrow}\sigma{\cap}\rho{\rightarrow}\tau$, then it is easy to check that $\mu \in [\![\lambda xy.xy]\!]$ and $\mu \notin [\![\lambda x.x]\!]$, so $[\![\lambda xy.xy]\!]$ is not contained in $[\![\lambda x.x]\!]$.

Notice that, if M is a closed term, $[\![M]\!]$ is infinite. If M is not closed, it may be that $[\![M]\!]_\xi$ is finite, since ξ can select finite filters as well. However, we can limit $\mathcal{F}_S$ by selecting only infinite strict filters. Notice that this would still give us a lambda model different from $\mathcal{F}$.

Theorem 4.2.10 If M is in normal form, then there are B and σ such that $B \vdash_s M{:}\sigma$, and in this derivation ω does not occur.

Proof: By induction on the structure of lambda terms in normal form.

i) $M \equiv x$. Take σ strict, such that ω does not occur in σ. Then $\{x{:}\sigma\} \vdash_s x{:}\sigma$.

ii) $M \equiv \lambda x.M'$, with M' in normal form. By induction there are B, τ such that $B \vdash_s M'{:}\tau$ and ω does not occur in this derivation. In order to perform the ($\rightarrow$I)-step, B must contain (whether or not x is free in M') a statement with subject x and predicate, say, σ. But then, of course, $B\backslash x \vdash_s \lambda x.M'{:}\sigma{\rightarrow}\tau$, and ω does not occur in this derivation.

iii) $M \equiv xM_1 \ldots M_n$, with $M_1, \ldots, M_n$ in normal form. By induction there are $B_1, \ldots, B_n$ and $\sigma_1, \ldots, \sigma_n$ such that for every $1 \leq i \leq n$ $B_i \vdash_s M_i{:}\sigma_i$, and ω does not occur in these derivations. Take τ strict, such that ω does not occur in τ, and

$$B = \Pi\{B_1, \ldots, B_n, \{x{:}\sigma_1{\rightarrow}\cdots{\rightarrow}\sigma_n{\rightarrow}\tau\}\}.$$

Then $B \vdash_s xM_1 \ldots M_n{:}\tau$, and in this derivation ω does not occur. □

Theorem 4.2.11 If M is in head normal form, then there are B and σ such that $B \vdash_s M{:}\sigma$.

Proof: By induction on the structure of lambda terms in head normal form.

i) $M \equiv x$. Take σ strict, then $\{x{:}\sigma\} \vdash_s x{:}\sigma$.

ii) $M \equiv \lambda x.M'$, with M' in head normal form. By induction there are B and τ such that $B \vdash_s M'{:}\tau$. As in the previous theorem, B must contain a statement with subject x and predicate, say, σ. But then, of course, $B\backslash x \vdash_s \lambda x.M'{:}\sigma{\rightarrow}\tau$.

iii) $M \equiv xM_1 \ldots M_n$, with $M_1, \ldots, M_n$ lambda terms. Take τ strict, then also (with n times ω) $\omega{\rightarrow}\omega{\rightarrow}\cdots{\rightarrow}\omega{\rightarrow}\tau$ is strict, and $\{x{:}\omega{\rightarrow}\omega{\rightarrow}\cdots{\rightarrow}\omega{\rightarrow}\tau\} \vdash_s xM_1 \ldots M_n{:}\tau$. □

Theorem 4.2.12 $\exists\, B, \sigma\, [\, B \vdash_s M{:}\sigma \;\&\; B, \sigma\ \omega\text{-free} \,] \iff M$ has a normal form.

Proof: $\Rightarrow$) If $B \vdash_s M{:}\sigma$ and B, σ ω-free, then $B \vdash_{BCD} M{:}\sigma$ and B, σ ω-free. Then by property 2.3.4 (iv) M has a normal form.

$\Leftarrow$) By theorem 4.2.10, and corollary 4.2.8. □

Notice that in the second part of the proof, because of corollary 4.2.8 we can only state that if $M =_\beta N$ and $B \vdash_s M{:}\sigma$, then $B \vdash_s N{:}\sigma$. From theorem 4.2.10 we can conclude that B and σ do not contain ω, but the property that ω does not occur at all in the derivation is, in general, lost. (See also the remark after lemma 7.4.2.)

Theorem 4.2.13 $\exists\, B, \sigma\, [\, B \vdash_s M{:}\sigma \,] \iff M$ has a head normal form.

Proof: $\Rightarrow$) If $B \vdash_s M{:}\sigma$, then $B \vdash_{BCD} M{:}\sigma$ and $\sigma \neq \omega$. Then by property 2.3.4 (iii), M has a head normal form.

$\Leftarrow$) By theorem 4.2.11 and corollary 4.2.8. □

Corollary 4.2.14 i) $\exists\, B, \sigma\, [\, B \vdash_S M{:}\sigma \;\&\; B, \sigma\ \omega\text{-free} \,] \iff M$ has a normal form.

ii) $\exists\, B, \sigma\, [\, B \vdash_S M{:}\sigma \;\&\; \sigma \neq \omega \,] \iff M$ has a head normal form. □

4.3 *The relation between* $\vdash_S$ *and* $\vdash_{BCD}$

The BCD-type assignment is not conservative over the strict type assignment. So the following does not hold:

Suppose all types occurring in B and σ are in $\mathcal{T}_S$. Then $B \vdash_S M{:}\sigma \Leftrightarrow B \vdash_{BCD} M{:}\sigma$.

As a counter example for $\Leftarrow$, take $\{x{:}\sigma{\rightarrow}\sigma\} \vdash_{BCD} x{:}\sigma{\cap}\tau{\rightarrow}\sigma$. It is not possible to derive the statement $x{:}\sigma{\cap}\tau{\rightarrow}\sigma$ from the basis $\{x{:}\sigma{\rightarrow}\sigma\}$ in $\vdash_S$. Of course the implication in the other direction holds: $B \vdash_S M{:}\sigma$ implies $B \vdash_{BCD} M{:}\sigma$. The relation between the two systems is stronger, however. The strict system turns out to be the nucleus of the BCD-system: we will show that, if in a derivation for $M{:}\sigma$ the derivation rule $(\leq)$ is used, the same statement can be derived using a derivation in which the derivation rule $(\leq)$ is at the most only used at the very end of the derivation (theorem 4.3.5). The proof is based on the fact that for every $\sigma \in \mathcal{T}_{BCD}$ there is a $\sigma^* \in \mathcal{T}_S$ such that $\sigma \sim \sigma^*$ (lemma 4.3.1; the same result was stated in [Hindley '82], §4), and the approximation theorem as given in [Ronchi della Rocca & Venneri '84].

Lemma 4.3.1 (cf. [Hindley '82]) For every $\sigma \in \mathcal{T}_{BCD}$ there is a $\sigma^* \in \mathcal{T}_S$ such that $\sigma \sim \sigma^*$.

Proof: By induction on the structure of types in $\mathcal{T}_{BCD}$.

i) $\sigma = \omega$, or σ is a type-variable: trivial.

ii) $\sigma = \rho{\rightarrow}\tau$. By induction there are ρ^* and $\tau^* \in \mathcal{T}_S$ such that $\rho \sim \rho^*$ and $\tau \sim \tau^*$.
   a) $\tau^* = \omega$. Take $\sigma^* = \omega$.
   b) $\tau^* = \tau_1{\cap}\cdots{\cap}\tau_m$, each $\tau_i \in \mathcal{T}_s$. Take $\sigma^* = (\rho^*{\rightarrow}\tau_1){\cap}\cdots{\cap}(\rho^*{\rightarrow}\tau_n)$.
   c) τ^* is strict, then take $\sigma^* = \rho^*{\rightarrow}\tau^*$.

iii) $\sigma = \rho{\cap}\tau$. By induction there are ρ^* and $\tau^* \in \mathcal{T}_S$ such that $\rho \sim \rho^*$ and $\tau \sim \tau^*$.
   a) $\rho^* = \omega$. Take $\sigma^* = \tau^*$.
   b) $\tau^* = \omega$. Take $\sigma^* = \rho^*$.
   c) $\rho^* \neq \omega$ & $\tau^* \neq \omega$. Take $\sigma^* = \rho^*{\cap}\tau^*$. □

Notice that lemma 4.3.1 is not a proof for the statement that $\mathcal{T}_S$ modulo $\sim_S$ is isomorphic to $\mathcal{T}_{BCD}$ modulo $\sim$. For example, take $\sigma_1 = \sigma{\rightarrow}\tau$ and $\sigma_2 = (\sigma{\rightarrow}\tau){\cap}(\sigma{\cap}\rho{\rightarrow}\tau)$; then $\sigma_1 \sim \sigma_2$, $\sigma_1{}^* = \sigma_1$, $\sigma_2{}^* = \sigma_2$, but not $\sigma_1 \sim_S \sigma_2$.

As for the CDV-system and the BCD-system, the type assignment rules of the strict system are generalized to elements of $\mathcal{N}$ by allowing for the terms to be elements of $\Lambda\bot$. Notice that, because strict type assignment is syntax directed, if $\bot$ occurs in a term M and $B \vdash_S M{:}\sigma$, then either $\sigma = \omega$, or in the derivation for $M{:}\sigma$, $\bot$ appears in the right hand subterm of an application on which the derivation rule $({\rightarrow}E)$ is used with $n = 0$.

We will now prove the main theorem of this section by showing that the $\vdash_S$-system is in fact the nucleus of the BCD-system. We will do this by proving first, for terms in $\mathcal{N}$, that the derivation rule $(\leq)$ can be transferred to the very end of a derivation, and afterwards generalizing this result to arbitrary lambda terms.

Theorem 4.3.2 If A is in $\lambda\bot$-normal form and $B \vdash_{BCD} A{:}\sigma$ then there are $B', \sigma' \in \mathcal{T}_S$ such that $B' \vdash_S A{:}\sigma'$, $\sigma' \leq \sigma$ and $B' \geq B$.

Proof: The proof is given by induction on the structure of terms in $\lambda\bot$-normal form.

All cases where $\sigma \sim \omega$ are trivial, because then we can take $B' = \emptyset$ and $\sigma' = \omega$. Therefore, in the rest of the proof, we will assume $\sigma \neq \omega$.

i) $B \vdash_{BCD} x{:}\sigma$. By property 2.3.4 (v) there is an $x{:}\rho \in B$ such that $\rho \leq \sigma$. By lemma 4.3.1 $\{x{:}\rho^*\} \geq B$ and $\rho^* \leq \sigma$.

ii) $B \vdash_{BCD} \lambda x.A'{:}\sigma$, with $A' \neq \bot$. Then there are $\rho_1, \ldots, \rho_n, \mu_1, \ldots, \mu_n$ such that

$$\sigma = (\rho_1{\rightarrow}\mu_1){\cap}\cdots{\cap}(\rho_n{\rightarrow}\mu_n).$$

So, by $({\cap}E)$ and property 2.3.4 (ii) for every $1 \leq i \leq n$ we have $B \cup \{x{:}\rho_i\} \vdash_{BCD} A'{:}\mu_i$. By induction for $1 \leq i \leq n$ there are B_i and $\rho_i{}', \mu_i{}' \in \mathcal{T}_S$ such that

$$B_i \cup \{x{:}\rho_i{}'\} \vdash_S A'{:}\mu_i{}',\ \mu_i{}' \leq \mu_i \text{ and } B_i \cup \{x{:}\rho_i{}'\} \geq B \cup \{x{:}\rho_i\}.$$

We can assume, without loss of generality, that each $\mu_i{}'$ is an element of $\mathcal{T}_s$. Then:

$$\forall\, 1 \leq i \leq n\ B_i \vdash_s \lambda x.A'{:}\rho_i{}'{\rightarrow}\mu_i{}',\ \rho_i{}'{\rightarrow}\mu_i{}' \leq \rho_i{\rightarrow}\mu_i \text{ and } B_i \geq B.$$

So $\Pi\{B_1, \ldots, B_n\} \vdash_S \lambda x.A'{:}(\rho_1{}'{\rightarrow}\mu_1{}'){\cap}\cdots{\cap}(\rho_n{}'{\rightarrow}\mu_n{}')$, $\Pi\{B_1, \ldots, B_n\} \geq B$, and $(\rho_1{}'{\rightarrow}\mu_1{}'){\cap}\cdots{\cap}(\rho_n{}'{\rightarrow}\mu_n{}') \leq \sigma$.

iii) $B \vdash_{BCD} xA_1 \ldots A_n{:}\sigma$. By property 2.3.4 (i) there are $\tau_1, \ldots, \tau_n \in \mathcal{T}_{BCD}$ such that

$$B \vdash_{BCD} x{:}\tau_1{\rightarrow}\cdots{\rightarrow}\tau_n{\rightarrow}\sigma, \text{ and for every } 1 \leq i \leq n\ B \vdash_{BCD} A_i{:}\tau_i.$$

By induction for $1 \leq i \leq n$ there are B_i, τ_i such that $B_i \vdash_S A_i{:}\tau_i{}'$, $\tau_i{}' \leq \tau_i$ and $B_i \geq B$. Also $\Pi\{B_1, \ldots, B_n\} \geq B$.
Let $\sigma^* = \sigma_1{\cap}\cdots{\cap}\sigma_k$ where each $\sigma_i \in \mathcal{T}_s$ and $k \geq 1$. By lemma 4.3.1

$$\tau_1{\rightarrow}\cdots{\rightarrow}\tau_n{\rightarrow}\sigma \leq (\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_1){\cap}\cdots{\cap}(\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_k)$$

and because of property 2.3.4 (v), we have

$$\Pi\{B_1, \ldots, B_n, \{x{:}(\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_1){\cap}\cdots{\cap}(\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_k)\}\} \geq B.$$

For every $1 \leq f \leq k$ $\Pi\{B_1, \ldots, B_n, \{x{:}\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_f\}\} \vdash_S xA_1 \ldots A_n{:}\sigma_f$, so

$$\Pi\{B_1, \ldots, B_n, \{x{:}(\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_1){\cap}\cdots{\cap}(\tau_1{}'{\rightarrow}\cdots{\rightarrow}\tau_n{}'{\rightarrow}\sigma_k)\}\} \vdash_S xA_1 \ldots A_n{:}\sigma'.$$ □

Lemma 4.3.3 i) If $A_1, A_2 \in \mathcal{A}(M)$, then there is an A' such that A_1 and A_2 both match A' except for occurrences of $\bot$, and $A' \in \mathcal{A}(M)$.

ii) If for every $1 \leq i \leq n$ there is an $A \in \mathcal{A}(M)$ such that $B_i \vdash_S A{:}\sigma_i$, then there is an $A \in \mathcal{A}(M)$ such that for $1 \leq i \leq n$ $B_i \vdash_S A{:}\sigma_i$.

Proof: i) This is a consequence of proposition 10.2.2, in [Barendregt '84].

ii) If for every $1 \leq i \leq n$ there is an $A_i \in \mathcal{A}(M)$ such that $B_i \vdash_S A_i{:}\sigma_i$, then by part (i) there is an $A \in \mathcal{A}(M)$, such that for every $1 \leq i \leq n$, A_i matches A except for occurrences of $\bot$. Since $\bot$ occurs only in subterms that are typed with ω, also for every $1 \leq i \leq n$, $B_i \vdash_S A{:}\sigma_i$. □

Theorem 4.3.4 $B \vdash_S M{:}\sigma \Leftrightarrow \exists\, A \in \mathcal{A}(M)\ [\ B \vdash_S A{:}\sigma\ ]$.

Proof: $\Rightarrow$) By straightforward induction on the structure of derivations, using lemma 4.3.3 (ii).

$\Leftarrow$) If $B \vdash_{\mathrm{S}} A{:}\sigma$, then by the remark made before theorem 4.3.2, $\perp$ appears only in subterms that are typed by ω. Since $A \in \mathcal{A}(M)$, there is an M' such that $M' =_\beta M$ and A matches M' except for occurrences of $\perp$. Then obviously $B \vdash_{\mathrm{S}} M'{:}\sigma$, and by corollary 4.2.8 also $B \vdash_{\mathrm{S}} M{:}\sigma$. $\square$

Theorem 4.3.5 If $B \vdash_{\mathrm{BCD}} M{:}\sigma$, then there are $B', \sigma' \in \mathcal{T}_{\mathrm{S}}$ such that $B' \vdash_{\mathrm{S}} M{:}\sigma'$, $\sigma' \leq \sigma$, and $B' \geq B$.

Proof: $B \vdash_{\mathrm{BCD}} M{:}\sigma \;\Rightarrow$ (2.3.2.7)
$\exists\, A \in \mathcal{A}(M)\ [\ B \vdash_{\mathrm{BCD}} A{:}\sigma\ ] \;\Rightarrow$ (4.3.2)
$\exists\, A \in \mathcal{A}(M), B', \sigma' \in \mathcal{T}_{\mathrm{S}}\ [\ B' \vdash_{\mathrm{S}} A{:}\sigma'\ \&\ \sigma' \leq \sigma\ \&\ B' \geq B\ ] \;\Rightarrow$ (4.3.4)
$\exists\, B', \sigma' \in \mathcal{T}_{\mathrm{S}}\ [\ B' \vdash_{\mathrm{S}} M{:}\sigma'\ \&\ \sigma' \leq \sigma\ \&\ B' \geq B\ ]$. $\square$

4.4 *Soundness and completeness of strict type assignment*

In this section we will prove completeness for the strict system. Recall definitions 2.3.1.2 and 2.3.1.3.

As shown in [Barendregt *et al.* '83], the BCD-type assignment is sound and complete with respect to the simple type semantics. In this section we will show that soundness is lost if – instead of simple type semantics – the inference type semantics as defined in [Mitchell '88] is used. By using the latter, we are able to prove soundness and completeness without having the necessity of introducing $\leq$. This is done in a way very similar to the one used in [Barendregt *et al.* '83], using the strict filter lambda model $\mathcal{F}_{\mathrm{S}}$.

At one very crucial point, the completeness proof in this section differs from the one in [Barendregt *et al.* '83]. In this paper, the simple type semantic was inductively defined, whereas our approach will be to give a map from $\mathcal{T}_{\mathrm{S}}$ to $\wp(\mathcal{F}_{\mathrm{S}})$ and prove that it is a type interpretation. It will be a different kind of type interpretation than the one used in [Barendregt *et al.* '83], because the latter would not suffice in our case.

Theorem 4.4.1 *Soundness.* $B \vdash_{\mathrm{S}} M{:}\sigma \;\Rightarrow\; B \models M{:}\sigma$.

Proof: By induction on the structure of derivations. $\square$

The notion of derivability $\vdash_{\mathrm{BCD}}$ is not sound for $\models$. Take, for example, the statement $\lambda x.x{:}(\sigma{\to}\sigma){\to}\sigma{\cap}\tau{\to}\sigma$. This statement is derivable in the system $\vdash_{\mathrm{BCD}}$, but it is not valid in the strict filter lambda model.

Definition 4.4.2 i) We define a map $\upsilon_0 : \mathcal{T}_{\mathrm{S}} \to \wp(\mathcal{F}_{\mathrm{S}})$ by $\upsilon_0(\sigma) = \{\ d \in \mathcal{F}_{\mathrm{S}} \mid \sigma \in d\ \}$.
ii) $\xi_B(x) = \{\ \sigma \in \mathcal{T}_{\mathrm{S}} \mid B \vdash_{\mathrm{S}} x{:}\sigma\ \}$.

Theorem 4.4.3 The map υ_0 is a type interpretation.

Proof: We check condition (a) of definition 2.3.1.2 (i). (Condition (b) is easy, (c) is trivial.)
$\forall\, e\ [\ e \in \upsilon_0(\sigma) \Rightarrow d \cdot e \in \upsilon_0(\tau)\ ] \;\Rightarrow$
$\forall\, e\ [\ e \in \upsilon_0(\sigma) \Rightarrow \varepsilon \cdot d \cdot e \in \upsilon_0(\tau)\ ] \;\Rightarrow$ (take $e = \uparrow\!\sigma$)
$\tau \in \varepsilon \cdot d \cdot \uparrow\!\sigma \;\Rightarrow$ (4.2.2 (ii))

$\exists\, \rho \in \uparrow\!\sigma, \nu \in d, \eta\ [\ \nu \leq_{\mathrm{S}} \eta{\to}\tau\ \&\ \rho \leq_{\mathrm{S}} \eta\ ] \;\Rightarrow$ (4.2.2 (iii))
$\exists\, \nu \in d, \eta\ [\ \nu \leq_{\mathrm{S}} \eta{\to}\tau\ \&\ \sigma \leq_{\mathrm{S}} \eta\ ] \;\Rightarrow$
$\sigma{\to}\tau \in \uparrow\!\{\ \rho{\to}\mu \mid \exists\, \nu \in d, \eta\ [\ \nu \leq_{\mathrm{S}} \eta{\to}\mu\ \&\ \rho \leq_{\mathrm{S}} \eta\ ]\ \} \;\Rightarrow$
$\sigma{\to}\tau \in \uparrow\!\{\ \eta \mid \exists\, \nu \in d\ [\ \nu{\to}\eta \in \varepsilon\ ]\ \} \;\Rightarrow$
$\varepsilon \cdot d \in \upsilon_0(\sigma{\to}\tau)$. $\square$

Notice that, if part (iv.b) of definition 4.1.1 is omitted, the sets $\upsilon_0(\sigma{\cap}\tau{\to}\sigma)$ and $\upsilon_0(\tau{\cap}\sigma{\to}\sigma)$ are incompatible. Then, we can only show that both contain

$$\{\ \varepsilon \cdot d \mid \forall\, e\ [\ e \in \upsilon_0(\sigma) \cap \upsilon_0(\tau) \;\Rightarrow\; d \cdot e \in \upsilon_0(\sigma)\ ]\ \}$$

and that they are both contained in

$$\{\ d \mid \forall\, e\ [\ e \in \upsilon_0(\sigma) \cap \upsilon_0(\tau) \;\Rightarrow\; d \cdot e \in \upsilon_0(\sigma)\ ]\ \}.$$

If the relation $\sim_{\mathrm{S}}$ is defined as in definition 4.1.1 then $\sigma{\cap}\tau{\to}\sigma \sim_{\mathrm{S}} \tau{\cap}\sigma{\to}\sigma$. Since filters are closed for $\sim_{\mathrm{S}}$, $\sigma{\cap}\tau{\to}\sigma \in \uparrow\!\tau{\cap}\sigma{\to}\sigma$ and $\tau{\cap}\sigma{\to}\sigma \in \uparrow\!\sigma{\cap}\tau{\to}\sigma$, so $\upsilon_0(\sigma{\cap}\tau{\to}\sigma) = \upsilon_0(\tau{\cap}\sigma{\to}\sigma)$.

Lemma 4.4.4 i) $B \vdash_{\mathrm{S}} M{:}\sigma$ if and only if $B_{\xi_B} \vdash_{\mathrm{S}} M{:}\sigma$.
ii) $\mathcal{F}_{\mathrm{S}}, \xi_B, \upsilon_0 \models B$.

Proof: i) Because for every x, $\xi_B(x)$ is a strict filter.
ii) $x{:}\sigma \in B \;\Rightarrow$ (i) $\sigma \in \{\ \tau \mid B_{\xi_B} \vdash_{\mathrm{S}} x{:}\tau\ \} \;\Rightarrow\; \sigma \in [\![x]\!]_{\xi_B}$.
So $[\![x]\!]_{\xi_B} \in \{\ d \in \mathcal{F}_{\mathrm{S}} \mid \sigma \in d\ \} = \upsilon_0(\sigma)$. $\square$

The system of [Barendregt *et al.* '83] proved to be complete with respect to the simple type semantics. The system $\vdash_{\mathrm{S}}$, however, is not complete in this semantics, due to the fact that, if we take υ to be a type interpretation from $\mathcal{T}_{\mathrm{S}}$ to $\wp(\mathcal{F}_{\mathrm{S}})$, the set

$$\{\ d \mid \forall\, e\ [\ e \in \upsilon(\sigma) \;\Rightarrow\; d \cdot e \in \upsilon(\tau)\ ]\ \}$$

is not contained in $\upsilon(\sigma{\to}\tau)$, since we don't allow for ω or an intersection type scheme at the right hand side of an arrow type scheme. If, instead, we use the notion of type interpretation as defined in definition 2.3.1.2 (i), because of theorem 4.4.3, completeness can be proved.

Theorem 4.4.5 *Completeness.* Let $\sigma \in \mathcal{T}_{\mathrm{S}}$, then $B \models M{:}\sigma \;\Rightarrow\; B \vdash_{\mathrm{S}} M{:}\sigma$.

Proof: $B \models M{:}\sigma \;\Rightarrow$ (2.3.1.3 (ii.c.1), 4.4.4 (ii) & 4.4.3)
$\mathcal{F}_{\mathrm{S}}, \xi_B, \upsilon_0 \models M{:}\sigma \;\Rightarrow$ (2.3.1.3 (ii.a) & 4.4.3)
$[\![M]\!]_{\xi_B} \in \upsilon_0(\sigma) \;\Rightarrow$ (4.4.2 (i))
$\sigma \in [\![M]\!]_{\xi_B} \;\Rightarrow$ (4.2.7)
$B_{\xi_B} \vdash_{\mathrm{S}} M{:}\sigma \;\Rightarrow$ (4.4.4 (i))
$B \vdash_{\mathrm{S}} M{:}\sigma$. $\square$

Chapter 5 *The Essential Intersection Type Assignment System*

The strict system is not closed under η-reduction. So, the following does not hold:

$$B \vdash_{\mathrm{S}} M{:}\sigma \;\&\; M \rightarrow_{\eta} N \;\Rightarrow\; B \vdash_{\mathrm{S}} N{:}\sigma.$$

For example, take as in example 4.2.9 the terms $\lambda xy.xy$ and $\lambda x.x$, and notice that $\lambda xy.xy \rightarrow_{\eta} \lambda x.x$. It is easy to check that:

$$\vdash_{\mathrm{S}} \lambda xy.xy{:}(\sigma\rightarrow\tau)\rightarrow\sigma\cap\rho\rightarrow\tau \text{ and not } \vdash_{\mathrm{S}} \lambda x.x{:}(\sigma\rightarrow\tau)\rightarrow\sigma\cap\rho\rightarrow\tau.$$

In this chapter, we will show that the straightforward extension of $\leq_{\mathrm{S}}$ to a relation that is also defined over arrow-types is sufficient to create a system that is closed under η-reduction. We call this notion of type assignment *essential*, to emphasize that it is the smallest restriction (that is not strict) of the BCD-system that satisfies all its properties. We will compare this notion of type assignment with the one defined in [Hindley '82], and prove a completeness result.

5.1 *Essential type assignment*

The relation $\leq_{\mathrm{E}}$ on $\mathcal{T}_{\mathrm{S}}$ is a natural extension of the relation $\leq_{\mathrm{S}}$, that was only defined for intersection types. Notice that, as in the relation $\leq$, in the definition of $\leq_{\mathrm{E}}$, the arrow type constructor is contravariant in its left argument and covariant in its right argument.

Definition 5.1.1 i) We define the relation $\leq_{\mathrm{E}}$ on $\mathcal{T}_{\mathrm{S}}$ like $\leq_{\mathrm{S}}$, but add the last alternative.

a) $\sigma \leq_{\mathrm{E}} \sigma$.
b) $\sigma \leq_{\mathrm{E}} \omega$.
c) $\sigma\cap\tau \leq_{\mathrm{E}} \sigma \;\&\; \sigma\cap\tau \leq_{\mathrm{E}} \tau$.
d) $\sigma \leq_{\mathrm{E}} \tau \leq_{\mathrm{E}} \rho \Rightarrow \sigma \leq_{\mathrm{E}} \rho$.
e) $\sigma \leq_{\mathrm{E}} \rho \;\&\; \sigma \leq_{\mathrm{E}} \tau \Rightarrow \sigma \leq_{\mathrm{E}} \rho\cap\tau$.
f) $\rho \leq_{\mathrm{E}} \sigma \;\&\; \tau \leq_{\mathrm{E}} \mu \Rightarrow \sigma\rightarrow\tau \leq_{\mathrm{E}} \rho\rightarrow\mu$.

ii) On $\mathcal{T}_{\mathrm{S}}$, the relation $\sim_{\mathrm{E}}$ is defined by: $\sigma \sim_{\mathrm{E}} \tau \Leftrightarrow \sigma \leq_{\mathrm{E}} \tau \leq_{\mathrm{E}} \sigma$.

$\mathcal{T}_{\mathrm{S}}$ may be considered modulo $\sim_{\mathrm{E}}$. As before, $\leq_{\mathrm{E}}$ becomes a partial order, and in this chapter we consider types modulo $\sim_{\mathrm{E}}$.

As with the relations $\leq$ (definition 2.3.2) and $\leq_{\mathrm{S}}$, we extend the relation $\leq_{\mathrm{E}}$ to bases.

Lemma 5.1.2 For the relation $\leq_{\mathrm{E}}$, the following properties hold:

i) $\sigma \leq_{\mathrm{S}} \tau \Rightarrow \sigma \leq_{\mathrm{E}} \tau$.
ii) $\varphi \leq_{\mathrm{E}} \sigma \Leftrightarrow \sigma = \varphi$. So $\{\, \sigma \mid \sigma \sim_{\mathrm{E}} \varphi \,\} = \{\varphi\}$.
iii) $\omega \leq_{\mathrm{E}} \sigma \Leftrightarrow \sigma = \omega$. So $\{\, \sigma \mid \sigma \sim_{\mathrm{E}} \omega \,\} = \{\omega\}$.

iv) $\sigma\rightarrow\tau \leq_{\mathrm{E}} \rho \in \mathcal{T}_{\mathrm{s}} \Leftrightarrow \exists\, \alpha \in \mathcal{T}_{\mathrm{S}}, \beta \in \mathcal{T}_{\mathrm{s}}\, [\, \rho = \alpha\rightarrow\beta \;\&\; \alpha \leq_{\mathrm{E}} \sigma \;\&\; \tau \leq_{\mathrm{E}} \beta \,]$.
v) $\sigma_1\cap\cdots\cap\sigma_n \leq_{\mathrm{E}} \tau \in \mathcal{T}_{\mathrm{s}} \Rightarrow \exists\, 1 \leq i \leq n\, [\, \sigma_i \leq_{\mathrm{E}} \tau \,]$.

Proof: Easy. □

Using this lemma, we can prove the following:

Lemma 5.1.3 i) $\sigma_1\cap\cdots\cap\sigma_n \leq_{\mathrm{E}} \tau \in \mathcal{T}_{\mathrm{S}} \Rightarrow$
$\exists\, \tau_1, \ldots, \tau_m \in \mathcal{T}_{\mathrm{s}}\, [\, \tau = \tau_1\cap\cdots\cap\tau_m \;\&\; \forall\, 1 \leq j \leq m\; \exists\, 1 \leq i \leq n\, [\, \sigma_i \leq_{\mathrm{E}} \tau_j \,]\,]$.
ii) $\sigma \leq_{\mathrm{E}} \tau \;\&\; \sigma \in \mathcal{T}_{\mathrm{s}} \Rightarrow$
$\exists\, \tau_1, \ldots, \tau_m\; (m \geq 0)\, [\, \tau = \tau_1\cap\cdots\cap\tau_m \;\&\; \forall\, 1 \leq j \leq m\, [\, \sigma \leq_{\mathrm{E}} \tau_j \,]\,]$.
iii) $B' \leq_{\mathrm{E}} B \leq_{\mathrm{S}} \{x{:}\sigma\} \;\&\; \sigma \in \mathcal{T}_{\mathrm{s}} \Rightarrow \exists\, \sigma' \in \mathcal{T}_{\mathrm{s}}\, [\, B' \leq_{\mathrm{S}} \{x{:}\sigma'\} \;\&\; \sigma' \leq_{\mathrm{E}} \sigma \,]$.

Proof: Easy. □

Notice that $\sim_{\mathrm{S}}$ is a true subrelation of $\sim_{\mathrm{E}}$, since for example $\sigma\rightarrow\tau \sim_{\mathrm{E}} (\sigma\rightarrow\tau)\cap(\sigma\cap\rho\rightarrow\tau)$, but this does not hold in $\sim_{\mathrm{S}}$.

The essential type assignment system is constructed from the set of strict types, and a minor extension of the derivation rules as in definition 4.1.3 (i). This way, a system is obtained that satisfies the main properties of the BCD- and the strict system, which is not surprising: type assignment is closed under β-equality, the set of terms typeable with type σ from a basis B such that ω does not occur in B and σ is the set of normalizable terms, and the set of terms typeable with type $\sigma \neq \omega$ is the set of terms having a head normal form.

Definition 5.1.4 i) *Essential type assignment* and *essential derivations* are defined by the following natural deduction system (where all types displayed are strict, except σ in the derivation rules $(\rightarrow\mathrm{I})$ and $(\leq_{\mathrm{E}})$):

$$(\rightarrow\mathrm{I}): \quad \frac{\begin{array}{c}[x{:}\sigma]\\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\rightarrow\tau}\ \text{(a)} \qquad\qquad (\leq_{\mathrm{E}}): \quad \frac{x{:}\sigma \qquad \sigma \leq_{\mathrm{E}} \tau}{x{:}\tau}$$

$$(\rightarrow\mathrm{E}): \quad \frac{M{:}\sigma_1\cap\cdots\cap\sigma_n\rightarrow\tau \qquad N{:}\sigma_1 \;\ldots\; N{:}\sigma_n}{MN{:}\tau}\ (n \geq 0)$$

(a) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

If $M{:}\sigma$ is derivable from B using an essential derivation, we write $B \vdash_{\mathrm{e}} M{:}\sigma$.

ii) We define $\vdash_{\mathrm{E}}$ by: $B \vdash_{\mathrm{E}} M{:}\sigma$ if and only if: there are $\sigma_1, \ldots, \sigma_n$ $(n \geq 0)$ such that $\sigma = \sigma_1\cap\cdots\cap\sigma_n$, and for every $1 \leq i \leq n$, $B \vdash_{\mathrm{e}} M{:}\sigma_i$.

Notice that derivation rule $(\cap\mathrm{E})$ from the strict system is not required in this definition, since it is derivable from $(\leq_{\mathrm{E}})$.

The difference between the strict system and the essential system does not lie in the set of types that can occur in derivations, but only in the extension of the derivation rule $(\cap\mathrm{E})$ to the one for $(\leq_{\mathrm{E}})$.

For this notion of type assignment, the following properties hold:

Lemma 5.1.5 i) $B \vdash_e MN{:}\sigma \iff \exists\, \tau \in \mathcal{T}_S\ [\ B \vdash_e M{:}\tau{\rightarrow}\sigma\ \&\ B \vdash_E N{:}\tau\ ]$.

ii) $B \vdash_e \lambda x.M{:}\sigma \iff \exists\, \rho \in \mathcal{T}_S, \mu \in \mathcal{T}_s\ [\ \sigma = \rho{\rightarrow}\mu\ \&\ B \cup \{x{:}\rho\} \vdash_e M{:}\mu\ ]$.

iii) $B \vdash_E M{:}\sigma \iff \exists\, \sigma_1, \ldots, \sigma_n\ (n \geq 0)\ [\ \sigma = \sigma_1 \cap \cdots \cap \sigma_n\ \&\ \forall\, 1 \leq i \leq n\ [\ B \vdash_e M{:}\sigma_i\ ]\ ]$.

iv) $B \vdash_E x{:}\sigma \iff \exists\, \rho \in \mathcal{T}_S\ [\ x{:}\rho \in B\ \&\ \rho \leq_E \sigma\ ]$.

v) $B \vdash_E M{:}\sigma\ \&\ B' \leq_E B \Rightarrow B' \vdash_E M{:}\sigma$.

Proof: Easy. □

Although the derivation rule $(\leq_E)$ is not allowed on all terms, we can prove the following:

Lemma 5.1.6 If $B \vdash_E M{:}\sigma$, and $\sigma \leq_E \tau$, then $B \vdash_E M{:}\tau$.

Proof: By induction on $\vdash_E$.

i) $\sigma = \omega$. Then by lemma 5.1.2 (iii) $\tau = \omega$. Obviously $B \vdash_E M{:}\tau$.

ii) $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$. Then by lemma 5.1.5 (iii), for every $1 \leq i \leq n$, $B \vdash_e M{:}\sigma_i$. By lemma 5.1.3 (i) there are $\tau_1, \ldots, \tau_m \in \mathcal{T}_s$ such that $\tau = \tau_1 \cap \cdots \cap \tau_m$, and for every $1 \leq j \leq m$ there is a $1 \leq i \leq n$ such that $\sigma_i \leq_E \tau_j$. By induction for every $1 \leq j \leq m$ $B \vdash_e M{:}\tau_j$. But then $B \vdash_E M{:}\tau$.

iii) $\sigma \in \mathcal{T}_s$. This part is proved by induction on M.

a) $M \equiv x$. Then $B \leq_E \{x{:}\sigma\} \leq_E \{x{:}\tau\}$, so by lemma 5.1.5 (iv) $B \vdash_E x{:}\tau$.

b) $M \equiv \lambda x.M'$. Then by lemma 5.1.5 (ii), there are $\rho \in \mathcal{T}_S$, $\mu \in \mathcal{T}_s$ such that $\sigma = \rho{\rightarrow}\mu$, and $B \cup \{x{:}\rho\} \vdash_e M'{:}\mu$. By lemmas 5.1.3 (ii) and 5.1.2 (iv), there are $\rho_1, \ldots, \rho_n, \mu_1, \ldots, \mu_n$ such that $\tau = (\rho_1{\rightarrow}\mu_1) \cap \cdots \cap (\rho_n{\rightarrow}\mu_n)$, and for $1 \leq i \leq n$ $\rho_i \leq_E \rho$ and $\mu \leq_E \mu_i$. By lemma 5.1.5 (v), for $1 \leq i \leq n$ $B \cup \{x{:}\rho_i\} \vdash_e M'{:}\mu$, and by induction $B \cup \{x{:}\rho_i\} \vdash_e M'{:}\mu_i$. So by lemma 5.1.5 (ii), for every $1 \leq i \leq n$ $B \vdash_e \lambda x.M'{:}\rho_i{\rightarrow}\mu_i$, so $B \vdash_E \lambda x.M'{:}\tau$.

c) $M \equiv M_1 M_2$. Then by lemma 5.1.5 (i), there is a $\mu \in \mathcal{T}_S$ such that $B \vdash_e M_1{:}\mu{\rightarrow}\sigma$, and $B \vdash_E M_2{:}\mu$. Since $\sigma \leq_E \tau$, also $\mu{\rightarrow}\sigma \leq_E \mu{\rightarrow}\tau$, and by induction $B \vdash_e M_1{:}\mu{\rightarrow}\tau$. But then, by lemma 5.1.5 (i), $B \vdash_E M_1 M_2{:}\tau$. □

Now it is easy to prove that type assignment in this system is closed under η-reduction.

Theorem 5.1.7 If $B \vdash_E M{:}\sigma$ and $M \rightarrow_\eta N$, then $B \vdash_E N{:}\sigma$.

Proof: We will only show the part $\sigma \in \mathcal{T}_s$. The proof is completed by induction on the definition of $\rightarrow_\eta$, of which we only show the part $\lambda x.Mx \rightarrow_\eta M$, where x does not occur free in M. The other parts are dealt with by straightforward induction.

$B \vdash_e \lambda x.Mx{:}\sigma \Rightarrow$ (5.1.5 (ii))
$\exists\, \rho, \mu\ [\ \sigma = \rho{\rightarrow}\mu\ \&\ B \cup \{x{:}\rho\} \vdash_e Mx{:}\mu\ ] \Rightarrow$ (5.1.5 (i))
$\exists\, \tau, \rho, \mu\ [\ \sigma = \rho{\rightarrow}\mu\ \&\ B \cup \{x{:}\rho\} \vdash_e M{:}\tau{\rightarrow}\mu\ \&\ B \cup \{x{:}\rho\} \vdash_E x{:}\tau\ ] \Rightarrow$
(5.1.5 (iv), x not in M)
$\exists\, \tau, \rho, \mu\ [\ \sigma = \rho{\rightarrow}\mu\ \&\ B \vdash_e M{:}\tau{\rightarrow}\mu\ \&\ \rho \leq_E \tau\ ] \Rightarrow$ (5.1.1)
$\exists\, \tau, \rho, \mu\ [\ \sigma = \rho{\rightarrow}\mu\ \&\ B \vdash_e M{:}\tau{\rightarrow}\mu\ \&\ \tau{\rightarrow}\mu \leq_E \rho{\rightarrow}\mu\ ] \Rightarrow$ (5.1.6)
$B \vdash_e M{:}\sigma$. □

It is easy to check that $\vdash_E \lambda xy.xy{:}(\sigma{\rightarrow}\tau){\rightarrow}\sigma{\cap}\rho{\rightarrow}\tau$ and $\vdash_E \lambda x.x{:}(\sigma{\rightarrow}\tau){\rightarrow}\sigma{\cap}\rho{\rightarrow}\tau$.

5.2 *The relation between* $\vdash_S$, $\vdash_E$ *and* $\vdash_{BCD}$

As with the BCD-type assignment system and the strict system, essential type assignment is not conservative over strict type assignment. So the following does not hold:

$$B \vdash_S M{:}\sigma \iff B \vdash_E M{:}\sigma.$$

As a counter example for $\Leftarrow$, take $\{x{:}\sigma{\rightarrow}\sigma\} \vdash_E x{:}\sigma{\cap}\tau{\rightarrow}\sigma$. Notice that $\sigma{\rightarrow}\sigma \leq_E \sigma{\cap}\tau{\rightarrow}\sigma$. It is not possible to derive the statement $x{:}\sigma{\cap}\tau{\rightarrow}\sigma$ from the basis $\{x{:}\sigma{\rightarrow}\sigma\}$ in $\vdash_S$. Of course the implication in the other direction holds: $B \vdash_S M{:}\sigma$ implies $B \vdash_E M{:}\sigma$.

In a way similar to that of section 4.3, we can show that the essential type assignment system is the nucleus of the BCD-system; we can also show that the strict system is the nucleus of the essential system. The relation between the different notions of type assignment is formulated as follows:

Theorem 5.2.1 i) If $B \vdash_{BCD} M{:}\sigma$, then there are B', σ' such that $B' \vdash_E M{:}\sigma'$, $\sigma \sim \sigma'$ and $B \sim B'$.

ii) If $B \vdash_E M{:}\sigma$, then there are B', σ' such that $B' \vdash_S M{:}\sigma'$, $\sigma' \leq_E \sigma$ and $B \leq_E B'$.

Proof: i) As the proof of theorem 4.3.5.

ii) By easy induction, using lemma 5.1.5. □

In part (i) in particular $\sigma' \leq \sigma$ and $B \leq B'$.

As in the previous chapter, it is possible to prove that the essential type assignment system satisfies the main properties of the BCD-system and of the strict system.

Property 5.2.2 i) $B \vdash_E M{:}\sigma\ \&\ M =_\beta N \Rightarrow B \vdash_E N{:}\sigma$.

ii) $\exists\, B, \sigma\ [\ B \vdash_E M{:}\sigma\ \&\ B, \sigma\ \omega\text{-free}\ ] \iff M$ has a normal form.

iii) $\exists\, B, \sigma\ [\ B \vdash_E M{:}\sigma\ \&\ \sigma \neq \omega\ ] \iff M$ has a head normal form. □

5.3 *Soundness and completeness of essential type assignment*

For this essential system, it is possible to prove completeness of type assignment with respect to the simple type semantics the same way as done in [Barendregt *et al.* '83]. Since such a proof would be obtained in exactly the same way as in [Barendregt *et al.* '83], we will not present it here. Instead, we will prove a completeness result using results proven in [Hindley '82]. In this paper, some restrictions of the BCD-system were investigated, and one of them proved to be essentially the same as the BCD-system.

Definition 5.3.1 [Hindley '82] i) The set $\mathcal{T}_N$ of *normal intersection types* is defined by:

a) Type-variables and ω are in $\mathcal{T}_N$.

b) If $\sigma, \tau \in \mathcal{T}_N - \{\omega\}$, then $\sigma{\cap}\tau \in \mathcal{T}_N$.

c) If $\sigma \in \mathcal{T}_N$, and $\tau \in \mathcal{T}_N$-$\{\omega$, intersections$\}$, then $\sigma{\rightarrow}\tau \in \mathcal{T}_N$.

ii) On $\mathcal{T}_{\rm N}$, the relation $\leq_{\rm N}$ is obtained by restricting the definition of $\leq$ to $\mathcal{T}_{\rm N}$.

iii) The notion of type assignment $\vdash_{\rm N}$ is defined as $\vdash_{\rm BCD}$, but by adding:

a) All types are in $\mathcal{T}_{\rm N}$.

b) Derivation rule $(\cap{\rm E})$ never immediately follows $(\cap{\rm I})$.

c) Derivation rules $(\cap{\rm E})$ and $(\leq_{\rm N})$ are only used with atomic subjects.

It is straightforward to show that $\mathcal{T}_{\rm N} = \mathcal{T}_{\rm S}$, and $\leq_{\rm N} \ = \ \leq_{\rm E}$.

Proposition 5.3.2 (cf. [Hindley '82]) i) If $\sigma, \tau \in \mathcal{T}_{\rm S}$, then $\sigma \leq \tau \iff \sigma \leq_{\rm E} \tau$.

ii) Let * be defined as in lemma 4.3.1. $B \vdash_{\rm BCD} M{:}\sigma \iff B^* \vdash_{\rm N} M{:}\sigma^*$. $\square$

We will now prove that BCD-type assignment is conservative over essential type assignment.

Theorem 5.3.3 *Conservativity*. Let B and σ contain types in $\mathcal{T}_{\rm S}$. If $B \vdash_{\rm BCD} M{:}\sigma$, then $B \vdash_{\rm E} M{:}\sigma$.

Proof: $B \vdash_{\rm BCD} M{:}\sigma \Rightarrow$ (5.2.1 (i))

$\exists\, B', \sigma'\, [\, B' \vdash_{\rm E} M{:}\sigma' \ \&\ \sigma' \sim \sigma \ \&\ B \sim B'\,] \Rightarrow$ (5.3.2 (i))

$\exists\, B', \sigma'\, [\, B' \vdash_{\rm E} M{:}\sigma' \ \&\ \sigma' \leq_{\rm E} \sigma \ \&\ B \leq_{\rm E} B'\,] \Rightarrow$ (5.1.5 (v) & 5.1.6)

$B \vdash_{\rm E} M{:}\sigma$. $\square$

Of course the implication in the other direction also holds: If $B \vdash_{\rm E} M{:}\sigma$, then $B \vdash_{\rm BCD} M{:}\sigma$.

We will now show that $\vdash_{\rm N}$ and $\vdash_{\rm E}$ are equivalent.

Theorem 5.3.4 $B \vdash_{\rm N} M{:}\sigma \iff B \vdash_{\rm E} M{:}\sigma$.

Proof: $\Rightarrow$) $B \vdash_{\rm N} M{:}\sigma \Rightarrow$ (5.3.2 (ii)) $B \vdash_{\rm BCD} M{:}\sigma \Rightarrow$ (5.3.3) $B \vdash_{\rm E} M{:}\sigma$.

$\Leftarrow$) $B \vdash_{\rm E} M{:}\sigma \Rightarrow B \vdash_{\rm BCD} M{:}\sigma \Rightarrow$ (5.3.2 (ii)) $B \vdash_{\rm N} M{:}\sigma$. $\square$

Now, soundness and completeness of essential type assignment are easy to prove.

Theorem 5.3.5 *Soundness and completeness of essential type assignment*. Let B and σ contain types in $\mathcal{T}_{\rm S}$. Then $B \vdash_{\rm E} M{:}\sigma \iff B \models_{\rm s} M{:}\sigma$.

Proof: $\Rightarrow$) $B \vdash_{\rm E} M{:}\sigma \Rightarrow B \vdash_{\rm BCD} M{:}\sigma \Rightarrow$ (2.3.1.5 (i)) $B \models_{\rm s} M{:}\sigma$.

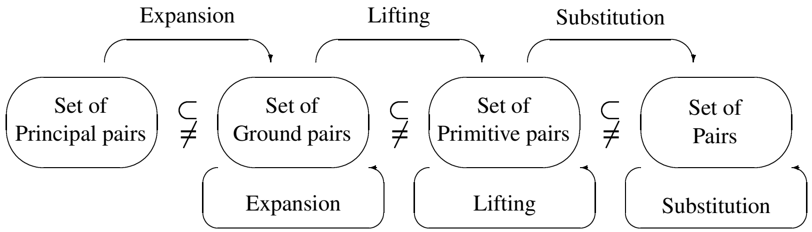
$\Leftarrow$) $B \models_{\rm s} M{:}\sigma \Rightarrow$ (2.3.1.5 (ii)) $B \vdash_{\rm BCD} M{:}\sigma \Rightarrow$ (5.3.3) $B \vdash_{\rm E} M{:}\sigma$. $\square$

Since $\vdash_{\rm N}$ and $\vdash_{\rm E}$ are equivalent, soundness and completeness of essential type assignment could also be proved using a similar result proved in [Hindley '82] for $\vdash_{\rm N}$.

Chapter 6 *Principal Type Schemes for the Strict Type Assignment System*

In this chapter, we will prove that the Strict Type Assignment System as presented in chapter four has the principal type property. For each lambda term the principal pair (of basis and type) will be defined. We will define three operations on pairs of basis and types, namely strict substitution, strict expansion, and lifting, that are correct and sufficient to generate all derivable pairs for lambda terms. The operation of lifting is the strict counterpart of rise, and the operation of substitution is a slight modification of the ones defined before. The operation of expansion coincides with the CDV- and RV-expansions. Substitution and expansion will prove to be sound on all pairs; we will also show that there is no operation of lifting that is sound on all pairs of basis and type.

In order to prove that the operations defined are sufficient, we will define a hierarchy of pairs consisting of principal pairs, ground pairs, primitive pairs, and pairs. We will show that these form a true hierarchy, that the set of ground pairs is closed under the operation of expansion, and that the set of primitive pairs is closed under the operation of lifting. In a diagram, this construction looks like:



There is an arrow from one set of pairs to another, if the operation mentioned above the arrow maps a pair in the set at the beginning of the arrow to a pair in the set at the end. When these sets are different, the result of applying the operation in general is not a pair in the set at the beginning of the arrow. The symbols between the sets indicate that the set on the left is a true subset of the set on the right.

The principal pair for a lambda term M in the Strict Type Assignment System, $PP_{\rm S}(M)$, turns out to be the same as $PP_{\rm CDV}(M)$ and $PP_{\rm RV}(M)$; the definition of ground pairs coincides with the one for CDV-ground pairs. As will be shown in this chapter, the results of [Coppo *et al.* '80] and [Ronchi della Rocca & Venneri '84] do not provide principal types for the strict system. Although the type assignment systems differ only in details, none of the results of either paper can be applied to the strict system. All results presented in this chapter with proofs are therefore new, although we are using techniques that are similar to the ones used in these two papers.

Since the strict system is closed under β-equality, type assignment in this system is not decidable, but semi-decidable. This means that, in an implementation of the strict system, some restrictions have to be made, as is done, for example, in [Coppo & Giannini '92].

We will show that the three operations defined in this chapter are complete: if $<B, \sigma>$ is a suitable pair for a term A in $\lambda\bot$-normal form, and $<P, \pi>$ is the principal pair for A, then there are an operation of lifting L, sequences of operations of expansion $\vec{E}$, and substitution $\vec{S}$, such that $<B, \sigma> = \vec{S}(L(\vec{E}(<P, \pi>)))$.

We will generalize these results to arbitrary lambda terms, and, finally, show that also the essential type assignment system has the principal type property.

In proving that the strict system has the principal pair property, we could have used the same technique as [Margaria & Zacchi '90]. In this paper, principal typings for the type assignment system as presented in [Jacobs *et al.* '92] were studied. This system is a combination of the BCD-system and the polymorphic type discipline as presented in [Girard '86]. This combination can be seen as an extension of the BCD-system by adding the type constructor '$\forall$': if φ is a type-variable and σ is a type, then $\forall\varphi.\sigma$ is a type. Also, the type inclusion relation $\leq$ is extended in a natural way. The type assignment rules consist of $(\rightarrow\mathrm{I})$, $(\rightarrow\mathrm{E})$, $(\forall\mathrm{I})$, $(\cap\mathrm{I})$, $(\leq)$, and (ω). The rules $(\cap\mathrm{E})$ and $(\forall\mathrm{E})$ are omitted since they can be derived from $(\leq)$.

The technique used in [Margaria & Zacchi '90] is the following: For every A in $\lambda\bot$-normal form, a relation $\subseteq_A$ is defined on the inductively defined set of pairs $<B, \sigma>$ admissible for A (i.e. such that $B \vdash A{:}\sigma$). This relation satisfies: if $<B_1, \sigma_1> \subseteq_A <B_2, \sigma_2>$, then $<B_1, \sigma_1>$ *and* $<B_2, \sigma_2>$ are *both* admissible pairs for A. The principal pairs of terms in $\lambda\bot$-normal form are defined by induction on the structure of such terms (similar to definitions 2.2.2.10, 2.3.2.5 and 6.1.3 of this thesis). The proof is completed by showing that, if $<B, \sigma>$ is an admissible pair for A, and $<P, \pi>$ is the principal pair of A, then $<P, \pi> \subseteq_A <B, \sigma>$.

A major difference between this technique and the one used in this chapter, is that the latter yields, given an admissible pair $<B, \sigma>$ for A, a sequence of operations that will transform the principal pair of A into $<B, \sigma>$. The technique of [Margaria & Zacchi '90] is sufficient to show that, for every term in $\lambda\bot$-normal form, there exists a principal type; it does not specify how to create an admissible pair from the principal one.

6.1 *Principal pairs for terms in $\lambda\bot$-normal form*

In [Coppo *et al.* '80], principal pairs were defined for a type assignment system that is, at first sight, very similar to the strict system, and in [Ronchi della Rocca & Venneri '84] principal pairs were defined for the BCD-system. In order to understand the necessity of defining principal pairs for the the strict system, we focus on the small but important differences between the strict system and the other two.

In the type assignment system as presented in [Coppo *et al.* '80], the types that occur in bases can only be type-variables or arrow types. Instead of using intersections in bases, it is allowed for to let a basis contain several statements the subject of which is a variable. If $B \vdash_{\mathrm{CDV_P}} M{:}\tau$, and x occurs in B, the $(\rightarrow\mathrm{I})$-rule of this system collects all types that are predicates for x and are used in the derivation of $M{:}\tau$. This system does not contain an $(\cap\mathrm{E})$-rule of any kind, so, in this system, it is impossible to derive the type $\varphi_0\cap\varphi_1\rightarrow\varphi_0$ for the lambda term $\lambda x.x$. This type is derivable for this term in the strict system.

Moreover, if $B \vdash_{\mathrm{CDV_P}} M{:}\tau$, and the variable x does not occur in B, then for $\lambda x.M$ only the type $\omega\rightarrow\tau$ can be derived. Therefore, in that system it is impossible to derive $\varphi_0\rightarrow\varphi_1\rightarrow\varphi_0$ for

the lambda term $\lambda ab.a$. This type is derivable for this term in the strict system.

The restriction being made from the BCD-system to the strict system consists of eliminating the derivation rule $(\leq)$. This rule plays an important part in [Ronchi della Rocca & Venneri '84], in which the derivation rule $(\cap\mathrm{E})$ of the BCD-system is left out because it is in fact derivable from $(\leq)$. In this system, the statement $\lambda x.x{:}\varphi_0\cap\varphi_1\rightarrow\varphi_0$ can be derived by applying the $(\leq)$-rule to the derivation for

$$\vdash_{\mathrm{BCD}} \lambda x.x{:}\varphi_0\rightarrow\varphi_0$$

(allowed for since $\varphi_0\rightarrow\varphi_0 \leq \varphi_0\cap\varphi_1\rightarrow\varphi_0$). Similarly, the statement $\lambda ab.a{:}\varphi_0\rightarrow\varphi_1\rightarrow\varphi_0$ can be derived by applying the $(\leq)$-rule to the derivation for

$$\vdash_{\mathrm{BCD}} \lambda ab.a{:}\varphi_0\rightarrow\omega\rightarrow\varphi_0$$

(since $\varphi_0\rightarrow\omega\rightarrow\varphi_0 \leq \varphi_0\rightarrow\varphi_1\rightarrow\varphi_0$). Although the $(\leq)$-rule is not allowed in the strict system, it could of course be that the operation of rise – which works on pairs and produces pairs – is sound for strict pairs. This, however, is not true. Take, for example, the rise

$$R = <<\emptyset, \varphi_0\rightarrow\omega\rightarrow\varphi_0>, <\emptyset, \varphi_0\rightarrow\varphi_1\rightarrow\varphi_0>>.$$

Then $R(<\{x{:}\varphi_0\rightarrow\omega\rightarrow\varphi_0\}, \varphi_0\rightarrow\omega\rightarrow\varphi_0>) = <\{x{:}\varphi_0\rightarrow\omega\rightarrow\varphi_0\}, \varphi_0\rightarrow\varphi_1\rightarrow\varphi_0>$. It is not possible to derive $x{:}\varphi_0\rightarrow\varphi_1\rightarrow\varphi_0$ from the basis $\{x{:}\varphi_0\rightarrow\omega\rightarrow\varphi_0\}$ in the strict system.

So, the results of [Coppo *et al.* '80] and [Ronchi della Rocca & Venneri '84] do not provide principal types for the strict system. The technique used in these papers to construct the principal pairs and to define the operations that generate all other derivable pairs for a lambda term, is, however, similar to the one used here.

Before we come to the definition of principal pairs, we introduce the notion of a basis used for a statement. The idea is that, in an average derivation, a number of types attached to term-variables in the basis is not needed in the derivation at all: there is, for example, no $(\cap\mathrm{E})$-rule that selects these types. In constructing a used basis of a derivation, we collect all (and nothing but) the types that are actually used in the derivation. This notion makes the proofs of subsection 6.2.2 easier.

Definition 6.1.1 i) The *used bases of* $B \vdash_{\mathrm{S}} M{:}\sigma$ are inductively defined by:

a) If $\sigma = \sigma_1\cap\cdots\cap\sigma_n$ $(n \geq 0)$, then for every $1 \leq i \leq n$ $B \vdash_{\mathrm{s}} M{:}\sigma_i$. Let, for every $1 \leq i \leq n$, B_i be a used basis of $B \vdash_{\mathrm{s}} M{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

b) $\sigma \in \mathcal{T}_{\mathrm{s}}$.

1) $M \equiv x$. Take $\{x{:}\sigma\}$.

2) $M \equiv \lambda x.M'$. Then $\sigma = \alpha\rightarrow\beta$, and $B \cup \{x{:}\alpha\} \vdash_{\mathrm{S}} M'{:}\beta$. Let B' be a used basis of $B \cup \{x{:}\alpha\} \vdash_{\mathrm{S}} M'{:}\beta$. If $x{:}\alpha' \in B'$, take $B' \backslash x$, otherwise take B'.

3) $M \equiv M_1M_2$. Then there is a τ such that $B \vdash_{\mathrm{s}} M_1{:}\tau\rightarrow\sigma$ and $B \vdash_{\mathrm{S}} M_2{:}\tau$. Let B_1 be a used basis of $B \vdash_{\mathrm{s}} M_1{:}\tau\rightarrow\sigma$, and B_2 be a used basis of $B \vdash_{\mathrm{S}} M_2{:}\tau$. Take $\Pi\{B_1, B_2\}$.

ii) A basis B is *used for* $M{:}\sigma$ if and only if there is a basis B' such that $B' \vdash_{\mathrm{S}} M{:}\sigma$, and B is a used basis of $B' \vdash_{\mathrm{S}} M{:}\sigma$.

Notice that in part (i.a), if $n = 0$, then $\sigma = \omega$, and $\Pi\{B_1, \ldots, B_n\} = \emptyset$.

Notice that constructing a used basis from a derivation is not the same as constructing the minimal basis needed to derive the conclusion of the derivation. Also, a used basis for a derivable statement $M{:}\sigma$ is not unique, but depends on the derivation used. This is caused by the fact that in part (i.b.3), for example, the type τ is not fixed. Choosing another suitable τ could give another used basis. The results we present with used bases do not depend on the actual structure of a used basis, however; they only depend on its existence.

For used bases, the following properties hold.

Lemma 6.1.2 i) If B is used for $M{:}\sigma$, then $B \vdash_S M{:}\sigma$.

ii) $B \vdash_S M{:}\sigma \Leftrightarrow \exists B' [\, B \leq_S B' \;\&\; B'$ is used for $M{:}\sigma$].

iii) B is used for $\lambda x.M{:}\sigma{\rightarrow}\tau \Leftrightarrow \exists \sigma' [\, \sigma \leq_S \sigma' \;\&\; B \cup \{x{:}\sigma'\}$ is used for $M{:}\tau$].

iv) B is used for $xM_1 \ldots M_n{:}\sigma \Leftrightarrow \exists B_1, \ldots, B_n, \sigma_1, \ldots, \sigma_n [\, \forall 1 \leq i \leq n [\, B_i$ is used for $M_i{:}\sigma_i$] & $B = \Pi\{B_1, \ldots, B_n, \{x{:}\sigma_1{\rightarrow}\cdots{\rightarrow}\sigma_n{\rightarrow}\sigma\}\}$].

Proof: By lemmas 4.1.5 and 4.1.6, and definition 6.1.1. □

Principal pairs for the Strict Type Assignment System are defined by:

Definition 6.1.3 i) Let $A \in \mathcal{N}$. *PP*$_S(A)$, the *strict principal pair* of A, is defined by:

a) *PP*$_S(\bot) = <\emptyset, \omega>$.

b) *PP*$_S(x) = <\{x{:}\varphi\}, \varphi>$.

c) If $A \neq \bot$, and *PP*$_S(A) = <P, \pi>$, then:

1) If x occurs free in A, and $x{:}\sigma \in P$, then *PP*$_S(\lambda x.A) = <P\backslash x, \sigma{\rightarrow}\pi>$.

2) Otherwise *PP*$_S(\lambda x.A) = <P, \omega{\rightarrow}\pi>$.

d) If *PP*$_S(A_i) = <P_i, \pi_i>$ for $1 \leq i \leq n$ (we choose trivial variants that are disjoint in pairs), then *PP*$_S(xA_1 \ldots A_n) = <\Pi\{P_1, \ldots, P_n, \{x{:}\pi_1{\rightarrow}\cdots{\rightarrow}\pi_n{\rightarrow}\varphi\}\}, \varphi>$, where φ is a type-variable that does not occur in *PP*$_S(A_i)$ for $1 \leq i \leq n$.

ii) $\mathcal{P}_S = \{<P, \pi> \mid \exists A \in \mathcal{N} [\,$ *PP*$_S(A) = <P, \pi>$] $\}$.

Notice that, this definition is, apart from the notion of sequence, the same as the definition for *PP*$_{CDV}(M)$, and equivalent to the one for *PP*$_{RV}(M)$, so we can say that $\mathcal{P}_S = \mathcal{P}_{RV}$.

Notice that, if $<P, \pi> \in \mathcal{P}_S$, then $\pi \in \mathcal{T}_S$, and that if *PP*$_S(A) = <P, \pi>$, then P is used for $A{:}\pi$. The notion of principal pairs for $\lambda\bot$-normal forms will be generalized to arbitrary lambda terms in subsection 6.3.2. This is done the same way as in [Coppo *et al.* '80] and [Ronchi della Rocca & Venneri '84].

6.2 *Operations on pairs*

In this section, we present three different operations on pairs of <*basis*, *type*>, namely strict substitution, strict expansion, and lifting. The operation of strict substitution is a slight modification of the one normally used; this modification is required to make sure that substitution is closed on strict types. The operation of strict expansion coincides with the one given in [Coppo *et al.* '80, Ronchi della Rocca & Venneri '84]. The operation of lifting forms the strict counterpart of the operation of rise. It deals with the introduction of extra (types to) statements in the basis of a derivation, or introduces extra types to term-variables that are bound.

For the operations of substitution and expansion we will prove soundness: for every $A \in \mathcal{N}$, if $B \vdash_S A{:}\sigma$, and $<B', \sigma'>$ can be obtained from $<B, \sigma>$ by an operation of substitution or expansion, then $B' \vdash_S A{:}\sigma'$. For the operation of lifting we will prove a more restricted result: if $<B, \sigma>$ is a primitive pair for A (then also $B \vdash_S A{:}\sigma$), and $<B', \sigma'>$ can be obtained from $<B, \sigma>$ by lifting, then $<B', \sigma'>$ is a primitive pair for A. We will also show that there is no operation of lifting that is sound on all pairs.

We will define a hierarchy of pairs, consisting of (in order): principal pairs, ground pairs, and primitive pairs. We will show that the set of ground pairs is closed under the operation of expansion, and that the set of primitive pairs is closed under the operation of lifting. These results are required in the completeness proof of section 6.3.

6.2.1 *Strict substitution*

Substitution on types is normally defined as the operation that replaces type-variables by types. For strict types, this definition would not be correct. For example, the replacement of φ by ω would transform $\sigma{\rightarrow}\varphi$ (or $\sigma{\cap}\varphi$) into $\sigma{\rightarrow}\omega$ ($\sigma{\cap}\omega$), which is not a strict type. Therefore, substitution on strict types is not defined as an operation that replaces type-variables by types, but as a mapping from types to types.

Definition 6.2.1.1 i) The *strict substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_S \rightarrow \mathcal{T}_S$, where φ is a type-variable and $\alpha \in \mathcal{T}_s \cup \{\omega\}$, is defined by:

a) $(\varphi \mapsto \alpha)(\varphi) = \alpha$.

b) $(\varphi \mapsto \alpha)(\varphi') = \varphi'$, if $\varphi \neq \varphi'$.

c) $(\varphi \mapsto \alpha)(\sigma{\rightarrow}\tau) = \omega$, if $(\varphi \mapsto \alpha)(\tau) = \omega$.

d) $(\varphi \mapsto \alpha)(\sigma{\rightarrow}\tau) = (\varphi \mapsto \alpha)(\sigma) \rightarrow (\varphi \mapsto \alpha)(\tau)$, if $(\varphi \mapsto \alpha)(\tau) \neq \omega$.

e) $(\varphi \mapsto \alpha)(\sigma_1{\cap}\cdots{\cap}\sigma_n) = (\varphi \mapsto \alpha)(\sigma_1{}') \cap\cdots\cap (\varphi \mapsto \alpha)(\sigma_m{}')$, where $\{\sigma_1{}', \ldots, \sigma_m{}'\} = \{\, \sigma_i \in \{\sigma_1, \ldots, \sigma_n\} \mid (\varphi \mapsto \alpha)(\sigma_i) \neq \omega \,\}$.

ii) If S_1 and S_2 are strict substitutions, then so is $S_1{\circ}S_2$, where $S_1{\circ}S_2(\sigma) = S_1(S_2(\sigma))$.

iii) $S(B) = \{\, x{:}S(\alpha) \mid x{:}\alpha \in B \;\&\; S(\alpha) \neq \omega \,\}$.

iv) $S(<B, \sigma>) = <S(B), S(\sigma)>$.

Notice that in part (i.e), if for $1 \leq i \leq n$ $(\varphi \mapsto \alpha)(\sigma_i) = \omega$, then $(\varphi \mapsto \alpha)(\sigma_1{\cap}\cdots{\cap}\sigma_n) = \omega$.

Since no confusion is possible, we will throughout this chapter speak of 'substitution' instead of 'strict substitution'. For substitutions, the following properties hold:

Lemma 6.2.1.2 Let S be a substitution.

i) If $\sigma \leq_S \tau$, then $S(\sigma) \leq_S S(\tau)$.

ii) If $\sigma \in \mathcal{T}_s$, then $S(\sigma) \neq \omega \Rightarrow S(\sigma) \in \mathcal{T}_s$.

Proof: Easy. □

The following lemma is needed in the proof of theorem 6.3.1.2.

Lemma 6.2.1.3 Let S be a substitution such that $S(\tau) = \tau'$, $\tau \in \mathcal{T}_s$, and $\tau' \neq \omega$, then:

i) If $S(B \cup \{x{:}\sigma\}) = B' \cup \{x{:}\sigma'\}$, then $S(<B, \sigma\to\tau>) = <B', \sigma'\to\tau'>$.

ii) If, for every $1 \leq i \leq n$, $S(<B_i, \sigma_i>) = <B_i', \sigma_i'>$, then

$$S(<\Pi\{B_1, \ldots, B_n, \{x{:}\sigma_1\to\cdots\to\sigma_n\to\tau\}\}, \tau>) = \\ <\Pi\{B_1', \ldots, B_n', \{x{:}\sigma_1'\to\cdots\to\sigma_n'\to\tau'\}\}, \tau'>.$$

Proof: Immediately by definition 6.2.1.1. □

We will now prove that the operation of substitution is sound on all pairs of basis and type.

Theorem 6.2.1.4 If $B \vdash_S A{:}\sigma$, then for every substitution S: if $S(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_S A{:}\sigma'$.

Proof: By induction on the definition of $\vdash_S$.

i) $B \vdash_S A{:}\omega$. $S(\omega) = \omega$, and obviously $B' \vdash_S A{:}\omega$.

ii) $B \vdash_S A{:}\sigma_1\cap\cdots\cap\sigma_n$. Then, for every $1 \leq i \leq n$, $B \vdash_s A{:}\sigma_i$. Let $\sigma_1', \ldots, \sigma_m'$ be the elements in $\{\sigma_1, \ldots, \sigma_n\}$ such that $S(\sigma_i') \not= \omega$. By induction and lemma 6.2.1.2 (ii), for every $1 \leq i \leq m$, $B' \vdash_s A{:}S(\sigma_i')$. Then $B' \vdash_S A{:}S(\sigma_1')\cap\cdots\cap S(\sigma_m')$, so $B' \vdash_S A{:}S(\sigma)$.

iii) $B \vdash_s A{:}\sigma$. Then $\sigma \in \mathcal{T}_s$. This part is proven by induction on elements of $\mathcal{N}$. The case $S(\sigma) = \omega$ is trivial, so in the rest of the proof $S(\sigma) \not= \omega$, so by lemma 6.2.1.2 (ii) $S(\sigma) \in \mathcal{T}_s$.

a) $A \equiv x$. Then there is an α such that $x{:}\alpha \in B$, and $\alpha \leq_S \sigma$. Then $x{:}S(\alpha) \in B'$, and, by lemma 6.2.1.2 (i), $S(\alpha) \leq_S S(\sigma) \in \mathcal{T}_s$, so $B' \vdash_s x{:}S(\sigma)$.

b) $A \equiv \lambda x.A'$. Then $\sigma = \alpha\to\beta$, $B \cup \{x{:}\alpha\} \vdash_s A'{:}\beta$, and

$$S(<B \cup \{x{:}\alpha\}, \beta>) = <B' \cup \{x{:}S(\alpha)\}, S(\beta)>.$$

Since $S(\sigma) \in \mathcal{T}_s$, also $S(\beta) \in \mathcal{T}_s$, so by induction: $B' \cup \{x{:}S(\alpha)\} \vdash_s A'{:}S(\beta)$, so $B' \vdash_s \lambda x.A'{:}S(\alpha) \to S(\beta)$, and, therefore, $B' \vdash_s \lambda x.A'{:}S(\sigma)$.

c) $A \equiv xA_1 \ldots A_m$. Then there are $\tau_1, \ldots, \tau_m$, such that $x{:}\tau_1\to\cdots\to\tau_m\to\sigma \in B$, and for every $1 \leq j \leq m$ $B \vdash_S A_j{:}\tau_j$. $S(\sigma) \in \mathcal{T}_s$, so $x{:}S(\tau_1)\to\cdots\to S(\tau_m)\to S(\sigma) \in B'$. By induction for every $1 \leq j \leq m$, $B' \vdash_S A_j{:}S(\tau_j)$. But then also $B' \vdash_S xA_1 \ldots A_m{:}S(\sigma)$. □

6.2.2 *Strict expansion*

The operation of strict expansion as defined in this subsection corresponds to the CDV-expansion, and is a simplified version of the RV-expansion. It is an operation on types that deals with the replacement of (sub)types by an intersection of a number of copies of that type. Strict expansion on types corresponds to the duplication of (sub)derivations: a subderivation in either the right hand side of an $(\to E)$-step – or the final step in a derivation in $\vdash_S$ – is expanded by copying. In this process, the types that occur in the subderivation are also copied: the types in the conclusion and the ones in the basis of the subderivation will be instantiated into a number of copies.

A strict expansion not only indicates the type to be expanded, but also the number of copies that has to be generated. Like in the original definition, strict expansion is a complex operation, possibly affecting more types than just the one which is to be expanded occurs in. Suppose that μ is a subtype of σ that is to be expanded into n copies. If $\tau\to\mu$ is also a subtype of σ, then, just replacing μ by an intersection of copies of μ would generate $\tau\to\mu_1\cap\cdots\cap\mu_n$. This is not a legal strict type.

Defining an operation of strict expansion by saying that it should replace the subtype $\tau\to\mu$ by $(\tau\to\mu_1)\cap\cdots\cap(\tau\to\mu_n)$ (which is by definition of $\leq$ a type equivalent to $\tau\to\mu_1\cap\cdots\cap\mu_n$) would give an expansion that is sound, but not sufficient: it would not be closed for ground pairs, a property we need in the proof of theorem 6.3.1.2.

Therefore, the subtype $\tau\to\mu$ will be expanded into $(\tau_1\to\mu_1)\cap\cdots\cap(\tau_n\to\mu_n)$, where $\tau_1, \ldots, \tau_n$ are copies of τ. This means that all other occurrences of τ should also be expanded into $\tau_1\cap\cdots\cap\tau_n$, with possibly the same effect on other types. Moreover, if φ is a type-variable that occurs in μ (or τ), then all occurrences of φ outside μ (τ) will be expanded into $\varphi_1\cap\cdots\cap\varphi_n$, and all types of the shape $\rho\to\varphi$ will be expanded into $(\rho_1\to\varphi_1)\cap\cdots\cap(\rho_n\to\varphi_n)$, etc.

So, again the strict expansion of μ can have a more than local effect on σ. Therefore, the strict expansion of a type is defined in a such a way that, before replacing types by intersections, all type-variables are collected that are affected by the strict expansion of μ. Then, types are traversed top down, and subtypes are replaced if they end with one of the type-variables found.

The definition of the operation of strict expansion can be found in definition 6.2.2.6; it is based on the definition of a type-expansion, as given in definition 6.2.2.2. The definition of expansion as presented here is slightly different from the ones given in definitions 2.2.2.5 and 2.3.2.2. In those definitions subtypes are collected, whereas this definition collects type-variables.

Definition 6.2.2.1 i) a) The *last type-variable* of a strict type is defined by:

1) The last type-variable of φ is φ.

2) The last type-variable of $\sigma_1\cap\cdots\cap\sigma_n\to\tau$ $(n \geq 0)$ is the last type-variable of τ.

b) A strict type σ is said to *end with* φ, if φ is the last type-variable of σ.

ii) If B is a basis and $\sigma \in \mathcal{T}_S$, then $\mathcal{T}_{<B,\sigma>}$ is the set of all strict subtypes occurring in the pair $<B, \sigma>$.

Definition 6.2.2.2 For every $\mu \in \mathcal{T}_s$, $n \geq 2$, basis B and $\sigma \in \mathcal{T}_S$, the quadruple $<\mu, n, B, \sigma>$ determines a *type-expansion* $T_{<\mu,n,B,\sigma>} : \mathcal{T}_S \to \mathcal{T}_S$, that is constructed as follows.

i) The set of type-variables $\mathcal{V}_\mu(<B, \sigma>)$ is constructed by:

a) If φ occurs in μ, then $\varphi \in \mathcal{V}_\mu(<B, \sigma>)$.

b) Let $\tau \in \mathcal{T}_{<B,\sigma>}$ with last type-variable φ_0. If $\varphi_0 \in \mathcal{V}_\mu(<B, \sigma>)$, then for all type-variables φ that occur in τ: $\varphi \in \mathcal{V}_\mu(<B, \sigma>)$.

ii) Suppose $\mathcal{V}_\mu(<B, \sigma>) = \{\varphi_1, \ldots, \varphi_m\}$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i does not occur in $\mathcal{T}_{<B,\sigma>}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Let S_i be the substitution that replaces every φ_j by φ_j^i.

iii) a) $T_{<\mu,n,B,\sigma>}(\alpha) = S_1(\alpha)\cap\cdots\cap S_n(\alpha)$, if the last type-variable of α is in $\mathcal{V}_\mu(<B, \sigma>)$.

b) $T_{<\mu,n,B,\sigma>}(\alpha\to\beta) = T_{<\mu,n,B,\sigma>}(\alpha) \to T_{<\mu,n,B,\sigma>}(\beta)$, if the last type-variable of β is not in $\mathcal{V}_\mu(<B, \sigma>)$.

c) $T_{<\mu,n,B,\sigma>}(\varphi) = \varphi$, if $\varphi \notin \mathcal{V}_\mu(<B, \sigma>)$.

d) $T_{<\mu,n,B,\sigma>}(\alpha_1\cap\cdots\cap\alpha_n) = T_{<\mu,n,B,\sigma>}(\alpha_1)\cap\cdots\cap T_{<\mu,n,B,\sigma>}(\alpha_n)$.

iv) $T_{<\mu,n,B,\sigma>}(B') = \{\, x{:}T_{<\mu,n,B,\sigma>}(\rho) \mid x{:}\rho \in B' \,\}$.

v) $T_{<\mu,n,B,\sigma>}(<B', \sigma'>) = <T_{<\mu,n,B,\sigma>}(B'), T_{<\mu,n,B,\sigma>}(\sigma')>$.

Instead of $T_{<\mu,n,B,\sigma>}$ we will write $<\mu, n, B, \sigma>$.

For an operation of type-expansion the following properties hold:

Lemma 6.2.2.3 Let $T = <\mu, n, B, \sigma>$ be a type-expansion.

i) If $\tau \in \mathcal{T}_{<B,\sigma>}$, then either:
 a) $T(\tau) = \tau_1 \cap \cdots \cap \tau_n$ with for every $1 \leq i \leq n$ τ_i is a trivial variant of τ, or:
 b) $T(\tau) \in \mathcal{T}_{\mathrm{s}}$.
ii) $T(\Pi\{B_1, \ldots, B_n\}) = \Pi\{T(B_1), \ldots, T(B_n)\}$.
iii) If $\tau \leq_{\mathrm{S}} \rho$, then $T(\tau) \leq_{\mathrm{S}} T(\rho)$.
iv) If $B' \leq_{\mathrm{S}} B''$, then $T(B') \leq_{\mathrm{S}} T(B'')$.

Proof: Immediately by 6.2.2.2. □

Variants of the following lemmas are proved in [Coppo *et al.* '80] and [Ronchi della Rocca & Venneri '84]. They are needed in the proofs of the following theorems.

Lemma 6.2.2.4 Let B' be used for $A{:}\tau$, where $\tau \in \mathcal{T}_{\mathrm{s}}$, and $<\mu, n, B, \sigma>$ be a type-expansion such that $\mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>}$. If τ ends with $\varphi \in \mathcal{V}_\mu(<B, \sigma>)$, then, for every type-variable φ' that occurs in $<B', \tau>$, $\varphi' \in \mathcal{V}_\mu(<B, \sigma>)$.

Proof: By induction on elements of $\mathcal{N}$.

i) $A \equiv x$, then $B' = \{x{:}\tau\}$. Since the last type-variable of τ is in $\mathcal{V}_\mu(<B, \sigma>)$, and $\tau \in \mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>}$, all type-variables that occur in τ are in $\mathcal{V}_\mu(<B, \sigma>)$.
ii) $A \equiv \lambda x.A'$, then $\tau = \alpha \rightarrow \beta$. Since B' is used for $\lambda x.A'{:}\alpha\rightarrow\beta$, there is an α' such that $\alpha \leq_{\mathrm{S}} \alpha'$, and $B' \cup \{x{:}\alpha'\}$ is used for $A'{:}\beta$. Because the last type-variable of $\alpha\rightarrow\beta$ is in $\mathcal{V}_\mu(<B, \sigma>)$, and $\alpha\rightarrow\beta \in \mathcal{T}_{<B',\,\alpha\rightarrow\beta>} \subseteq \mathcal{T}_{<B,\sigma>}$, all type-variables of $\alpha\rightarrow\beta$ are in $\mathcal{V}_\mu(<B, \sigma>)$. Since the last type-variable of $\alpha\rightarrow\beta$ is the last type-variable of β, and
$$\mathcal{T}_{<B'\cup\{x:\alpha'\},\,\beta>} \subseteq \mathcal{T}_{<B'\cup\{x:\alpha\},\,\beta>} \subseteq \mathcal{T}_{<B',\,\alpha\rightarrow\beta>} \subseteq \mathcal{T}_{<B,\sigma>},$$
by induction: all type-variables in $<B \cup \{x{:}\alpha'\}, \beta>$ are in $\mathcal{V}_\mu(<B, \sigma>)$. So all type-variables in $<B, \alpha\rightarrow\beta>$ are in $\mathcal{V}_\mu(<B, \sigma>)$.
iii) $A \equiv xA_1 \ldots A_m$. Then there are $\tau_1, \ldots, \tau_m$, and $B_1, \ldots, B_m$, such that for every $1 \leq j \leq m$ B_j is used for $A_j{:}\tau_j$ and $B' = \Pi\{B_1, \ldots, B_m, \{x{:}\tau_1\rightarrow\cdots\rightarrow\tau_m\rightarrow\tau\}\}$. Since the last type-variable of τ is in $\mathcal{V}_\mu(<B, \sigma>)$, and
$$\tau_1\rightarrow\cdots\rightarrow\tau_m\rightarrow\tau \in \mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>},$$
every type-variable in $\tau_1\rightarrow\cdots\rightarrow\tau_m\rightarrow\tau$ is in $\mathcal{V}_\mu(<B, \sigma>)$. Assume, without loss of generality, that each $\tau_j \in \mathcal{T}_{\mathrm{s}}$, then for every $1 \leq j \leq m$ the last type-variable of τ_j is in $\mathcal{V}_\mu(<B, \sigma>)$, and
$$\mathcal{T}_{<B_j,\,\tau_j>} \subseteq \mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>},$$
so by induction all type-variables in $<B_j, \tau_j>$ are in $\mathcal{V}_\mu(<B, \sigma>)$. So all type-variables in $<\Pi\{B_1, \ldots, B_m, \{x{:}\tau_1\rightarrow\cdots\rightarrow\tau_m \rightarrow\tau\}\}, \tau>$ are in $\mathcal{V}_\mu(<B, \sigma>)$. □

Lemma 6.2.2.5 Let B' be used for $A{:}\tau$, where $\tau \in \mathcal{T}_{\mathrm{s}}$, and $<\mu, n, B, \sigma>$ be a type-expansion such that $\mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>}$. Then

i) There are $B_1, \ldots, B_n, \tau_1 \cap \cdots \cap \tau_n$, such that
$$<\mu, n, B, \sigma>(<B', \tau>) = <\Pi\{B_1, \ldots, B_n\}, \tau_1 \cap \cdots \cap \tau_n>$$
and for every $1 \leq i \leq n$ $<B_i, \tau_i>$ is a trivial variant of $<B', \tau>$, or
ii) $<\mu, n, B, \sigma>(<B', \tau>) = <B'', \tau'>$, with $\tau' \in \mathcal{T}_{\mathrm{s}}$.

Proof: By lemma 6.2.2.4. □

Notice that in particular this lemma holds for the case that $<B', \tau> = <B, \sigma>$.

We will now define the operation of strict expansion, that maps pairs to pairs.

Definition 6.2.2.6 Let $\mathcal{P}$*airs* be the set of all pairs of basis and type. For every $\mu \in \mathcal{T}_{\mathrm{s}}$ and $n \geq 2$, the pair $<\mu, n>$ determines a *strict expansion* $E_{<\mu,n>} : \mathcal{P}\mathit{airs} \rightarrow \mathcal{P}\mathit{airs}$, defined by:
$$E_{<\mu,n>}(<B, \sigma>) = <\mu, n, B, \sigma>(<B, \sigma>).$$
Instead of $E_{<\mu,n>}$ we write $<\mu, n>$, and we will say expansion instead of strict expansion.

The following lemma is needed in the proofs of theorems 6.2.2.10 and 6.3.1.2.

Lemma 6.2.2.7 Let E be an expansion, and $\tau \in \mathcal{T}_{\mathrm{s}}$.

i) $E(<B \cup \{x{:}\sigma\}, \tau>) = <B' \cup \{x{:}\sigma'\}, \tau'>$, where $\tau' \in \mathcal{T}_{\mathrm{s}}$, if and only if $E(<B, \sigma\rightarrow\tau>) = <B', \sigma'\rightarrow\tau'>$.
ii) Let $E(<B_i, \sigma_i>) = <B_i{}', \sigma_i{}'>$ for every $1 \leq i \leq n$. Then
$$E(<\Pi\{B_1, \ldots, B_n, \{x{:}\sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\varphi\}\}, \varphi>) =$$
$$<\Pi\{B_1{}', \ldots, B_n{}', \{x{:}\sigma_1{}'\rightarrow\cdots\rightarrow\sigma_n{}'\rightarrow\varphi\}\}, \varphi>.$$

Proof: Easy, using definitions 6.2.2.6 and 6.2.2.2, and lemma 6.2.2.3 (ii). □

We will now prove that the operation of expansion is closed on the set of strict ground pairs, as defined in the next definition. This property is needed in the proof that the operations defined in this chapter are complete (as given in subsection 6.3.1).

Definition 6.2.2.8 The pair $<B, \sigma>$ is a *strict ground pair* for $A \in \mathcal{N}$ if and only if:

i) If $\sigma = \omega$, then $B = \emptyset$ and $A \equiv \bot$.
ii) If $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$, then there are $B_1, \ldots, B_n$ such that $B = \Pi\{B_1, \ldots, B_n\}$, the $<B_i, \sigma_i>$ are disjoint in pairs and for $1 \leq i \leq n$ $<B_i, \sigma_i>$ is a ground pair for A.
iii) If $\sigma \in \mathcal{T}_{\mathrm{s}}$, then:
 a) If $A \equiv x$, then $\sigma = \varphi$, and $B = \{x{:}\varphi\}$.
 b) If $A \equiv \lambda x.A'$, then:
  1) If $x \in \mathrm{FV}(A')$, then $\sigma = \alpha\rightarrow\beta$, and $<B \cup \{x{:}\alpha\}, \beta>$ is a ground pair for A'.
  2) If $x \notin \mathrm{FV}(A')$, then $\sigma = \omega\rightarrow\beta$, and $<B, \beta>$ is a ground pair for A'.
 c) If $A \equiv xA_1 \ldots A_m$, then $\sigma = \varphi$, and there are $B_1, \ldots, B_m, \tau_1, \ldots, \tau_m$ such that $B = \Pi\{B_1, \ldots, B_m, \{x{:}\tau_1\rightarrow\cdots\rightarrow\tau_m\rightarrow\varphi\}\}$, the $<B_j, \tau_j>$ are disjoint in pairs, and for every $1 \leq j \leq m$ φ does not occur in $<B_j, \tau_j>$, and $<B_j, \tau_j>$ is a ground pair for A_j.

Notice that this definition is the same as the one for CDV-ground pairs (apart from the notion of sequence).

Lemma 6.2.2.9 i) If $<B,\sigma>$ is a ground pair for A, then B is used for $A{:}\sigma$.

ii) For every A, $PP_S(A)$ is a ground pair for A.

Proof: Easy. □

The following theorem states that expansion is closed on the set of ground pairs.

Theorem 6.2.2.10 If $<B,\sigma>$ is a ground pair for A, and $<\mu,n>(<B,\sigma>) = <B',\sigma'>$, then $<B',\sigma'>$ is a ground pair for A.

Proof: By induction on the definition of ground pairs. We will only show the part $\sigma \in \mathcal{T}_s$.

Notice that, because of lemma 6.2.2.9 (i), we already know that B is used for $A{:}\sigma$, and, therefore, by lemma 6.2.2.5 either:

i) $\sigma' = \sigma_1\cap\cdots\cap\sigma_n$, $B' = \Pi\{B_1,\ldots,B_n\}$ where each $<B_i,\sigma_i>$ is a trivial variant of $<B,\sigma>$ and, therefore, a ground pair for A. Because of definition 6.2.2.6, the $<B_i,\sigma_i>$ are disjoint in pairs. So $<B',\sigma'>$ is a ground pair for A.

ii) $\sigma' \in \mathcal{T}_s$. This part is proved by induction on elements of $\mathcal{N}$. Notice that we need not consider the case that $A \equiv \bot$.

a) $A \equiv x$, $B = \{x{:}\varphi\}$, and $\sigma = \varphi$.
Since $\sigma' \in \mathcal{T}_s$, $\varphi \notin \mathcal{V}_\mu(<B,\sigma>)$ and $<B',\sigma'> = <\{x{:}\varphi\},\varphi>$.

b) $A \equiv \lambda x.A'$, $\sigma = \alpha\to\beta$, and $<B\cup\{x{:}\alpha\},\beta>$ is a ground pair for A'. Let $\sigma' = \alpha'\to\beta'$. (Notice that if $x \notin FV(A')$, then $\alpha = \alpha' = \omega$). By lemma 6.2.2.7 (i)

$$<\mu,n>(<B\cup\{x{:}\alpha\},\beta>) = <B'\cup\{x{:}\alpha'\},\beta'>,$$

which is by induction a ground pair for A'. Because $\beta' \in \mathcal{T}_s$, also $<B',\alpha'\to\beta'>$ is a ground pair for $\lambda x.A'$.

c) $A \equiv xA_1\ldots A_m$, $\sigma = \varphi$, and $B = \Pi\{B_1,\ldots,B_m,\{x{:}\tau_1\to\cdots\to\tau_m\to\varphi\}\}$, the (disjoint in pairs) $<B_j,\tau_j>$ are ground pairs for A_j, in which φ does not occur. Since $\sigma' \in \mathcal{T}_s$, $\sigma' = \varphi$. Let for every $1 \le j \le m$

$$<\mu,n>(<B_j,\tau_j>) = <B_j{}',\tau_j{}'>,$$

which is by induction a ground pair for A_j. Then by lemma 6.2.2.7 (ii)

$$<\mu,n>(<\Pi\{B_1,\ldots,B_m,\{x{:}\tau_1\to\cdots\to\tau_m\to\varphi\}\},\varphi>) = <\Pi\{B_1{}',\ldots,B_m{}',\{x{:}\tau_1{}'\to\cdots\to\tau_m{}'\to\varphi\}\},\varphi>.$$

Since the $<B_j,\tau_j>$ are disjoint in pairs, the $<B_j{}',\tau_j{}'>$ are too, because of definition 6.2.2.6, and φ does not occur in any of the $<B_j{}',\tau_j{}'>$. So

$$<\Pi\{B_1{}',\ldots,B_m{}',\{x{:}\tau_1{}'\to\cdots\to\tau_m{}'\to\varphi\}\},\varphi>$$

is a ground pair for $xA_1\ldots A_m$. □

Example 6.2.2.11 (cf. [Coppo *et al.* '80]) Take the pair $<\emptyset,(\omega\to(\varphi_0\to\varphi_0)\to\varphi_1)\to\varphi_1>$, which is the principal pair of $\lambda x.x\bot(\lambda y.y)$.

$$\dfrac{\dfrac{\dfrac{[x{:}\omega\to(\varphi_0\to\varphi_0)\to\varphi_1]}{x\bot{:}(\varphi_0\to\varphi_0)\to\varphi_1}\qquad\dfrac{[y{:}\varphi_0]}{\lambda y.y{:}\varphi_0\to\varphi_0}}{x\bot(\lambda y.y){:}\varphi_1}}{\lambda x.x\bot(\lambda y.y){:}(\omega\to(\varphi_0\to\varphi_0)\to\varphi_1)\to\varphi_1}$$

Take the expansion $E = <\varphi_0\to\varphi_0, 2>$. Then $\mathcal{V}_{\varphi_0\to\varphi_0}(<\emptyset,(\omega\to(\varphi_0\to\varphi_0)\to\varphi_1)\to\varphi_1>) = \{\varphi_0\}$ and $E(<\emptyset,(\omega\to(\varphi_0\to\varphi_0)\to\varphi_1)\to\varphi_1>) = <\emptyset,(\omega\to(\varphi_2\to\varphi_2)\cap(\varphi_3\to\varphi_3)\to\varphi_1)\to\varphi_1>$, which is by the previous lemma a ground pair for the term $\lambda x.x\bot(\lambda y.y)$.

$$\dfrac{\dfrac{\dfrac{[x{:}\omega\to(\varphi_2\to\varphi_2)\cap(\varphi_3\to\varphi_3)\to\varphi_1]}{x\bot{:}(\varphi_2\to\varphi_2)\cap(\varphi_3\to\varphi_3)\to\varphi_1}\qquad\dfrac{[y{:}\varphi_2]}{\lambda y.y{:}\varphi_2\to\varphi_2}\qquad\dfrac{[y{:}\varphi_3]}{\lambda y.y{:}\varphi_3\to\varphi_3}}{x\bot(\lambda y.y){:}\varphi_1}}{\lambda x.x\bot(\lambda y.y){:}(\omega\to(\varphi_2\to\varphi_2)\cap(\varphi_3\to\varphi_3)\to\varphi_1)\to\varphi_1}$$

We can also prove that the operation of expansion is sound on all pairs. In order to do so, we first need to prove that the operation of type-expansion is sound on pairs $<B,\sigma>$ for the terms $A\in\mathcal{N}$ such that B is used for $A{:}\sigma$. Notice that the possible type-expansions are limited to those that at least compute the effect of the expansion on all types in $\mathcal{T}_{<B,\sigma>}$.

Theorem 6.2.2.12 For every $A\in\mathcal{N}$: if $<\mu,n,B,\sigma>(<B_1,\sigma_1>) = <B_2,\sigma_2>$, B_1 is used for $A{:}\sigma_1$, and $\mathcal{T}_{<B_1,\,\sigma_1>} \subseteq \mathcal{T}_{<B,\sigma>}$, then $B_2 \vdash_S A{:}\sigma_2$.

Proof: Again we only show the part $\sigma_1 \in \mathcal{T}_s$. Because of lemma 6.2.2.5, we know that either:

i) $\sigma_2 = \sigma_1{}'\cap\cdots\cap\sigma_n{}'$, $B_2 = \Pi\{B_1{}',\ldots,B_n{}'\}$, and for every $1\le i\le n$ $<B_i{}',\sigma_i{}'>$ is a trivial variant of $<B_1,\sigma_1>$. So $B_2 \vdash_S A{:}\sigma_2$.

ii) $\sigma_2 \in \mathcal{T}_s$. This part is proved by induction on elements of $\mathcal{N}$. Again we need not consider the case that $A\equiv\bot$.

a) $A \equiv x$. Then $B_1 = \{x{:}\sigma_1\}$, and $B_2 = \{x{:}\sigma_2\}$. Then $B_2 \vdash_s x{:}\sigma_2$.

b) $A \equiv \lambda x.A'$. Then $\sigma_1 = \alpha\to\beta$, and $B_1\cup\{x{:}\alpha\} \vdash_s A'{:}\beta$. Since $\sigma_2\in\mathcal{T}_s$, $\sigma_2 = \alpha'\to\beta'$, and by definition 6.2.2.2, $\alpha' = <\mu,n,B,\sigma>(\alpha)$ and $\beta' = <\mu,n,B,\sigma>(\beta)$, so:

$$<\mu,n,B,\sigma>(<B_1\cup\{x{:}\alpha\},\beta>) = <B_2\cup\{x{:}\alpha'\},\beta'>.$$

B_1 is used for $\lambda x.A'{:}\alpha\to\beta$, so by lemma 6.1.2 (iii) there is a ρ such that $B_1\cup\{x{:}\rho\}$ is used for $A'{:}\beta$, and $\alpha \le_S \rho$. Let $\rho' = <\mu,n,B,\sigma>(\rho)$, then

$$<\mu,n,B,\sigma>(<B_1\cup\{x{:}\rho\},\beta>) = <B_2\cup\{x{:}\rho'\},\beta'>,$$

and since

$$\mathcal{T}_{<B_1\cup\{x:\rho\},\beta>} \subseteq \mathcal{T}_{<B_1\cup\{x:\alpha\},\beta>} \subseteq \mathcal{T}_{<B_1,\alpha\to\beta>} \subseteq \mathcal{T}_{<B,\sigma>},$$

and $\beta'\in\mathcal{T}_s$, by induction $B_2\cup\{x{:}\rho'\} \vdash_s A'{:}\beta'$.
By lemmas 6.2.2.3 (iv) and 4.1.6 also $B_2\cup\{x{:}\alpha'\} \vdash_s A'{:}\beta'$. Then $B_2 \vdash_s \lambda x.A'{:}\sigma_2$.

c) $A \equiv xA_1\ldots A_m$. Then $B_1 = \Pi\{B_1{}',\ldots,B_m{}',\{x{:}\tau_1\to\cdots\to\tau_m\to\sigma_1\}\}$, where $B_j{}'$ is used for $A_j{:}\tau_j$ (notice that τ_j need not be strict). Take

$$<B_j{}'',\tau_j{}''> = <\mu,n,B,\sigma>(<B_j{}',\tau_j>).$$

Since

$$\mathcal{T}_{<B_j',\tau_j>} \subseteq \mathcal{T}_{<B_1,\sigma_1>} \subseteq \mathcal{T}_{<B,\sigma>},$$

by induction: $B_j{}'' \vdash_{\mathrm{S}} A_j{:}\tau_i{}''$ for every $1 \leq j \leq m$.
By definition 6.2.2.2 $\tau_1{}''\to\cdots\to\tau_m{}''\to\sigma_2 = {<}\mu,n,B,\sigma{>}\,(\tau_1\to\cdots\to\tau_m\to\sigma_1)$, so

$$<B_2,\sigma_2> = <\Pi\{B_1{}'',\ldots,B_m{}'',\{x{:}\tau_1{}''\to\cdots\to\tau_m{}''\to\sigma_2\}\},\sigma_2>.$$

So $B_2 \vdash_{\mathrm{S}} xA_1\ldots A_m{:}\sigma_2$. □

The next theorem was also proved in [Ronchi della Rocca & Venneri '84] for the BCD-ystem; it states that expansion is a sound operation on pairs.

Theorem 6.2.2.13 If $B \vdash_{\mathrm{S}} A{:}\sigma$, and ${<}\mu,n{>}\,({<}B,\sigma{>}) = {<}B',\sigma'{>}$, then $B' \vdash_{\mathrm{S}} A{:}\sigma'$.

Proof: By definition 6.2.2.6 ${<}\mu,n{>}\,({<}B,\sigma{>}) = {<}\mu,n,B,\sigma{>}\,({<}B,\sigma{>})$. If $B \vdash_{\mathrm{S}} A{:}\sigma$, then by 6.1.2 (ii) there is a B_1 such that $B \leq_{\mathrm{S}} B_1$ (so $\mathcal{T}_{<B_1,\sigma>} \subseteq \mathcal{T}_{<B,\sigma>}$), and B_1 is used for $A{:}\sigma$. By theorem 6.2.2.12, if ${<}B_2,\sigma'{>} = {<}\mu,n,B,\sigma{>}\,({<}B_1,\sigma{>})$, then $B_2 \vdash_{\mathrm{S}} A{:}\sigma'$. Since $B \leq_{\mathrm{S}} B_1$, by lemma 6.2.2.3 (iv) $B' \leq_{\mathrm{S}} B_2$, so by lemma 4.1.6 $B' \vdash_{\mathrm{S}} A{:}\sigma'$. □

6.2.3 *Lifting*

The operation on pairs defined in this subsection forms the strict counterpart of the operation of rise, that consists of applying an extra derivation rule ($\leq$) to a derivation. As mentioned in the beginning of this chapter, in the BCD-system the derivation rule ($\leq$) can be applied anywhere in the derivation, and the derivation rule ($\cap$E) can be omitted since it is derivable from rule ($\leq$).

The main difference between the BCD-system and the strict one is that, in the latter, the derivation rule ($\leq$) is omitted. In the Strict Type Assignment System the only part of the ($\leq$)-rule that is kept is the derivation rule ($\cap$E). Moreover, this rule can only be applied to term-variables. Therefore, the strict counterpart of the operation of rise will in fact be introducing extra ($\cap$E)-rules (or extra types in the upper half of that rule) for premisses.

In general, the derivation

$$\cfrac{\begin{array}{c}[x{:}\sigma]\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\to\tau}\;(\to\mathrm{I}) \qquad \text{will be transformed into:} \qquad \cfrac{\begin{array}{c}\cfrac{[x{:}\sigma\cap\rho]}{x{:}\sigma}\;(\cap\mathrm{E})\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\cap\rho\to\tau}\;(\to\mathrm{I})$$

The notion of lifting as defined in this section is not sound on all pairs ${<}B,\sigma{>}$. Take, for example, the pair ${<}\emptyset,(\alpha\to\beta)\to\alpha\to\beta{>}$. This is a pair for both $\lambda xy.xy$ and $\lambda z.z$.

$$\cfrac{\cfrac{\cfrac{[x{:}\alpha\to\beta] \qquad [y{:}\alpha]}{xy{:}\beta}}{\lambda y.xy{:}\alpha\to\beta}}{\lambda xy.xy{:}(\alpha\to\beta)\to\alpha\to\beta} \qquad\qquad \cfrac{[z{:}\alpha\to\beta]}{\lambda z.z{:}(\alpha\to\beta)\to\alpha\to\beta}$$

Suppose there is an operation *O* which enables to express that in the derivation for

$$\lambda xy.xy{:}(\alpha\to\beta)\to\alpha\to\beta,$$

the predicate for the term-variable y is lifted to $\alpha\cap\gamma$, which corresponds to the following derivation:

$$\cfrac{\cfrac{\cfrac{[x{:}\alpha\to\beta] \qquad \cfrac{[y{:}\alpha\cap\gamma]}{y{:}\alpha}}{xy{:}\beta}}{\lambda y.xy{:}\alpha\cap\gamma\to\beta}}{\lambda xy.xy{:}(\alpha\to\beta)\to\alpha\cap\gamma\to\beta}$$

Then *O* should be such that *O* $({<}\emptyset,(\alpha\to\beta)\to\alpha\to\beta{>}) = {<}\emptyset,(\alpha\to\beta)\to\alpha\cap\gamma\to\beta{>}$. However, ${<}\emptyset,(\alpha\to\beta)\to\alpha\cap\gamma\to\beta{>}$ is not a suitable pair for $\lambda z.z$: there exists no derivation in $\vdash_{\mathrm{S}}$ for

$$\lambda z.z{:}(\alpha\to\beta)\to\alpha\cap\gamma\to\beta.$$

So it is not true that for every M, B and σ: if $B \vdash_{\mathrm{S}} M{:}\sigma$, and *O* $({<}B,\sigma{>}) = {<}B',\sigma'{>}$, then also $B' \vdash_{\mathrm{S}} M{:}\sigma'$. Therefore, we are not able to prove that this *O* produces correct results for all lambda terms. We will however show that the operation defined here is closed for a certain class of pairs, being the primitive pairs. The above problem is solved by allowing liftings only on primitive pairs for terms: the pair ${<}\emptyset,(\alpha\to\beta)\to\alpha\to\beta{>}$ is a primitive pair for the term $\lambda xy.xy$, not for the term $\lambda z.z$.

We now come to the definition of lifting. It is based on the relation $\leq_{\mathrm{E}}$, in the same way as the operation of rise is based on $\leq$.

Definition 6.2.3.1 A *lifting L* is an operation denoted by a pair of pairs ${<}{<}B_0,\tau_0{>},{<}B_1,\tau_1{>}{>}$ such that $\tau_0 \leq_{\mathrm{E}} \tau_1$ and $B_1 \leq_{\mathrm{E}} B_0$, and is defined by:

i) a) $L(\sigma) = \tau_1$, if $\sigma = \tau_0$.
   b) $L(\sigma) = \sigma$, otherwise.
ii) a) $L(B) = B_1$, if $B = B_0$.
   b) $L(B) = B$, otherwise.
iii) $L({<}B,\sigma{>}) = {<}L(B),L(\sigma){>}$.

For operations of lifting, the following properties hold:

Lemma 6.2.3.2 i) ${<}{<}B\cup\{x{:}\sigma\},\tau{>},{<}B'\cup\{x{:}\sigma'\},\tau'{>}{>}$ is a lifting (where $\tau,\tau' \in \mathcal{T}_{\mathrm{S}}$), if and only if ${<}{<}B,\sigma\to\tau{>},{<}B',\sigma'\to\tau'{>}{>}$ is a lifting.
ii) ${<}{<}B_i,\sigma_i{>},{<}B_i{}',\sigma_i{}'{>}{>}$ is a lifting for every $1 \leq i \leq n$, if and only if
$${<}{<}\Pi\{B_1,\ldots,B_n,\{x{:}\sigma_1\to\cdots\to\sigma_n\to\varphi\}\},\varphi{>},$$
$${<}\Pi\{B_1{}',\ldots,B_n{}',\{x{:}\sigma_1{}'\to\cdots\to\sigma_n{}'\to\varphi\}\},\varphi{>}{>}$$
is a lifting.

Proof: By definitions 5.1.1 and 6.2.3.1. □

Although the strict type assignment is not closed for $\leq_{\mathrm{E}}$, we will show that there is a set of primitive pairs – a subset of the set of pairs – for which the operation of lifting is sound. The definition for primitive pairs we will give here is based on the definition of ground pairs given in definition 6.2.2.8. The main difference between ground pairs and primitive pairs is that, in a primitive pair, a predicate for a term-variable x (bound or free) is not the smallest type needed, but it can contain some additional, irrelevant types. Crucial in the definition is that a pair is called a primitive pair for an application if the type in the pair is a type-variable, and that term-variables that are not head-variables of a term $xA_1 \ldots A_m$ are also typed with type-variables.

Definition 6.2.3.3 The pair $\langle B, \sigma \rangle$ is a *primitive pair* for A if and only if:

i) If $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$ $(n \geq 0)$, then for every $1 \leq i \leq n$, $\langle B, \sigma_i \rangle$ is a primitive pair for A.

ii) If $\sigma \in \mathcal{T}_s$, then:

a) If $A \equiv x$, then $\sigma = \varphi$, and $B \leq_S \{x{:}\varphi\}$.

b) If $A \equiv \lambda x.A'$, then there are $\alpha \in \mathcal{T}_S$ and $\beta \in \mathcal{T}_s$, such that: $\sigma = \alpha \to \beta$, and $\langle B \cup \{x{:}\alpha\}, \beta \rangle$ is a primitive pair for A'.

c) If $A \equiv xA_1 \ldots A_m$, then there are $\tau_1, \ldots, \tau_m \in \mathcal{T}_S$, and a type-variable φ such that $\sigma = \varphi$, $B \leq_S \{x{:}\tau_1 \to \cdots \to \tau_m \to \varphi\}$, and for every $1 \leq j \leq m$, $\langle B, \tau_j \rangle$ is a primitive pair for A_j.

Notice that in this definition the relation $\leq_S$ is used, not $\leq_E$.

For primitive pairs, the following properties hold.

Lemma 6.2.3.4 i) If $\langle B, \sigma \rangle$ is a primitive pair for A, then $B \vdash_S A{:}\sigma$.

ii) For every A, every ground pair for A is a primitive pair for A.

Proof: By induction on the definitions of primitive pairs and ground pairs. □

The next theorem states that a lifting performed on a primitive pair for A produces a primitive pair for A.

Theorem 6.2.3.5 For all $A \in \mathcal{N}$, liftings L: if $L(\langle B, \sigma \rangle) = \langle B', \sigma' \rangle$, and $\langle B, \sigma \rangle$ is a primitive pair for A, then $\langle B', \sigma' \rangle$ is a primitive pair for A.

Proof: By induction on the definition of primitive pairs.

i) $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$ $(n \geq 0)$, and for every $1 \leq i \leq n$ $\langle B, \sigma_i \rangle$ is a primitive pair for A. Since $\sigma_1 \cap \cdots \cap \sigma_n \leq_E \sigma'$, by lemma 5.1.3 (i) there are $\sigma_1', \ldots, \sigma_m'$ such that $\sigma' = \sigma_1' \cap \cdots \cap \sigma_m'$, and for every $1 \leq j \leq m$ there is a $1 \leq i_j \leq n$ such that $\sigma_{i_j} \leq_E \sigma_j'$. Take for every $1 \leq j \leq m$ the lifting

$$L_j = \langle \langle B, \sigma_{i_j} \rangle, \langle B', \sigma_j' \rangle \rangle,$$

then by induction for every $1 \leq j \leq m$ $\langle B', \sigma_j' \rangle$ is a primitive pair for A. So $\langle B', \sigma_1' \cap \cdots \cap \sigma_m' \rangle$ is a primitive pair for A.

ii) $\sigma \in \mathcal{T}_s$. This part is proved by induction on elements of $\mathcal{N}$. Notice that we need not consider the case that $A \equiv \bot$.

a) $A \equiv x$, and $\sigma = \varphi$. By lemmas 5.1.2 (ii) and 5.1.3 (ii) $\sigma' = \varphi$, so by lemma 5.1.3 (iii) $B' \leq_S \{x{:}\varphi\}$. So $\langle B', \varphi \rangle$ is a primitive pair for x.

b) $A \equiv \lambda x.A'$. Then there are α and β, such that $\sigma = \alpha \to \beta$ and $\langle B \cup \{x{:}\alpha\}, \beta \rangle$ is a primitive pair for A'. Then by lemmas 5.1.2 (iv) and 5.1.3 (ii) there are $\rho_1, \ldots, \rho_n$, $\mu_1, \ldots, \mu_n$ such that $\sigma' = (\rho_1 \to \mu_1) \cap \cdots \cap (\rho_n \to \mu_n)$, and for $1 \leq i \leq n$, $\rho_i \leq_E \alpha$ and $\beta \leq_E \mu_i$. Take, for every $1 \leq i \leq n$, the lifting

$$L_i = \langle \langle B \cup \{x{:}\alpha\}, \beta \rangle, \langle B' \cup \{x{:}\rho_i\}, \mu_i \rangle \rangle,$$

then by induction $\langle B' \cup \{x{:}\rho_i\}, \mu_i \rangle$ is a primitive pair for A' for every $1 \leq i \leq n$. So for every $1 \leq i \leq n$, $\langle B', \rho_i \to \mu_i \rangle$ is a primitive pair for A, so $\langle B', \sigma' \rangle$ is a primitive pair for A.

c) $A \equiv xA_1 \ldots A_m$, there are $\tau_1, \ldots, \tau_m$ and φ such that $B \leq_S \{x{:}\tau_1 \to \cdots \to \tau_m \to \varphi\}$, $\sigma = \varphi$, and for every $1 \leq j \leq m$, $\langle B_j, \tau_j \rangle$ is a primitive pair for A_j. Then by lemmas 5.1.2 (ii) and 5.1.3 (ii) $\sigma' = \varphi$, and by lemmas 5.1.2 (iv) and 5.1.3 (iii) there are τ_1', $\ldots, \tau_m' \in \mathcal{T}_S$ such that

$$B' \leq_S \{x{:}\tau_1' \to \cdots \to \tau_m' \to \varphi\}, \text{ and for } 1 \leq j \leq m \ \tau_j \leq_E \tau_j'.$$

Take for $1 \leq j \leq m$ the lifting $L_j = \langle \langle B, \tau_j \rangle, \langle B', \tau_j' \rangle \rangle$, then by induction $\langle B', \tau_j' \rangle$ is a primitive pair for A_j. So $\langle B', \varphi \rangle$ is a primitive pair for A. □

Example 6.2.3.6 i) Take the pair $\langle \{y{:}(\varphi_0 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2\}, \varphi_2 \rangle$, which is a primitive pair for the lambda term $y(\lambda x.x)$.

$$\dfrac{y{:}(\varphi_0 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2 \qquad \dfrac{[x{:}\varphi_0]}{\lambda x.x{:}\varphi_0 \to \varphi_0} \qquad \dfrac{[x{:}\varphi_1]}{\lambda x.x{:}\varphi_1 \to \varphi_1}}{y(\lambda x.x){:}\varphi_2}$$

Since $(\varphi_0 \cap \varphi_3 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2 \leq_E (\varphi_0 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2$, the pair

$$\langle \{y{:}(\varphi_0 \cap \varphi_3 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2\}, \varphi_2 \rangle$$

is a primitive pair for $y(\lambda x.x)$.

$$\dfrac{y{:}(\varphi_0 \cap \varphi_3 \to \varphi_0) \cap (\varphi_1 \to \varphi_1) \to \varphi_2 \qquad \dfrac{\dfrac{[x{:}\varphi_0 \cap \varphi_3]}{x{:}\varphi_0}}{\lambda x.x{:}\varphi_0 \cap \varphi_3 \to \varphi_0} \qquad \dfrac{[x{:}\varphi_1]}{\lambda x.x{:}\varphi_1 \to \varphi_1}}{y(\lambda x.x){:}\varphi_2}$$

ii) Take the pair $\langle \emptyset, ((\varphi_0 \to \omega \to \varphi_0) \to \varphi_1) \to \varphi_1 \rangle$, which is a primitive pair for the lambda term $\lambda z.z(\lambda xy.x)$.

$$\dfrac{\dfrac{[z{:}(\varphi_0 \to \omega \to \varphi_0) \to \varphi_1] \qquad \dfrac{\dfrac{[x{:}\varphi_0]}{\lambda y.x{:}\omega \to \varphi_0}}{\lambda xy.x{:}\varphi_0 \to \omega \to \varphi_0}}{z(\lambda xy.x){:}\varphi_1}}{\lambda z.z(\lambda xy.x){:}((\varphi_0 \to \omega \to \varphi_0) \to \varphi_1) \to \varphi_1}$$

Since $((\varphi_0 \to \omega \to \varphi_0) \to \varphi_1) \to \varphi_1 \leq_E ((\varphi_0 \to \varphi_2 \to \varphi_0) \to \varphi_1) \to \varphi_1$, the pair

$$\langle \emptyset, ((\varphi_0 \to \varphi_2 \to \varphi_0) \to \varphi_1) \to \varphi_1 \rangle$$

is a primitive pair for $\lambda z.z(\lambda xy.x)$.

$$\dfrac{\dfrac{[z{:}(\varphi_0 \to \varphi_2 \to \varphi_0) \to \varphi_1] \qquad \dfrac{\dfrac{[x{:}\varphi_0]}{\lambda y.x{:}\varphi_2 \to \varphi_0}}{\lambda xy.x{:}\varphi_0 \to \varphi_2 \to \varphi_0}}{z(\lambda xy.x){:}\varphi_1}}{\lambda z.z(\lambda xy.x){:}((\varphi_0 \to \varphi_2 \to \varphi_0) \to \varphi_1) \to \varphi_1}$$

6.3 *Completeness for lambda terms*

Although lifting is not sound on all pairs, we are able to prove that the three operations defined in the previous section are sufficient (i.e. complete): for every pair $\langle B, \sigma\rangle$ and $A \in \mathcal{N}$, if $B \vdash_{\mathrm{S}} A{:}\sigma$, then there exists a number of operations of expansion, lifting, and substitution, such that $\langle B, \sigma\rangle$ can be obtained from $PP_{\mathrm{S}}(A)$ by performing these operations in sequence. In the next section we will generalize this result to arbitrary lambda terms.

6.3.1 *Soundness and completeness for terms in λ⊥-normal form*

Definition 6.3.1.1 i) A *chain* is an object $\langle O_1, \ldots, O_n\rangle$, where each O_i is an operation of expansion, lifting, or substitution, and

$$\langle O_1, \ldots, O_n\rangle(\langle B, \sigma\rangle) = O_n(\cdots(O_1(\langle B, \sigma\rangle))\cdots).$$

ii) On chains we denote the operation of concatenation by $*$, and

$$\langle O_1, \ldots, O_i\rangle * \langle O_{i+1}, \ldots, O_n\rangle = \langle O_1, \ldots, O_n\rangle.$$

iii) A *strict chain* is a chain of a number of expansions, one lifting, and a number of substitutions, in that order. So a strict chain $C = \langle E_1, \ldots, E_m, L, S_1, \ldots, S_n\rangle$, where $m \geq 0$, and $n \geq 0$. We also write $C = \vec{E} * \langle L\rangle * \vec{S}$.

We will now prove that, for every suitable pair for a term A, there exists a strict chain such that the result of the application of this chain on the principal pair of A produces the desired pair. Part (i) of the lemmas 6.2.1.3, 6.2.2.7, and 6.2.3.2 are needed for the inductive step in case of an abstraction term, part (iii.b) of the proof, part (ii) of these lemma's are needed for the inductive step in case of an application term, part (iii.c). Notice that, by construction, all operations mentioned in this part satisfy the conditions required by these lemmas.

Theorem 6.3.1.2 If $B \vdash_{\mathrm{S}} A{:}\sigma$, and $PP_{\mathrm{S}}(A) = \langle P, \pi\rangle$, then there exists a strict chain C such that $C(\langle P, \pi\rangle) = \langle B, \sigma\rangle$.

Proof: By induction on the definition of $\vdash_{\mathrm{S}}$.

i) $B \vdash_{\mathrm{S}} A{:}\omega$. Assume, without loss of generality, that P and B are disjoint. Take

$$L = \langle\langle P, \pi\rangle, \langle\Pi\{P, B\}, \omega\rangle\rangle,$$

and S the substitution that replaces all type-variables occurring in P by ω.
Take $C = \langle L, S\rangle$.

ii) $B \vdash_{\mathrm{S}} A{:}\sigma_1\cap\cdots\cap\sigma_n$. Then $B \vdash_{\mathrm{S}} A{:}\sigma_i$. For $1 \leq i \leq n$, take $\langle B_i, \tau_i\rangle$, which are trivial variants of $\langle B, \sigma_i\rangle$ that are disjoint in pairs. Take S such that

$$S(\langle\Pi\{B_1, \ldots, B_n\}, \tau_1\cap\cdots\cap\tau_n\rangle) = \langle B, \sigma_1\cap\cdots\cap\sigma_n\rangle,$$

and let $E = \langle\pi, n\rangle$, then

$$E(\langle P, \pi\rangle) = \langle\Pi\{P_1, \ldots, P_n\}, \pi_1\cap\cdots\cap\pi_n\rangle,$$

with $PP_{\mathrm{S}}(A) = \langle P_i, \pi_i\rangle$ for every $1 \leq i \leq n$. By induction there exist strict chains $C_1, \ldots, C_n$ such that for $1 \leq i \leq n$ $C_i(\langle P_i, \pi_i\rangle) = \langle B_i, \tau_i\rangle$.
Let for $1 \leq i \leq n$, $C_i = \vec{E}_i * \langle L_i\rangle * \vec{S}_i$. Let $\vec{E} = \vec{E}_1 * \cdots * \vec{E}_n$, and $\vec{S} = \vec{S}_1 * \cdots * \vec{S}_n$.
Let for $1 \leq i \leq n$, $\vec{E}_i(\langle P_i, \pi_i\rangle) = \langle B_i', \tau_i'\rangle$; by construction there are B_i'', τ_i'', such that $L_i = \langle\langle B_i', \tau_i'\rangle, \langle B_i'', \tau_i''\rangle\rangle$. Take

$$L = \langle\langle\Pi\{B_1', \ldots, B_n'\}, \tau_1'\cap\cdots\cap\tau_n'\rangle, \langle\Pi\{B_1'', \ldots, B_n''\}, \tau_1''\cap\cdots\cap\tau_n''\rangle\rangle.$$

Take $C = \langle E\rangle * \vec{E} * \langle L\rangle * \vec{S} * \langle S\rangle$.

iii) $B \vdash_{\mathrm{s}} A{:}\sigma$, so $\sigma \in \mathcal{T}_{\mathrm{s}}$. This part is proved by induction on the structure of elements of $\mathcal{N}$. Notice that we need not consider the case that $A \equiv \bot$.

a) $A \equiv x$, $B \leq_{\mathrm{S}} \{x{:}\sigma\}$, $P = \{x{:}\varphi\}$, and $\pi = \varphi$. Assume, without loss of generality, that P and B are disjoint. Take $L = \langle\langle P, \varphi\rangle, \langle\Pi\{P, B\}, \varphi\rangle\rangle$, and $S = (\varphi := \sigma)$. Take $C = \langle L, S\rangle$.

b) $A \equiv \lambda x.A'$. Then there are α, β such that $\sigma = \alpha{\to}\beta$, and $B \cup \{x{:}\alpha\} \vdash_{\mathrm{S}} A'{:}\beta$.
We will distinguish two cases:

1) $x \in \mathrm{FV}(A')$. Then $PP_{\mathrm{S}}(\lambda x.A') = \langle P, \mu{\to}\pi\rangle$, with $PP_{\mathrm{S}}(A') = \langle P{\cup}\{x{:}\mu\}, \pi\rangle$.
By induction there exists a strict chain $C' = \vec{E} * \langle L'\rangle * \vec{S}$ such that

$$C'(\langle P \cup \{x{:}\mu\}, \pi\rangle) = \langle B \cup \{x{:}\alpha\}, \beta\rangle.$$

Let

$$\vec{E}(\langle P \cup \{x{:}\mu\}, \pi\rangle) = \langle B_1 \cup \{x{:}\alpha_1\}, \beta_1\rangle, \text{ and}$$
$$L' = \langle\langle B_1 \cup \{x{:}\alpha_1\}, \beta_1\rangle, \langle B_2 \cup \{x{:}\alpha_2\}, \beta_2\rangle\rangle.$$

Since $\beta \in \mathcal{T}_{\mathrm{s}}$, by construction also $\beta_2 \in \mathcal{T}_{\mathrm{s}}$ and by lemma 6.2.2.7 (i) $\vec{E}(\langle P, \mu{\to}\pi\rangle) = \langle B_1, \alpha_1{\to}\beta_1\rangle$. Take

$$L = \langle\langle B_1, \alpha_1{\to}\beta_1\rangle, \langle B_2, \alpha_2{\to}\beta_2\rangle\rangle$$

which is by lemma 6.2.3.2 (i) a lifting.

2) $x \notin \mathrm{FV}(A')$. Then $PP_{\mathrm{S}}(\lambda x.A') = \langle P, \omega{\to}\pi\rangle$, where $PP_{\mathrm{S}}(A') = \langle P, \pi\rangle$.
By induction there exists a strict chain $C' = \vec{E} * \langle L'\rangle * \vec{S}$ such that

$$C'(\langle P, \pi\rangle) = \langle B \cup \{x{:}\alpha\}, \beta\rangle.$$

Let $\vec{E}(\langle P, \pi\rangle) = \langle B_1, \beta_1\rangle$, and $L' = \langle\langle B_1, \beta_1\rangle, \langle B_2 \cup \{x{:}\alpha_2\}, \beta_2\rangle\rangle$.
Since $\beta \in \mathcal{T}_{\mathrm{s}}$, by construction also $\beta_2 \in \mathcal{T}_{\mathrm{s}}$ and by lemma 6.2.2.7 (i) $\vec{E}(\langle P, \omega{\to}\pi\rangle) = \langle B_1, \omega{\to}\beta_1\rangle$. Take

$$L = \langle\langle B_1, \omega{\to}\beta_1\rangle, \langle B_2, \alpha_2{\to}\beta_2\rangle\rangle$$

which is by lemma 6.2.3.2 (i) a lifting.

Notice that in both cases by theorems 6.2.2.10 and 6.2.3.5, $\langle B_2, \alpha_2{\to}\beta_2\rangle$ is a primitive pair for $\lambda x.A'$, and by lemma 6.2.1.3 (i) $\vec{S}(\langle B_2, \alpha_2{\to}\beta_2\rangle) = \langle B, \alpha{\to}\beta\rangle$.
Take $C = \vec{E} * \langle L\rangle * \vec{S}$.

c) $A \equiv xA_1 \ldots A_m$. Then there are $\sigma_1, \ldots, \sigma_m$ such that $B \leq_{\mathrm{S}} \{x{:}\sigma_1{\to}\cdots{\to}\sigma_m{\to}\sigma\}$, and for every $1 \leq j \leq m$ $B \vdash_{\mathrm{S}} A_j{:}\sigma_j$, and

$$P = \Pi\{P_1, \ldots, P_m, \{x{:}\pi_1{\to}\cdots{\to}\pi_m{\to}\varphi\}\}, \ \pi = \varphi,$$

with for every $1 \leq j \leq m$, $PP_{\mathrm{S}}(A_j) = \langle P_j, \pi_j\rangle$, in which φ does not occur. Take $\langle B_j, \tau_j\rangle$, trivial variants of $\langle B, \sigma_j\rangle$, that are disjoint in pairs. Let S be such that

$$S(\langle\Pi\{B_1, \ldots, B_m\}, \tau_1\cap\cdots\cap\tau_m\rangle) = \langle B, \sigma_1\cap\cdots\cap\sigma_m\rangle.$$

By induction there are strict chains $C_1, \ldots, C_m$ such that for $1 \leq j \leq m$ $C_j(\langle P_j, \pi_j\rangle) = \langle B_j, \tau_j\rangle$. Notice that the strict chains $C_1, \ldots, C_m$ do not interfere, and that φ does not occur in any of the operations or pairs. Let for $1 \leq j \leq m$, $C_j = \vec{E}_j * \langle L_j\rangle * \vec{S}_j$, and $\vec{E} = \vec{E}_1 * \cdots * \vec{E}_m$, and $\vec{S} = \vec{S}_1 * \cdots * \vec{S}_m$. Let for $1 \leq j \leq m$, $\vec{E}_i\,(<P_j, \pi_j>) = <B_j', \tau_j'>$, and $L_j = <<B_j', \tau_j'>, <B_j'', \tau_j''>>$. Then by lemma 6.2.2.7 (ii)

$$\vec{E}\,(<\Pi\{P_1, \ldots, P_m, \{x{:}\pi_1\rightarrow\cdots\rightarrow\pi_m\rightarrow\varphi\}\}, \varphi>) = \\ <\Pi\{B_1', \ldots, B_m', \{x{:}\tau_1'\rightarrow\cdots\rightarrow\tau_m'\rightarrow\varphi\}\}, \varphi>.$$

Take

$$L = <<\Pi\{B_1', \ldots, B_m', \{x{:}\tau_1'\rightarrow\cdots\rightarrow\tau_m'\rightarrow\varphi\}\}, \varphi>>, \\ <\Pi\{B_1'', \ldots, B_m'', \{x{:}\tau_1''\rightarrow\cdots\rightarrow\tau_m''\rightarrow\varphi\}\}, \varphi>,$$

which is by lemma 6.2.3.2 (ii) a lifting. By lemma 6.2.1.3 (ii)

$$\vec{S}\,(<\Pi\{B_1'', \ldots, B_m'', \{x{:}\tau_1''\rightarrow\cdots\rightarrow\tau_m''\rightarrow\varphi\}\}, \varphi>) = \\ <\Pi\{B_1, \ldots, B_m, \{x{:}\sigma_1\rightarrow\cdots\rightarrow\sigma_m\rightarrow\varphi\}\}, \varphi>.$$

Take $C = \vec{E} * <L> * \vec{S} * <S, (\varphi := \sigma)>$. □

Theorem 6.3.1.3 i) *Soundness of strict chains.* If $PP_S\,(A) = <P, \pi>$, and there exists a strict chain C such that $C\,(<P, \pi>) = <B, \sigma>$, then $B \vdash_S A{:}\sigma$.

ii) *Completeness of strict chains.* If $B \vdash_S A{:}\sigma$, and $PP_S\,(A) = <P, \pi>$, then there exists a strict chain C such that $C\,(<P, \pi>) = <B, \sigma>$.

Proof: i) By theorems 6.2.2.10, 6.2.3.5, and 6.2.1.4.

ii) By theorem 6.3.1.2. □

6.3.2 *Principal pairs for lambda terms*

We will conclude the proofs of this chapter by, like in subsections 2.2.2 and 2.3.2, generalizing the concept of principal pairs to arbitrary lambda terms, using that $<B, \sigma>$ is a suitable pair for M, if and only if there is an $A \in \mathcal{A}(M)$ such that $<B, \sigma>$ is a suitable pair for A.

Definition 6.3.2.1 (cf. [Ronchi della Rocca & Venneri '84]) i) For all terms M we define the set $\Pi_S\,(M)$ as follows:

$$\Pi_S\,(M) = \{\ <P, \pi> \mid \exists\, A \in \mathcal{A}(M)\ [\ PP_S\,(A) = <P, \pi>\ ]\ \}$$

ii) Like in definition 2.3.2.6 (iii), on $\mathcal{P}_S$ it is possible to define the preorder relation $\sqsubseteq_\omega$ by:

$$<P, \pi> \sqsubseteq_\omega <P', \pi'> \Leftrightarrow \\ \exists\, \varphi_1, \ldots, \varphi_n\ [\ <P, \pi> = (\varphi_1 := \omega)\circ\cdots\circ(\varphi_n := \omega)\,(<P', \pi'>)\ ].$$

Since our definition of principal pairs of an approximate normal form coincides with the corresponding definition in [Coppo *et al.* '80] and [Ronchi della Rocca & Venneri '84], we can use the following result proved there:

Theorem 6.3.2.2 [Coppo *et al.* '80, Ronchi della Rocca & Venneri '84] i) $\mathcal{P}_S, \sqsubseteq_\omega$ is a meet semilattice isomorphic to $\mathcal{N}, \leq$.

ii) $\Pi_S\,(M)$ is an ideal in $\mathcal{P}_S$ and therefore:

a) If $\Pi_S\,(M)$ is finite, there exists a pair $<P, \pi> = \sqcup\Pi_S\,(M)$, where $<P, \pi> \in \mathcal{P}_S$. This pair is then called the principal pair of M.

b) If $\Pi_S\,(M)$ is infinite, $\sqcup\Pi_S\,(M)$ does not exist in $\mathcal{P}_S$. The principal pair of M will then be the infinite set of pairs $\Pi_S\,(M)$. □

By theorems 4.3.4 and 6.3.1.3, we have the following:

Theorem 6.3.2.3 (cf. [Ronchi della Rocca & Venneri '84]) Let $M \in \Lambda$, and B and σ such that $B \vdash_S M{:}\sigma$.

i) $\mathcal{A}(M)$ is finite. Let $<P, \pi>$ be the principal pair of M. Then there exists a strict chain C such that $C\,(<P, \pi>) = <B, \sigma>$.

ii) $\mathcal{A}(M)$ is infinite. Then there exist a pair $<P, \pi> \in \Pi_S\,(M)$, and a strict chain C such that $C\,(<P, \pi>) = <B, \sigma>$. □

6.4 *Principal pairs for the essential type assignment system*

It is possible to prove the principal type property for the essential type assignment system, the same way as done in [Ronchi della Rocca & Venneri '84] for the BCD-system. The operations needed for this proof would be strict substitution, strict expansion, and lifting, and it is possible to show that all pairs for a term can be generated by chains that exist of expansions, and substitutions (in that order) and end with one lifting. Since these results would be obtained in exactly the same way as in [Ronchi della Rocca & Venneri '84], we will not present them here. We just remark that all three operations can be proved to be sound on all pairs; part of the proofs needed for these results can be found in section 10.2.

It is worthwhile to observe that the set of operations for both systems would be exactly the same. The only difference between the chains that produce pairs for both systems is the place of the allowed operations of lifting. As for the strict system, they are only allowed directly after expansions, and *before* substitutions; for the essential system they are allowed after expansions *and* substitutions.

Chapter 7 *The Barendregt-Coppo-Dezani Type Assignment System without ω*

In this chapter we will present a type assignment system that is a restriction of the BCD-system. Its major feature is the elimination of the type constant ω, and it is an extension of the CD-system.

While building a derivation $B \vdash_{\text{BCD}} M{:}\sigma$ (where ω does not occur in σ and B) for a lambda term M that has a normal form, the type ω is only needed to type sub-terms that will be erased while reducing M to its normal form and that cannot be typed starting from B. This, together with the results of [Coppo & Dezani-Ciancaglini '80], gives rise to the idea that, if we limit ourselves to the set of lambda terms where no sub-terms will be erased – i.e. the λI-calculus – the type ω is not really needed for terms that have a normal form. We will show that the intersection type assignment system without ω yields a filter model for the λI-calculus. We will also show that for the λI-calculus the BCD-type assignment is conservative over the one without ω, and prove that this type assignment is complete for the λI-calculus with respect to the simple type semantics. Furthermore we will prove that the set of all terms typeable by the system without ω is the set of all strongly normalizable terms.

While obtaining these results, we could of course use the result of chapter four and look at the system without $\leq$ and ω, but, since this is a more restricted system, we prefer the approach we use in this chapter. Also, the proofs of various lemmas in section 7.4 are greatly facilitated by the presence of derivation rule ($\leq$); the technique used for the proof of the main theorem of this section requirers a notion of type assignment that is closed under η-reduction. In fact, the strong normalization property for the system without $\leq$ and ω follows immediately from the results of 7.4. Moreover, we could prove a completeness result for this system with respect to the inference semantics.

7.1 *ω-free derivations*

In this section we present a restriction of the BCD-system in which the type ω is removed. This system yields a filter λI-model.

Definition 7.1.1 i) $\mathcal{T}_{-\omega}$, the set of ω*-free types* is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{-\omega}$.

b) If $\sigma, \tau \in \mathcal{T}_{-\omega}$, then $\sigma\cap\tau, \sigma\rightarrow\tau \in \mathcal{T}_{-\omega}$.

ii) On $\mathcal{T}_{-\omega}$ the relation $\leq$ is as defined in definition 2.3.1 (ii), but without parts (ii.b) and (ii.c).

iii) If $M{:}\sigma$ is derivable from a basis B, using only ω-free types and the derivation rules ($\cap$I), ($\cap$E), ($\rightarrow$I), ($\rightarrow$E) or ($\leq$) of the BCD-system, we write $B \vdash_{-\omega} M{:}\sigma$.

Lemma 7.1.2 i) $B \vdash_{-\omega} MN{:}\sigma \iff \exists\, \tau\, [\, B \vdash_{-\omega} M{:}\tau\rightarrow\sigma \;\&\; B \vdash_{-\omega} N{:}\tau \,]$.

ii) $B \vdash_{-\omega} \lambda x.M{:}\sigma\rightarrow\tau \iff B\backslash x \cup \{x{:}\sigma\} \vdash_{-\omega} M{:}\tau$.

iii) If $B \vdash_{-\omega} \lambda x.M{:}\rho$, then there are types σ_i, and τ_i ($1 \leq i \leq n$) such that $\rho = (\sigma_1\rightarrow\tau_1)\cap\cdots\cap(\sigma_n\rightarrow\tau_n)$.

iv) If $B\backslash z \cup \{z{:}\sigma\} \vdash_{-\omega} Mz{:}\tau$ and $z \notin \text{FV}(M)$, then $B \vdash_{-\omega} M{:}\sigma\rightarrow\tau$.

Proof: By induction on the structure of derivations, using property 2.3.4 (ii) to prove (ii). To prove part (iv), the derivation rule ($\leq$) is needed. □

Definition 7.1.3 i) A subset d of $\mathcal{T}_{-\omega}$ is called an *I-filter* if

a) $\sigma, \tau \in d \Rightarrow \sigma\cap\tau \in d$.

b) $\sigma \geq \tau \in d \Rightarrow \sigma \in d$.

ii) $\mathcal{F}_{-\omega} = \{\, d \subseteq \mathcal{T}_{-\omega} \mid d$ is an I-filter $\}$. We define application on $\mathcal{F}_{-\omega}$, $\cdot{:}\mathcal{F}_{-\omega} \times \mathcal{F}_{-\omega} \rightarrow \mathcal{F}_{-\omega}$ by:

$$d \cdot e = \{\, \tau \mid \exists\, \sigma \in e\, [\, \sigma\rightarrow\tau \in d\,]\, \}.$$

iii) If V is a subset of $\mathcal{T}_{-\omega}$, then $\uparrow_\omega V$ is the smallest I-filter that contains V. Also $\uparrow_\omega\sigma = \uparrow_\omega\{\sigma\}$. Since no confusion is possible, we will omit the subscript on $\uparrow$.

Notice that the empty set, $\emptyset$, is the bottom element of $\mathcal{F}_{-\omega}$.

Let $<\mathcal{D}, \leq>$ be a cpo with least element $\perp$. The set of strict functions is defined as usual, i.e. as the set of continuous functions that at least map $\perp$ onto $\perp$. We denote by $[\mathcal{D} \rightarrow_\perp \mathcal{D}]$ the set of strict functions from $\mathcal{D}$ to $\mathcal{D}$.

Definition 7.1.4 We define F: $\mathcal{F}_{-\omega} \rightarrow [\mathcal{F}_{-\omega}\rightarrow_\perp\mathcal{F}_{-\omega}]$ and G: $[\mathcal{F}_{-\omega}\rightarrow_\perp\mathcal{F}_{-\omega}] \rightarrow \mathcal{F}_{-\omega}$ by:

i) F $d\ e = d \cdot e$.

ii) G $f = \uparrow\{\, \sigma\rightarrow\tau \in \mathcal{T}_{-\omega} \mid \tau \in f(\uparrow\sigma)\, \}$.

It is, again, easy to check that F and G are continuous.

Definition 7.1.5 [Honsell & Ronchi della Rocca '84] Let $\cdot$ be a binary relation on the set $\mathcal{D}$. The structure $<\mathcal{D}, \cdot, \varepsilon>$ is called a λI-*model* if and only if, in $\mathcal{D}$, there are five elements **i**, **b**, **c**, **s** and ε that satisfy:

i) $\mathbf{i} \cdot d = d$.

ii) $((\mathbf{b} \cdot d) \cdot e) \cdot f = d \cdot (e \cdot f)$.

iii) $((\mathbf{c} \cdot d) \cdot e) \cdot f = (d \cdot f) \cdot e$.

iv) $((\mathbf{s} \cdot d) \cdot e) \cdot f = (d \cdot f) \cdot (e \cdot f)$.

v) $(\varepsilon \cdot d) \cdot e = d \cdot e \;\&\; \forall\, d \in \mathcal{D}\, [\, e \cdot d = f \cdot d \Rightarrow \varepsilon \cdot e = \varepsilon \cdot f\,] \;\&\; \varepsilon \cdot \varepsilon = \varepsilon$.

Moreover, in [Dezani-Ciancaglini *et al.* '86] the following is stated:

Proposition 7.1.6 [Dezani-Ciancaglini *et al.* '86] If $<\mathcal{D}, \leq>$ is a cpo and there are continuous maps F: $\mathcal{D}\rightarrow[\mathcal{D} \rightarrow_\perp \mathcal{D}]$ and G: $[\mathcal{D} \rightarrow_\perp \mathcal{D}]\rightarrow\mathcal{D}$ such that:

i) F∘G = $id_{[\mathcal{D}\rightarrow_\perp\mathcal{D}]}$

ii) G∘F $\in [\mathcal{D} \rightarrow_\perp \mathcal{D}]$

Then $\mathcal{D}$ is a λI-model. $\square$

Theorem 7.1.7 F and G as defined in definition 7.1.4 yield a λI-model.

Proof: It is sufficient to check that the conditions of proposition 7.1.6 are met.

i) $\mathrm{F{\circ}G}\, f\ d = \{\ \mu \mid \exists\, \rho \in d\ [\ \rho{\rightarrow}\mu \in {\uparrow}\{\sigma{\rightarrow}\tau \mid \tau \in f({\uparrow}\sigma)\}\ ]\ \} =$
$\{\ \mu \mid \exists\, \rho \in d\ [\ \mu \in f({\uparrow}\rho)\ ]\ \} = f(d)$.

ii) $\mathrm{G{\circ}F}\, \emptyset = {\uparrow}\{\ \rho{\rightarrow}\mu \mid \mu \in \{\ \sigma \mid \exists\, \tau \in {\uparrow}\rho\ [\ \tau{\rightarrow}\sigma \in \emptyset\ ]\ \}\ \} = \emptyset$. $\square$

That the type discipline without ω gives rise to a model for the λI-calculus is also proved in [Honsell & Ronchi della Rocca '84]. The technique used there is to build – using Scott's inverse limit construction – a model M2 satisfying the equation $\mathcal{D} \simeq \mathcal{P}_\omega{\times}[\mathcal{D} \rightarrow_\perp \mathcal{D}]$, with $\mathcal{D}_0 = \mathcal{P}_\omega$ (the powerset of natural numbers) and i: $\mathcal{D}_0{\rightarrow}\mathcal{P}_\omega{\times}[\mathcal{D}_0 \rightarrow_\perp \mathcal{D}_0]$ is defined by $i(d) = {<}d,\, \lambda x.\perp{>}$ (see also [Barendregt '84], exercise 18.4.26 and [Plotkin & Smyth '78].) It is straightforward to verify that $\mathcal{F}_{-\omega}$ is a solution of the same domain equation.

Definition 7.1.8 Let ξ be a valuation of term-variables in $\mathcal{F}_{-\omega}$.

i) $[\![M]\!]_\xi$, the interpretation of λI-terms in $\mathcal{F}_{-\omega}$ via ξ is inductively defined by:

a) $[\![x]\!]_\xi = \xi(x)$.

b) $[\![MN]\!]_\xi = \mathrm{F}\, [\![M]\!]_\xi [\![N]\!]_\xi$.

c) $[\![\lambda x.M]\!]_\xi = \mathrm{G}\, (\lambda\!\!\lambda\, v \in \mathcal{F}_{-\omega}.[\![M]\!]_{\xi(v/x)})$.

ii) $B_\xi = \{x{:}\sigma \mid \sigma \in \xi(x)\}$.

Notice that $\lambda\!\!\lambda$ is well defined in λI-models, since $(\lambda\!\!\lambda\, v \in \mathcal{F}_{-\omega}.[\![M]\!]_{\xi(v/x)})\ \emptyset = \emptyset$.

Theorem 7.1.9 For all $M \in \Lambda\mathrm{I}$, ξ: $[\![M]\!]_\xi = \{\ \sigma \mid B_\xi \vdash_{-\omega} M{:}\sigma\ \}$.

Proof: By induction on the structure of lambda terms.

i) $[\![x]\!]_\xi = \xi(x)$. Since $\{\ y{:}\rho \mid \rho \in \xi(y)\ \} \vdash_{-\omega} x{:}\sigma \iff \sigma \in \xi(x)$.

ii) $[\![MN]\!]_\xi = \{\ \tau \mid \exists\, \sigma\ [\ B_\xi \vdash_{-\omega} N{:}\sigma\ \&\ B_\xi \vdash_{-\omega} M{:}\sigma{\rightarrow}\tau\ ]\ \} =$ (7.1.2 (i))
$\{\ \tau \mid B_\xi \vdash_{-\omega} MN{:}\tau\ \}$.

iii) $[\![\lambda x.M]\!]_\xi = {\uparrow}\{\ \sigma{\rightarrow}\tau \mid \tau \in \{\ \rho \mid B_{\xi({\uparrow}\sigma/x)} \vdash_{-\omega} M{:}\rho\ \}\ \} =$
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid B_{\xi({\uparrow}\sigma/x)} \vdash_{-\omega} M{:}\tau\ \} =$ ($B_{\xi'} = B_\xi \backslash x$)
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid B_{\xi'} \cup \{x{:}\rho \mid \rho \in {\uparrow}\sigma\ \} \vdash_{-\omega} M{:}\tau\} =$
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid \exists\, \mu \in {\uparrow}\sigma\ [\ B_{\xi'} \cup \{x{:}\mu\} \vdash_{-\omega} M{:}\tau\ ]\ \} =$ (7.1.2 (ii))
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid \exists\, \mu \in {\uparrow}\sigma\ [\ B_{\xi'} \vdash_{-\omega} \lambda x.M{:}\mu{\rightarrow}\tau\ ]\ \} =$ ($\leq$)
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid B_{\xi'} \vdash_{-\omega} \lambda x.M{:}\sigma{\rightarrow}\tau\ \} =$
${\uparrow}\{\ \sigma{\rightarrow}\tau \mid B_\xi \vdash_{-\omega} \lambda x.M{:}\sigma{\rightarrow}\tau\ \} =$ (7.1.2 (iii))
$\{\ \rho \mid B_\xi \vdash_{-\omega} \lambda x.M{:}\rho\ \}$. $\square$

Notice that F and G do not yield a lambda model. For example: take $O = \lambda xy.y$, $D = \lambda x.xx$ and $I = \lambda x.x$. Then, clearly, $O(DD) =_\beta I$ and $\vdash_{-\omega} I{:}\sigma{\rightarrow}\sigma$, but we cannot give a derivation without ω for $O(DD){:}\sigma{\rightarrow}\sigma$.

7.2 *The relation between $\vdash_{-\omega}$ and $\vdash_{\mathrm{BCD}}$*

Type assignment in the BCD-system is not fully conservative over the type assignment without ω. If, for example, we have $B \vdash_{\mathrm{BCD}} M{:}\sigma$ such that B and σ are ω-free, but M contains a sub-term that has no normal form, ω is needed in the derivation. (See the final remark of the previous section.) However, we can prove that for every lambda-term M such that $B \vdash_{\mathrm{BCD}} M{:}\sigma$ with B and σ ω-free, there is an M' such that $M \rightarrow_\beta M'$ and $B \vdash_{-\omega} M'{:}\sigma$. We will show this by proving for terms in normal form, that each ω-free predicate, starting from a ω-free basis, can be derived in $\vdash_{-\omega}$, and afterwards by using property 2.3.4 (iv). We will use the same technique to prove a conservativity result.

Lemma 7.2.1 If M is in normal form and $B \vdash_{\mathrm{BCD}} M{:}\sigma$ with B and σ ω-free, then $B \vdash_{-\omega} M{:}\sigma$.

Proof: The proof is given by induction on the structure of terms in normal form.

i) $B \vdash_{\mathrm{BCD}} x{:}\sigma$. Then by property 2.3.4 (v) there is $x{:}\tau \in B$ such that $\tau \leq \sigma$.
Obviously $B \vdash_{-\omega} x{:}\sigma$.

ii) $B \vdash_{\mathrm{BCD}} \lambda x.M'{:}\sigma$. Then $\sigma \equiv (\rho_1{\rightarrow}\mu_1) \cap \cdots \cap (\rho_n{\rightarrow}\mu_n)$ ($n \geq 1$), and by property 2.3.4 (ii) for every $1 \leq i \leq n$ $B \cup \{x{:}\rho_i\} \vdash_{\mathrm{BCD}} M'{:}\mu_i$.
By induction for every $1 \leq i \leq n$, $B \cup \{x{:}\rho_i\} \vdash_{-\omega} M'{:}\mu_i$. So $B \vdash_{-\omega} \lambda x.M'{:}\sigma$.

iii) $B \vdash_{\mathrm{BCD}} xM_1 \ldots M_n{:}\sigma$. By property 2.3.4 (i) there are $\tau_1, \ldots, \tau_n$ such that

$$B \vdash_{\mathrm{BCD}} x{:}\tau_1{\rightarrow}\cdots{\rightarrow}\tau_n{\rightarrow}\sigma, \text{ and for every } 1 \leq i \leq n\ B \vdash_{\mathrm{BCD}} M_i{:}\tau_i.$$

By property 2.3.4 (v) there is a $x{:}\rho \in B$ such that $\rho \leq \tau_1{\rightarrow}\cdots{\rightarrow}\tau_n{\rightarrow}\sigma$. By property 2.3.4 (vi) this implies

$$\rho = (\tau_1^1{\rightarrow}\cdots{\rightarrow}\tau_n^1{\rightarrow}\sigma^1) \cap \cdots \cap (\tau_1^s{\rightarrow}\cdots{\rightarrow}\tau_n^s{\rightarrow}\sigma^s) \cap \rho',$$

for $\tau_1^j, \ldots, \tau_n^j, \sigma^j$ such that $\tau_i^j \geq \tau_i$ with $1 \leq i \leq n$, $1 \leq j \leq s$, and $\sigma^1 \cap \cdots \cap \sigma^s \leq \sigma$. Then by ($\leq$) and ($\cap$I) for every $1 \leq i \leq n$, we have $B \vdash_{\mathrm{BCD}} M_i{:}\tau_i^1 \cap \cdots \cap \tau_i^s$.
Since each τ_i^j occurs in a statement in the basis, the induction hypothesis is applicable, and, for every $1 \leq i \leq n$, we have $B \vdash_{-\omega} M_i{:}\tau_i^1 \cap \cdots \cap \tau_i^s$. Also

$$(\tau_1^1{\rightarrow}\cdots{\rightarrow}\tau_n^1{\rightarrow}\sigma^1) \cap \cdots \cap (\tau_1^s{\rightarrow}\cdots{\rightarrow}\tau_n^s{\rightarrow}\sigma^s) \leq \\ \tau_1^1 \cap \cdots \cap \tau_1^s \rightarrow \cdots \rightarrow \tau_n^1 \cap \cdots \cap \tau_n^s \rightarrow \sigma^1 \cap \cdots \cap \sigma^s,$$

so $B \vdash_{\mathrm{BCD}} x{:}\tau_1^1 \cap \cdots \cap \tau_1^s \rightarrow \cdots \rightarrow \tau_n^1 \cap \cdots \cap \tau_n^s \rightarrow \sigma^1 \cap \cdots \cap \sigma^s$ and by part (i)

$$B \vdash_{-\omega} x{:}\tau_1^1 \cap \cdots \cap \tau_1^s \rightarrow \cdots \rightarrow \tau_n^1 \cap \cdots \cap \tau_n^s \rightarrow \sigma^1 \cap \cdots \cap \sigma^s.$$

But then by ($\rightarrow$E) $B \vdash_{-\omega} xM_1 \ldots M_n{:}\sigma_1 \cap \cdots \cap \sigma_n$ and by ($\leq$) $B \vdash_{-\omega} xM_1 \ldots M_n{:}\sigma$. $\square$

Theorem 7.2.2 If $B \vdash_{\mathrm{BCD}} M{:}\sigma$, where ω does not occur in B and σ, then there is an M' such that $M \twoheadrightarrow_\beta M'$ and $B \vdash_{-\omega} M'{:}\sigma$.

Proof: If $B \vdash_{\mathrm{BCD}} M{:}\sigma$, where ω does not occur in B and σ, then, by property 2.3.4 (iv), M has a normal form M'. Then also $B \vdash_{\mathrm{BCD}} M'{:}\sigma$. By the previous lemma we have that $B \vdash_{-\omega} M'{:}\sigma$. $\square$

As was remarked in the beginning of this chapter, if we are interested in deriving types without ω occurrences, the type constant ω will only be needed in the BCD-system to type sub-terms N of M that will be erased while reducing M. In fact, if there is a type ρ such that $B \vdash_{-\omega} N{:}\rho$, then, even for this N, we would not need ω. Unfortunately, there are lambda terms M that contain a sub-term N that must be typed with ω in $B \vdash_{\mathrm{BCD}} M{:}\sigma$, even if ω does not occur in B and σ. We can even find strongly normalizable lambda terms that contain such a sub-term (see also the remark made after lemma 7.4.2). So, to prove theorem 7.2.2 we have to go down all the way to the set of lambda terms in normal form, since only these do not contain sub-terms that will be erased.

Theorem 7.2.3 *Conservativity*. If M is a λI-term and $B \vdash_{\mathrm{BCD}} M{:}\sigma$ where ω does not occur in B and σ, then $B \vdash_{-\omega} M{:}\sigma$.

Proof: If $B \vdash_{\mathrm{BCD}} M{:}\sigma$ and B, σ are ω-free, then, by property 2.3.4 (iv), M has a normal form M'. Then, also, $B \vdash_{\mathrm{BCD}} M'{:}\sigma$. By lemma 7.2.1 we have $B \vdash_{-\omega} M'{:}\sigma$. Because M and M' are λI-terms, by corollary 7.4.3 we obtain $B \vdash_{-\omega} M{:}\sigma$. $\square$

7.3 *The type assignment without ω is complete for the λI-calculus*

In this section, completeness of type assignment without ω for the λI-calculus will be proved using the method of [Barendregt *et al.* '83]. The notions of type interpretation as defined in definition 2.3.1.2 lead also for the λI-calculus in a natural way to the following definitions for semantic satisfiability.

Definition 7.3.1 As in definition 2.3.1.3, we define $\models$ by: (where $\mathcal{M}$ is a λI-model, ξ a valuation and υ a type interpretation)

i) $\mathcal{M}, \xi, \upsilon \models M{:}\sigma \iff [\![M]\!]^{\mathcal{M}}_{\xi} \in \upsilon(\sigma)$.

ii) $\mathcal{M}, \xi, \upsilon \models B \iff \mathcal{M}, \xi, \upsilon \models x{:}\sigma$ for every $x{:}\sigma \in B$.

iii) a) $B \models M{:}\sigma \iff \forall\, \mathcal{M}, \xi, \upsilon\ [\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

b) $B \models_{\mathrm{s}} M{:}\sigma \iff$

$\forall\, \mathcal{M}, \xi$, simple type interpretations υ $[\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

c) $B \models_{\mathrm{F}} M{:}\sigma \iff \forall\, \mathcal{M}, \xi$, *F* type interpretations υ $[\ \mathcal{M}, \xi, \upsilon \models B \Rightarrow \mathcal{M}, \xi, \upsilon \models M{:}\sigma\ ]$.

We will only consider the simple type semantics, since $\vdash_{-\omega}$ is not sound for all type interpretations. For example: $\{y{:}(\varphi_1 \cap \varphi_2 \rightarrow \varphi_3) \rightarrow \varphi_4,\ x{:}\varphi_1 \rightarrow \varphi_3\} \vdash_{-\omega} yx{:}\varphi_4$, but this is not semantically valid for all type interpretations.

Theorem 7.3.2 *Soundness*. If $B \vdash_{-\omega} M{:}\sigma$ then $B \models_{\mathrm{s}} M{:}\sigma$.

Proof: By induction on the structure of derivations. $\square$

Definition 7.3.3 i) We define a map $\upsilon_1 : \mathcal{T}_{-\omega} \rightarrow \wp(\mathcal{F}_{-\omega})$ by $\upsilon_1(\sigma) = \{\ d \in \mathcal{F}_{-\omega} \mid \sigma \in d\ \}$.

ii) $\xi_B(x) = \{\ \sigma \in \mathcal{T}_{-\omega} \mid B \vdash_{-\omega} x{:}\sigma\ \}$.

Theorem 7.3.4 i) The map υ_1 is a simple type interpretation.

ii) $B \vdash_{-\omega} M{:}\sigma \iff B_{\xi_B} \vdash_{-\omega} M{:}\sigma$.

iii) $\mathcal{F}_{-\omega}, \xi_B, \upsilon_1 \models_{\mathrm{s}} B$.

Proof: i) Easy.

ii) Because for every x, $\xi_B(x)$ is an I-filter.

iii) $x{:}\sigma \in B \Rightarrow \sigma \in \{\ \tau \mid B_{\xi_B} \vdash_{-\omega} x{:}\tau\ \} \Rightarrow \sigma \in [\![x]\!]_{\xi_B}$.
So $[\![x]\!]_{\xi_B} \in \{\ d \in \mathcal{F}_{-\omega} \mid \sigma \in d\ \}$. $\square$

Theorem 7.3.5 *Completeness*. Let M be a λI-term, and suppose ω does not occur in B and σ. If $B \models_{\mathrm{s}} M{:}\sigma$, then $B \vdash_{-\omega} M{:}\sigma$.

Proof: $B \models_{\mathrm{s}} M{:}\sigma \Rightarrow$ (7.3.1 (iii.b), 7.3.4 (i) & 7.3.4 (iii))
$\mathcal{F}_{-\omega}, \xi_B, \upsilon_1 \models_{\mathrm{s}} M{:}\sigma \Rightarrow$ (7.3.1 (i) & 7.3.4 (i))
$[\![M]\!]_{\xi_B} \in \upsilon_1(\sigma) \Rightarrow$ (7.3.3 (i))
$\sigma \in [\![M]\!]_{\xi_B} \Rightarrow$ (7.1.9)
$B_{\xi_B} \vdash_{-\omega} M{:}\sigma \Rightarrow$ (7.3.4 (ii))
$B \vdash_{-\omega} M{:}\sigma$. $\square$

7.4 *Strong normalization result for the system without ω*

In this section, we show that the set of lambda terms typeable by means of the derivation rules ($\cap$I), ($\cap$E), ($\rightarrow$I) and ($\rightarrow$E) of the BCD-system is exactly the set of strongly normalizable terms. The same result was presented in [Coppo *et al.* '81], [Leivant '83] and [Pottinger '80]. However, the proof in [Leivant '83] was too brief, the proof in [Pottinger '80] gave few details and the proof in [Coppo *et al.* '81] was not complete. In this section, we will present a complete and formal proof. The same result has also been proved in [Krivine '90], using the technique of saturated sets.

In order to prove that each term typeable by the rules ($\cap$I), ($\cap$E), ($\rightarrow$I) and ($\rightarrow$E) is strongly normalizable, we will prove even more: we will show that if $B \vdash_{-\omega} M{:}\sigma$ (i.e. using derivation rule ($\leq$) as well), then M is strongly normalizable. In [Ronchi della Rocca '88], a similar result was proved: $B \vdash_{-\omega} M{:}\sigma \iff M$ is strongly normalizable. In this paper, this result was given in corollary 6.3 and was obtained from the theorem that the procedure PP' (as defined in [Ronchi della Rocca '88], section 6) finds a principal pair for all and nothing but the strongly normalizable terms. In this section, we will present a proof for the same result, different from the one given in [Ronchi della Rocca '88]. The proof that all strongly normalizable terms are typeable in the system without ω and ($\leq$) will be given in theorem 7.4.4.

Notice that an I-filter can be empty. A direct result of the main theorem of this section will be that $[\![\cdots]\!]$ as defined in definition 7.1.8 will map all unsolvable terms onto the empty filter ('unsolvable' in the λI-calculus equals to 'not having a normal form,' as well as that 'normalizable' and 'strongly normalizable' coincide).

Notice, also, that we no longer restrict ourselves to the λI-terms, but prove the statement for the full λK-calculus.

Fact 7.4.1 In the sequel, we will accept the following without proof:

i) If $xM_1 \ldots M_n$ and N are strongly normalizable, then so is $xM_1 \ldots M_n N$.

ii) If Mz is strongly normalizable (where z does not occur free in M), then so is M.
iii) If $M[x := N]$ and N are strongly normalizable, then so is $(\lambda x.M)N$. □

Lemma 7.4.2 If $B \vdash_{-\omega} \mathrm{C}[M[x := N]]{:}\tau$ and $B \vdash_{-\omega} N{:}\rho$, then $B \vdash_{-\omega} \mathrm{C}[(\lambda x.M)N]{:}\tau$, where $\mathrm{C}[\cdots]$ is the notation for a context.

Proof: By induction on the structure of contexts. We will omit the case that the context is an application, since it is trivial.

i) $\mathrm{C}[M[x := N]] \equiv M[x := N]$. We can assume that x does not occur in B.
  a) N occurs n times in $M[x := N]$, each time typed by, say, σ_i.
  $B \vdash_{-\omega} M[x := N]{:}\tau \;\Rightarrow$
  $B \cup \{x{:}\sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{-\omega} M{:}\tau \;\&\; B \vdash_{-\omega} N{:}\sigma_1 \cap \cdots \cap \sigma_n \;\Rightarrow$
  $B \vdash_{-\omega} \lambda x.M{:}\sigma_1 \cap \cdots \cap \sigma_n \to \tau \;\&\; B \vdash_{-\omega} N{:}\sigma_1 \cap \cdots \cap \sigma_n \;\Rightarrow$
  $B \vdash_{-\omega} (\lambda x.M)N{:}\tau$.
  b) N does not occur in $M[x := N]$, so $x \notin \mathrm{FV}(M)$.
  $B \vdash_{-\omega} M{:}\tau \;\&\; B \vdash_{-\omega} N{:}\rho \;\Rightarrow$ $(x \notin \mathrm{FV}(M))$
  $B \cup \{x{:}\rho\} \vdash_{-\omega} M{:}\tau \;\&\; B \vdash_{-\omega} N{:}\rho \;\Rightarrow$
  $B \vdash_{-\omega} \lambda x.M{:}\rho \to \tau \;\&\; B \vdash_{-\omega} N{:}\rho \;\Rightarrow$
  $B \vdash_{-\omega} (\lambda x.M)N{:}\tau$.

ii) $B \vdash_{-\omega} \lambda y.C[M[x := N]]{:}\tau \;\&\; B \vdash_{-\omega} N{:}\rho \;\Rightarrow$ (7.1.2 (ii) & (iii))
$\exists\, \rho_1, \ldots, \rho_n, \mu_1, \ldots, \mu_n\ [\ \tau = (\rho_1 \to \mu_1) \cap \cdots \cap (\rho_n \to \mu_n)\ \&$
$\forall\, 1 \leq i \leq n\ [\ B \cup \{y{:}\rho_i\} \vdash_{-\omega} \mathrm{C}[M[x := N]]{:}\mu_i\ ]\ ]\ \&\ B \vdash_{-\omega} N{:}\rho \;\Rightarrow$ (IH)
$\exists\, \rho_1, \ldots, \rho_n, \mu_1, \ldots, \mu_n\ [\ \tau = (\rho_1 \to \mu_1) \cap \cdots \cap (\rho_n \to \mu_n)\ \&$
$\forall\, 1 \leq i \leq n\ [\ B \cup \{y{:}\rho_i\} \vdash_{-\omega} \mathrm{C}[(\lambda x.M)N]{:}\mu_i\ ]\ ]\ \Rightarrow$
$B \vdash_{-\omega} \lambda y.C[(\lambda x.M)N]{:}\tau$. □

Notice that the condition $B \vdash_{-\omega} N{:}\rho$ in the formulation of the lemma is essential. As a counter example, take the two lambda terms $\lambda yz.(\lambda b.z)(yz)$ and $\lambda yz.z$. Notice that the first strongly reduces to the latter. We know that

$$\{z{:}\sigma, y{:}\tau\} \vdash_{-\omega} z{:}\sigma,$$

but it is impossible to give a derivation for $(\lambda b.z)(yz){:}\sigma$ from the same basis without using ω. This is caused by the fact that we can only type $(\lambda b.z)(yz)$ in the system without ω from a basis in which the predicate for y is an arrow type scheme. We can, for example, derive

$$\{z{:}\sigma, y{:}\sigma \to \tau\} \vdash_{-\omega} (\lambda b.z)(yz){:}\sigma.$$

We can therefore only state that we can derive

$$\vdash_{-\omega} \lambda yz.(\lambda b.z)(yz){:}(\sigma \to \tau) \to \sigma \to \sigma \text{ and } \vdash_{-\omega} \lambda yz.z{:}\tau \to \sigma \to \sigma$$

but that we are not able to give a derivation without ω for the statement

$$\lambda yz.(\lambda b.z)(yz){:}\tau \to \sigma \to \sigma.$$

So the type assignment without ω is not closed for β-equality, but of course this is not imperative. We only want to be able to derive *a* type for each strongly normalizable term, no matter what basis or type is used. Notice that, for the λI-calculus, part 7.4.2 (i.b) is not applicable and the condition $B \vdash_{-\omega} N{:}\rho$ is not needed. So, the following is an immediate result:

Corollary 7.4.3 Let M and M' be λI-terms, such that $M \twoheadrightarrow_\beta M'$, and there are B and σ such that $B \vdash_{-\omega} M'{:}\sigma$. Then $B \vdash_{-\omega} M{:}\sigma$. □

Lemma 7.4.2 is also essentially the proof for the statement that each strongly normalizable term can be typed in the system $\vdash_{-\omega}$.

Theorem 7.4.4 If M is strongly normalizable, then there are B and σ such that $B \vdash_{-\omega} M{:}\sigma$ and in this derivation the rule $(\leq)$ is not used.

Proof: If M is strongly normalizable, then the normal form of M will be reached using the inside-out reduction strategy (see [Barendregt '84].14.2.11). This strategy has the special property that a redex $(\lambda x.M)N$ can only be contracted if N is in normal form. The proof is completed by induction on the inside-out reduction path, using lemma 7.4.2 and theorem 4.2.10. □

In order to prove that each term typeable in $\vdash_{-\omega}$ is strongly normalizable, we will introduce a notion of computability. We will abbreviate 'M is strongly normalizable' by SN(M).

Definition 7.4.5 (cf. [Pottinger '80]) Comp (B, M, ρ) is inductively defined by:
i) Comp $(B, M, \varphi) \Leftrightarrow B \vdash_{-\omega} M{:}\varphi$ & SN(M).
ii) Comp $(B, M, \sigma \to \tau) \Leftrightarrow$ (Comp $(B', N, \sigma) \Rightarrow$ Comp $(B \cup B', MN, \tau)$).
iii) Comp $(B, M, \sigma \cap \tau) \Leftrightarrow$ (Comp (B, M, σ) & Comp (B, M, τ)).

Lemma 7.4.6 Take σ, and τ such that $\sigma \leq \tau$. Then Comp $(B, M, \sigma) \Rightarrow$ Comp (B, M, τ).

Proof: By straightforward induction on the definition of $\leq$. □

Theorem 7.4.7 i) $B \vdash_{-\omega} xM_1 \ldots M_n{:}\rho$ & SN($xM_1 \ldots M_n$) $\Rightarrow$ Comp $(B, xM_1 \ldots M_n, \rho)$.
ii) Comp $(B, M, \rho) \Rightarrow B \vdash_{-\omega} M{:}\rho$ & SN(M).

Proof: Simultaneously by induction on the structure of types. The only interesting case is when $\rho = \sigma \to \tau$; the other cases are dealt with by induction.

i) $B \vdash_{-\omega} xM_1 \ldots M_n{:}\sigma \to \tau$ & SN($xM_1 \ldots M_n$) $\Rightarrow$ ((ii))
(Comp $(B', N, \sigma) \Rightarrow$
  $B \vdash_{-\omega} xM_1 \ldots M_n{:}\sigma \to \tau$ & SN($xM_1 \ldots M_n$) & $B' \vdash_{-\omega} N{:}\sigma$ & SN(N)) $\Rightarrow$ (7.4.1 (i))
(Comp $(B', N, \sigma) \Rightarrow B \cup B' \vdash_{-\omega} xM_1 \ldots M_nN{:}\tau$ & SN($xM_1 \ldots M_nN$)) $\Rightarrow$ (IH)
(Comp $(B', N, \sigma) \Rightarrow$ Comp $(B \cup B', xM_1 \ldots M_nN, \tau)) \Rightarrow$ (7.4.5 (ii))
Comp $(B, xM_1 \ldots M_n, \sigma \to \tau)$.

ii) Comp $(B, M, \sigma \to \tau)$ & $z \notin \mathrm{FV}(M) \Rightarrow$ ((i))
Comp $(B, M, \sigma \to \tau)$ & Comp $(\{z{:}\sigma\}, z, \sigma)$ & $z \notin \mathrm{FV}(M) \Rightarrow$ (7.4.5 (ii))
Comp $(B \cup \{z{:}\sigma\}, Mz, \tau)$ & $z \notin \mathrm{FV}(M) \Rightarrow$ (IH)
$B \cup \{z{:}\sigma\} \vdash_{-\omega} Mz{:}\tau$ & SN(Mz) & $z \notin \mathrm{FV}(M) \Rightarrow$ (7.1.2 (iv) & 7.4.1 (ii))
$B \vdash_{-\omega} M{:}\sigma \to \tau$ & SN(M). □

Lemma 7.4.8 Comp $(B \cup B', \mathrm{C}[M[x := N]], \sigma)$ & Comp $(B', N, \rho) \Rightarrow$ Comp $(B \cup B', \mathrm{C}[(\lambda x.M)N], \sigma)$.

Proof: By induction on the structure of types. We will only consider the case that σ is a type-variable:

Comp $(B \cup B', \mathrm{C}[M[x := N]], \varphi)$ & Comp $(B', N, \rho) \Rightarrow$ (7.4.7 (ii))
$B \cup B' \vdash_{-\omega} \mathrm{C}[M[x := N]]{:}\varphi$ & SN(C[$M[x := N]$]) & $B' \vdash_{-\omega} N{:}\rho$ & SN(N) $\Rightarrow$ (7.4.2 & 7.4.1 (iii))
$B \cup B' \vdash_{-\omega} \mathrm{C}[(\lambda x.M)N]{:}\varphi$ & SN(C[$(\lambda x.M)N$]) $\Rightarrow$ (7.4.5 (i))
Comp $(B \cup B', \mathrm{C}[(\lambda x.M)N], \varphi)$. □

Theorem 7.4.9 If $B = \{x_1{:}\mu_1, \ldots, x_n{:}\mu_n\}$ and for every $1 \leq i \leq n$ Comp (B_i, N_i, μ_i) and $B \vdash_{-\omega} M{:}\sigma$, then Comp $(B_1 \cup \cdots \cup B_n, M[x_1 := N_1, \ldots, x_n := N_n], \sigma)$.

Proof: By induction on the structure of derivations. We will only show the non-trivial parts.

i) ($\to$I). Then $M \equiv \lambda y.M'$, $\sigma = \rho\to\tau$, and $B \cup \{y{:}\rho\} \vdash_{-\omega} M'{:}\tau$.
$B = \{x_1{:}\mu_1, \ldots, x_n{:}\mu_n\}$ &
$\forall\, 1 \leq i \leq n$ [Comp (B_i, N_i, μ_i)] & $B \cup \{y{:}\rho\} \vdash_{-\omega} M'{:}\tau \Rightarrow$ (IH)
(Comp $(B', N, \rho) \Rightarrow$
Comp $(B_1 \cup \cdots \cup B_n \cup B', M'[x_1 := N_1, \ldots, x_n := N_n, y := N], \tau)) \Rightarrow$ (7.4.8)
(Comp $(B', N, \rho) \Rightarrow$
Comp $(B_1 \cup \cdots \cup B_n \cup B', (\lambda y.M'[x_1 := N_1, \ldots, x_n := N_n])N, \tau)) \Rightarrow$ (7.4.5 (ii))
Comp $(B_1 \cup \cdots \cup B_n, (\lambda y.M')[x_1 := N_1, \ldots, x_n := N_n], \rho\to\tau)$.

ii) ($\to$E). Then $M \equiv M_1M_2$, $B \vdash_{-\omega} M_1{:}\rho\to\tau$ and $B \vdash_{-\omega} M_2{:}\rho$.
$B = \{x_1{:}\mu_1, \ldots, x_n{:}\mu_n\}$ & $\forall\, 1 \leq i \leq n$ [Comp (B_i, N_i, μ_i)] &
$B \vdash_{-\omega} M_1{:}\rho\to\tau$ & $B \vdash_{-\omega} M_2{:}\rho \Rightarrow$ (IH)
Comp $(B_1 \cup \cdots \cup B_n, M_1[x_1 := N_1, \ldots, x_n := N_n], \rho\to\tau)$ &
Comp $(B_1 \cup \cdots \cup B_n, M_2[x_1 := N_1, \ldots, x_n := N_n], \rho) \Rightarrow$ (7.4.5 (ii))
Comp $(B_1 \cup \cdots \cup B_n, (M_1M_2)[x_1 := N_1, \ldots, x_n := N_n], \tau)$. □

Theorem 7.4.10 If $B \vdash_{-\omega} M{:}\sigma$, then SN($M$).

Proof: $B \vdash_{-\omega} M{:}\sigma \Rightarrow$ (7.4.9) Comp $(B, M, \sigma) \Rightarrow$ (7.4.7 (ii)) SN(M). □

We can now prove the main theorem of this section.

Theorem 7.4.11 $\{M \mid M$ is typeable by means of the derivation rules ($\cap$I), ($\cap$E), ($\to$I) and ($\to$E)$\}$ = $\{M \mid M$ is strongly normalizable$\}$.

Proof: $\subseteq$) If M is typeable by means of the derivation rules ($\cap$I), ($\cap$E), ($\to$I) and ($\to$E), then certainly $B \vdash_{-\omega} M{:}\sigma$. Then, by theorem 7.4.10, M is strongly normalizable.

$\supseteq$) If M is strongly normalizable, then, by theorem 7.4.4, there are B, σ such that $B \vdash_{-\omega} M{:}\sigma$; in this derivation the derivation rule ($\leq$) is not used. □

Chapter 8 *The Rank 2 Intersection Type Assignment System*

The notion of type assignment presented in this chapter will be based on the strict type assignment system (restricted to intersection types of Rank 2, as suggested by D. Leivant in [Leivant '83]), and the CD-system. We will define intersection types of Rank 2, and we will present two operations on pairs of basis and type, namely Rank 2 substitution and duplication. We will show that the presented notion of type assignment has the principal type property: for every typeable term M, there are a basis P and type π such that the pair $<P, \pi>$ is the principal pair for M, and $P \vdash_{\mathrm{R}} M{:}\pi$. We will show that for every basis B and type σ such that $B \vdash_{\mathrm{R}} M{:}\sigma$, there is a type-chain of operations $\mathbf{C}$ (consisting of nothing but substitutions and duplications) such that $\mathbf{C}(<P, \pi>) = <B, \sigma>$. We will obtain this result by defining a unification algorithm for intersection types of Rank 2; using this algorithm, for every term M we define the principal pair for M.

The system presented in this chapter is perhaps more general than needed to obtain Rank 2 intersection type assignment for the lambda calculus. For example, the definition of duplication in definition 8.3.2.1 could be formulated in a slightly simplified way. However, in chapter twelve we will formulate a notion of type assignment based on the one presented in this chapter, for which this more general approach is appropriate.

To avoid confusion, we would like to point out that there also exists a notion of type assignment that is called the Rank 2 *polymorphic* type assignment system, defined in [Kfoury & Tiuryn '89]. This system is an extension of Milner's system, by allowing the $\forall$-type constructor to occur on the left hand side of an arrow type as well, instead of only at top level. (It is also a restriction of the polymorphic type discipline [Girard '86], in which types are restricted to polymorphic types of Rank 2.) The definition of types allowed in this system is similar to that of Rank 2 intersection types, and $\forall$ can occur only at the left of the 'top'-arrow. As in the system presented here, type assignment in this system is decidable.

8.1 *Coppo-Dezani type assignment versus ML type assignment*

In [Leivant '83] was remarked that (part of) the Milner's type assignment system can be seen as a restriction of the Coppo-Dezani type discipline, by limitation of the set of types to intersection types of Rank 2. This observation can be understood through the following argument: suppose the (LET)-rule is used to derive a type for a term. Then there is a Curry type σ, such that

$$\dfrac{B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} M{:}\tau \qquad \dfrac{B \vdash_{\mathrm{ML}} N{:}\sigma}{B \vdash_{\mathrm{ML}} N{:}\overline{\sigma}}\ (\mathrm{GEN})}{B \vdash_{\mathrm{ML}} (\mathsf{let}\ x = N\ \mathsf{in}\ M){:}\tau}\ (\mathrm{LET})$$

is part of the derivation, and $\overline{\sigma} = \forall\varphi_1 \cdots \forall\varphi_m.\sigma$. (When assigning a type to the term (let $x = N$

in M), first the 'operand' N is typed by the Curry-type σ.) In deriving $B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} M{:}\tau$, $\overline{\sigma}$ is instantiated using the derivation rule (INST) (otherwise the rules (ABS) and (COMB) cannot be used) into the Curry-types $\sigma_1, \ldots, \sigma_n$.

$$\dfrac{\dfrac{\dfrac{B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} x{:}\overline{\sigma} \quad \overline{\sigma} > \sigma_1}{B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} x{:}\sigma_1} \;\cdots\; \dfrac{B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} x{:}\overline{\sigma} \quad \overline{\sigma} > \sigma_n}{B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} x{:}\sigma_n}\,(\mathrm{INST})}{\vdots \atop B \cup \{x{:}\overline{\sigma}\} \vdash_{\mathrm{ML}} M{:}\tau} \quad \dfrac{B \vdash_{\mathrm{ML}} N{:}\sigma}{B \vdash_{\mathrm{ML}} N{:}\overline{\sigma}}\,(\mathrm{GEN})}{B \vdash_{\mathrm{ML}} (\mathsf{let}\ x = N\ \mathsf{in}\ M){:}\tau}\,(\mathrm{LET})$$

By definition of the relation '$>$' on ML-type schemes, there are Curry-substitutions $S_1, \ldots, S_n$ such that for every $1 \leq i \leq n$, $S_i\,(\sigma) = \sigma_i$.

Under those conditions, however, the term $(\lambda x.M)N$ can be typed in the Coppo-Dezani type assignment system, because in this system the term $\lambda x.M$ can be typed by $\sigma_1 \cap \cdots \cap \sigma_n \to \tau$; it is easy to verify that N is typeable by every σ_i, so by derivation rules $(\cap\mathrm{I})$ and $(\to\mathrm{E})$ the term $(\lambda x.M)N$ is typeable by τ.

$$\dfrac{\dfrac{\dfrac{[x{:}\sigma_1 \cap \cdots \cap \sigma_n]}{x{:}\sigma_1} \;\cdots\; \dfrac{[x{:}\sigma_1 \cap \cdots \cap \sigma_n]}{x{:}\sigma_n}\,(\cap\mathrm{E})}{\vdots \atop M{:}\tau}}{\lambda x.M{:}\sigma_1 \cap \cdots \cap \sigma_n \to \tau}\,(\to\mathrm{I}) \quad \dfrac{N{:}\sigma_1 \quad \cdots \quad N{:}\sigma_n}{N{:}\sigma_1 \cap \cdots \cap \sigma_n}\,(\cap\mathrm{I})$$
$$\overline{\qquad (\lambda x.M)N{:}\tau \qquad}\,(\to\mathrm{E})$$

So, when using intersection types, the let-construct is not needed. Notice that this derivation can also be given in the BCD-system and in the strict system.

Notice, moreover, that the construction sketched above only uses intersections of Curry-types, and that such an intersection can only occur on the left hand side of an $\to$-type constructor (Leivant speaks of 'intersection types of Rank 2'). This gives rise to the idea that the ML type assignment system (and, in particular, the unification algorithm for that system) and the limitation of the intersection type assignment system to Rank 2 are, as far as decidability is concerned, equivalent.

The Rank 2 system and Milner's system are not really equivalent, because there are terms that are typeable in the former and not typeable in the latter (see example 8.4.2.2). But not only is the class of typeable terms significantly extended when intersection types of Rank 2 are used, also more accurate types can be deduced for terms. For example, the term $S\,K\,S\,I$ (where S, K and I are the well know lambda terms) has a more general principal type in the Rank 2 Intersection Type Assignment System than in the ML system (see also example 8.4.2.2).

When the lambda calculus is extended with the fixed-point combinator Y, and the notion of reduction $\to_\beta$ is extended by: $Y M \to_\beta M(Y M)$ for all terms M, and Y is assumed to have the principal type $(\varphi\to\varphi)\to\varphi$ (so all occurrences of Y have a type that is a substitution instance of this type), then all Milner-typeable terms in Exp can be translated into terms in this extended calculus with Rank 2 intersection types. This construction cannot be given for the

Mycroft-typeable terms, since in this approach it is not possible to give Y *one* type from which all other types can be generated; if Y is used in Mycroft's system, then for every occurrence of an Y there are $\sigma_1, \ldots, \sigma_n$ and σ, such that Y is typed with $(\sigma_1 \cap \cdots \cap \sigma_n \to \sigma)\to\sigma$, and all σ_i are substitution instances of σ.

8.2 *Rank 2 intersection type assignment*

In this section, we will give a formal presentation of *a* Rank 2 intersection type assignment system. Since Leivant only presented an observation, it is not clear whether the system presented here is the one he had in mind. Moreover, when we would formally define intersection types of Rank 2 using the general definition of the 'rank of a type', then we would obtain a notion of type assignment that is more general than the one we will present in this chapter: then, for example, $\omega\to\sigma$ would also be a Rank 2 type. The system we will present here is designed in such a way, that it has enough expressive power to get a straightforward proof that every term in Exp is typeable in it, after let-expressions are replaced by redexes, and Y is added to the calculus.

Allowing of more general rank 2 types would unnecessarily obscure this chapter; also, it would not be possible to extend the notion of type assignment defined here to the one to be defined in chapter twelve.

Intersection types of Rank 2 are a true subset of the set of BCD-types and the set of strict types and only a minor extension of the set of Curry-types. They are defined by:

Definition 8.2.1 i) $\mathcal{T}_1$ is defined by: If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_{\mathrm{C}}$ $(n \geq 1)$, then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_1$.

ii) $\mathcal{T}_2$ is inductively defined by:

a) If $\sigma \in \mathcal{T}_{\mathrm{C}}$, then $\sigma \in \mathcal{T}_2$.

b) If $\sigma \in \mathcal{T}_1$, $\tau \in \mathcal{T}_2$, then $\sigma\to\tau \in \mathcal{T}_2$.

iii) $\mathcal{T}_{\mathrm{R}}$, the set of intersection types of Rank 2 is defined by: if $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_2$ $(n \geq 1)$ then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_{\mathrm{R}}$.

In the next definition we will define a partial order relation $\leq_{\mathrm{R}}$ on $\mathcal{T}_{\mathrm{R}}$ that is, like $\leq_{\mathrm{S}}$, induced by intersections. This relation does not really play a role in this chapter, but it will be important in chapter twelve. We will also define an equivalence relation $\sim_{\mathrm{R}}$ on types. Types σ and τ are equivalent under this relation, if σ can be obtained from τ by permuting subtypes that are part of an intersection subtype.

Definition 8.2.2 i) On $\mathcal{T}_{\mathrm{R}}$, the relation $\leq_{\mathrm{R}}$ is defined by:

a) $\sigma \leq_{\mathrm{R}} \sigma$.

b) $\sigma\cap\tau \leq_{\mathrm{R}} \sigma$ & $\sigma\cap\tau \leq_{\mathrm{R}} \tau$.

c) $\sigma \leq_{\mathrm{R}} \tau \leq_{\mathrm{R}} \rho \Rightarrow \sigma \leq_{\mathrm{R}} \rho$.

d) $\sigma \leq_{\mathrm{R}} \rho$ & $\sigma \leq_{\mathrm{R}} \tau \Rightarrow \sigma \leq_{\mathrm{R}} \rho\cap\tau$.

ii) On $\mathcal{T}_{\mathrm{R}}$, the relation $\sim_{\mathrm{R}}$ is defined by:

a) For $\sigma, \tau \in \mathcal{T}_{\mathrm{R}}$: $\sigma \leq_{\mathrm{R}} \tau \leq_{\mathrm{R}} \sigma \Rightarrow \sigma \sim_{\mathrm{R}} \tau$.

b) For $\sigma\to\tau$, $\rho\to\mu \in \mathcal{T}_{\mathrm{R}}$: $\sigma \sim_{\mathrm{R}} \rho$ & $\tau \sim_{\mathrm{R}} \mu \;\Rightarrow\; \sigma\to\tau \sim_{\mathrm{R}} \rho\to\mu$.

In this chapter we consider types modulo $\sim_{\mathrm{R}}$. Therefore, $\rho\cap(\sigma\cap\tau) = (\rho\cap\sigma)\cap\tau$, and $\sigma\to\tau = \sigma\cap\sigma\to\tau$. Unless stated otherwise, if $\sigma_1\cap\cdots\cap\sigma_n$ is used to denote a type, then all $\sigma_1, \ldots, \sigma_n$ are assumed to be in $\mathcal{T}_2$.

Definition 8.2.3 i) A *Rank 2 statement* is an expression of the form $M{:}\sigma$, where $M \in \Lambda$ and $\sigma \in \mathcal{T}_{\mathrm{R}}$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

ii) A *Rank 2 basis* is a set of Rank 2 statements with term-variables, not necessarily distinct, as subjects and types in $\mathcal{T}_{\mathrm{C}}$ as predicates.

The definition of bases is not the standard one, since we do not allow for all types in $\mathcal{T}_{\mathrm{R}}$ as predicates for Rank 2 statements that have a term-variable as subject. (See also the remark after theorem 8.4.2.5.) Moreover, we allow of more than one statement for term-variables.

The Rank 2 Intersection Type Assignment System is defined as follows:

Definition 8.2.4 i) *Rank 2 type assignment* and *Rank 2 derivations* are defined by the following natural deduction system:

$$(\to\mathrm{I}){:}\ \frac{\begin{array}{c}[x{:}\sigma_1]\cdots[x{:}\sigma_n]\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma_1\cap\cdots\cap\sigma_n\to\tau}\ (a) \qquad (\to\mathrm{E}){:}\ \frac{M{:}\sigma_1\cap\cdots\cap\sigma_n\to\tau \quad N{:}\sigma_1 \ \ldots\ N{:}\sigma_n}{MN{:}\tau}$$

(a) If $x{:}\sigma_1, \ldots, x{:}\sigma_n$ are all and nothing but the statements about x on which $M{:}\tau$ depends. If x does not occur free in M, so no statement with subject x is used to obtain $M{:}\tau$, then $n = 1$.

If B is a basis such that $M{:}\sigma$ is derivable from B using a Rank 2 derivation, and B contains nothing but the statements about term-variables needed to obtain $M{:}\sigma$, we write $B \vdash_2 M{:}\sigma$.

ii) We define $\vdash_{\mathrm{R}}$ by: $B \vdash_{\mathrm{R}} M{:}\sigma$ if and only if: there are $\sigma_1, \ldots, \sigma_n, B_1, \ldots, B_n$ $(n \geq 1)$ such that $\sigma = \sigma_1\cap\cdots\cap\sigma_n$, $B = B_1\cup\cdots\cup B_n$ and for every $1 \leq i \leq n$ $B_i \vdash_2 M{:}\sigma_i$.

Notice that in both the derivation rules, for $1 \leq i \leq n$, $\sigma_i \in \mathcal{T}_{\mathrm{C}}$.

For the notions of type assignment defined here, the following properties hold:

Lemma 8.2.5 i) $B \vdash_2 \lambda x.M{:}\sigma \iff \exists\, \tau_1, \ldots, \tau_n \in \mathcal{T}_{\mathrm{C}}, \tau \in \mathcal{T}_2\ [\ \sigma = \tau_1\cap\cdots\cap\tau_n\to\tau$ &
$(\ x \in \mathrm{FV}(M) \ \Rightarrow\ B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\} \vdash_2 M{:}\tau\ )$ &
$(\ x \notin \mathrm{FV}(M) \ \Rightarrow\ B \vdash_2 M{:}\tau$ & $n = 1\ )\ ]$.

ii) $B \vdash_2 MN{:}\sigma \iff \exists\, \tau \in \mathcal{T}_1, B_1, B_2\ [\ B_1 \vdash_2 M{:}\tau\to\sigma$ & $B_2 \vdash_{\mathrm{R}} N{:}\tau$ & $B = B_1\cup B_2\ ]$.

Proof: Immediately from definition 8.2.4. □

Notice that the notion '$B \vdash_2 M{:}\sigma$' is a more restricted relation between basis, term and type than in the previous chapters. In these chapters, it was sufficient for a basis to just *contain* all types needed in the derivation, while in the Rank 2 intersection system we must require that a basis contain *nothing more* than these types. (So for this notion of type assignment, the definition of 'B is used for $M{:}\sigma$' would be superfluous.) If we drop this restriction, then we should also be more permissive in the constraints on derivation rule $(\to\mathrm{I})$, and, for example, allow of derivations for

$$\{x{:}\varphi_1, x{:}\varphi_2\} \vdash_2 x{:}\varphi_1, \text{ and } \vdash_2 \lambda x.x{:}\varphi_1\cap\varphi_2\to\varphi_1.$$

Since it is not possible to derive this last type for $\lambda x.x$ in Milner's system, we have chosen not to allow it for the Rank 2 Intersection Type Assignment System. Notice that being more permissive forces to define an operation of lifting, something not needed in the approach taken here. In fact this restriction is only needed for the Rank 2 Intersection Type Assignment System for the lambda calculus. We will drop it in chapter twelve, in which we will extend the system defined in this chapter to one for Term Rewriting Systems.

8.3 *Operations on pairs*

In this section we will define two operations on pairs of basis and type, namely substitution and duplication. In theorem 8.3.3.3 (i) we will prove that these operations are sound, and in theorem 8.4.2.5 we will prove that these are complete (i.e. are sufficient to generate all pairs for a term from its principal pair).

8.3.1 *Rank 2 substitution*

In this chapter, substitution will be defined as a Curry-substitution, the operation that replaces type-variables by elements of $\mathcal{T}_{\mathrm{C}}$. Although perhaps this is a more restricted kind of substitution than could be expected, it is a sound operation and will prove to be sufficient.

Definition 8.3.1.1 i) The *Rank 2 substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_{\mathrm{R}} \to \mathcal{T}_{\mathrm{R}}$, where φ is a type-variable and $\alpha \in \mathcal{T}_{\mathrm{C}}$, is defined by:

a) $(\varphi \mapsto \alpha)\,(\varphi) = \alpha$.

b) $(\varphi \mapsto \alpha)\,(\varphi') = \varphi'$, if $\varphi \neq \varphi'$.

c) $(\varphi \mapsto \alpha)\,(\sigma\to\tau) = (\varphi \mapsto \alpha)\,(\sigma) \to (\varphi \mapsto \alpha)\,(\tau)$.

d) $(\varphi \mapsto \alpha)\,(\sigma_1\cap\cdots\cap\sigma_n) = (\varphi \mapsto \alpha)\,(\sigma_1)\cap\cdots\cap(\varphi \mapsto \alpha)\,(\sigma_n)$.

ii) If S_1 and S_2 are Rank 2 substitutions, then so is $S_1{\circ}S_2$, where $S_1{\circ}S_2\,(\sigma) = S_1\,(S_2\,(\sigma))$.

iii) $S\,(B) = \{x{:}S\,(\sigma) \mid x{:}\sigma \in B\}$.

iv) $S\,(\langle B, \sigma\rangle) = \langle S\,(B), S\,(\sigma)\rangle$.

Substitution is normally defined as the operation that replaces type-variables by types, without restriction. This definition would not be correct for the Rank 2 system, since, for example, the replacement of the type-variable φ in $\varphi\to\varphi$ by the type $(\sigma\to\tau)\cap\sigma\to\tau$ would give a type that is not an element of $\mathcal{T}_{\mathrm{R}}$. It is easy to verify that the above defined operation is well defined.

For substitutions, the following properties hold:

Lemma 8.3.1.2 Let S be a Rank 2 substitution. Then:

i) If $\sigma \leq_{\mathrm{R}} \tau$, then $S(\sigma) \leq_{\mathrm{R}} S(\tau)$.

ii) If $\sigma \in \mathcal{T}_{\mathrm{C}}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{\mathrm{R}})$, then $S(\sigma) \in \mathcal{T}_{\mathrm{C}}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{\mathrm{R}})$.

Proof: Easy. □

Notice that, because of part (ii) of this lemma, substitution is well defined on bases.

The following theorem will show that substitution is a sound operation.

Theorem 8.3.1.3 *Soundness of substitution.* If $B \vdash_{\mathrm{R}} M{:}\sigma$, then for every Rank 2 substitution S: $S(B) \vdash_{\mathrm{R}} M{:}S(\sigma)$.

Proof: i) $\sigma \in \mathcal{T}_2$, so $B \vdash_2 M{:}\sigma$. By induction on the structure of M.

a) $M \equiv x$, so $B = \{x{:}\sigma\}$, and $\sigma \in \mathcal{T}_{\mathrm{C}}$. Then $S(\sigma) \in \mathcal{T}_{\mathrm{C}}$, and $S(B) \vdash_2 x{:}S(\sigma)$.

b) $M \equiv \lambda x.M'$. Then by lemma 8.2.5 (i) there are $\tau_1, \ldots, \tau_n \in \mathcal{T}_{\mathrm{C}}$, $\tau \in \mathcal{T}_2$ such that $\sigma = \tau_1 \cap \cdots \cap \tau_n \rightarrow \tau$ and either

1) $x \in \mathrm{FV}(M)$. Then $B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\} \vdash_2 M'{:}\tau$. By induction

$$S(B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\}) \vdash_2 M'{:}S(\tau),$$

so by definition 8.3.1.1 $S(B) \cup \{x{:}S(\tau_1), \ldots, x{:}S(\tau_n)\} \vdash_2 M'{:}S(\tau)$, so by lemma 8.2.5 (i) and definition 8.3.1.1 $S(B) \vdash_2 \lambda x.M{:}S(\sigma)$.

2) $x \notin \mathrm{FV}(M)$. Then $B \vdash_2 M'{:}\tau$, and $n = 1$. By induction $S(B) \vdash_2 M'{:}S(\tau)$, and by lemma 8.2.5 (i) $S(B) \vdash_2 \lambda x.M'{:}S(\tau_1) \rightarrow S(\tau)$, so by definition 8.3.1.1 $S(B) \vdash_2 \lambda x.M{:}S(\sigma)$.

c) $M \equiv M_1 M_2$. By lemma 8.2.5 (ii) there are B_1, B_2 and $\tau \in \mathcal{T}_1$, such that:

$$B = B_1 \cup B_2,\ B_1 \vdash_2 M_1{:}\tau \rightarrow \sigma, \text{ and } B_2 \vdash_{\mathrm{R}} M_2{:}\tau.$$

Then $S(B) = S(B_1) \cup S(B_2)$, and by lemma 8.3.1.2 (ii) $S(\tau) \in \mathcal{T}_1$, and since $S(\tau \rightarrow \sigma) = S(\tau) \rightarrow S(\sigma)$, also by induction:

$$S(B_1) \vdash_2 M_1{:}S(\tau) \rightarrow S(\sigma) \text{ and } S(B_2) \vdash_{\mathrm{R}} M_2{:}S(\tau).$$

Then by lemma 8.2.5 (ii) $S(B) \vdash_2 M_1 M_2{:}S(\sigma)$.

ii) $\sigma \in \mathcal{T}_{\mathrm{R}}$. Then by definition 8.2.4 (ii) there are $\sigma_1, \ldots, \sigma_n, B_1, \ldots, B_n$ ($n \geq 1$) such that $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$, and $B = B_1 \cup \cdots \cup B_n$ and for every $1 \leq i \leq n$ $B_i \vdash_2 M{:}\sigma_i$.
Then $S(B) = S(B_1) \cup \cdots \cup S(B_n)$, and by induction for $1 \leq i \leq n$ $S(B_i) \vdash_2 M{:}S(\sigma_i)$.
Again by definition 8.2.4 (ii) we have $S(B) \vdash_2 M{:}S(\sigma)$. □

8.3.2 *Duplication*

We now come to the definition of duplication. It can be seen as a very simple version of the various operations of expansion as defined before; duplication is a total expansion, that is not 'computed': all type-variables occurring in basis and type are copied.

Definition 8.3.2.1 Let B be a basis, $\sigma \in \mathcal{T}_{\mathrm{R}}$, and $n \geq 1$. The triple $<n, B, \sigma>$ determines a *duplication* $D_{<n,B,\sigma>} : \mathcal{T}_{\mathrm{R}} \rightarrow \mathcal{T}_{\mathrm{R}}$, which is constructed as follows:

i) a) Suppose $V = \{\varphi_1, \ldots, \varphi_m\}$ is the set of all type-variables occurring in $<B, \sigma>$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i ($1 \leq i \leq n$, $1 \leq j \leq m$) does not occur in V. Let S_i be the substitution that replaces every φ_j by φ_j^i.

b) $D_{<n,B,\sigma>}(\tau) = S_1(\tau) \cap \cdots \cap S_n(\tau)$.

ii) $D_{<n,B,\sigma>}(B') = S_1(B') \cup \cdots \cup S_n(B')$.

iii) $D_{<n,B,\sigma>}(<B', \sigma'>) = <D_{<n,B,\sigma>}(B'), D_{<n,B,\sigma>}(\sigma')>$.

Instead of $D_{<n,B,\sigma>}$, we will simply write $<n, B, \sigma>$.

Notice that if τ does not contain type-variables that occur in V, then $<n, B, \sigma>(\tau) = \tau \cap \cdots \cap \tau$, which is by definition of $\sim_{\mathrm{R}}$ the same as τ.

For an operation of duplication, the following properties hold:

Lemma 8.3.2.2 Let $D = <n, B, \sigma>$.

i) If $\rho \leq_{\mathrm{R}} \tau$, then $D(\rho) \leq_{\mathrm{R}} D(\tau)$.

ii) $D(<B', \sigma'>) = <B_1 \cup \cdots \cup B_n, \sigma_1 \cap \cdots \cap \sigma_n>$ with for every $1 \leq i \leq n$ there is a substitution S_i such that $S_i(<B', \sigma'>) = <B_i, \sigma_i>$.

iii) $D(<B, \sigma>) = <B_1 \cup \cdots \cup B_n, \sigma_1 \cap \cdots \cap \sigma_n>$, with for every $1 \leq i \leq n$, $<B_i, \sigma_i>$ is a trivial variant of $<B, \sigma>$, and the $<B_i, \sigma_i>$ are disjoint in pairs.

iv) If $\tau \in \mathcal{T}_{\mathrm{C}}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{\mathrm{R}})$, then $D(\tau) \in \mathcal{T}_1(\mathcal{T}_1, \mathcal{T}_{\mathrm{R}}, \mathcal{T}_{\mathrm{R}})$.

Proof: Immediately by definition 8.3.2.1. □

We will now prove that the operation of duplication is sound.

Theorem 8.3.2.3 *Soundness of duplication.* If $B \vdash_{\mathrm{R}} M{:}\sigma$, then for every duplication D: If $D(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_{\mathrm{R}} M{:}\sigma'$.

Proof: By lemma 8.3.2.2 (ii) there are $B_1, \ldots, B_n, \sigma_1, \ldots, \sigma_n$ such that: $<B', \sigma'> = <B_1 \cup \cdots \cup B_n, \sigma_1 \cap \cdots \cap \sigma_n>$, and for every $1 \leq i \leq n$ there is a substitution S_i such that $S_i(<B, \sigma>) = <B_i, \sigma_i>$. The proof is completed by theorem 8.3.1.3. □

8.3.3 *Type-chains*

As before we will define chains as sequences of operations. To avoid confusion, we will call the chains needed in this chapter type-chains.

Definition 8.3.3.1 A *type-chain* is a chain $<O_1, \ldots, O_n>$, where each O_i is an operation of Rank 2 substitution or duplication.

Type-chains have the following effect on types.

Lemma 8.3.3.2 Let C be a type-chain.

i) If $\sigma \leq_{\mathrm{R}} \tau$, then $C(\sigma) \leq_{\mathrm{R}} C(\tau)$.

ii) If $\tau \in \mathcal{T}_{\mathrm{C}}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{\mathrm{R}})$, then $C(\tau) \in \mathcal{T}_1(\mathcal{T}_1, \mathcal{T}_{\mathrm{R}}, \mathcal{T}_{\mathrm{R}})$.

iii) If $\tau \in \mathcal{T}_C$, and $C(\tau) \in \mathcal{T}_C$, then there is a substitution S such that $C(\tau) = S(\tau)$. Without loss of generality, we can even assume that there is also a type-chain C' such that $C = \langle S \rangle * C'$.

Proof: By lemmas 8.3.1.2 and 8.3.2.2. □

Type-chains have the following effect on pairs.

Theorem 8.3.3.3 Let C be a type-chain.

i) *Soundness of type-chains.* If $B \vdash_R M{:}\sigma$ and $C(\langle B, \sigma \rangle) = \langle B', \sigma' \rangle$, then $B' \vdash_R M{:}\sigma'$.

ii) If $C(\langle B \backslash x \cup \{x{:}\sigma_1, \ldots, x{:}\sigma_n\}, \sigma \rangle) = \langle B' \backslash x \cup \{x{:}\tau_1, \ldots, x{:}\tau_m\}, \tau \rangle$, and $\tau \in \mathcal{T}_2$, then $C(\langle B \backslash x, \sigma_1 \cap \cdots \cap \sigma_n \rightarrow \sigma \rangle) = \langle B' \backslash x, \tau_1 \cap \cdots \cap \tau_m \rightarrow \tau \rangle$.

Proof: i) By theorems 8.3.1.3 and 8.3.2.3.

ii) By definitions 8.3.1.1 and 8.3.2.1. □

8.4 *Principal type property*

In this section, we will show that the Rank 2 Intersection Type Assignment System has the principal type property: for every typeable term M there exists a pair $PP_R(M)$, the principal pair of M, and for every pair $\langle B, \sigma \rangle$ such that $B \vdash_R M{:}\sigma$, there exists a type-chain C such that

$$C(PP_R(M)) = \langle B, \sigma \rangle.$$

As for the Curry type assignment system, principal types for terms will be defined using a notion of unification.

8.4.1 *Unification of intersection types of Rank 2*

Unification is a procedure normally used to find a common instance for demanded and provided types for application, i.e.: if M_1 has type $\sigma \rightarrow \tau$, and M_2 has type α, then unification will look for a common instance of both the types σ and α, such that $M_1 M_2$ can be typed properly.

The unification algorithm unify_1 we will present in the next definition deals with just that problem. This means that it is not a full unification algorithm for intersection types of Rank 2, but only an algorithm that finds the most general unifying type-chain for demanded and provided type. It is defined using Robinson's unification algorithm unify_R.

The algorithm is used in the definition of principal pairs for M; in finding the principal pair for the term $M_1 M_2$ by construction, the demanded type σ in $\sigma \rightarrow \tau$ is in $\mathcal{T}_1$ and the provided type α is in $\mathcal{T}_2$. The unification algorithm looks for types that can be assigned to the terms M_1 and M_2, such that the derivation rule $(\rightarrow E)$ can be applied. Therefore, the result of the unification of σ and α – a type-chain C – should always be such that $C(\alpha) \in \mathcal{T}_1$. However, by lemma 8.3.3.2 (ii), if $\alpha \notin \mathcal{T}_C$, then $C(\alpha) \notin \mathcal{T}_1$. To overcome this difficulty, we have inserted an extra algorithm $\mathit{to}\mathcal{T}_C$ that, when called with the type α, returns a type-chain of operations that removes, if possible, intersections in α, and unify_1 is called with the types σ and α', the latter being α in which the intersections are removed, so $\alpha' = \mathit{to}\mathcal{T}_C(\alpha)(\alpha)$.

It is possible that $\sigma \notin \mathcal{T}_C$, so it can be that α' must be duplicated. Since such an operation affects the basis as well, the third argument of unify_1 is a basis.

Definition 8.4.1.1 *Rank 2 unification.* Let $\mathcal{B}$ be the set of all bases, $\mathcal{C}$ the set of all type-chains, and let Id_S be the substitution that replaces all type-variables by themselves.

i) $\mathit{unify}_1 : \mathcal{T}_1 \times \mathcal{T}_C \times \mathcal{B} \rightarrow \mathcal{C}$

$$\begin{array}{ll} \mathit{unify}_1(\sigma, \alpha, B) = \mathit{unify}_R(\sigma, \alpha), & \text{if } \sigma \in \mathcal{T}_C \\ \mathit{unify}_1(\sigma_1 \cap \cdots \cap \sigma_n, \alpha, B) = \langle D, S_1, \ldots, S_n \rangle, & \text{otherwise} \end{array}$$
$$\begin{array}{lll} \text{where } & D & = \langle n, B, \alpha \rangle, \\ & \alpha_1 \cap \cdots \cap \alpha_n & = D(\alpha), \\ & S_i & = \mathit{unify}_R(\langle S_1, \ldots, S_{i-1} \rangle(\sigma_i), \alpha_i) \text{ for every } 1 \leq i \leq n. \end{array}$$

ii) $\mathit{to}\mathcal{T}_C : \mathcal{T}_2 \rightarrow \mathcal{C}$

$$\begin{array}{ll} \mathit{to}\mathcal{T}_C(\sigma) = Id_S, & \text{if } \sigma \in \mathcal{T}_C \\ \mathit{to}\mathcal{T}_C(\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \sigma) = \langle S_1, \ldots, S_{n-1} \rangle * C, & \text{otherwise} \end{array}$$
$$\begin{array}{ll} \text{where } S_i & = \mathit{unify}_R(\langle S_1, \ldots, S_{i-1} \rangle(\sigma_i), \langle S_1, \ldots, S_{i-1} \rangle(\sigma_{i+1})) \\ & \quad \text{for every } 1 \leq i \leq n\text{-}1, \\ C & = \mathit{to}\mathcal{T}_C(\langle S_1, \ldots, S_{n-1} \rangle(\sigma)). \end{array}$$

Notice that unify_1 and $\mathit{to}\mathcal{T}_C$ may fail because unify_R may fail, and that $\langle n, B, \alpha \rangle$ never fails. Because of this relation between unify_1 and $\mathit{to}\mathcal{T}_C$, and unify_R, the procedures defined here are terminating and type assignment in the system defined in this chapter is decidable.

Lemma 8.4.1.2 i) For every $\sigma \in \mathcal{T}_2$, type-chain C: if $C(\sigma) = \tau \in \mathcal{T}_1$, then there is a type-chain C' such that: $\mathit{to}\mathcal{T}_C(\sigma) * C'(\sigma) = \tau$. (Without loss of generality, we can assume that $C = \mathit{to}\mathcal{T}_C(\sigma) * C'$.)

ii) For every $\sigma \in \mathcal{T}_1$, $\alpha \in \mathcal{T}_C$ that are disjoint: if there is a type-chain C such that $C(\sigma) = C(\alpha)$, then for every basis B that shares no type-variables with σ there is a type-chain C' such that

$$C(\sigma) = \mathit{unify}_1(\sigma, \alpha, B) * C'(\sigma) = \mathit{unify}_1(\sigma, \alpha, B) * C'(\alpha) = C(\alpha).$$

(Without loss of generality, we can assume that $C = \mathit{unify}_1(\sigma, \alpha, B) * C'$.)

Proof: By induction on definition 8.4.1.1, using property 1.6. □

8.4.2 *Principal pairs for terms*

The definition of principal pairs for lambda terms in the Rank 2 system will then look like:

Definition 8.4.2.1 We define, for every term M, the Rank 2 principal pair by defining the notion $PP_R(M) = \langle P, \pi \rangle$ inductively by:

i) For all x, φ: $PP_R(x) = \langle \{x{:}\varphi\}, \varphi \rangle$.

ii) If $PP_R(M) = \langle P, \pi \rangle$, then:

a) If x occurs free in M and $P = P \backslash x \cup \{x{:}\sigma_1, \ldots, x{:}\sigma_n\}$, then $PP_R(\lambda x.M) = \langle P \backslash x, \sigma_1 \cap \cdots \cap \sigma_n \rightarrow \pi \rangle$.

b) otherwise $PP_R(\lambda x.M) = \langle P, \varphi \rightarrow \pi \rangle$, where φ is a type-variable that does not occur in $\langle P, \pi \rangle$.

iii) If $PP_{\mathrm{R}}(M_1) = <P_1, \pi_1>$ and $PP_{\mathrm{R}}(M_2) = <P_2, \pi_2>$ (we choose if necessary trivial variants such that the $<P_i, \pi_i>$ are disjoint in pairs), and $S_2 = to\mathcal{T}_{\mathrm{C}}(\pi_2)$, then

a) If $\pi_1 = \varphi$, then:

$$PP_{\mathrm{R}}(M_1 M_2) = <S_2, S_1> (<P_1 \cup P_2, \varphi'>),$$
$$\text{where } S_1 = unify_{\mathrm{R}}(\varphi, S_2(\pi_2) \rightarrow \varphi')$$

and φ' is a type-variable not occurring in any other type.

b) If $\pi_1 = \sigma \rightarrow \tau$, then:

$$PP_{\mathrm{R}}(M_1 M_2) = <S_2> * C(<P_1 \cup P_2, \tau>),$$
$$\text{where } C = unify_1(\sigma, S_2(\pi_2), S_2(P_2)).$$

Example 8.4.2.2 Using Rank 2 intersection types, the term $S\ K\ S\ I$ (where S, K and I are the well known lambda terms) has a smaller principal type than using Curry types. Notice that:

$PP_{\mathrm{R}}(S) = (1 \rightarrow 2 \rightarrow 3) \rightarrow (4 \rightarrow 2) \rightarrow 1 \cap 4 \rightarrow 3$,
$PP_{\mathrm{R}}(K) = 5 \rightarrow 6 \rightarrow 5$, and
$PP_{\mathrm{R}}(I) = 7 \rightarrow 7$,

and by definition 8.4.2.1 it is easy but laborious to check that $PP_{\mathrm{R}}(S\ K\ S\ I) = 8 \rightarrow 8$ (in Curry's system – and in ML – this term has the principal type $(9 \rightarrow 10) \rightarrow 9 \rightarrow 10$).

If we define $D = \lambda x.xx$, then we can even check that for example $PP_{\mathrm{R}}(D\,(S\ K\ S\ I)) = PP_{\mathrm{R}}(D\ I) = 9 \rightarrow 9$. Notice that the term $I\ D$ is not typeable in this system.

Lemma 8.4.2.3 If $PP_{\mathrm{R}}(M) = <P, \pi>$, then $P \vdash_2 M{:}\pi$, so $\pi \in \mathcal{T}_2$.

Proof: By induction on the definition of $PP_{\mathrm{R}}(M)$, using lemma 8.3.3.2. □

The following lemma is needed in the proof of theorem 8.4.2.5, and formulates that if a type-chain maps the principal pairs of terms in an application to pairs that allow the use of derivation rule ($\rightarrow$E), then these pairs can also be obtained by first performing a unification.

Lemma 8.4.2.4 Let $\sigma \in \mathcal{T}_2$, and for $i = 1, 2$ $PP_{\mathrm{R}}(M_i) = <P_i, \pi_i>$, such that these pairs are disjoint, and let C be a type-chain such that:

$$C(PP_{\mathrm{R}}(M_1)) = <B_1, \tau \rightarrow \sigma>, \text{ and } C(PP_{\mathrm{R}}(M_2)) = <B_2, \tau>.$$

Then there are type-chains C_g and C', and type $\alpha \in \mathcal{T}_2$ such that:

$$PP_{\mathrm{R}}(M_1 M_2) = C_g(<P_1 \cup P_2, \alpha>), \text{ and } C'(PP_{\mathrm{R}}(M_1 M_2)) = <B_1 \cup B_2, \sigma>.$$

Proof: Since $C(\pi_2) \in \mathcal{T}_1$, by lemma 8.4.1.2 (i) there is a C_1 such that $C = <S_2> * C_1$, with $S_2 = to\mathcal{T}_{\mathrm{C}}(\pi_2)$.

i) $\pi_1 = \varphi$. Take $S_1 = unify_{\mathrm{R}}(\varphi, S_2(\pi_2) \rightarrow \varphi')$, where φ' is a type-variable not occurring in any other type. Assume, without loss of generality, that $C_1(\varphi') = \sigma$. Then by definition 8.4.2.1 (iii.a)

$$PP_{\mathrm{R}}(M_1 M_2) = <S_2, S_1> (<P_1 \cup P_2, \varphi'>).$$

Since $\varphi \in \mathcal{T}_{\mathrm{C}}$ and $\tau \rightarrow \sigma \in \mathcal{T}_2$, by lemma 8.3.3.2 (ii) $\tau \rightarrow \sigma \in \mathcal{T}_1$, so $\tau \rightarrow \sigma \in \mathcal{T}_{\mathrm{C}}$ and $\tau \in \mathcal{T}_{\mathrm{C}}$. So $C_1(S_2(\pi_2) \rightarrow \varphi') \in \mathcal{T}_{\mathrm{C}}$, and by lemma 8.3.3.2 (iii) there are a substitution S_3 and a type-chain C_2 such that:

$$S_3(S_2(\pi_2) \rightarrow \varphi') = \tau \rightarrow \sigma, \text{ and } C_1 = <S_3> * C_2.$$

Assume, without loss of generality, that $S_3(\varphi) = \tau \rightarrow \sigma$. Then by property 1.6 there is a substitution S_4 such that $S_3 = S_4 \circ S_1$. So

$$C = <S_2> * C_1 = <S_2> * <S_3> * C_2 = <S_2, S_1> * <S_4> * C_2.$$

Take $C_g = <S_2, S_1>$, $C' = <S_4> * C_2$ and $\alpha = \varphi'$.

ii) $\pi_1 = \rho \rightarrow \mu$. Since the pairs $<P_1, \rho \rightarrow \mu>$ and $<P_2, \pi_2>$ are disjoint, $C_1(\rho \rightarrow \mu) = \tau \rightarrow \sigma$. Since $C_1(\rho) = C_1(S_2(\pi_2))$, and $S_2(P_2)$ shares no type-variables with ρ, by lemma 8.4.1.2 (ii) there are type-chains C_u and C_2 such that

$$C_u = unify_1(\rho, S_2(\pi_2), S_2(P_2)), \text{ and } C_1 = C_u * C_2.$$

By definition 8.4.2.1 (iii.b)

$$PP_{\mathrm{R}}(M_1 M_2) = <S_2> * C_u(<P_1 \cup P_2, \mu>).$$

Then $C = <S_2> * C_1 = <S_2> * C_u * C_2$. Take $C_g = <S_2> * C_u$, $C' = C_2$, and $\alpha = \mu$. □

The following theorem will show that type-chains are sufficient to generate all possible pairs for a typeable term.

Theorem 8.4.2.5 *Completeness of type-chains.* If $B \vdash_{\mathrm{R}} M{:}\sigma$, then there are a basis P, type π and a type-chain C such that $PP_{\mathrm{R}}(M) = <P, \pi>$, and $C(<P, \pi>) = <B, \sigma>$.

Proof: i) $\sigma \in \mathcal{T}_2$. By induction on the structure of M.

a) $M \equiv x$. Then $B = \{x{:}\sigma\}$, $\sigma \in \mathcal{T}_{\mathrm{C}}$, and $PP_{\mathrm{R}}(x) = <\{x{:}\varphi\}, \varphi>$. Take $C = <(\varphi \mapsto \sigma)>$.

b) $M \equiv \lambda x.M'$. Then by lemma 8.2.5 (i) either:

1) x occurs free in M' and there are $\tau_1, \ldots, \tau_n \in \mathcal{T}_{\mathrm{C}}$, $\tau \in \mathcal{T}_2$ such that

$$\sigma = \tau_1 \cap \cdots \cap \tau_n \rightarrow \tau, \text{ and } B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\} \vdash_2 M'{:}\tau.$$

By induction there are P, π, and a type-chain C such that

$$PP_{\mathrm{R}}(M') = <P, \pi>, \text{ and } C(<P, \pi>) = <B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\}, \tau>.$$

By definition 8.4.2.1 (ii.a) there are $\sigma_1, \ldots, \sigma_m$ $(m \leq n)$ such that

$$P = P \backslash x \cup \{x{:}\sigma_1, \ldots, x{:}\sigma_m\} \text{ and } PP_{\mathrm{R}}(M) = <P \backslash x, \sigma_1 \cap \cdots \cap \sigma_m \rightarrow \pi>.$$

Since $C(<P \backslash x \cup \{x{:}\sigma_1, \ldots, x{:}\sigma_m\}, \pi>) = <B \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\}, \tau>$, by theorem 8.3.3.3 (ii) $C(<P \backslash x, \sigma_1 \cap \cdots \cap \sigma_m \rightarrow \pi>) = <B, \sigma>$.

2) x does not occur free in M' and there are $\tau_1 \in \mathcal{T}_{\mathrm{C}}$, $\tau \in \mathcal{T}_2$ such that

$$\sigma = \tau_1 \rightarrow \tau, \text{ and } B \vdash_2 M'{:}\tau.$$

By induction there are P, π, and a type-chain C' such that

$$PP_{\mathrm{R}}(M') = <P, \pi>, \text{ and } C'(<P, \pi>) = <B, \tau>.$$

By definition 8.4.2.1 (ii.b) there is a φ not occurring in $<P, \pi>$ such that $PP_{\mathrm{R}}(M) = <P, \varphi \rightarrow \pi>$.
Take $C = (\varphi := \tau_1) * C'$, then $C(<P, \varphi \rightarrow \pi>) = <B, \sigma>$.

c) $M \equiv M_1 M_2$. Then by lemma 8.2.5 (ii) there are a $\tau \in \mathcal{T}_1$ and B_1, B_2 such that:

$$B_1 \vdash_2 M_1{:}\tau \rightarrow \sigma, B_2 \vdash_{\mathrm{R}} M_2{:}\tau, \text{ and } B = B_1 \cup B_2.$$

By induction for $i = 1, 2$ there are P_i, π_i, and type-chains C_i such that:

$$PP_R(M_1) = <P_1, \pi_1>, C_1(PP_R(M_1)) = <B_1, \tau\rightarrow\sigma>,$$
$$PP_R(M_2) = <P_2, \pi_2>, \text{ and } C_2(PP_R(M_2)) = <B_2, \tau>.$$

Assume, without loss of generality, that the pairs $<P_i, \pi_i>$ are disjoint. Then the type-chains C_i do not interfere, so

$$C_1 * C_2(PP_R(M_1)) = <B_1, \tau\rightarrow\sigma>, \text{ and } C_1 * C_2(PP_R(M_2)) = <B_2, \tau>.$$

Then, by lemma 8.4.2.4 there is a type-chain C such that $C(PP_R(M_1M_2)) = <B, \sigma>$.

ii) $\sigma = \sigma_1\cap\cdots\cap\sigma_n$. By definition 8.2.4 (ii) there are $B_1, \ldots, B_n$ such that $B = B_1\cup\cdots\cup B_n$, and for every $1 \leq i \leq n$, $B_i \vdash_2 M{:}\sigma_i$. Take pairs $<B_i', \sigma_i'>$, trivial variants of $<B_i, \sigma_i>$ and disjoint in pairs. Take S such that

$$S(<B_1'\cup\cdots\cup B_n', \sigma_1'\cap\cdots\cap\sigma_n'>) = <B_1\cup\cdots\cup B_n, \sigma_1\cap\cdots\cap\sigma_n>.$$

By induction there are P, π, such that $PP_R(M) = <P, \pi>$. Let $D = <n, P, \pi>$, then

$$D(<P, \pi>) = <P_1\cup\cdots\cup P_n, \pi_1\cap\cdots\cap\pi_n>,$$

and by lemma 8.3.2.2 (iii) the $<P_i, \pi_i>$ are disjoint in pairs, and for $1 \leq i \leq n$ $PP_R(M) = <P_i, \pi_i>$. By induction there are type-chains $C_1, \ldots, C_n$ such that

$$\text{for } 1 \leq i \leq n\ C_i(<P_i, \pi_i>) = <B_i', \sigma_i'>.$$

Since the $<P_i, \pi_i>$ and the $<B_i', \sigma_i'>$ are disjoint in pairs, the C_i do not interfere. Take $C = <D> * C_1 * \cdots * C_n * <S>$. □

At this stage, we can more clearly explain why we choose to define a basis as in definition 8.2.3 (ii). Take the term xy. If all types were allowed for term-variables, then the following would be a correct statement:

$$\{x{:}\sigma\cap\tau\rightarrow\rho, y{:}\sigma, y{:}\tau\} \vdash_2 xy{:}\rho.$$

Notice that

$$PP_R(xy) = <\{x{:}1\rightarrow 2, y{:}1\}, 2>.$$

If we want our notion of principal pair to be correct, we should show that we have a way of obtaining the first pair for xy from its principal one, so we need a type-chain of operations C such that:

$$C(<\{x{:}1\rightarrow 2, y{:}1\}, 2>) = <\{x{:}\sigma\cap\tau\rightarrow\rho, y{:}\sigma, y{:}\tau\}, \rho>.$$

However, there exists no type-chain of operations as defined in this chapter that can replace the type $1\rightarrow 2$ by $\sigma\cap\tau\rightarrow\rho$; therefore, we have chosen to restrict the possible predicates for statements that have a term-variable as subject to types in $\mathcal{T}_C$.

Chapter 9 *Applicative Term Rewriting Systems*

In this chapter we will define Applicative Term Rewriting Systems (ATRS), a slight extension of the Term Rewriting Systems as defined in [Klop '90], as term rewriting systems that contain a special binary operator Ap. The Applicative Term Rewriting Systems to be defined in this chapter are extensions to those suggested by most functional programming languages, in that they do not discriminate against the varieties of function symbols that can be used in patterns. As such, there is no distinction between function symbols (e.g. append and plus) and constructor symbols (e.g. cons and succ); the extension made consists of allowing not only constructor-symbols in the operand space of the left hand side of rewrite rules, but all function symbols.

The definition of *applicative* systems as will be done in this chapter is motivated by the following observation: there is a clear translation (embedding) of combinator systems into Term Rewriting Systems, in which the implicit application of the world of combinators is made explicit. The kind of term rewriting system that is needed for such a translation contains only one function symbol, called Ap, and is therefore often called an applicative term rewriting system. A translation of, for example, Combinatory Logic (CL)

$$\begin{array}{ll} S\ x\ y\ z & = x\ z\ (y\ z) \\ K\ x\ y & = x \\ I\ x & = x \end{array}$$

into such a term rewriting system then looks like:

$$\begin{array}{lll} Ap(Ap(Ap(S,x),y),z) & \rightarrow & Ap(Ap(x,z),Ap(y,z)) \\ Ap(Ap(K,x),y) & \rightarrow & x \\ Ap(I,x) & \rightarrow & x \end{array}$$

The definition of applicative systems we will present in this thesis is, however, more general: in the systems we consider, Ap is a *special* function symbol; in particular it is *one of the function symbols*, not the only one. We prefer to see an applicative term rewriting system as a term rewriting system with a *predefined* symbol, which we call Ap. In order to distinguish between the term rewriting systems that contain *only* the function symbol Ap and those that contain Ap next to other function symbols, we call the former the *Pure* Applicative Term Rewriting Systems.

We prefer to see the symbols S, K and I as functions, with 3, 2 and 1 operands, respectively. If we try to capture this view in our translation, then a first attempt would give:

$$\begin{array}{lll} S(x,y,z) & \rightarrow & Ap(Ap(x,z),Ap(y,z)) \\ K(x,y) & \rightarrow & x \\ I(x) & \rightarrow & x. \end{array}$$

Then a term like $S(K,S,I)$ would be illegal, since the functions that appear in operand position are used without the necessary arguments; it would not be possible to translate all combinator expressions. This means that we have to introduce extra rewrite rules to express the

Curried versions of these symbols. Moreover, to get some computational power, some rewrite rule starting with Ap should be added. Such an extended CL system might look like:

$$\begin{array}{lll} S\,(x,y,z) & \rightarrow & Ap\,(Ap\,(x,z), Ap\,(y,z)) \\ Ap\,(Ap\,(Ap\,(S_0,x),y),z) & \rightarrow & S\,(x,y,z) \\ K\,(x,y) & \rightarrow & x \\ Ap\,(Ap\,(K_0,x),y) & \rightarrow & K\,(x,y) \\ I\,(x) & \rightarrow & x \\ Ap\,(I_0,x) & \rightarrow & I\,(x), \end{array}$$

or like,

$$\begin{array}{lll} S\,(x,y,z) & \rightarrow & Ap\,(Ap\,(x,z), Ap\,(y,z)) \\ Ap\,(S_2\,(x,y),z) & \rightarrow & S\,(x,y,z) \\ Ap\,(S_1\,(x),y) & \rightarrow & S_2\,(x,y) \\ Ap\,(S_0,x) & \rightarrow & S_1\,(x) \\ K\,(x,y) & \rightarrow & x \\ Ap\,(K_1\,(x),y) & \rightarrow & K\,(x,y) \\ Ap\,(K_0,x) & \rightarrow & K_1\,(x) \\ I\,(x) & \rightarrow & x \\ Ap\,(I_0,x) & \rightarrow & I\,(x). \end{array}$$

The rules with S_0, K_0, I_0, etc. give in fact the 'Curried' versions of respectively S, K and I.

We will consider the applicative rewriting systems, because they are far more general than the subclass of pure applicative systems. However, all results obtained in the next three chapters are also valid for that subclass. Moreover, every term rewriting system is an applicative term rewriting system.

9.1 *The defined symbol of rewrite rules*

We take the view that in a rewrite rule a certain symbol is defined; it is this symbol to which the structure of the rewrite rule gives a type. The reason for treating Ap as a predefined symbol is the following: It is clear that the rules added to obtain the Curried versions of symbols in the translation of CL into a rewriting system are not intended as definitions for Ap, but as definitions for those Curried versions. However, in general, Term Rewriting Systems are not sensitive for the names used for functions symbols and function symbols can be replaced by others, as long as this is done in a consistent way. So the translation of CL is in fact the same as the one obtained by replacing all Ap's by F:

$$\begin{array}{lll} S\,(x,y,z) & \rightarrow & F\,(F\,(x,z), F\,(y,z)) \\ F\,(S_2\,(x,y),z) & \rightarrow & S\,(x,y,z) \\ F\,(S_1\,(x),y) & \rightarrow & S_2\,(x,y) \\ F\,(S_0,x) & \rightarrow & S_1\,(x) \\ K\,(x,y) & \rightarrow & x \\ F\,(K_1\,(x),y) & \rightarrow & K\,(x,y) \\ F\,(K_0,x) & \rightarrow & K_1\,(x) \end{array}$$

$$\begin{array}{lll} I\,(x) & \rightarrow & x \\ F\,(I_0,x) & \rightarrow & I\,(x). \end{array}$$

Now, all rewrite rules starting with F could be seen as rules that define F. Because this is in fact the same term rewriting system, there should be no difference in the intended meaning, so all rules starting with Ap should then be rules that define Ap. In order to avoid this problem, we will regard those rewriting systems that have a 'predefined' binary function, called Ap, which then cannot be renamed. As suggested above, the symbol Ap is neglected when we are looking for the symbol that is defined in a rewrite rule.

We will consider rewriting systems that are Curry closed. We could have defined a closure operation on ATRS's, by adding rules and extending the set of function symbols, but it is easier to assume that every ATRS is closed. When presenting a rewrite system, however, we will only show the rules that are essential; we will not show the rules that define the Curried versions.

9.2 *Partial type assignment*

The type assignment systems we will present in the next three chapters are partial systems in the sense of [Pfenning '88]. Not only will we define how terms and rewrite rules can be typed, but we will also assume that every function symbol already has a type, which structure is usually motivated by a rewrite rule. There are several reasons to do so.

First of all a term rewriting system can contain a symbol that is not the defined symbol of a rewrite rule (such a symbol is called a constant). A constant can appear in a rewrite rule more or less as a symbol that 'has to be there', but for which it is impossible to determine any functional characterization, apart from what is demanded by the immediate context. If we provide a type for every constant, then we can formulate some consistency requirement by saying that the types used for a constant must be related to the provided type.

Moreover, even for every defined symbol there must be some way of determining what type can be used for an occurrence. Normally the rewrite rules that define such a symbol are investigated, and from analyzing the structure of those rules the 'most general type' for that symbol can be constructed. Instead of investigating all the defining rules for a defined symbol every time the symbol is encountered, we can store the type of the symbol in a mapping from symbols to types, and use this mapping instead. Of course it makes no difference to assume the existence from the start of such a mapping from symbols (both defined and constant) to types, and to define type assignment using that mapping (in the following such a mapping is called an 'environment').

In fact, the approach we take here is very much the same as the one taken by Hindley in [Hindley '69], where he defines the principal Curry-type scheme of an object in Combinatory Logic. Even his notion of type assignment could be regarded as a partial one. Moreover, since combinator systems can easily be translated into (Left Linear) Applicative Term Rewriting Systems, the results of chapter eleven (when restricting the allowed rewrite rules to those that correspond to combinators), are the same as in [Hindley '69].

9.3 *Definitions*

The following definitions will be based on definitions given in [Klop '90]. Definition 9.3.1 will define Applicative Term Rewriting Systems the same way as the definition given by Klop for Term Rewriting Systems, extended with part (i.c) to express the existence of the predefined symbol Ap. Definition 9.3.2 will define a notion of rewriting on Applicative Term Rewriting Systems, the same way as the definition of rewriting given by Klop for Term Rewriting Systems, extended with part (ii.a.3) to express that the possible use of the symbol Ap in the left hand side is restricted, and part (iii) to define the notion of defined symbol of a rewrite rule. In fact, parts (ii.a.3) and (iii) are related. Part (iv) is added to express that only Curry closed rewrite systems are considered.

We will introduce in some parts a notation different from Klop's, because some of the symbols or definitions Klop used were already used in this thesis with a different meaning. For example, we will use the word 'replacement' for the operation that replaces term-variables by terms, instead of the word 'substitution', which was used for operations that replace type-variables by types. (Substitution and replacement are also operations defined in [Curry & Feys '58]. They both are defined as operations on terms; substitution was defined as the operation that replaces term-variables by terms, and replacement was defined as the operation that replaces occurrences of subterms by terms. Note that our definition therefore differs also from the one given in [Curry & Feys '58].) To denote a replacement, we will use capital characters like 'R', instead of Greek characters like 'σ', which were used to denote types. We will use the symbol '$\rightarrow$' for the rewriting symbol, instead of '$\rightarrow$' which was used as a type constructor. We will use the notion 'constant symbol' for a symbol that cannot be rewritten (for which there is no rewrite rule that defines that symbol), instead of for a function symbol with arity 0.

Definition 9.3.1 (cf. [Klop '90]) An *Applicative Term Rewriting System* (ATRS) is a pair (Σ, **R**) of an *alphabet* or *signature* Σ and a set of *rewrite rules* **R**.

i) The alphabet Σ consists of:
   a) A countable infinite set of variables $x_1, x_2, x_3, \ldots$ (or $x, y, z, x', y', \ldots$).
   b) A non empty set $\mathcal{F}$ of *function symbols* or *operator symbols* $F, G, \ldots$, each equipped with an 'arity' (a natural number), i.e. the number of 'arguments' it is supposed to have. We have 0-ary, unary, binary, ternary etc. function symbols.
   c) A special binary operator, called *application* (Ap).
ii) The set of *terms* (or *expressions*) 'over' Σ is $\mathrm{T}(\mathcal{F}, \mathcal{X})$ and is defined inductively:
   a) $x, y, z, \ldots \in \mathrm{T}(\mathcal{F}, \mathcal{X})$.
   b) If $F \in \mathcal{F} \cup \{Ap\}$ is an n-ary symbol ($n \geq 0$), and $T_1, \ldots, T_n \in \mathrm{T}(\mathcal{F}, \mathcal{X})$, then
   $$F(T_1, \ldots, T_n) \in \mathrm{T}(\mathcal{F}, \mathcal{X}).$$
   The T_i ($i = 1, \ldots, n$) are the arguments of the last term.

Definition 9.3.2 (cf. [Klop '90]) Let (Σ, **R**) be an ATRS.

i) A *replacement* R is a map from $\mathrm{T}(\mathcal{F}, \mathcal{X})$ to $\mathrm{T}(\mathcal{F}, \mathcal{X})$ satisfying
   $$\mathrm{R}(F(T_1, \ldots, T_n)) = F(\mathrm{R}(T_1), \ldots, \mathrm{R}(T_n))$$
   for every n-ary function symbol F (here $n \geq 0$). So, R is determined by its restriction to the set of variables. We also write T^{R} instead of $\mathrm{R}(T)$.

ii) a) A *rewrite rule* $\in$ **R** is a pair (Lhs, Rhs) of terms $\in \mathrm{T}(\mathcal{F}, \mathcal{X})$. Often, a rewrite rule will get a name, e.g. **r**, and we write **r** : $Lhs \rightarrow Rhs$. Three conditions will be imposed:
      1) Lhs is not a variable.
      2) The variables occurring in Rhs are contained in Lhs.
      3) For every Ap in Lhs, the left hand argument is not a variable.
   b) A rewrite rule **r** : $Lhs \rightarrow Rhs$ determines a set of *rewrites* $Lhs^{\mathrm{R}} \rightarrow Rhs^{\mathrm{R}}$ for all replacements R. The left hand side Lhs^{R} is called a *redex*; it may be replaced by its '*contractum*' Rhs^{R} inside a context C[ ]; this gives rise to *rewrite steps*:
   $$\mathrm{C}[\ Lhs^{\mathrm{R}}\ ] \rightarrow_{\mathbf{r}} \mathrm{C}[\ Rhs^{\mathrm{R}}\ ].$$
   c) We call $\rightarrow_{\mathbf{r}}$ the *one-step rewrite relation* generated by **r**. Concatenating rewrite steps we have (possibly infinite) *rewrite sequences* $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots$ or *rewrites* for short. If $T_0 \rightarrow \cdots \rightarrow T_n$ we also write $T_0 \rightarrow^* T_n$, and T_n is a *rewrite* of T_0.
iii) a) In a rewrite rule, the leftmost, outermost symbol in the left hand side that is not an Ap, is called *the defined symbol* of that rule.
   b) If the symbol F is the defined symbol of the rewrite rule **r**, then **r** *defines* F.
   c) F is *a defined symbol*, if there is a rewrite rule that defines F.
   d) $Q \in \mathcal{F}$ is called a *constant symbol* if Q is not a defined symbol.
iv) For every defined symbol F with arity $n \geq 1$, there are n additional rewrite rules that define the function symbols F_0 upto F_{n-1} as follows:
   $$\begin{array}{lcl} Ap(F_{n-1}(x_1, \ldots, x_{n-1}), x_n) & \rightarrow & F(x_1, \ldots, x_n) \\ Ap(F_{n-2}(x_1, \ldots, x_{n-2}), x_{n-1}) & \rightarrow & F_{n-1}(x_1, \ldots, x_{n-1}) \\ \vdots & & \\ Ap(F_1(x_1), x_2) & \rightarrow & F_2(x_1, x_2) \\ Ap(F_0, x_1) & \rightarrow & F_1(x_1) \end{array}$$
   The added rules with $F_{n-1}, \ldots, F_1, F_0$, etc. give in fact the 'Curried'-versions of F.

Because of part (iv) in this thesis we will consider only rewriting systems that are called *Curry closed*.

Part (ii.a.3) of definition 9.3.2 was added in order to avoid rewrite rules with left hand sides like $Ap(x, y)$, because such a rule would not have a defined symbol. It was also added to avoid the kind of problem as mentioned in [van Oostrom '90]. In this paper a lambda calculus with patterns was defined: when arbitrary applications like $Ap(x, y)$ are allowed in patterns, then, for example, $\lambda(xy).(yx)$ is a correct term (it changes the order of terms in an application). Using this term in this calculus it is then possible to show that $K = I$. Moreover, this restriction on occurrences of Ap in left hand sides makes it possible to give the following structural definition of those terms.

Definition 9.3.3 *LHS*, the set of terms that are allowed in the left hand side of a rewrite rule, is inductively defined by:

i) If $T_1 \in$ *LHS*, and $T_2 \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ such that either:
   a) $T_2 \equiv x$ for some term-variable x, or

b) $T_2 \in$ *LHS*,

then $Ap(T_1, T_2) \in$ *LHS*.

ii) If $F \in \mathcal{F}$ with arity n, and $M_1, \ldots, M_n \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ such that for every $1 \leq i \leq n$ either:

a) $T_i \equiv x$ for some term-variable x, or

b) $T_i \in$ *LHS*,

then $F(T_1, \ldots, T_n) \in$ *LHS*.

It is easy to show that *LHS* is exactly the set of terms allowed by definition 9.3.2 (ii.a).

We can call a rewrite rule $\mathbf{r} : Lhs \rightarrow Rhs$ with defined symbol F *recursive* if the defined symbol also occurs in other nodes than the defining one, and then call such an F a *recursive symbol*. This definition is, however, not accurate. Take, for example, the rewrite system

$$
\begin{array}{l}
F(x) = G(x) \\
G(x) = F(x).
\end{array}
$$

Then these rewrite rules are not recursive according to the definition. However, apart from the length of rewrite sequences, this rewrite system is in fact the same as

$$
\begin{array}{l}
F(x) = G(x) \\
F(x) = F(x) \\
G(x) = F(x)
\end{array}
$$

and now of course F is a recursive symbol. Therefore, the first system too is regarded as a recursive one, and the two rewrite rules are called *mutually recursive*. If allowing this kind of recursion, we are in fact forced to give a different notion of defined symbol, since both rules in the first system define F and G *simultaneously*. Therefore, for the sake of simplicity, we will assume that rules are not mutually recursive.

Proposition 9.3.4 Let F be the defined symbol of the rewrite rule $\mathbf{r} : Lhs \rightarrow Rhs$. Then there are $n \geq j \geq 0$, and $T_1, \ldots, T_n \in \mathrm{T}(\mathcal{F}, \mathcal{X})$, such that F has arity j and:

$$Lhs = Ap(Ap(\cdots Ap(F(T_1, \ldots, T_j), T_{j+1}), \cdots), T_n)$$

and $T_1, \ldots, T_n$ are called the *patterns* of $\mathbf{r}$.

Proof: Easy. □

In chapter eleven a notion of type assignment on Left Linear Applicative Term Rewriting Systems will be defined. These systems are defined as above, extended with parts to express the left linearity of rewrite rules.

Definition 9.3.5 A *Left Linear Applicative Term Rewriting System* (LLATRS) is a pair $(\Sigma, \mathbf{R})$ of an *alphabet* or *signature* Σ and a set of *rewrite rules* $\mathbf{R}$ and is defined as in definition 9.3.1, extended with:

iii) Terms in which no variable occurs twice or more, are called *linear*.

Definition 9.3.6 Let $(\Sigma, \mathbf{R})$ be a LLATRS. The notion of rewriting on LLATRS's is defined as is definition 9.3.2, extended with:

ii) a) 4) Lhs is linear.

In the following definition we will give a special Applicative Term Rewriting System.

Definition 9.3.7 *Applicative Combinatory Logic* (ACL) is the ATRS $(\Sigma, \mathbf{R})$, where $\mathcal{F} = \{S, S_2, S_1, S_0, K, K_1, K_0, I, I_0\}$, and $\mathbf{R}$ contains the rewrite rules

$$
\begin{array}{lcl}
S(x, y, z) & \rightarrow & Ap(Ap(x, z), Ap(y, z)) \\
K(x, y) & \rightarrow & x \\
I(x) & \rightarrow & x.
\end{array}
$$

Notice that ACL is even a LLATRS.

For ACL we have, for example, the following rewriting sequence:

$$
\begin{array}{ll}
S(K_0, S_0, I_0) & \rightarrow \\
Ap(Ap(K_0, I_0), Ap(S_0, I_0)) & \rightarrow \\
Ap(K_1(I_0), Ap(S_0, I_0)) & \rightarrow \\
K(I_0, Ap(S_0, I_0)) & \rightarrow \\
I_0. &
\end{array}
$$

Notice that a term like $K_1(I_0)$ itself cannot be rewritten. This corresponds to the fact that in CL the term $K\,I$ is not a redex.

Because ACL is Curry-closed, it is in fact combinatory complete: every lambda term can be translated into a term in ACL; for details of such a translation, see [Barendregt '84, Dezani-Ciancaglini & Hindley '92].

Example 9.3.8 Rewrite rules can of course be more complicated than was illustrated above by the rules for ACL. In general, if the left hand side of a rewrite rule is $F(T_1, \ldots, T_n)$, then the T_i need not be simple variables but can be terms as well, as, for example, in the rewrite rule

$$H(S_2(x, y)) \rightarrow S_2(I_0, y).$$

It is also possible that for a certain symbol F, there is more than one rewrite rule that defines F. For example, the rewrite rules:

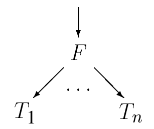
$$
\begin{array}{lcl}
F(x) & \rightarrow & x \\
F(x) & \rightarrow & Ap(x, x)
\end{array}
$$

are legal.

9.4 *Tree-representation of terms and rewrite rules*

The three different notions of type assignment on ATRS's as defined in the next three chapters will in fact be defined on the tree-representation of terms and rewrite rules of these systems. In this section we will present this representation in a formal way.

Definition 9.4.1 i) The tree-representation of terms and rewrite rules is in a straightforward way obtained by representing a term $F(T_1, \ldots, T_n)$ by:



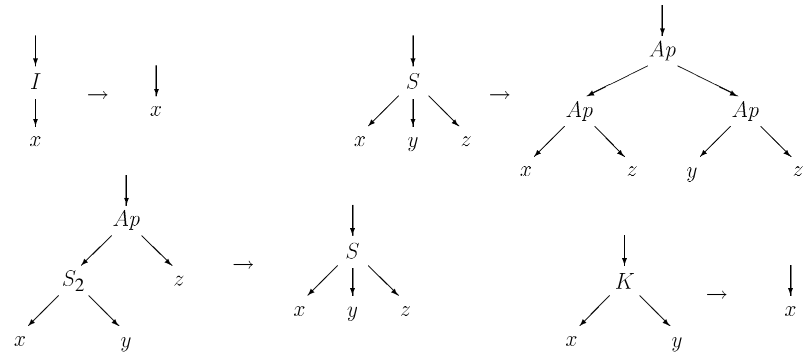
ii) The *spine* of a term-tree is defined as usual, i.e.: the root node of the term-tree is on the spine, and if a node is on the spine, then its left most descendant is on the spine.

iii) In the tree-representation of a rewrite rule, the first node on the spine of the left hand side (starting from the root node) that does not contain an Ap is called the *defining node* of that rule. Notice that if F is the defined symbol of the rule, then it occurs in the defining node.

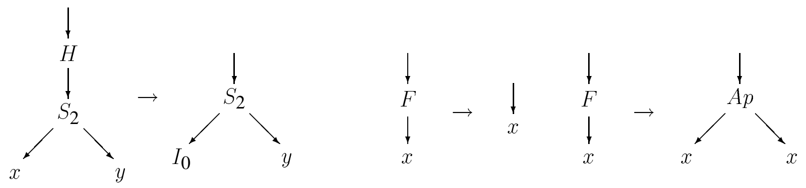
iv) The edge pointing to the the root of a term is called the *root edge*.

v) A node containing a term-variable (a function symbol $F \in \mathcal{F}$, the symbol Ap) will be called a *variable node* (*function node*, *application node*).

Example 9.4.2 We give some of the rewrite rules of ACL in tree-representation:



Example 9.4.3 We give the tree-representations of the rewrite rules given in example 9.3.8.



Chapter 10 *Partial Intersection Type Assignment in Applicative Term Rewriting Systems*

In this chapter and the following two, we will define three different notions of partial type assignment on the tree-representation of terms and rewrite rules. The only constraints on these systems are local, and are imposed by the relation between the type assigned to a node and those assigned to its incoming and out-going edges. Assigning types to an ATRS will consist of assigning types to function symbols via a mapping that is called an environment, and labelling the nodes and edges in the tree-representation of terms with type information. Left- and right hand side of rewrite rules will be typed as terms, and conditions will be formulated that the types assigned to those terms should satisfy.

Types are labelled to nodes to capture the notion of 'type of a function', 'type of a constant' or 'type of a variable', and are assigned to edges to capture the notion of 'type of a subterm' (or term-tree). The type assigned to the root edge of a term-tree is the type assigned to the term that is represented by the term-tree.

In this chapter we will present a notion of type assignment on ATRS's that is based on the essential type assignment system for the lambda calculus and Milner's system for ML. Milner's system plays a role in recursive definitions.

As remarked in section 6.4, the essential type assignment system for the lambda calculus has the principal type property. The operations on types for this system are strict substitution, strict expansion, and lifting, and all pairs for a lambda term can be generated by chains that exist of strict expansions, strict substitutions (in that order), and that end with one lifting. But, since all those operations can be proved sound on all pairs, we will, in this section, allow the operations to appear in arbitrary order, so we will use just ordinary chains. Moreover, in this section we will drop the distinction between the operations of expansions and type-expansions made in section 6.2.2; we will call both expansions.

The three operations on types are used to define type assignment on ATRS's: types that can be assigned to occurrences of function symbols can be obtained from the type provided by the environment using a chain of operations. These operations will be proved sound on typeable term-trees. For the operations of strict substitution and strict expansion we will prove a notion of soundness on rewrite rules, and we will show that it is not possible to show such a result for the operation of lifting.

It will be shown that type assignment in term rewriting systems in general does not satisfy the subject reduction property: i.e. types are not preserved under rewriting. For the notions of type assignment presented in this chapter and the next two we will formulate a sufficient condition rewrite rules should satisfy to obtain subject reduction. For the system of chapter eleven, even this condition will be proved necessary. For the system presented in this chapter this condition is not decidable. It is so for the systems of chapters eleven and twelve.

We will discuss the differences between the notions of type assignment defined for ATRS's that are based on Milner's or Mycroft's way of dealing with recursion.

10.1 *Essential type assignment in ATRS's*

Partial intersection type assignment on an ATRS (Σ, R) is defined as the labelling of nodes and edges in the tree-representation of terms and rewrite rules with types in $\mathcal{T}_{\mathrm{S}}$. In this labelling, we will use that there is a mapping that provides a type in $\mathcal{T}_{\mathrm{s}}$ for every $F \in \mathcal{F} \cup \{Ap\}$. Such a mapping is called an environment.

Definition 10.1.1 Let $(\Sigma, \mathbf{R})$ be an ATRS.

i) A mapping $\mathcal{E} : \mathcal{F} \cup \{Ap\} \to \mathcal{T}_{\mathrm{s}}$ is called an *environment* if $\mathcal{E}(Ap) = (1{\to}2){\to}1{\to}2$, and for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$.

ii) For $F \in \mathcal{F}$ with arity $n \geq 0$, $\sigma \in \mathcal{T}_{\mathrm{s}}$, and $\mathcal{E}$ an environment, the environment $\mathcal{E}[F := \sigma]$ is defined by:

   a) $\mathcal{E}[F := \sigma](G) = \sigma$, if $G \in \{F, F_{n-1}, \ldots, F_0\}$.

   b) $\mathcal{E}[F := \sigma](G) = \mathcal{E}(G)$, otherwise.

The condition that $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$ is not essential. It is only introduced for reasons of efficiency; we want to be able to assign the same set of types to all Curried versions of a function symbol. It could be dropped without loss of results.

Since $\mathcal{E}$ maps all $F \in \mathcal{F}$ to types in $\mathcal{T}_{\mathrm{s}}$, no function symbol in particular is mapped to the type constant ω.

Type assignment on applicative term rewriting systems will be defined in two stages. In the next definition we will define type assignment on terms, in definition 10.1.7 we will define type assignment on term rewrite rules.

Definition 10.1.2 Let $(\Sigma, \mathbf{R})$ be an ATRS, and $\mathcal{E}$ an environment.

i) We say that $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ *is typeable by* $\sigma \in \mathcal{T}_{\mathrm{S}}$ *with respect to* $\mathcal{E}$, if there exists an assignment of types to edges and nodes that satisfies the following constraints:

   a) The root edge of T is typed with σ; if $\sigma = \omega$, then the root edge is the only thing in the term-tree that is typed.

   b) The type assigned to a function node containing $F \in \mathcal{F} \cup \{Ap\}$ (where F has arity $n \geq 0$) is $\tau_1 \cap \cdots \cap \tau_m$, if and only if for every $1 \leq i \leq m$ there are $\sigma_1^i, \ldots, \sigma_n^i \in \mathcal{T}_{\mathrm{S}}$, and $\sigma_i \in \mathcal{T}_{\mathrm{s}}$, such that $\tau_i = \sigma_1^i {\to} \cdots {\to} \sigma_n^i {\to} \sigma_i$, the type assigned to the j-th $(1 \leq j \leq n)$ out-going edge is $\sigma_j^1 \cap \cdots \cap \sigma_j^m$, and the type assigned to the incoming edge is $\sigma_1 \cap \cdots \cap \sigma_m$.

$$
\begin{array}{c}
\downarrow \sigma_1 \cap \cdots \cap \sigma_m \\
F{:}(\sigma_1^1 {\to} \cdots {\to} \sigma_n^1 {\to} \sigma_1) \cap \cdots \cap (\sigma_1^m {\to} \cdots {\to} \sigma_n^m {\to} \sigma_m) \\
\swarrow \sigma_1^1 \cap \cdots \cap \sigma_1^m \quad \swarrow \sigma_2^1 \cap \cdots \cap \sigma_2^m \quad \cdots \quad \searrow \sigma_{n-1}^1 \cap \cdots \cap \sigma_{n-1}^m \quad \searrow \sigma_n^1 \cap \cdots \cap \sigma_n^m
\end{array}
$$

   c) If the type assigned to a function node containing $F \in \mathcal{F} \cup \{Ap\}$ is τ, then there is a chain C such that $C(\mathcal{E}(F)) = \tau$.

ii) Let $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ be typeable by σ with respect to $\mathcal{E}$. If B is a basis such that for every statement $x{:}\tau$ occurring in the typed term-tree there is a $x{:}\tau' \in B$ such that $\tau' \leq_{\mathrm{E}} \tau$, we write $B \vdash_{\mathcal{E}} T{:}\sigma$.

Notice that if $B \vdash_{\mathcal{E}} T{:}\sigma$, then B can contain more statements than needed to obtain $T{:}\sigma$. Notice also that parts (i.a) and (ii) are not in conflict, so for every B and T: $B \vdash_{\mathcal{E}} T{:}\omega$.

A typical example for part (i.b) of definition 10.1.2 is the symbol Ap, for which every environment provides the type $(1{\to}2){\to}1{\to}2$. So for every occurrence of Ap in a term-tree, there are σ and τ such that the following is part of the typed term-tree.

$$
\begin{array}{c}
\downarrow \tau \\
Ap{:}\ (\sigma{\to}\tau){\to}\sigma{\to}\tau \\
\sigma{\to}\tau \swarrow \qquad \searrow \sigma
\end{array}
$$

Notice that the type the environment provides for Ap is crucial; it is the type suggested by the $({\to}\mathrm{E})$ derivation rule, and gives structure to the type assignment. As remarked by F.J. de Vries of the C.W.I. in Amsterdam, the Netherlands (private communication), it is possible to define type assignment on term rewriting systems in a trivial way: We could define an environment in such a way that the only requirement is that for every $F \in \mathcal{F} \cup \{Ap\}$ with arity n there are $\sigma_1, \ldots, \sigma_n$, and σ such that $\mathcal{E}(F) = \sigma_1 {\to} \cdots {\to} \sigma_n {\to} \sigma$. Then it would be, for example, possible to define it in such a way that the type provided for Ap is $1{\to}2{\to}3$. Moreover, it would then be possible to assign the type $0{\to}0{\to}\cdots{\to}0{\to}0$ $(n+1$ times) to all F with arity n, and all structure would be lost. Also, such a system would then not be a natural extension of the essential type assignment system for the lambda calculus.

Instead of saying 'typeable with respect to $\mathcal{E}$', we will just say 'typeable', and instead of saying 'the type assigned to the node containing a function symbol (variable)' we will often speak of 'the type assigned to a function symbol (variable)'.

Example 10.1.3 The term $S(K_0, S_0, I_0)$ can be typed with the type $7{\to}7$, under the assumption that:

$$
\begin{array}{ll}
\mathcal{E}(S) & = (1{\to}2{\to}3){\to}(4{\to}2){\to}1{\cap}4{\to}3 \\
\mathcal{E}(K) & = 5{\to}\omega{\to}5 \\
\mathcal{E}(I) & = 6{\to}6.
\end{array}
$$

$$
\begin{array}{c}
\downarrow 7{\to}7 \\
S{:}((7{\to}7){\to}\omega{\to}7{\to}7) \to \omega \to (7{\to}7) \to 7{\to}7 \\
\swarrow \qquad \downarrow \qquad \searrow \\
K_0{:}(7{\to}7){\to}\omega{\to}7{\to}7 \qquad S_0 \qquad I_0{:}7{\to}7
\end{array}
$$

Notice that to obtain the type $((7{\to}7){\to}\omega{\to}7{\to}7){\to}\omega{\to}(7{\to}7){\to}7{\to}7$ for S, we used the chain $<(1 := 7{\to}7), (2 := \omega), (3 := 7{\to}7), (4 := \omega)>$, and that the node containing S_0 is not typed since the incoming edge is typed with ω.

If we define $D(x) \to Ap(x, x)$, then we can even check that, for example, $D(S(K_0, S_0, I_0))$ and $D(I_0)$ are typeable by $8{\to}8$.

For the notion of type assignment as defined in definition 10.1.2 the following properties hold:

Lemma 10.1.4 i) $B \vdash_{\mathcal{E}} T{:}\sigma_1 \cap \cdots \cap \sigma_n \iff \forall\, 1 \leq i \leq n\ [\ B \vdash_{\mathcal{E}} T{:}\sigma_i\ ]$.

ii) If $x{:}\tau$ is a statement occurring in the typed term-tree for $B \vdash_{\mathcal{E}} T{:}\sigma$, then $B \leq_{\mathrm{E}} \{x{:}\tau\}$.

Proof: Immediately from definition 10.1.2. □

In the next definition we will introduce the notion of essentially used bases, which is, because of the relation $\leq_{\mathrm{E}}$, more specific than the notion of used basis. The idea of this notion is to collect all types assigned to term-variables that are actually used for the typed term-tree. A difference, however, with the notion of used basis is that the collected types need not occur in the original bases themselves.

Definition 10.1.5 i) The *essentially used bases of* $B \vdash_{\mathcal{E}} T{:}\sigma$ are inductively defined by:

a) $\sigma \in \mathcal{T}_{\mathrm{S}}$.

1) $T \equiv x$. Take $\{x{:}\sigma\}$.

2) $T \equiv F(T_1, \ldots, T_n)$. There are $\sigma_1, \ldots, \sigma_n$ such that for every $1 \leq i \leq n$ $B \vdash_{\mathcal{E}} T_i{:}\sigma_i$. Let for $1 \leq i \leq n$, B_i be an essentially used basis of $B \vdash_{\mathcal{E}} T_i{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

b) If $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$ $(n \geq 0)$, then by lemma 10.1.4 (i) for every $1 \leq i \leq n$ $B \vdash_{\mathcal{E}} T{:}\sigma_i$. Let for every $1 \leq i \leq n$, B_i be an essentially used basis of $B \vdash_{\mathcal{E}} T{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

ii) A basis B *is essentially used for* $T{:}\sigma$ *with respect to* $\mathcal{E}$ if and only if there is a basis B' such that $B' \vdash_{\mathcal{E}} T{:}\sigma$ and B is an essentially used basis of $B' \vdash_{\mathcal{E}} T{:}\sigma$.

Notice that in part (i.b), if $n = 0$, then $\sigma = \omega$, and $\Pi\{B_1, \ldots, B_n\} = \emptyset$.

We will say 'B is essentially used for $T{:}\sigma$' instead of 'B is essentially used for $T{:}\sigma$ with respect to $\mathcal{E}$'. As before, an essentially used basis for a statement $T{:}\sigma$ is not unique, but again the results of this chapter do not depend on the actual structure of such a basis, only on its existence.

For essentially used bases, the following properties hold.

Lemma 10.1.6 i) If B is essentially used for $T{:}\sigma$, then $B \vdash_{\mathcal{E}} T{:}\sigma$.

ii) $B \vdash_{\mathcal{E}} T{:}\sigma \iff \exists\, B'\ [\ B \leq_{\mathrm{E}} B'\ \&\ B'$ is essentially used for $T{:}\sigma\ ]$.

Proof: By induction on definition 10.1.5. □

Thanks to the notion of essentially used basis, we can give a clear definition of a typeable rewrite rule and a typeable rewrite system. The condition 'B is essentially used for $Lhs{:}\sigma$' in definition 10.1.7 (i.a) is crucial. Just saying:

> We say that $Lhs \to Rhs \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis B, type $\sigma \in \mathcal{T}_{\mathrm{S}}$, and an assignment of types to nodes and edges such that: $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$

would give a notion of type assignment that is not closed under rewriting (i.e. does not satisfy the subject reduction property), and is not a natural extension of the essential intersection type assignment system for the λ-calculus. As an example of the first, take the rewrite system

$$
\begin{array}{ll}
I(x) & \to x \\
K(x,y) & \to x \\
F(I_0) & \to I_0 \\
G(x) & \to F(x).
\end{array}
$$

Take the environment $\mathcal{E}$:

$$
\begin{array}{ll}
\mathcal{E}(I) & = 1 \to 1 \\
\mathcal{E}(K) & = 2 \to \omega \to 2 \\
\mathcal{E}(F) & = (3 \to 3) \to 4 \to 4 \\
\mathcal{E}(G) & = (5 \to \omega \to 5) \to 6 \to 6.
\end{array}
$$

Take $B = \{x{:}(7\to7) \cap (5\to\omega\to5)\}$, then $B \vdash_{\mathcal{E}} G(x){:}6\to6$, and $B \vdash_{\mathcal{E}} F(x){:}6\to6$. Notice that $\vdash_{\mathcal{E}} G(K_0){:}7\to7$, but not $\vdash_{\mathcal{E}} F(K_0){:}7\to7$.

Therefore, a minimal requirement for subject reduction will be to demand that all types assigned to term-variables in the typed term-tree for the right hand side of a rewrite rule already occurred in the typed term-tree for the left hand side. This is accomplished by restricting the possible bases to those that contain nothing but the types actually used for the left hand side.

As an example of the second, take the rewrite rule

$$E(x,y) \to Ap(x,y).$$

Let $\mathcal{E}(E) = 3\to1\to4$. Take $B = \{x{:}3 \cap (1\to4), y{:}1\}$, then $B \vdash_{\mathcal{E}} E(x,y){:}4$ and $B \vdash_{\mathcal{E}} Ap(x,y){:}4$. This rewrite rule for E corresponds to the lambda term $\lambda xy.xy$, but $3\to1\to4$ is not a correct type for this term in the type assignment system of chapter five.

Definition 10.1.7 Let $(\Sigma, \mathbf{R})$ be an ATRS, and $\mathcal{E}$ an environment.

i) We say that $Lhs \to Rhs \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis B, type $\sigma \in \mathcal{T}_{\mathrm{S}}$, and an assignment of types to nodes and edges such that:

a) B is essentially used for $Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$.

b) In $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$, all nodes containing F are typed with $\mathcal{E}(F)$.

ii) We say that $(\Sigma, \mathbf{R})$ *is typeable with respect to* $\mathcal{E}$, if every $\mathbf{r} \in \mathrm{R}$ is typeable with respect to $\mathcal{E}$.

Condition (i.b) of definition 10.1.7 is in fact added to ensure that the type provided by the environment for a function symbol F is not in conflict with the rewrite rules that define F. By restricting the type that can be assigned to the defined symbol to the type provided by the environment, we ensure that the rewrite rule is typed using that type, and not using some other type. By part (i.b) of definition 10.1.7, all occurrences of the defined symbol in a rewrite rule are typed with the same type, so type assignment of rewrite rules is actually defined using Milner's solution for recursion.

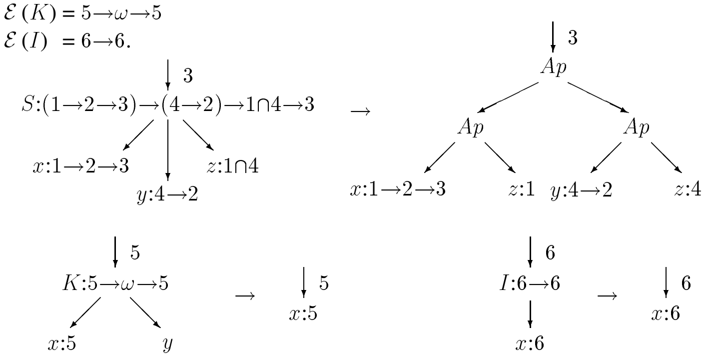
It is easy to check that if F is a function symbol with artity n, and all rewrite rules that define F are typeable, then there are $\gamma_1, \ldots, \gamma_n, \gamma$ such that $\mathcal{E}(F) = \gamma_1 \to \cdots \to \gamma_n \to \gamma$.

The use of an environment corresponds to the use of 'axiom-schemes', and part (i.c) of definition 10.1.2 to the use of 'axioms' as in [Hindley '69], and to the use of a 'combinator basis' and the axioms in definition 3.2 of [Dezani-Ciancaglini & Hindley '92]. Also, because of these definitions the type assignment system we present in this chapter is a partial system; each function symbol *has* a type provided by the environment, and the connection between that type and the type assigned to that symbol in a term is given in definition 10.1.2 (i.c).

The combination of these two definitions also introduces a notion of polymorphism into our type assignment system. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type.

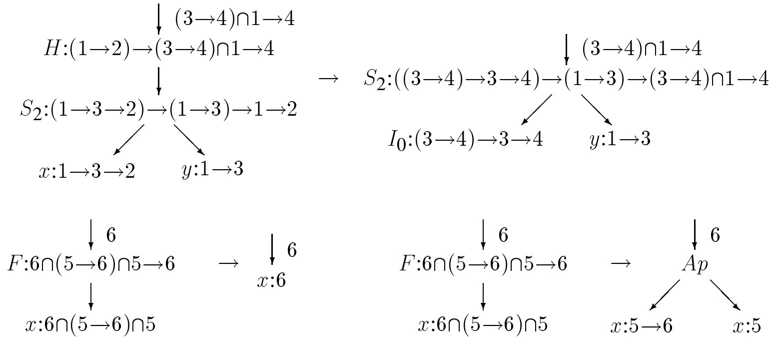
Example 10.1.8 Typed versions of some of the rewrite rules for ACL, assuming that:

$\mathcal{E}(S) = (1\to2\to3)\to(4\to2)\to1\cap4\to3$
$\mathcal{E}(K) = 5\to\omega\to5$
$\mathcal{E}(I) = 6\to6.$



Example 10.1.9 Typed versions of the the rewrite rules given in example 9.4.3, using:

$\mathcal{E}(H) = (1\to2)\to(3\to4)\cap1\to4$
$\mathcal{E}(F) = 6\cap(5\to6)\cap5\to6$
$\mathcal{E}(S) = (7\to8\to9)\to(10\to8)\to7\cap10\to9$
$\mathcal{E}(I) = 11\to11.$



10.2 *Soundness of strict operations*

In this section we will show that the three strict operations on pairs (strict substitution, strict expansion, and lifting) are sound on typed term-trees. For the operations of strict substitution and strict expansion we will also show that part (i.c) of definition 10.1.2 is sound in the following sense: if there is an operation *O* (either a strict substitution or a strict expansion) such that

$O(\mathcal{E}(F)) = \sigma$, then for every type $\tau \in \mathcal{T}_S$ such that $\sigma \leq_S \tau$, the rewrite rules that define F are typeable with respect to the changed environment $\mathcal{E}[F := \tau]$.

We will also show that we cannot prove such a property for the operation of lifting.

Lemma 10.2.1 Let *S* be a strict substitution.

i) If $\sigma \leq_E \tau$, then $S(\sigma) \leq_E S(\tau)$.
ii) If $B \leq_E B'$, then $S(B) \leq_E S(B')$.

Proof: Easy. □

The following theorem will show that strict substitution is a sound operation on typed term-trees and rewrite rules.

Theorem 10.2.2 *Soundness of strict substitution.* Let *S* be a strict substitution.

i) If $B \vdash_{\mathcal{E}} T{:}\sigma$, then $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma)$.
ii) If B is essentially used for $T{:}\sigma$, then $S(B)$ is essentially used for $T{:}S(\sigma)$.
iii) Let **r**: $Lhs \to Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of **r**. Then **r** is typeable with respect to $\mathcal{E}[F := S(\mathcal{E}(F))]$.

Proof: i) Similar to that for theorem 6.2.1.4, by induction on $\mathcal{T}_S$.

a) $\sigma \in \mathcal{T}_s$. This part is proved by induction on the structure of terms. The case $S(\sigma) = \omega$ is trivial, so, in the rest of the proof, $S(\sigma) \neq \omega$, and, therefore, by lemma 6.2.1.2 (ii), $S(\sigma) \in \mathcal{T}_s$.
   1) $T \equiv x$. Then $B \leq_E \{x{:}\sigma\}$. By lemma 10.2.1 (ii) $S(B) \leq_E S(\{x{:}\sigma\}) = \{x{:}S(\sigma)\}$, so $S(B) \vdash_{\mathcal{E}} x{:}S(\sigma)$.
   2) $T \equiv F(T_1, \ldots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_S$, and a chain C such that for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} T_i{:}\sigma_i$, and $C(\mathcal{E}(F)) = \sigma_1\to\cdots\to\sigma_n\to\sigma$. By induction for every $1 \leq i \leq n$, $S(B) \vdash_{\mathcal{E}} T_i{:}S(\sigma_i)$; since $C * \langle S \rangle$ is a chain and
   $$C * \langle S \rangle(\mathcal{E}(F)) = S(\sigma_1)\to\cdots\to S(\sigma_n)\to S(\sigma),$$
   we obtain $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma)$.
b) $\sigma = \sigma_1\cap\cdots\cap\sigma_n$. Then, for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} T{:}\sigma_i$. Let $\sigma_1{}', \ldots, \sigma_m{}'$ be the elements in $\{\sigma_1, \ldots, \sigma_n\}$ such that $S(\sigma_i{}') \neq \omega$. By induction for every $1 \leq i \leq m$, $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma_i{}')$. Then $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma_1{}')\cap\cdots\cap S(\sigma_m{}')$, so $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma)$.
c) $\sigma = \omega$, then $S(\sigma) = \omega$. Obviously $S(B) \vdash_{\mathcal{E}} T{:}\omega$.

ii) As the proof of part (i).
iii) If **r** is a rewrite rule typeable with respect to $\mathcal{E}$, then, by definition 10.1.7 (i), there is a basis B and $\sigma \in \mathcal{T}_s$, and an assignment f of types to nodes and edges such that B is essentially used for $Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$. Moreover, in $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$ all nodes containing F are typed with $\mathcal{E}(F)$.
We will check the requirements of definition 10.1.7 (i) for $\mathcal{E}[F := S(\mathcal{E}(F))]$.
Let $B' = S(B)$, $\sigma' = S(\sigma)$, and take the assignment f' of types to nodes and edges such that $f'(\text{node/edge}) = S(f(\text{node/edge}))$. Then:
   a) By part (ii): B' is essentially used for $Lhs{:}\sigma'$, and by part (i): $B' \vdash_{\mathcal{E}} Rhs{:}\sigma'$.
   b) All occurrences of F in $B' \vdash_{\mathcal{E}} Lhs{:}\sigma'$ and $B' \vdash_{\mathcal{E}} Rhs{:}\sigma'$ are typed with $S(\mathcal{E}(F))$. □

In the ATRS-world we are not able to prove the counterpart of lemma 6.2.2.4. As a counter example, take the term $F(x)$ and assume that $\mathcal{E}(F) = \varphi_1 \rightarrow \varphi_2$. Then we have:

$$\{x{:}\varphi_3\} \vdash_{\mathcal{E}} F(x){:}\varphi_4.$$

In the construction of the expansion $T_{<\varphi_4,\, n,\, \{x:\varphi_3\},\, \varphi_4>}$, the set $\mathcal{V}_{\varphi_4}(<\{x{:}\varphi_3\},\ \varphi_4>) = \{\varphi_4\}$. This difference is caused by the fact that the set of $\lambda\bot$-normal forms does not contain redexes; from the type assignment point of view the term $F(x)$ is a redex. However, lemma 6.2.2.4 is only needed to prove that strict expansion is closed on ground pairs, in other words to prove completeness of operations in the strict type assignment system. We will not show such a property for the type assignment system of this chapter, and, therefore, the loss of this lemma is harmless. Moreover, we can show that strict expansion is a sound operation on typed term-trees. In order to do so, we will only need a more general formulation of the lemmas 6.2.2.3 and 6.2.2.5.

Lemma 10.2.3 Let $B' \vdash_{\mathcal{E}} T{:}\tau$, where $\tau \in \mathcal{T}_{\mathrm{s}}$, and $<\mu, n, B, \sigma>$ be a strict expansion such that $\mathcal{T}_{<B',\,\tau>} \subseteq \mathcal{T}_{<B,\sigma>}$, and $\rho \in \mathcal{T}_{<B,\sigma>}$. Then

i) a) For $1 \leq i \leq n$, there are ρ_i and a strict substitution S_i such that $S_i(\rho) = \rho_i$ and $<\mu, n, B, \sigma>(\rho) = \rho_1 \cap \cdots \cap \rho_n$, or
   b) $<\mu, n, B, \sigma>(\rho) \in \mathcal{T}_{\mathrm{s}}$.

ii) a) For $1 \leq i \leq n$, there are B_i, τ_i, and a strict substitution S_i such that $S_i(<B', \tau>) = <B_i, \tau_i>$, and $<\mu, n, B, \sigma>(<B', \tau>) = <\Pi\{B_1, \ldots, B_n\},\ \tau_1 \cap \cdots \cap \tau_n>$, or
   b) $<\mu, n, B, \sigma>(<B', \tau>) = <B'', \tau'>$, with $\tau' \in \mathcal{T}_{\mathrm{s}}$.

Proof: By definition 6.2.2.2. □

Lemma 10.2.4 Let E be a strict expansion.

i) If $\sigma \leq_{\mathrm{E}} \tau$, then $E(\sigma) \leq_{\mathrm{E}} E(\tau)$.

ii) If $B \leq_{\mathrm{E}} B'$, then $E(B) \leq_{\mathrm{E}} E(B')$.

Proof: Easy. □

We can now prove that strict expansion is a sound operation on typed term-trees and rewrite rules.

Theorem 10.2.5 *Soundness of strict expansion.* Let E be a strict expansion such that $E(<B, \sigma>) = <B', \sigma'>$.

i) If $B \vdash_{\mathcal{E}} T{:}\sigma$, then $B' \vdash_{\mathcal{E}} T{:}\sigma'$.

ii) If B is essentially used for $T{:}\sigma$, then B' is essentially used for $T{:}\sigma'$.

iii) Let **r**: $Lhs \rightarrow Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of **r**. If $E(\mathcal{E}(F)) = \tau \in \mathcal{T}_{\mathrm{S}}$, then, for every $\mu \in \mathcal{T}_{\mathrm{s}}$ such that $\tau \leq_{\mathrm{S}} \mu$, **r** is typeable with respect to $\mathcal{E}[F := \mu]$.

Proof: i) By induction on $\mathcal{T}_{\mathrm{S}}$. We will only show the part $\sigma \in \mathcal{T}_{\mathrm{s}}$. Then, by lemma 10.2.3 either:

a) $\sigma' = \tau_1 \cap \cdots \cap \tau_m$, $B' = \Pi\{B_1, \ldots, B_m\}$, and for every $1 \leq j \leq m$ there is a strict substitution S such that $S(<B, \sigma>) = <B_j, \tau_j>$. Then, by theorem 10.2.2 (i), for every $1 \leq j \leq m$, $B_j \vdash_{\mathcal{E}} T{:}\tau_j$, so $B' \vdash_{\mathcal{E}} T{:}\sigma'$.

b) $\sigma' \in \mathcal{T}_{\mathrm{s}}$. This part is proved by induction on the structure of terms.
   1) $T \equiv x$. Then $B \leq_{\mathrm{E}} \{x{:}\sigma\}$. By lemma 10.2.4 (ii) $B' \leq_{\mathrm{E}} \{x{:}\sigma'\}$, so $B' \vdash_{\mathcal{E}} x{:}\sigma'$.
   2) $T \equiv F(T_1, \ldots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_n, \sigma_1{}', \ldots, \sigma_n{}' \in \mathcal{T}_{\mathrm{S}}$, and a chain C such that $C(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$, and, for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} T_i{:}\sigma_i$ and $E(\sigma_i) = \sigma_i{}'$. By induction for every $1 \leq i \leq n$, $B' \vdash_{\mathcal{E}} T_i{:}\sigma_i{}'$; since $C * <E>$ is a chain and $C * <E>(\mathcal{E}(F)) = \sigma_1{}' \rightarrow \cdots \rightarrow \sigma_n{}' \rightarrow \sigma'$, we obtain $B' \vdash_{\mathcal{E}} T{:}\sigma$.

ii) Similar to the proof of part (i).

iii) Since $\mathcal{E}(F) \in \mathcal{T}_{\mathrm{s}}$, by lemma 10.2.3 either:

a) $\tau = \tau_1 \cap \cdots \cap \tau_n$; notice that for $1 \leq i \leq n$, $\tau \leq_{\mathrm{S}} \tau_i$. For every $1 \leq i \leq n$, there is a strict substitution S such that $S(\mathcal{E}(F)) = \tau_i$. The proof is completed by theorem 10.2.2 (iii).

b) $\tau \in \mathcal{T}_{\mathrm{s}}$. If **r** is a rewrite rule typeable with respect to $\mathcal{E}$, then, by definition 10.1.7 (i), there is a basis B_1, and $\tau \in \mathcal{T}_{\mathrm{s}}$, and an assignment f of types to nodes and edges such that B_1 is essentially used for $Lhs{:}\sigma$ and $B_1 \vdash_{\mathcal{E}} Rhs{:}\sigma$, and in $B_1 \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B_1 \vdash_{\mathcal{E}} Rhs{:}\sigma$ all nodes containing F are typed with $\mathcal{E}(F)$.
We will check the requirements of definition 10.1.7 (i) for $\mathcal{E}[F := E(\mathcal{E}(F))]$.
Let $E(<B_1, \tau>) = <B_2, \sigma'>$, and take the assignment f' of types to nodes and edges such that $f'(\text{node/edge}) = E(f(\text{node/edge}))$. Then:
   1) By part (ii): B_2 is essentially used for $Lhs{:}\sigma'$ and by part (i): $B_2 \vdash_{\mathcal{E}} Rhs{:}\sigma'$.
   2) All occurrences of F in $B_2 \vdash_{\mathcal{E}} Lhs{:}\sigma'$ and $B_2 \vdash_{\mathcal{E}} Rhs{:}\sigma'$ are typed with $E(\mathcal{E}(F))$. □

Notice that in part (iii.a) the relation $\leq_{\mathrm{S}}$ is used, not $\leq_{\mathrm{E}}$.

Lifting is a sound operation on typed term-trees, i.e:

Theorem 10.2.6 *Soundness of lifting*. If $B \vdash_{\mathcal{E}} T{:}\sigma$ and L is a lifting , then $L(B) \vdash_{\mathcal{E}} T{:}L(\sigma)$.

Proof: By induction on $\mathcal{T}_{\mathrm{S}}$. Remember that $L(B) \leq_{\mathrm{E}} B$, and $\sigma \leq_{\mathrm{E}} L(\sigma)$.

i) $\sigma \in \mathcal{T}_{\mathrm{s}}$. This part is proved by induction on the structure of terms.

a) $T \equiv x$. Then $B \leq_{\mathrm{E}} \{x{:}\sigma\}$. Because $L(B) \leq_{\mathrm{E}} B \leq_{\mathrm{E}} \{x{:}\sigma\} \leq_{\mathrm{E}} \{x{:}L(\sigma)\}$, we obtain $L(B) \vdash_{\mathcal{E}} x{:}L(\sigma)$.

b) $T \equiv F(T_1, \ldots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_{\mathrm{S}}$, and a chain C such that for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} T_i{:}\sigma_i$ and $C(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$.
For $1 \leq i \leq n$, take $L_i = <<B, \sigma_i>, <L(B), \sigma_i>>$, then L_i is a lifting and, by induction, $L_i(B) \vdash_{\mathcal{E}} T_i{:}\sigma_i$. Take

$$L_0 = <<\emptyset, \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma>, <\emptyset, \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow L(\sigma)>>,$$

then L_0 is a lifting. Since $C * <L_0>$ is a chain and

$$C * <L_0>(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow L(\sigma),$$

we obtain $L(B) \vdash_{\mathcal{E}} T{:}L(\sigma)$.

ii) $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$. Then for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} T{:}\sigma_i$. By lemma 5.1.3 (i) there are $\tau_1, \ldots, \tau_m \in \mathcal{T}_{\mathrm{s}}$ such that $L(\sigma) = \tau_1 \cap \cdots \cap \tau_m$, and for every $1 \leq j \leq m$ there is a $1 \leq i_j \leq n$ such that $\sigma_{i_j} \leq_{\mathrm{E}} \tau_j$. Take, for $1 \leq j \leq m$, $L_j = <<B, \sigma_{i_j}>, <L(B), \tau_j>>$, which is a lifting, then by induction: $L(B) \vdash_{\mathcal{E}} T{:}\tau_j$. So also $L(B) \vdash_{\mathcal{E}} T{:}L(\sigma)$.

iii) $\sigma = \omega$, then $L(\sigma) = \omega$. Obviously $L(B) \vdash_{\mathcal{E}} T{:}\omega$. □

Obviously, not every lifting performed on a pair $<B, \sigma>$ such that B is essentially used for $T{:}\sigma$ produces a pair with this same property. Since type assignment on rewrite rules is defined using the notion of essentially used bases, it is clear that lifting cannot be a sound operation on rewrite rules. This can also be illustrated by the following:

Take the rewrite system

$$\begin{array}{ll} I\,(x) & \to x \\ F\,(I_0) & \to I_0 \end{array}$$

that is typeable with respect to the environment $\mathcal{E}_1$:

$$\begin{array}{l} \mathcal{E}_1\,(I) \;= 1\to1 \\ \mathcal{E}_1\,(F) = (2\to2)\to3\to3. \end{array}$$

Notice that $(2\to2)\to3\to3 \leq_{\mathrm{E}} (2\to2)\cap4\to3\to3$, so

$$<<\emptyset, (2\to2)\to3\to3>, <\emptyset, (2\to2)\cap4\to3\to3>>$$

is a lifting. It is impossible to show that the rewrite rule that defines F is typeable with respect to $\mathcal{E}[F := (2\to2)\cap4\to3\to3]$, since all types in $(2\to2)\cap4$ should be types for I. However, for rewrite rules of which the patterns consist of term-variables only, it is not difficult to show that lifting is a sound operation, so, therefore, it is not difficult to show that translating lambda terms into terms in ACL preserves types.

Combining the results proved above for the different operations, we have:

Theorem 10.2.7 i) If $B \vdash_{\mathcal{E}} T{:}\sigma$ then for every chain $\mathbf{C}$ such that $\mathbf{C}\,(<B, \sigma>) = <B', \sigma'>$, $B' \vdash_{\mathcal{E}} T{:}\sigma'$.

ii) If B is essentially used for $T{:}\sigma$, and $\mathbf{C}$ is a chain that contains no lifting, then: if $\mathbf{C}\,(<B, \sigma>) = <B', \sigma'>$, then B' is essentially used for $T{:}\sigma'$.

iii) Let **r**: $Lhs \to Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of **r**. If $\mathbf{C}$ is a chain that contains no lifting, then: if $\mathbf{C}\,(\mathcal{E}\,(F)) = \tau \in \mathcal{T}_{\mathrm{S}}$, then for every $\mu \in \mathcal{T}_{\mathrm{S}}$ such that $\tau \leq_{\mathrm{S}} \mu$, **r** is typeable with respect to $\mathcal{E}[F := \mu]$.

Proof: By theorems 10.2.2, 10.2.5, and 10.2.6. □

10.3 *The loss of the subject reduction property*

If a term T is rewritten to the term T' using the rewrite rule $Lhs \to Rhs$, there is a subterm T_0 of T, and a replacement R, such that $Lhs^{\mathrm{R}} = T_0$; T' is obtained by replacing T_0 by Rhs^{R}. The subject reduction property for this notion of reduction is:

$$\text{If } B \vdash_{\mathcal{E}} T{:}\sigma, \text{ and } T \text{ can be rewritten to } T', \text{ then } B \vdash_{\mathcal{E}} T'{:}\sigma.$$

This is, of course, an important property of reduction systems. To guarantee the subject reduction property, we should accept only those rewrite rules $Lhs \to Rhs$, that satisfy:

$$\text{For all replacements R, bases } B \text{ and types } \sigma\text{: if } B \vdash_{\mathcal{E}} Lhs^{\mathrm{R}}{:}\sigma, \text{ then } B \vdash_{\mathcal{E}} Rhs^{\mathrm{R}}{:}\sigma,$$

because only then are we sure that all possible rewrites are safe.

Definitions 10.1.1, 10.1.2 and 10.1.7 defined what a type assignment should be, just using the strategy as used in languages like, for example, Miranda. Unfortunately, it will not suffice to guarantee the subject reduction property. Even typeability cannot be kept under rewriting.

Take, for example, the definition of H as in example 9.3.8, and the following environment $\mathcal{E}_0$:

$$\begin{array}{l} \mathcal{E}_0\,(H) = ((1\to2)\to3)\to(1\to2)\to2 \\ \mathcal{E}_0\,(S) \;= (1\to2\to3)\to(1\to2)\to1\to3 \\ \mathcal{E}_0\,(I) \;\;= 1\to1 \end{array}$$

The rule that defines H is typeable with respect to $\mathcal{E}_0$:

$$\begin{array}{c} \downarrow (1\to2)\to2 \\ H{:}((1\to2)\to3)\to(1\to2)\to2 \\ \downarrow \\ S_2{:}((1\to2)\to1\to3)\to((1\to2)\to1)\to(1\to2)\to3 \\ \swarrow \qquad \searrow \\ x{:}(1\to2)\to1\to3 \qquad y{:}(1\to2)\to1 \end{array} \quad \to \quad \begin{array}{c} \downarrow (1\to2)\to2 \\ S_2{:}((1\to2)\to1\to2)\to((1\to2)\to1)\to(1\to2)\to2 \\ \swarrow \qquad \searrow \\ I_0{:}(1\to2)\to1\to2 \qquad y{:}(1\to2)\to1 \end{array}$$

If we take the term $H\,(S_2\,(K_0, I_0))$ then it is easy to see that the rewrite is allowed and that this term will be rewritten to $S_2\,(I_0, I_0)$.

Although the first term is typeable with respect to $\mathcal{E}_0$ in the following way:

$$\begin{array}{c} \downarrow (4\to5)\to5 \\ H{:}((4\to5)\to4\to5)\to(4\to5)\to5 \\ \downarrow \\ S_2{:}((4\to5)\to(4\to5)\to4\to5)\to((4\to5)\to4\to5)\to(4\to5)\to4\to5 \\ \swarrow \qquad \searrow \\ K_0{:}(4\to5)\to(4\to5)\to4\to5 \qquad I_0{:}(4\to5)\to4\to5 \end{array}$$

the term $S_2\,(I_0, I_0)$ is not typeable with respect to $\mathcal{E}_0$ with the type $(4\to5)\to5$. In fact, it is not typeable at all with respect to $\mathcal{E}_0$.

We emphasize that the loss of the subject reduction property is not caused by the fact that we use intersection types. The environment $\mathcal{E}_0$ maps function symbols to Curry-types, so even for a notion of type assignment based on Curry-types (as presented in the next chapter) types are not preserved under rewriting.

By the construction of this example, the impression could arise that the problem is caused by the fact that an environment is allowed to select more complicated types than needed to type the rewrite rules. This, however, is not true.

Take the rewrite system:

$$\begin{array}{ll} I\,(x) & \to x \\ K\,(x,y) & \to x \\ Z\,(x,y) & \to y \\ F\,(I_0) & \to I_0 \\ G\,(Z_1\,(x)) & \to F\,(x) \end{array}$$

that is typeable with respect to the environment $\mathcal{E}_1$:

$$\begin{array}{l} \mathcal{E}_1\,(I) \;= 1\to1 \\ \mathcal{E}_1\,(K) = 2\to\omega\to2 \end{array}$$

$\mathcal{E}_1(Z) = \omega \rightarrow 3 \rightarrow 3$
$\mathcal{E}_1(F) = (4 \rightarrow 4) \rightarrow 5 \rightarrow 5$
$\mathcal{E}_1(G) = (6 \rightarrow 6) \rightarrow 7 \rightarrow 7.$

Notice that $\mathcal{E}_1$ is more or less minimal for the rewrite system. If we take $T \equiv G(Z_1(K_0))$, then it is easy to check that T is typeable with respect to $\mathcal{E}_1$, that the rewrite rule that defines G matches T, but that $F(K_0)$ is not typeable with respect to $\mathcal{E}_1$.

In the next two chapters, we will discuss two restrictions (variants) of the notion of type assignment of the current chapter for which we can formulate a decidable and sufficient condition that rewrite rules should satisfy, in order to reach the subject reduction property. The first, presented in chapter eleven, will restrict the set of types to the Curry types (extended with type constants); for this system we will even prove that the condition is necessary. The second, in chapter twelve, will limit the possible types to intersection types of Rank 2.

The construction of these conditions will be made using a notion of principal pairs; the condition a rewrite rule should satisfy is that the principal pair for the left hand side term is also a pair for the right hand side term. For the notion of type assignment defined in this chapter, we are not able to formulate this condition in a constructive way, since it is not clear how we should define *the* principal pair for a term. Remember that, for the notion of essential type assignment in the lambda calculus, the principal pair for a lambda term was defined using the set of principal pairs for its approximants. There is no notion of approximants for terms in $\mathrm{T}(\mathcal{F}, \mathcal{X})$. This problem will be overcome in the next two chapters by defining a most general unification algorithm for types, and defining principal pairs using that algorithm, as was done for Curry's system and the Rank 2 system. At this moment there is no general unification algorithm for types in $\mathcal{T}_S$ which works well on all types, so we cannot take this approach.

For the notion of type assignment as defined in this chapter, the only result we can obtain is to show that *if* a left hand side of a rewrite rule has a principal pair and, using that pair, the rewrite rule can be typed, then rewriting using this rule is safe with respect to subject reduction.

Definition 10.3.1 i) Let $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$. A pair $<P, \pi>$ is called *an essential principal pair for T with respect to $\mathcal{E}$*, if $P \vdash_{\mathcal{E}} T{:}\pi$, and, for every B, and σ such that $B \vdash_{\mathcal{E}} T{:}\sigma$, there is a chain *C* such that $C(<P, \pi>) = <B, \sigma>$.

ii) The definition of a *safe type assignment with respect to $\mathcal{E}$* is the same as the one for a type assignment as defined in definition 10.1.7, by replacing condition (i.a) by:

$<B, \sigma>$ is an essential principal pair for Lhs with respect to $\mathcal{E}$, and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$.

Then rewrite rule $Lhs \rightarrow Rhs$ is called a *safe rewrite rule*.

Notice that we will not show that every typeable term *has* a principal pair with respect to $\mathcal{E}$; at the moment we cannot give a construction of such a pair for every term. But, even with this non-constructive approach we can show that the condition is sufficient.

First we will prove some preliminary results.

Lemma 10.3.2 Let $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$.

i) If $B \vdash_{\mathcal{E}} T{:}\sigma$, and R is a replacement and B' a basis such that for every statement $x{:}\rho \in B$ $B' \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho$, then $B' \vdash_{\mathcal{E}} T^{\mathrm{R}}{:}\sigma$.

ii) If, for replacement R, there are B and σ such that $B \vdash_{\mathcal{E}} T^{\mathrm{R}}{:}\sigma$, then, for every x occurring in T, there is a type ρ_x such that $\{\, x{:}\rho_x \mid x \text{ occurs in } T \,\} \vdash_{\mathcal{E}} T{:}\sigma$, and $B \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho_x$.

Proof: By induction on the structure of T.

i) We will show only the part $\sigma \in \mathcal{T}_S$.

a) $T \equiv x$. Trivial.

b) $T \equiv F(T_1, \ldots, T_n)$. If $B \vdash_{\mathcal{E}} F(T_1, \ldots, T_n){:}\sigma$, then there are $\sigma_1, \ldots, \sigma_n$, and a chain *C* such that

$$C(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma, \text{ and for every } 1 \leq i \leq n \; B \vdash_{\mathcal{E}} T_i{:}\sigma_i.$$

By induction for every $1 \leq i \leq n$, $B' \vdash_{\mathcal{E}} T_i^{\mathrm{R}}{:}\sigma_i$. By definition 9.3.2 (i)

$$F(T_1{}^{\mathrm{R}}, \ldots, T_1{}^{\mathrm{R}}) = F(T_1, \ldots, T_n)^{\mathrm{R}}.$$

So also $B' \vdash_{\mathcal{E}} F(T_1, \ldots, T_n)^{\mathrm{R}}{:}\sigma$.

ii) Trivial. □

In the following theorem, we prove that our solution is sufficient.

Theorem 10.3.3 *The condition is sufficient.* Let **r** : $Lhs \rightarrow Rhs$ be a safe rewrite rule. Then, for every replacement R , basis B and a type μ: if $B \vdash_{\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$, then $B \vdash_{\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$.

Proof: Since **r** is safe, there are P, and π such that $<P, \pi>$ is a principal pair for Lhs with respect to $\mathcal{E}$, and $P \vdash_{\mathcal{E}} Rhs{:}\pi$. Suppose R is a replacement such that there are basis B and type μ such that $B \vdash_{\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$. By lemma 10.3.2 (ii) there is a B' such that for every $x{:}\rho \in B'$, $B \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho$, and $B' \vdash_{\mathcal{E}} Lhs{:}\mu$. Since $<P, \pi>$ is a principal pair for Lhs with respect to $\mathcal{E}$, then, by definition 10.3.1 (i) there is a chain *C* such that $C(<P, \pi>) = <B', \mu>$. Since $P \vdash_{\mathcal{E}} Rhs{:}\pi$, then, by theorem 10.2.7 (i) we obtain $B' \vdash_{\mathcal{E}} Rhs{:}\mu$. Then, by lemma 10.3.2 (i), $B \vdash_{\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$. □

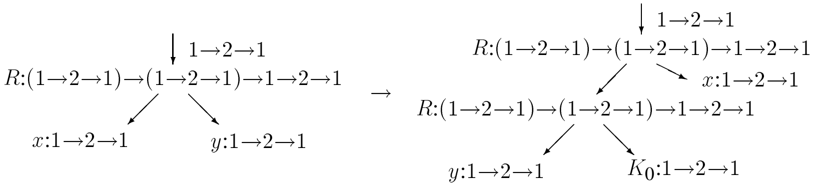
10.4 *About recursion*

Recall the two Exp-terms mentioned in section 3.3. The counterpart of the last of these two (that was presented to show a crucial difference between Milner's and Mycroft's notions of type assignment) in the applicative term rewriting world is the rule

$$R(x,y) \;\to\; R(R(y,K_0),x),$$

which can be typed using the environment

$\mathcal{E}(R) = (1\to2\to1)\to(1\to2\to1)\to1\to2\to1$
$\mathcal{E}(K) = 3\to4\to3.$

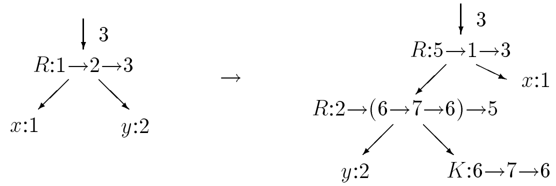


The main (and only) difference between Milner's type assignment system as used in this chapter and Mycroft's system lies in the way recursion is dealt with. If we had based our notion of type assignment on Mycroft's type assignment system, we would have required that *only* the occurrence of the defined symbol F of a rule *in the defining node* is typed with $\mathcal{E}(F)$. The rule given above is of course typeable in a type assignment system based on Mycroft's system, but in such a system it is even possible to use smaller types.

Take the environment

$\mathcal{E}(R) = 1\to2\to3$
$\mathcal{E}(K) = 4\to5\to4.$

In Mycroft's approach, the types assigned to the nodes containing R in the right hand side need only be 'instances' of the type $1\to2\to3$.



However, one important property is lost when Mycroft's approach is used: not every substitution instance of the type $1\to2\to3$ can be proved to be a correct type for R. (This is related to the problem discussed in section 3.3.) For example, when using $\mathcal{E}(R) = 4\to4\to5$, this rule cannot be typed. If first and second argument of R should have the same type, then the only types possible for R are the substitution instances of $(1\to2\to1)\to(1\to2\to1)\to1\to2\to1$. However, by definition 10.1.2 (i.c) it is possible that R occurs in a term with type $4\to4\to5$ since, using Mycroft's system, all occurrences of the defined symbol of a rule other than the occurrence in the defining node can be typed by a type that is an instance of the type assigned to the defining node, no matter where they appear. This implies that, although all types that can be obtained from the environment type for a defined symbol - by applying an operation

- are allowed for occurrences of this defined symbol, we cannot prove in a system based on Mycroft's approach that these types are correct for the rewrite rule.

Mycroft's type assignment system for Exp has the important property that Curry-substitution is a sound operation, both for terms and functions. When using Mycroft's system for term rewriting systems, we would want to obtain the same property. It is possible to prove this for typeable terms, but, for every defined symbol F and for every Curry-substitution S we should prove that the type $\mathcal{E}(F)$ in the environment can be replaced by $S(\mathcal{E}(F))$. Notice that this does not hold, as was illustrated above.

This, in fact, means that the Mycroft approach is not very well suited for type assignment in term rewriting systems.

But, even when using the notion of type assignment based on Milner's solution, changing the environment can affect typeability of rewrite systems. Take, for example, the rewrite system

$I(x) \;\to x$
$K(x,y) \;\to x$

that is typeable with respect to the environment $\mathcal{E}_2$:

$\mathcal{E}_2(I) = 1\to1$
$\mathcal{E}_2(K) = 2\to3\to2.$

Then we have $\vdash_{\mathcal{E}_2} I(K_0){:}4\to5\to4$. But this term is not typeable with respect to the environment $\mathcal{E}_2[I := (1\to1)\to1\to1]$. So, the only thing we can say about this apparent drawback of Mycroft's approach is that the loss of typeability becomes apparent *within* rewrite rules.

We can also remark that Mycroft's system avoids a complicated definition of 'defined symbol', and in such a system there is no need to assume that rewrite rules are not mutually recursive.

Like in the language Exp, it is possible to produce rewrite rules that are typeable when using Mycroft's system, and that are not typeable using Milner's. As an example for this, take the following rewrite rule:

$$G(x) \;\to\; K(G(I_0), G(K_0)).$$

This rule corresponds to the other Exp-term given in subsection 3.3. This rule can be typed using Mycroft's system with $\mathcal{E}(G) = 1\to2$, and is not typeable using Milner's system, because the types needed for G in the right hand side can never be the same.

The price we are paying for this freedom is that it is in general no longer possible to check the types for defined symbols, other than the type provided by the environment. This is of course a great disadvantage, but since it is used in the functional programming language Miranda (that basically uses Curry types), we will use it also in chapter eleven. In chapter twelve we will, again, present a system based on Milner's way of dealing with recursion, using Rank 2 intersection types.

Chapter 11 *Partial Curry Type Assignment in Left Linear Applicative Term Rewriting Systems*

In this chapter we will present a formal notion of type assignment on Left Linear Applicative Term Rewriting Systems, which is based on the extension defined by Mycroft of Curry's type assignment system. Since we use Mycroft's approach, the system of this chapter is not a restriction of the type assignment system of the previous chapter.

We will show that Curry-substitution is a sound operation: for every basis B, term T, type σ and Curry-substitution *S*: if $B \vdash_{C_\mathcal{E}} T{:}\sigma$, then also $S(B) \vdash_{C_\mathcal{E}} T{:}S(\sigma)$. We will show that, for every typeable term T, there is a principal pair $<P, \pi>$ for T; i.e. $P \vdash_{C_\mathcal{E}} T{:}\pi$, and for every pair $<B, \sigma>$ such that $B \vdash_{C_\mathcal{E}} T{:}\sigma$ there is a Curry-substitution *S* such that $S(<P, \pi>) = <B, \sigma>$.

We will formulate a condition that typeable rewrite rules should satisfy in order to guarantee the subject reduction property. This condition could be used to prove that all rewrite rules that can be defined in a language like Miranda are safe in this respect. Another result presented is that, if $<P, \pi>$ is the principal pair for T, T' is obtained from T by replacing (in a consistent way) term-variables by terms, and T' is typeable by τ, then there is a Curry-substitution *S* such that: $S(\pi) = \tau$, and for every $x{:}\rho$ occurring in P the replacement of x is typeable by $S(\rho)$. Using this result we will prove the formulated condition to be necessary and sufficient.

The most important difference between the notion of type assignment we will introduce in this chapter and Curry's type assignment system is the following: In Curry's system, a basis is defined as a mapping from term-variables to types, or, equivalently, as a set of statements with distinct term-variables as subjects. However, the bases allowed in the system we will present in this chapter can contain several different statements for the same term-variable. So, in the system we present, it is possible to assign a type to the term $Ap(x, x)$. This in fact corresponds to the definition of ML-type assignment as given in [Damas '85], and it was used there for tackling the let-construct. Unlike in lambda calculus, in term rewriting systems this causes no difficulties, since there is no notion of 'abstraction' in this world.

This more general definition of bases will force the restriction to the Left Linear Applicative Term Rewriting Systems (LLATRS) when defining Curry type assignment on term rewrite systems. In term rewriting systems, a term-variable x that occurs in the left hand side of a rewrite rule can be seen as the binding occurrence of x, binding the occurrences of x in the right hand side. However, in general x can occur more than once in the left hand side, making the notion of *the* binding occurrence obscure. In this chapter we will consider left linear rewriting systems, that contain only rewrite rules for which the left hand side is linear (term-variables occur only once), because for these rules the binding occurrence of a term-variable is unique. This is required because, with the used definition of bases, different occurrences can be typed with different types, and allowing of more than one type for a term variable in a rewrite rule could obviously give a notion of type assignment that is more an intersection approach than a Curry one: we will avoid this discrepancy by limiting the rewrite rules to the left linear ones.

The left linearity of rewrite rules will play a role in the proof of theorem 11.3.5 (i).

11.1 *Curry type assignment in LLATRS's*

In this section we will present a notion of partial type assignment on LLATRS's, based on the Mycroft type assignment system, which can also be seen as a generalization of Milner's type assignment system. Using this system, the only requirement for recursive definitions will be that the separate occurrences of the defined symbol within the rewrite rule (other than the occurrence in the defining node) are typed with types that are substitution instances of the type that is provided by the environment for the defined symbol.

The type system we will define in this section is based on the Curry type system, extended with type constants; we will in fact use ML-types but, nonetheless, we will use the symbol $\mathcal{T}_C$. Type constants will play a role in the proof of theorem 11.3.5 (i).

Definition 11.1.1 i) A *statement* is an expression of the form $T{:}\sigma$, where $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ and $\sigma \in \mathcal{T}_C$. T is the *subject* and σ the *predicate* of $T{:}\sigma$.

ii) A *basis* B is a set of statements with term-variables (not necessarily distinct) as subjects.

Curry type assignment on a LLATRS (Σ, R) will be defined as the labelling of nodes and edges in the tree-representation of terms and rewrite rules with Curry-types. In this labelling, we will use that there is an environment that provides a type in $\mathcal{T}_C$ for every $F \in \mathcal{F} \cup \{Ap\}$.

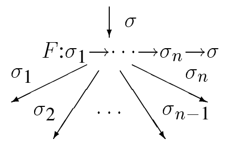
Definition 11.1.2 Let $(\Sigma, \mathbf{R})$ be a LLATRS. A mapping $\mathcal{E} : \mathcal{F} \cup \{Ap\} \rightarrow \mathcal{T}_C$ is called a *Curry-environment* if $\mathcal{E}(Ap) = (1\rightarrow 2)\rightarrow 1\rightarrow 2$, and, for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$.

Definition 11.1.3 Let $(\Sigma, \mathbf{R})$ be a LLATRS, and $\mathcal{E}$ a Curry-environment.

i) We will say that $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ *is Curry typeable by* $\sigma \in \mathcal{T}_C$ *with respect to* $\mathcal{E}$, if there exists an assignment of Curry-types to edges and nodes that satisfies the following constraints:

a) The root edge of T is typed with σ.

b) If a node contains a symbol $F \in \mathcal{F} \cup \{Ap\}$ that has arity n $(n \geq 0)$, then there are $\sigma_1, \ldots, \sigma_n$ and σ, such that this node is typed with $\sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\sigma$, the n out-going edges are from left to right typed with σ_1 upto σ_n, and the in-going edge is typed with σ.



c) If a node containing a symbol $F \in \mathcal{F} \cup \{Ap\}$ is typed with σ, then there is a Curry-substitution *S* such that $S(\mathcal{E}(F)) = \sigma$.

ii) Let $T \in \mathbf{T}(\mathcal{F}, \mathcal{X})$ be Curry typeable by σ with respect to $\mathcal{E}$. If B is a basis containing all statements with variables as subjects that appear in the typed term-tree for $T{:}\sigma$, we will write $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$.

Notice that if $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$, then B can contain more statements than needed to obtain $T{:}\sigma$.

The following theorem will show that substitution is a sound operation on term-trees. As illustrated in section 10.4, we cannot show such a result for rewrite rules.

Theorem 11.1.4 *Soundness of Curry-substitution.* If $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$, then for every Curry-substitution S: $S(B) \vdash_{\mathrm{C}_\mathcal{E}} T{:}S(\sigma)$.

Proof: By induction on the structure of T.

i) $T \equiv x$. If $B \vdash_{\mathrm{C}_\mathcal{E}} x{:}\sigma$, then $x{:}\sigma \in B$ and $x{:}S(\sigma) \in S(B)$, so $S(B) \vdash_{\mathrm{C}_\mathcal{E}} x{:}S(\sigma)$.

ii) $T \equiv F(T_1, \ldots, T_n)$. If $B \vdash_{\mathrm{C}_\mathcal{E}} F(T_1, \ldots, T_n){:}\sigma$, then are $\sigma_1, \ldots, \sigma_n$, such that for every $1 \leq i \leq n$, $B \vdash_{\mathrm{C}_\mathcal{E}} T_i{:}\sigma_i$, and there is a substitution S' such that $S'(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$. By induction for every $1 \leq i \leq n$, $S(B) \vdash_{\mathrm{C}_\mathcal{E}} T_i{:}S(\sigma_i)$. Since $S \circ S'$ is a substitution, and $S \circ S'(\mathcal{E}(F)) = S(\sigma_1) \rightarrow \cdots \rightarrow S(\sigma_n) \rightarrow S(\sigma)$, we obtain $S(B) \vdash_{\mathrm{C}_\mathcal{E}} F(T_1, \ldots, T_n){:}S(\sigma)$. □

As in chapter ten, in defining type assignment on rewrite rules we will restrict the possible bases for rewrite rules to those that contain nothing but the statements needed to type the left hand side. Therefore, we will define the notion of Curry used bases.

Definition 11.1.5 i) The *Curry used bases of* $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$ are inductively defined by:

a) $T \equiv x$. Take $\{x{:}\sigma\}$.

b) $T \equiv F(T_1, \ldots, T_n)$. There are $\sigma_1, \ldots, \sigma_n$ such that for every $1 \leq i \leq n$, $B \vdash_{\mathrm{C}_\mathcal{E}} T_i{:}\sigma_i$. Let, for $1 \leq i \leq n$, B_i be a Curry used basis of $B \vdash_{\mathrm{C}_\mathcal{E}} T_i{:}\sigma_i$. Take $B_1 \cup \cdots \cup B_n$.

ii) A basis B *is Curry used for* $T{:}\sigma$ *with respect to* $\mathcal{E}$, if and only if there is a basis B' such that $B' \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$ and B is a Curry used basis of $B' \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$.

We will say 'B is Curry used for $T{:}\sigma$' instead of 'B is Curry used for $T{:}\sigma$ with respect to $\mathcal{E}$'. As before, a Curry used basis for a statement $T{:}\sigma$ is not unique, but, again, the results of this chapter do not depend on the actual structure of such a basis, only on its existence.

For Curry used bases, the following properties hold.

Lemma 11.1.6 i) If B is Curry used for $T{:}\sigma$, then $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$.

ii) $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma \Leftrightarrow \exists\, B' \,[\, B' \subseteq B \;\&\; B'$ is Curry used for $T{:}\sigma$].

Proof: By induction on definition 11.1.5. □

Thanks to the notion of Curry used basis, we can give a clear definition of a typeable rewrite rule and a typeable rewrite system. The notion of 'Curry used basis' in this definition will play the same role as the notion of 'essentially used basis' in definition 10.1.7 (i.a).

Definition 11.1.7 Let $(\Sigma, \mathbf{R})$ be a LLATRS, and $\mathcal{E}$ a Curry-environment.

i) We say that $Lhs \rightarrow Rhs \in \mathbf{R}$ with defined symbol F *is naively Curry typeable with respect to* $\mathcal{E}$, if the following constraints hold:

a) There are $\sigma \in \mathcal{T}_{\mathrm{C}}$ and basis B such that B is Curry used for $Lhs{:}\sigma$, and $B \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}\sigma$.

b) The defining node of **r**, containing F, is typed with $\mathcal{E}(F)$.

ii) We say that $(\Sigma, \mathbf{R})$ *is naively Curry typeable with respect to* $\mathcal{E}$, if, for every $\mathbf{r} \in \mathrm{R}$, **r** is naively Curry typeable with respect to $\mathcal{E}$.

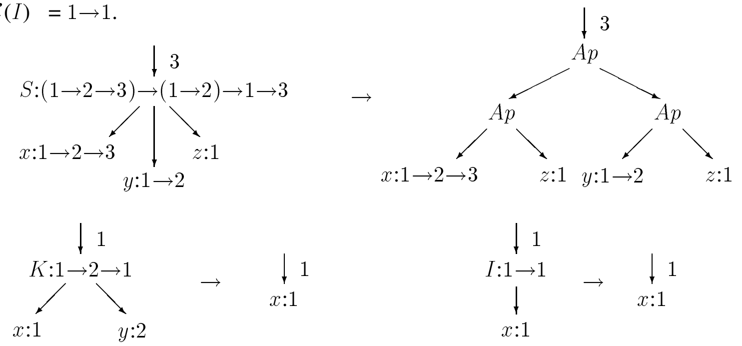
Notice that, by the formulation of part (i.b), type assignment on rewrite rules in this chapter is defined using Mycroft's approach.

As illustrated before definition 10.1.7, in order to avoid a conflict when modelling lambda abstraction, a natural thing to ask of a well-typed rewrite rule would be that all types assigned to term-variables in the right hand side have already been used in the left hand side. Condition (i.a) of definition 11.1.7 covers this problem. Notice that, since only left linear rewrite rules are considered in this chapter, by part (i.a) all nodes within a typeable rewrite rule **r** containing the same term-variable x are typed with the same type.

In the rest of this chapter we will say 'typeable', instead of saying '(naively) Curry typeable with respect to $\mathcal{E}$'.

Example 11.1.8 Typed variants of some of the rewrite rules given in example 9.4.2. We have only inserted those types that are not immediately clear. Notice that we assumed that

$\mathcal{E}(S) = (1\rightarrow2\rightarrow3)\rightarrow(1\rightarrow2)\rightarrow1\rightarrow3$
$\mathcal{E}(K) = 1\rightarrow2\rightarrow1$
$\mathcal{E}(I) = 1\rightarrow1$.



11.2 *The principal pair for a term*

In this section we will define the principal pair for a typeable term T with respect to $\mathcal{E}$, by defining the notion $PP_{\mathrm{C}_\mathcal{E}}(T)$ using Robinson's unification algorithm *unify*$_{\mathrm{R}}$. In the following we will show that, for every typeable term, this is a legal pair and it is indeed the most general one possible.

Definition 11.2.1 We define for every term T the Curry principal pair with respect to the environment $\mathcal{E}$, by inductively defining the notion $PP_{\mathrm{C}_\mathcal{E}}(T) = \langle P, \pi \rangle$ by:

i) For all x, φ: $PP_{\mathrm{C}_{\mathcal{E}}}(x) = \langle\{x{:}\varphi\}, \varphi\rangle$.

ii) If for every $1 \le i \le n$, $PP_{\mathrm{C}_{\mathcal{E}}}(T_i) = \langle P_i, \pi_i\rangle$ (we choose, when necessary, trivial variants such that the $\langle P_i, \pi_i\rangle$ are disjoint in pairs and share no type-variables with $\mathcal{E}(F)$), and

$$S = \mathit{unify}_{\mathrm{R}}(\mathcal{E}(F), \pi_1 \rightarrow \cdots \rightarrow \pi_n \rightarrow \varphi),$$

where φ does not occur in any of the pairs $\langle P_i, \pi_i\rangle$, neither in $\mathcal{E}(F)$, then:

$$PP_{\mathrm{C}_{\mathcal{E}}}(F(T_1, \ldots, T_n)) = S(\langle P_1 \cup \cdots \cup P_n, \varphi\rangle).$$

By induction on the definition of $PP_{\mathrm{C}_{\mathcal{E}}}(T)$, using theorem 11.1.4, it is easy to verify that $PP_{\mathrm{C}_{\mathcal{E}}}(T) = \langle P, \pi\rangle$ implies $P \vdash_{\mathrm{C}_{\mathcal{E}}} T{:}\pi$.

In the following theorem we will show that the operation of substitution is complete.

Theorem 11.2.2 *Completeness of Curry-substitution.* If $B \vdash_{\mathrm{C}_{\mathcal{E}}} T{:}\sigma$, then there are P, π, and a Curry-substitution S such that: $PP_{\mathrm{C}_{\mathcal{E}}}(T) = \langle P, \pi\rangle$, and $S(P) \subseteq B$, $S(\pi) = \sigma$.

Proof: By induction on the structure of T.

i) $T \equiv x$. Then $\{x{:}\sigma\} \subseteq B$. Then $PP_{\mathrm{C}_{\mathcal{E}}}(x) = \langle\{x{:}\varphi\}, \varphi\rangle$. Take $S = (\varphi \mapsto \sigma)$.

ii) $T \equiv F(T_1, \ldots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_n$, and a Curry-substitution S_0 such that for every $1 \le i \le n$, $B \vdash_{\mathrm{C}_{\mathcal{E}}} T_i{:}\sigma_i$, and $S_0(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$. By induction for every $1 \le i \le n$, there are $\langle P_i, \pi_i\rangle$ (disjoint in pairs) and a Curry-substitution S_i, such that

$$S_i(P_i) \subseteq B,\ S_i(\pi_i) = \sigma_i, \text{ and } PP_{\mathrm{C}_{\mathcal{E}}}(T_i) = \langle P_i, \pi_i\rangle.$$

Assume, without loss of generality, that none of the type-variables of the type $\mathcal{E}(F)$ occur in any pair $\langle P_i, \pi_i\rangle$. Let φ be a type-variable not occurring in any other type. Take $S' = S_n \circ \cdots \circ S_0 \circ (\varphi \mapsto \sigma)$, then:

$$S'(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma,\ S'(P_1 \cup \cdots \cup P_n) \subseteq B,\ S'(\varphi) = \sigma,$$
$$\text{and, for every } 1 \le i \le n,\ S'(\pi_i) = \sigma_i.$$

Since $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$ is a common instance of both $\mathcal{E}(F)$ and $\pi_1 \rightarrow \cdots \rightarrow \pi_n \rightarrow \varphi$, by property 1.6 there are Curry-substitutions S_g and S such that:

$$S_g = \mathit{unify}_{\mathrm{R}}(\mathcal{E}(F), \pi_1 \rightarrow \cdots \rightarrow \pi_n \rightarrow \varphi) \text{ and } S' = S \circ S_g.$$

By definition 11.2.1 (ii), $PP_{\mathrm{C}_{\mathcal{E}}}(F(T_1, \ldots, T_n)) = S_g(\langle P_1 \cup \cdots \cup P_n, \varphi\rangle)$.
Also, $S(S_g(P_1 \cup \cdots \cup P_n)) \subseteq B$, and $S(S_g(\varphi)) = \sigma$. □

11.3 *A necessary and sufficient condition for subject reduction*

In this section we will formulate a condition typeable rewrite rules should satisfy in order to obtain the subject reduction property. We will show that this condition is necessary and sufficient.

Definition 11.3.1 i) We call a rewrite rule $Lhs \rightarrow Rhs$ *safe* if:

If $PP_{\mathrm{C}_{\mathcal{E}}}(Lhs) = \langle P, \pi\rangle$, then $P \vdash_{\mathrm{C}_{\mathcal{E}}} Rhs{:}\pi$.

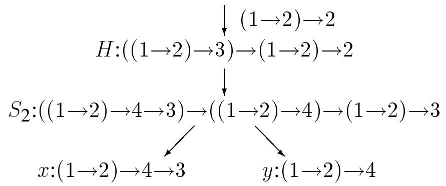
ii) The definition of a *safe Curry type assignment with respect to* $\mathcal{E}$ is the same as the one for a naive Curry type assignment, by replacing in definition 11.1.7 condition (i.a) by:

If $PP_{\mathrm{C}_{\mathcal{E}}}(Lhs) = \langle P, \pi\rangle$, then $P \vdash_{\mathrm{C}_{\mathcal{E}}} Rhs{:}\pi$.

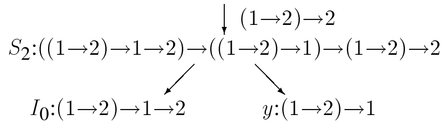
Notice that the notion $PP_{\mathrm{C}_{\mathcal{E}}}(T)$ is defined independently from the definition of typeable rewrite rules. Moreover, since type assignment as defined in this chapter is decidable, this safeness constraint on rewrite rules is decidable.

Example 11.3.2 As an example of a rule that is not safe, take the typed rewrite rule that defined H as in section 10.3: the types assigned to the nodes containing x and y are not the most general ones needed to find the type for the left hand side of the rewrite rule.

To obtain $PP_{\mathrm{C}_{\mathcal{E}}}(H(S_2(x, y)))$, we will assign types to nodes in the term-tree in the following way:



If $S_2(I_0, y))$ were to be typed with $(1\rightarrow2)\rightarrow2$, the node containing y must be typed with $(1\rightarrow2)\rightarrow1$.



Since types assigned to term-variables should be the same in left- and right hand side, we should replace the type-variable 4 by 1, so in the typed rewrite rule the most general pair for the left hand side is no longer used. In other words, this rule is not safe and should therefore be rejected.

Before we will come to the proof that the condition imposed on typeable rewrite rules as defined in definition 11.3.1 is necessary and sufficient, we will prove some preliminary results.

In the following lemma we will show that, if F is the defined symbol of a rewrite rule, then the type $\mathcal{E}(F)$ dictates not only the type for the left and right hand side of that rule, but also the principal type for the left hand side.

Lemma 11.3.3 If F is the defined symbol of the typeable rewrite rule $\mathbf{r} : Lhs \rightarrow Rhs$, then there are P, B, $\sigma_1, \ldots, \sigma_n$, and σ such that

$$\mathcal{E}(F) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma,\ PP_{\mathrm{C}_{\mathcal{E}}}(Lhs) = \langle P, \sigma\rangle,\ B \vdash_{\mathrm{C}_{\mathcal{E}}} Lhs{:}\sigma \text{ and } B \vdash_{\mathrm{C}_{\mathcal{E}}} Rhs{:}\sigma.$$

Proof: Since $\mathbf{r}$ is typeable, there are B and σ such that $B \vdash_{\mathrm{C}_{\mathcal{E}}} Lhs{:}\sigma$, and $B \vdash_{\mathrm{C}_{\mathcal{E}}} Rhs{:}\sigma$. By theorem 11.2.2 there are P, π, and S, such that $PP_{\mathrm{C}_{\mathcal{E}}}(Lhs) = \langle P, \pi\rangle$, $S(P) \subseteq B$, $S(\pi) = \sigma$.

If F is the defined symbol of $\mathbf{r}$, then, by proposition 9.3.4, there are $n \ge j \ge 0$ such that F has arity j and $Lhs = Ap(\cdots Ap(F(T_1, \ldots, T_j), T_{j+1}) \cdots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_j, \mu$ such that F is typed with $\sigma_1 \rightarrow \cdots \rightarrow \sigma_j \rightarrow \mu$ in $B \vdash_{\mathrm{C}_{\mathcal{E}}} Lhs{:}\sigma$.

For every Ap there are α, β such that Ap is typed with $(\alpha\rightarrow\beta)\rightarrow\alpha\rightarrow\beta$; therefore, there are $\sigma_{j+1}, \ldots, \sigma_n$, such that $\mu = \sigma_{j+1} \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$. But then F is typed with $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$ in

$B \vdash_{\mathrm{C}_\mathcal{E}} Lhs{:}\sigma$. Since F is the defined symbol of $\mathbf{r}$, $\mathcal{E}(F) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$.

Likewise, there are $\tau_1, \ldots, \tau_n$ such that F is typed with $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \pi$ in $P \vdash_{\mathrm{C}_\mathcal{E}} Lhs{:}\pi$. Since $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \pi$ is a substitution instance of $\mathcal{E}(F)$, and σ is a substitution instance of π, $\sigma = \pi$. □

The following lemma will formulate the relation between replacements performed on a term, and possible type assignments for that term.

Lemma 11.3.4 i) If $PP_{\mathrm{C}_\mathcal{E}}(T) = \langle P, \pi \rangle$, and for replacement R there are B and σ such that $B \vdash_{\mathrm{C}_\mathcal{E}} T^{\mathrm{R}}{:}\sigma$, then there is a Curry-substitution S such that $S(\pi) = \sigma$, and for every statement $x{:}\rho \in P$ $B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S(\rho)$.

ii) If $B \vdash_{\mathrm{C}_\mathcal{E}} T{:}\sigma$, R is a replacement, and B' a basis such that for every statement $x{:}\rho \in B$ $B' \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}\rho$, then $B' \vdash_{\mathrm{C}_\mathcal{E}} T^{\mathrm{R}}{:}\sigma$.

Proof: By induction on the structure of T.

i) a) $T \equiv x$. Then $P = \{x{:}\varphi\}$, and $\pi = \varphi$. Take $S = (\varphi \mapsto \sigma)$.

b) $T \equiv F(T_1, \ldots, T_n)$. If $B \vdash_{\mathrm{C}_\mathcal{E}} F(T_1, \ldots, T_n)^{\mathrm{R}}{:}\sigma$, then $B \vdash_{\mathrm{C}_\mathcal{E}} F(T_1{}^{\mathrm{R}}, \ldots, T_1{}^{\mathrm{R}}){:}\sigma$, and there are $\sigma_1, \ldots, \sigma_n$, and a Curry-substitution S_0 such that

for every $1 \leq i \leq n$, $B \vdash_{\mathrm{C}_\mathcal{E}} T_i^{\mathrm{R}}{:}\sigma_i$, and $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma = S_0(\mathcal{E}(F))$.

Let φ be a type-variable not occurring in any other type. If $PP_{\mathrm{C}_\mathcal{E}}(F(T_1, \ldots, T_n)) = \langle P, \pi \rangle$, then for $1 \leq i \leq n$, there are $\langle P_i, \pi_i \rangle$ (disjoint in pairs), such that for every $1 \leq i \leq n$, $PP_{\mathrm{C}_\mathcal{E}}(T_i) = \langle P_i, \pi_i \rangle$, and none of the type-variables of the type $\mathcal{E}(F)$ occur in any pair $\langle P_i, \pi_i \rangle$. By induction for every $1 \leq i \leq n$, there is a Curry-substitution S_i such that $S_i(\pi_i) = \sigma_i$, and, for every $x{:}\alpha \in P_i$, $B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S_i(\alpha)$. Take $S' = S_n \circ \cdots \circ S_0 \circ (\varphi \mapsto \sigma)$, then, for every $x{:}\alpha \in P_1 \cup \cdots \cup P_n$, $B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S'(\alpha)$, and $S'(\varphi) = \sigma$. Since $S'(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma = S'(\pi_1 \rightarrow \cdots \rightarrow \pi_n \rightarrow \varphi)$, by property 1.6 there are Curry-substitutions S and S_g such that

$$S_g = \mathit{unify}_{\mathrm{R}}(\mathcal{E}(F), \pi_1 \rightarrow \cdots \rightarrow \pi_n \rightarrow \varphi), \text{ and } S' = S \circ S_g$$

and $\langle P, \pi \rangle = S_g(\langle P_1 \cup \cdots \cup P_n, \varphi \rangle)$. Then,

for every $x{:}\beta \in S_g(P_1 \cup \cdots \cup P_n)$, $B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S(\beta)$, and $S(S_g(\varphi)) = \sigma$.

ii) a) $T \equiv x$. Trivial.

b) $T \equiv F(T_1, \ldots, T_n)$. If $B \vdash_{\mathrm{C}_\mathcal{E}} F(T_1, \ldots, T_n){:}\sigma$, then there are $\sigma_1, \ldots, \sigma_n$, and a Curry-substitution S such that $S(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$, and, for every $1 \leq i \leq n$, $B \vdash_{\mathrm{C}_\mathcal{E}} T_i{:}\sigma_i$. By induction for every $1 \leq i \leq n$, $B' \vdash_{\mathrm{C}_\mathcal{E}} T_i^{\mathrm{R}}{:}\sigma_i$. By definition 9.3.6 (i)

$$F(T_1{}^{\mathrm{R}}, \ldots, T_1{}^{\mathrm{R}}) = F(T_1, \ldots, T_n)^{\mathrm{R}}.$$

So, we obtain $B' \vdash_\mathcal{E} F(T_1, \ldots, T_n)^{\mathrm{R}}{:}\sigma$. □

In the following theorem we will prove that our solution is correct. The structure of the proof of the first part depends greatly on the fact that for every type σ we can trivially find an $Q \in \mathcal{F}$ such that $\mathcal{E}(Q) = \sigma$: we will just pick a constant Q that was not used previously.

Theorem 11.3.5 i) *The condition is necessary.* Let $Lhs, Rhs \in \mathrm{T}(\mathcal{F}, \mathcal{X})$, and $\mathbf{r} : Lhs \rightarrow Rhs$ be a typeable rewrite rule that is not safe. Then there there exist a replacement R and a type μ, such that $\vdash_{\mathrm{C}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$ and not $\vdash_{\mathrm{C}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$.

ii) *The condition is sufficient.* Let $\mathbf{r} : Lhs \rightarrow Rhs$ be a safe rewrite rule. Then, for every replacement R, basis B and a type μ, if $B \vdash_{\mathrm{C}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$, then $B \vdash_{\mathrm{C}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$.

Proof: i) Since $\mathbf{r}$ is typeable and left linear, we know that there are $\beta_1, \ldots, \beta_n, \tau$, and distinct $x_1, \ldots, x_n$, such that:

$$\{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\} \vdash_{\mathrm{C}_\mathcal{E}} Lhs{:}\tau \;\; \& \;\; \{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\} \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}\tau.$$

Then, by theorem 11.2.2 and lemma 11.3.3, we know that there are bases P_l, P_r, types $\rho_1, \ldots, \rho_n, \pi_r$ and a Curry-substitution S_0, such that:

$$\begin{aligned} PP_{\mathrm{C}_\mathcal{E}}(Lhs) = \langle P_l, \tau \rangle \;\&\; PP_{\mathrm{C}_\mathcal{E}}(Rhs) = \langle P_r, \pi_r \rangle \;\&\; P_l = \{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \\ \&\; S_0(\tau) = S_0(\pi_r) = \tau \;\&\; S_0(P_l) = S_0(P_r) = \{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\}. \end{aligned}$$

Let S_1 be the Curry-substitution such that, for every i, $S_1(\varphi_i) = c_i$ (the i-th type constant), μ be a type such that $S_1(\tau) = \mu$, $Q_1, \ldots, Q_n$ be constants such that, for every $1 \leq i \leq n$, $\mathcal{E}(Q_i) = S_1(\rho_i)$, and R be the replacement such that, for every $1 \leq i \leq n$, $x_i{}^{\mathrm{R}} = Q_i$. Then, by lemma 11.3.4 (ii), $\vdash_{\mathrm{C}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$. (Notice that Lhs^{R} does not contain term-variables.) Since $\mathbf{r}$ is not safe, we know that

$$\neg \{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}\tau.$$

Suppose (towards a contradiction) that $\vdash_{\mathrm{C}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$.

Then, by lemma 11.3.4 (i), there is a Curry-substitution S_2 such that

$$S_2(\pi_r) = \mu, \text{ and, for every } x{:}\gamma \in P_r, \; \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S_2(\gamma).$$

By definition 9.3.6 (ii.a.2), for every $x{:}\gamma \in P_r$ there is an $1 \leq i \leq n$ such that $x = x_i$. Since S_1 replaces type-variables by type constants, the type assigned to Q_i can only be $S_1(\rho_i)$. This implies that for every $x{:}\gamma \in P_r$ there is an $1 \leq i \leq n$ such that $x = x_i$ and $S_2(\gamma) = S_1(\rho_i)$. It is straightforward to verify that, since S_1 replaces type-variables by type constants, there is a Curry-substitution S_3 such that $S_2 = S_1 \circ S_3$. So, for every $x{:}\gamma \in P_r$ there is an $1 \leq i \leq n$ such that: $x = x_i$ and $\rho_i = S_3(\gamma)$. But then by theorem 11.1.4

$$\{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}S_3(\pi_r).$$

Moreover, $\mu = S_1(\tau) = S_2(\pi_r) = S_1 \circ S_3(\pi_r)$. Since S_1 replaces type-variables by distinct type constants, $\tau = S_3(\pi_r)$.

ii) Since $\mathbf{r}$ is safe, there are P, π such that: if $PP_{\mathrm{C}_\mathcal{E}}(Lhs) = \langle P, \pi \rangle$, then $P \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}\pi$. Suppose $PP_{\mathrm{C}_\mathcal{E}}(Lhs) = \langle P, \pi \rangle$, and R is a replacement such that there are basis B and type μ such that $B \vdash_{\mathrm{C}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$, then, by lemma 11.3.4 (i) there is a Curry-substitution S such that

$$S(\pi) = \mu \;\; \& \;\; \forall\, x{:}\rho \in P \; [\, B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S(\rho) \,].$$

But then by theorem 11.1.4

$$S(P) \vdash_{\mathrm{C}_\mathcal{E}} Rhs{:}S(\pi) \;\; \& \;\; \forall\, x{:}\rho \in P \; [\, B \vdash_{\mathrm{C}_\mathcal{E}} x^{\mathrm{R}}{:}S(\rho) \,].$$

So, by lemma 11.3.4 (ii) $B \vdash_{\mathrm{C}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$. □

Notice that, although the proof of part (i) explicitly used the presence of type constants, the problem of loss of subject reduction also arises if type constants are not in the type system. However, we do not believe that it is possible to prove that the condition is also necessary in this particular case.

Chapter 12 *Partial Rank 2 Intersection Type Assignment in Applicative Term Rewriting Systems*

The notion of type assignment presented in this chapter for Applicative Term Rewriting Systems will be based on the Rank 2 intersection type assignment system for the lambda calculus, and will be a restriction of the partial intersection type assignment system of chapter ten.

In the previous chapter, a partial type assignment system for Left Linear Applicative Term Rewriting Systems was presented. The system presented here can be seen as a variant of this system; the differences between these two are in the set of types that can be assigned to nodes and edges: Curry types in chapter eleven, intersection types of Rank 2 in this one. Another difference lies in the way we deal with recursively defined objects. The system in chapter eleven was based on Mycroft's extension of Curry's type assignment system, the one presented in this chapter will be based on Milner's. Also, because intersection types are used, the term rewrite rules need no longer be left linear.

We will define three operations on pairs: the operations of Rank 2 substitution and duplication are in fact the same as defined in chapter eight, the operation of weakening will replace a basis by a more informative one. We will define a notion of type assignment using Rank 2 intersection types. We will show that the defined operations are sound, i.e: for every basis B, term T, type σ and operation O: if $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$ and $O\,(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma'$.

We will show that the presented notion of type assignment has the principal type property. We will obtain this result by defining (using the unification algorithm for intersection types of Rank 2) for every term T the principal pair for T. We will show that for every typeable term T, there are a basis P and type π such that the pair $<P, \pi>$ is the principal pair for T, i.e. $P \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\pi$, and, for every basis B and type σ such that $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$, there is a chain of operations C such that $C\,(<P, \pi>) = <B, \sigma>$.

We will formulate a constraint typeable rewrite rules should satisfy in order to gain subject reduction. We will also show for every rewrite rule $Lhs \rightarrow Rhs$ that, if $<P, \pi>$ is the principal pair for Lhs, T is obtained from Lhs by replacing (in a consistent way) term-variables by terms, and T is typeable by τ, then there is a chain of operations C such that $C\,(\pi) = \tau$, and, for every $x{:}\alpha$ occurring in P, the replacement of x is typeable by $C\,(\alpha)$. With this result, we will prove the formulated condition to be sufficient.

We conclude this chapter by discussing some implementation aspects of Rank 2 type assignment in Applicative Term Rewriting Systems.

12.1 *Operations on pairs*

In this section we will discuss three operations on pairs of basis and type, namely substitution, duplication and weakening. Substitution is the same operation as defined in definition 8.3.1.1, with the same properties as, for example, in lemma 8.3.1.2. It will not be repeated here,

although, for example, the notion of basis is different for the notion of type assignment we discuss here. Duplication is very similar to the one defined in definition 8.3.2.1, and will be presented in definition 12.1.2.

The third operation we will present here is weakening. It was not needed for the results of chapter eight, but it plays a role in this chapter.

Definition 12.1.1 i) A *statement* is an expression of the form $T{:}\sigma$, where $T \in \mathrm{T}\,(\mathcal{F}, \mathcal{X})$ and $\sigma \in \mathcal{T}_{\mathrm{R}}$. T is the *subject* and σ the *predicate* of $T{:}\sigma$.

ii) A *basis* is a set of statements with distinct term-variables as subjects and types in $\mathcal{T}_1$ as predicates.

The definition of basis is not the same as in definition 8.2.3, since we do not allow for statements with the same term-variable. Unlike in chapter eight, this will cause no difficulties here. To make the Rank 2 system as we will define it in this chapter for Applicative Term Rewriting Systems fully comparable with the definition of the similar system for the lambda calculus in chapter eight, we have chosen to allow for types in $\mathcal{T}_1$ for term-variables only.

As in definition 2.3.2 we will extend the relation $\leq_{\mathrm{R}}$ to bases. Notice that $\Pi\{B_1, \ldots, B_n\}$ is well defined, since, if $\sigma_1, \ldots, \sigma_m$ are predicates of statements in $B_1 \cup \cdots \cup B_n$, then $\sigma_1 \in \mathcal{T}_1$, $\ldots, \sigma_m \in \mathcal{T}_1$, and $\sigma_1 \cap \cdots \cap \sigma_m \in \mathcal{T}_1$.

We now come to the definition of duplication. The differences between the one defined in this chapter and the one in definition 8.3.2.1 lies in part (ii), and is caused by the difference in the definition of bases.

Definition 12.1.2 Let B be a basis, $\sigma \in \mathcal{T}_{\mathrm{R}}$, and $n \geq 1$. The triple $<n, B, \sigma>$ determines a *duplication* $D_{<n,B,\sigma>} : \mathcal{T}_{\mathrm{R}} \rightarrow \mathcal{T}_{\mathrm{R}}$, which is constructed as follows:

i) a) Suppose $V = \{\varphi_1, \ldots, \varphi_m\}$ is the set of all type-variables occurring in $<B, \sigma>$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i $(1 \leq i \leq n, 1 \leq j \leq m)$ does not occur in V. Let S_i be the substitution that replaces every φ_j by φ_j^i.

b) $D_{<n,B,\sigma>}\,(\tau) = S_1\,(\tau) \cap \cdots \cap S_n\,(\tau)$.

ii) $D_{<n,B,\sigma>}\,(B') = \{x{:}D_{<n,B,\sigma>}\,(\rho) \mid x{:}\rho \in B'\}$.

iii) $D_{<n,B,\sigma>}\,(<B', \sigma'>) = <D_{<n,B,\sigma>}\,(B'), D_{<n,B,\sigma>}\,(\sigma')>$.

Instead of $D_{<n,B,\sigma>}$, we will write $<n, B, \sigma>$.

This definition satisfies the same properties as in lemma 8.3.2.2.

Lemma 12.1.3 Let $D = <n, B, \sigma>$.

i) If $\sigma \leq_{\mathrm{R}} \tau$, then $D\,(\sigma) \leq_{\mathrm{R}} D\,(\tau)$.

ii) $D\,(<B', \sigma'>) = <\Pi\{B_1, \ldots, B_n\}, \sigma_1 \cap \cdots \cap \sigma_n>$ with, for every $1 \leq i \leq n$, there is a substitution S_i such that $S_i\,(<B', \sigma'>) = <B_i, \sigma_i>$.

iii) $D\,(<B, \sigma>) = <\Pi\{B_1, \ldots, B_n\}, \sigma_1 \cap \cdots \cap \sigma_n>$ with, for every $1 \leq i \leq n$, $<B_i, \sigma_i>$ is a trivial variant of $<B, \sigma>$, and the $<B_i, \sigma_i>$ are disjoint in pairs.

iv) If $\tau \in \mathcal{T}_{\mathrm{C}}\,(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{\mathrm{R}})$, then $D\,(\tau) \in \mathcal{T}_1\,(\mathcal{T}_1, \mathcal{T}_{\mathrm{R}}, \mathcal{T}_{\mathrm{R}})$.

Proof: Immediately by definition 12.1.2. □

We now come to the definition of the operation of weakening; it replaces a basis by a more informative one.

Definition 12.1.4 A *weakening* W is an operation characterized by a pair of bases $<B_0, B_1>$ such that $B_1 \leq_R B_0$, and it is defined by:

i) $W(\sigma) = \sigma$.

ii) $W(<B, \sigma>) = <B_1, \sigma>$, if $B = B_0$.

iii) $W(<B, \sigma>) = <B, \sigma>$, if $B \neq B_0$.

Notice that a weakening $<B_0, B_1>$ corresponds to a lifting $<<B_0, \sigma>, <B_1, \sigma>>$, although they are not equivalent.

Similar to definition 8.3.3.1 we will define:

Definition 12.1.5 i) A *type-chain* is a chain of operations of substitution and duplication only.

ii) A *Rank 2 chain* is a type-chain concatenated with one operation of weakening.

This notion of type-chains also satisfies lemma 8.3.3.2.

12.2 *Rank 2 type assignment in ATRS's*

Rank 2 type assignment on an ATRS $(\Sigma, \mathbf{R})$ will be defined as the labelling of nodes (except those containing Ap) and edges in the tree-representation of terms and rewrite rules with types in $\mathcal{T}_R$. In this labelling, as before, we will use that there is a map that provides a type in $\mathcal{T}_2$ for every $F \in \mathcal{F}$. Such a map is called a Rank 2 environment.

Definition 12.2.1 Let $(\Sigma, \mathbf{R})$ be an ATRS.

i) A map $\mathcal{E} : \mathcal{F} \rightarrow \mathcal{T}_2$ is called a *Rank 2 environment* if for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$.

ii) For $F \in \mathcal{F}$ with arity $n \geq 0$, $\sigma \in \mathcal{T}_2$, and $\mathcal{E}$ an environment, the environment $\mathcal{E}[F := \sigma]$ is defined by:

a) $\mathcal{E}[F := \sigma](G) = \sigma$, if $G \in \{F, F_{n-1}, \ldots, F_0\}$.

b) $\mathcal{E}[F := \sigma](G) = \mathcal{E}(G)$, otherwise.

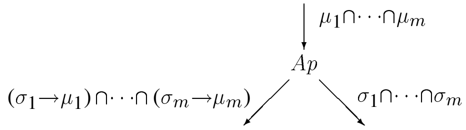
Rank 2 type assignment on Applicative Term Rewriting Systems will be defined, like before, in two stages. In the next definition we define Rank 2 type assignment on terms, and in definition 12.2.6 we will define Rank 2 type assignment on term rewrite rules.

Definition 12.2.2 Let $(\Sigma, \mathbf{R})$ be an ATRS, and $\mathcal{E}$ a Rank 2 environment.

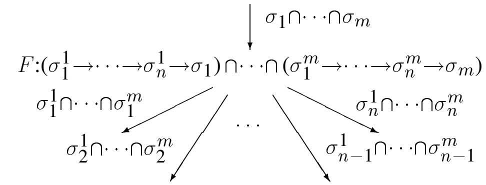
i) We say that $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ *is Rank 2 typeable by* $\sigma \in \mathcal{T}_R$ *with respect to* $\mathcal{E}$, if there exists an assignment of types to edges and nodes that satisfies the following constraints:

a) The root edge of T is typed with σ.

b) The type $\mu_1 \cap \cdots \cap \mu_m$ is assigned to the in-going edge of an application node, if and only if there are $\sigma_1, \ldots, \sigma_m \in \mathcal{T}_1$ such that $(\sigma_1 \rightarrow \mu_1) \cap \cdots \cap (\sigma_m \rightarrow \mu_m)$ is the type assigned to the left out-going edge, and the type assigned to the right out-going edge is $\sigma_1 \cap \cdots \cap \sigma_m$.



c) The type assigned to a function node containing $F \in \mathcal{F}$ (where F has arity $n \geq 0$) is $\tau_1 \cap \cdots \cap \tau_m$, if and only if, for every $1 \leq i \leq m$, there are $\sigma^i_1, \ldots, \sigma^i_n \in \mathcal{T}_1$ and $\sigma_i \in \mathcal{T}_2$, such that $\tau_i = \sigma^i_1 \rightarrow \cdots \rightarrow \sigma^i_n \rightarrow \sigma_i$, the type assigned to the j-th ($1 \leq j \leq n$) out-going edge is $\sigma^1_j \cap \cdots \cap \sigma^m_j$, and the type assigned to the in-going edge is $\sigma_1 \cap \cdots \cap \sigma_m$.



d) If the type assigned to a function node containing F is τ, then there is a type-chain C such that $C(\mathcal{E}(F)) = \tau$.

ii) Let $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ be Rank 2 typeable by σ with respect to $\mathcal{E}$. If B is a basis such that for every statement $x{:}\tau$ occurring in the typed term-tree, there is a $x{:}\tau' \in B$ such that $\tau' \leq_R \tau$, we write $B \vdash_{R_\mathcal{E}} T{:}\sigma$.

Notice that if $B \vdash_{R_\mathcal{E}} T{:}\sigma$, then B can contain more statements than needed to obtain $T{:}\sigma$.

A Rank 2 environment does not provide a type for Ap; instead, in part (i.b) of definition 12.2.2 it will be defined how the edges attached to an application node should be typed. So the node containing Ap itself is not typed; this is because although we know that, for every $1 \leq i \leq m$, $\sigma_i \rightarrow \mu_i$, $\mu_i \in \mathcal{T}_2$, and $\sigma_i \in \mathcal{T}_1$, not necessarily $(\sigma_i \rightarrow \mu_i) \rightarrow \sigma_i \rightarrow \mu_i \in \mathcal{T}_2$.

By definition 12.2.2 (i.d), the definition of $\mathcal{T}_R$ as in definition 8.2.1 is more general than actually needed. It would be sufficient to define $\mathcal{T}_R$ by:

iii) $\mathcal{T}_R$, the set of intersection types of Rank 2 is defined by: if $\sigma, \sigma_1, \ldots, \sigma_n \in \mathcal{T}_2$, and for every $1 \leq i \leq n$ there is a substitution S_i such that $S_i(\sigma) = \sigma_i$, then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_R$.

but this would have complicated the presentation of this chapter unnecessarily.

For the notion of type assignment as defined in definition 12.2.2, the following properties hold:

Lemma 12.2.3 i) $B \vdash_{R_\mathcal{E}} T{:}\sigma_1 \cap \cdots \cap \sigma_n$ if and only if for every $1 \leq i \leq n$, $B \vdash_{R_\mathcal{E}} T{:}\sigma_i$.

ii) If $x{:}\tau$ is a statement occurring in the typed term-tree for $B \vdash_{R_\mathcal{E}} T{:}\sigma$, then $B \leq_R \{x{:}\tau\}$.

Proof: Immediately from definition 12.2.2. □

Because of part (i) of this lemma, in most proofs over $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$ we will only show the part that assumes $\sigma \in \mathcal{T}_2$.

As in chapter ten in defining type assignment on rewrite rules, we will restrict the possible bases for rewrite rules to those that contain nothing but the statements needed to type the left hand side. In the next definition we will introduce the notion of Rank 2 used bases, which is similar to that of used bases. This notion will be convenient in several proofs and definitions in the rest of this chapter.

Definition 12.2.4 i) The *Rank 2 used bases of* $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$ are inductively defined by:

a) $\sigma \in \mathcal{T}_2$.

1) $T \equiv x$. Take $\{x{:}\sigma\}$.

2) $T \equiv Ap(T_1, T_2)$. Then there is a τ such that $B \vdash_{\mathrm{R}_\mathcal{E}} T_1{:}\tau\rightarrow\sigma$, and $B \vdash_{\mathrm{R}_\mathcal{E}} T_2{:}\tau$. Let B_1 be a Rank 2 used basis of $B \vdash_{\mathrm{R}_\mathcal{E}} T_1{:}\tau\rightarrow\sigma$, and B_2 be a Rank 2 used basis of $B \vdash_{\mathrm{R}_\mathcal{E}} T_2{:}\tau$. Take $\Pi\{B_1, B_2\}$.

3) $T \equiv F(T_1, \ldots, T_n)$. Then there are $\sigma_1, \ldots, \sigma_n$ such that, for every $1 \le i \le n$, $B \vdash_{\mathrm{R}_\mathcal{E}} T_i{:}\sigma_i$. Let, for $1 \le i \le n$, B_i be a Rank 2 used basis of $B \vdash_{\mathrm{R}_\mathcal{E}} T_i{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

b) If $\sigma = \sigma_1\cap\cdots\cap\sigma_n$, then, by lemma 12.2.3 (i), for every $1 \le i \le n$, $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma_i$. Let, for every $1 \le i \le n$, B_i be a Rank 2 used basis of $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

ii) A basis B *is Rank 2 used for* $T{:}\sigma$ *with respect to* $\mathcal{E}$ if and only if there is a basis B' such that $B' \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$, and B is a Rank 2 used basis of $B' \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$.

We will say 'B is Rank 2 used for $T{:}\sigma$' instead of 'B is Rank 2 used for $T{:}\sigma$ with respect to $\mathcal{E}$'.

As before, a Rank 2 used basis for a statement $T{:}\sigma$ is not unique, but, again, the results of this chapter do not depend on the actual structure of a Rank 2 used basis, only on its existence.

For Rank 2 used bases, the following properties hold.

Lemma 12.2.5 i) If B is Rank 2 used for $T{:}\sigma$, then $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$.

ii) $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma \Leftrightarrow \exists\, B' \,[\, B \le_{\mathrm{R}} B' \;\&\; B'$ is Rank 2 used for $T{:}\sigma$].

Proof: By induction on definition 12.2.4. □

Thanks to the notion of Rank 2 used basis, we can give a clear definition of a typeable rewrite rule and a typeable rewrite system. Notice that this notion plays the same role in this definition as the notion of 'essentially used basis' did in definition 10.1.7.

Definition 12.2.6 Let $(\Sigma, \mathbf{R})$ be an ATRS, and $\mathcal{E}$ a Rank 2 environment.

i) We say that $Lhs \rightarrow Rhs \in \mathbf{R}$ with defined symbol F *is Rank 2 typeable with respect to* $\mathcal{E}$, if there are basis B, type $\sigma \in \mathcal{T}_2$, such that

a) B is Rank 2 used for $Lhs{:}\sigma$ and $B \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}\sigma$.

b) In $B \vdash_{\mathrm{R}_\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}\sigma$, all nodes containing F are typed with $\mathcal{E}(F)$.

ii) We say that $(\Sigma, \mathbf{R})$ *is Rank 2 typeable with respect to* $\mathcal{E}$, if every $\mathbf{r} \in \mathrm{R}$ is Rank 2 typeable with respect to $\mathcal{E}$.

Proposition 12.2.7 Let x be a term-variable occurring in the rewrite rule $\mathbf{r}$: $Lhs \rightarrow Rhs$, and let $\mathbf{r}$ be typeable with basis B and type σ. If $\sigma_1, \ldots, \sigma_m$ are all and nothing but the types assigned to the nodes containing x in the typed term-tree for $B \vdash_{\mathrm{R}_\mathcal{E}} Lhs{:}\sigma$, and $\tau_1, \ldots, \tau_n$ are those for $B \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}\sigma$, then $x{:}\sigma_1\cap\cdots\cap\sigma_m \in B$ and $\sigma_1\cap\cdots\cap\sigma_m \le_{\mathrm{R}} \tau_1\cap\cdots\cap\tau_n$.

Proof: By definition 12.2.4 and lemma 12.2.3 (ii). □

Notice that, in the above case, not necessarily $\sigma_1\cap\cdots\cap\sigma_m = \tau_1\cap\cdots\cap\tau_n$.

12.3 *Soundness of operations on pairs*

In this section we will prove that the three operations on pairs of basis and type (substitution, duplication and weakening) are sound on typed term-trees. We will also show that part (i.d) of definition 12.2.2 is sound: if there is a type-chain C such that $C(\mathcal{E}(F)) = \sigma$, then, for every type $\tau \in \mathcal{T}_2$ such that $\sigma \le_{\mathrm{R}} \tau$, the rewrite rules that define F are typeable with respect to a changed environment, in which $\mathcal{E}(F)$ is replaced by τ.

The following lemma will show that Rank 2 substitution is a sound operation on typed term-trees and rewrite rules.

Theorem 12.3.1 *Soundness of Rank 2 substitution.* Let S be a Rank 2 substitution.

i) If $B \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma$, then $S(B) \vdash_{\mathrm{R}_\mathcal{E}} T{:}S(\sigma)$.

ii) If B is Rank 2 used for $T{:}\sigma$, then $S(B)$ is Rank 2 used for $T{:}S(\sigma)$.

iii) Let $\mathbf{r}$: $Lhs \rightarrow Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of $\mathbf{r}$. Then $\mathbf{r}$ is typeable with respect to $\mathcal{E}[F := S(\mathcal{E}(F))]$.

Proof: Very much like the one for theorem 10.2.2. We will, therefore, only show the proof of part (i), which is given by induction on the structure of T. (Assume $\sigma \in \mathcal{T}_2$.)

a) $T \equiv x$. Then by lemma 12.2.3 (ii) $B \le_{\mathrm{R}} \{x{:}\sigma\}$. By lemma 8.3.1.2 (i) $S(B) \le_{\mathrm{R}} S(\{x{:}\sigma\})$, so by 12.2.2 (ii) $S(B) \vdash_{\mathrm{R}_\mathcal{E}} x{:}S(\sigma)$.

b) $T \equiv Ap(T_1, T_2)$. As part (i.c) of the proof of theorem 8.3.1.3.

c) $T \equiv F(T_1, \ldots, T_n)$. By definition 12.2.2 (i) there are $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_1$, and a type-chain C such that $C(\mathcal{E}(F)) = \sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\sigma$, and, for every $1 \le i \le n$, $B \vdash_{\mathrm{R}_\mathcal{E}} T_i{:}\sigma_i$. By induction for every $1 \le i \le n$, $S(B) \vdash_{\mathrm{R}_\mathcal{E}} T_i{:}S(\sigma_i)$, and, by lemma 8.3.1.2 (ii), $S(\sigma_i) \in \mathcal{T}_1$. Since $C * \langle S\rangle$ is also a type-chain, and $S(\sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\sigma) = S(\sigma_1)\rightarrow\cdots\rightarrow S(\sigma_n)\rightarrow S(\sigma)$, we have $S(B) \vdash_{\mathrm{R}_\mathcal{E}} F(T_1, \ldots, T_n){:}S(\sigma)$. □

We will now prove that the operation of duplication is sound on typed term-trees and rewrite rules.

Theorem 12.3.2 *Soundness of duplication.* Let D be a duplication such that $D(\langle B', \sigma'\rangle) = \langle B'', \sigma''\rangle$.

i) If $B' \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma'$, then $B'' \vdash_{\mathrm{R}_\mathcal{E}} T{:}\sigma''$.

ii) If B' is Rank 2 used for $T{:}\sigma'$, then B'' is Rank 2 used for $T{:}\sigma''$.

iii) Let **r**: $Lhs \rightarrow Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of **r**. If $D(\mathcal{E}(F)) = \tau \in \mathcal{T}_R$, then, for every $\mu \in \mathcal{T}_2$ such that $\tau \leq_R \mu$, **r** is typeable with respect to $\mathcal{E}[F := \mu]$.

Proof: Very much like the proof of theorem 10.2.5, but easier since duplication is a simplified version of strict expansion.

i) By lemma 12.1.3 (i), there are $B_1, \ldots, B_n, \sigma_1, \ldots, \sigma_n$ such that

$$<B', \sigma'> = <\Pi\{B_1, \ldots, B_n\}, \sigma_1 \cap \cdots \cap \sigma_n>,$$

and for every $1 \leq i \leq n$ there is a substitution S_i such that $S_i(<B, \sigma>) = <B_i, \sigma_i>$.
Notice that for all $F \in \mathcal{F}$ occurring in T by definition 12.2.2 (i.d), there is a type-chain C such that F is typed with $C(\mathcal{E}(F))$. Since $C * <D>$ is also a type-chain, F can be typed with the type $D(C(\mathcal{E}(F)))$.
The proof is completed by theorem 12.3.1 (i) and lemma 12.2.3 (i).

ii) As the proof of part (i).

iii) For every $\mu \in \mathcal{T}_2$ such that $\tau \leq_R \mu$, by definition 12.1.2 there is a substitution S such that $\mu = S(\mathcal{E}(F))$. The proof is completed by theorem 12.3.1 (iii). □

The next theorem states that a weakening performed on a arbitrary pair, will produce a correct result.

Theorem 12.3.3 *Soundness of weakening.* For every $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$: if $B \vdash_{R_\mathcal{E}} T{:}\sigma$, then for every weakening W: if $W(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_{R_\mathcal{E}} T{:}\sigma'$.

Proof: By definition 12.2.2 (ii), for every $x{:}\tau$ occurring in the typed term-tree for $T{:}\sigma$ there is a $x{:}\mu \in B$ such that $\mu \leq_R \sigma$. By definition 12.1.4, $B' \leq_R B$, and $\sigma' = \sigma$. For every $x{:}\alpha \in B$ there is a $x{:}\alpha' \in B'$ such that $\alpha' \leq_R \alpha$. But then for every $x{:}\tau$ occurring in the typed term-tree for $T{:}\sigma$, there is a $x{:}\mu' \in B'$ such that $\mu' \leq_R \tau$, so $B' \vdash_{R_\mathcal{E}} T{:}\sigma$. □

Combining the results proved above, we know that Rank 2 chains and type-chains will have the following effect on pairs and rewrite rules:

Lemma 12.3.4 i) If $B \vdash_{R_\mathcal{E}} T{:}\sigma$, and C is a Rank 2 chain such that $C(<B, \sigma>) = <B', \sigma'>$, then $B' \vdash_{R_\mathcal{E}} T{:}\sigma'$.

ii) If B is Rank 2 used for $T{:}\sigma$, C is a type-chain, and $C(<B, \sigma>) = <B', \sigma'>$, then B' is Rank 2 used for $T{:}\sigma'$.

iii) Let **r**: $Lhs \rightarrow Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of **r**. If C is a type-chain and $C(\mathcal{E}(F)) = \tau \in \mathcal{T}_R$, then, for every $\mu \in \mathcal{T}_2$ such that $\tau \leq_R \mu$, **r** is typeable with respect to $\mathcal{E}[F := \mu]$.

Proof: By theorems 12.3.1, 12.3.2 and 12.3.3. □

12.4 *Principal type property*

In this section we will show that the partial Rank 2 type assignment system has the principal type property, by defining, for every typeable term T, a principal pair with respect to the environment $\mathcal{E}$, $PP_{R_\mathcal{E}}(T)$, and by showing that, for all pairs $<B, \sigma>$ such that $B \vdash_{R_\mathcal{E}} T{:}\sigma$, there is a Rank 2 chain C such that $C(PP_{R_\mathcal{E}}(T)) = <B, \sigma>$.

We will define the principal pair for a term T with respect to $\mathcal{E}$, consisting of basis P and type π, by defining $PP_{R_\mathcal{E}}(T)$ using the operations *unify*$_1$ and *unify*$_R$. In theorem 12.4.8 we will show that, for every term, this is indeed the principal one.

Definition 12.4.1 For every $T \in \mathrm{T}(\mathcal{F}, \mathcal{X})$ we define (using *unify*$_1$, *unify*$_R$, and *to*$\mathcal{T}_C$) the Rank 2 principal pair with respect to $\mathcal{E}$ by defining the notion $PP_{R_\mathcal{E}}(T) = <P, \pi>$ by:

i) $T \equiv x$. Then $<P, \pi> = <\{x{:}\varphi\}, \varphi>$.

ii) $T \equiv Ap(T_1, T_2)$. Let $PP_{R_\mathcal{E}}(T_1) = <P_1, \pi_1>$, $PP_{R_\mathcal{E}}(T_2) = <P_2, \pi_2>$, (we choose if necessary trivial variants such that these are disjoint in pairs), and $S_2 = to\mathcal{T}_C(\pi_2)$, then:

a) If $\pi_1 = \varphi$, then:

$$PP_{R_\mathcal{E}}(Ap(T_1, T_2)) = <S_2, S_1>(<\Pi\{P_1, P_2\}, \varphi'>),$$
$$\text{where } S_1 = unify_R(\varphi, S_2(\pi_2) \rightarrow \varphi').$$

and φ' is a type-variable not occurring in any other type.

b) If $\pi_1 = \sigma \rightarrow \tau$, then:

$$PP_{R_\mathcal{E}}(Ap(T_1, T_2)) = <S_2> * C(<\Pi\{P_1, P_2\}, \tau>),$$
$$\text{where } C = unify_1(\sigma, S_2(\pi_2), S_2(P_2)).$$

iii) $T \equiv F(T_1, \ldots, T_n)$. If $\mathcal{E}(F) = \gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$, and for every $1 \leq i \leq n$, $PP_{R_\mathcal{E}}(T_i) = <P_i, \pi_i>$, (we choose if necessary trivial variants such that the $<P_i, \pi_i>$ are disjoint in pairs and these pairs share no type-variables with $\gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$), then:

$$PP_{R_\mathcal{E}}(F(T_1, \ldots, T_n)) = C(<\Pi\{P_1, \ldots, P_n\}, \gamma>),$$
$$\begin{aligned} \text{where } C &= <S_1, \ldots, S_n> * C_1 * \cdots * C_n, \\ S_i &= to\mathcal{T}_C(\pi_i), \\ C_i &= unify_1(C_1 * \cdots * C_{i-1}(\gamma_i), S_i(\pi_i), S_i(P_i)). \end{aligned}$$

Notice that part (ii) of this definition is the same as part (iii) of definition 8.4.2.1.

Example 12.4.2 The typed rules for F (as in example 10.1.9) seem perhaps somewhat ad hoc, but using the environment:

$$\begin{aligned} \mathcal{E}(K) &= 1 \rightarrow 2 \rightarrow 1 \\ \mathcal{E}(Z) &= 3 \rightarrow 4 \rightarrow 4 \\ \mathcal{E}(I) &= 5 \rightarrow 5 \\ \mathcal{E}(F) &= 7 \cap (6 \rightarrow 7) \cap 6 \rightarrow 7 \end{aligned}$$

where Z is defined by $Z(x, y) \rightarrow y$, and using definition 12.4.1, the following can easily be checked:

i) $F(I_0)$ is typeable by $8 \rightarrow 8$, which is a type for both I_0 and $I(I_0)$.

ii) $F(Z_0)$ is typeable by $(8 \rightarrow 8) \rightarrow 8 \rightarrow 8$, which is a type for both Z_0 and $Z_1(Z_0)$.

iii) $F(K_0)$ is typeable by $(8 \rightarrow 9) \rightarrow 9 \rightarrow 8 \rightarrow 9$, which is a type for both K_0 and $K_1(K_0)$.

The given types are the principal types for respectively $F(I_0)$, $F(Z_0)$, and $F(K_0)$.

Example 12.4.3 Using Rank 2 intersection types, the term $S(K_0, S_0, I_0)$ has a smaller principal type than using Curry types. With the environment

$$\begin{aligned}\mathcal{E}\,(S) &= (1{\to}2{\to}3){\to}(4{\to}2){\to}1{\cap}4{\to}3\\ \mathcal{E}\,(K) &= 5{\to}6{\to}5\\ \mathcal{E}\,(I) &= 7{\to}7\end{aligned}$$

and definition 12.4.1 the following can easily be checked:

$$\downarrow 8{\to}8$$

$$\begin{aligned}S{:}((8{\to}8){\to}(((9{\to}10){\to}9){\to}(9{\to}10){\to}10){\to}8{\to}8) &\to\\ (((9{\to}10){\to}9{\to}10){\to}((9{\to}10){\to}9){\to}(9{\to}10){\to}10) &\to\\ (8{\to}8){\cap}((9{\to}10){\to}9{\to}10) &\to 8{\to}8\end{aligned}$$

$$\swarrow \qquad \downarrow \qquad \searrow$$

$$K_0{:}(8{\to}8){\to}(((9{\to}10){\to}9){\to}(9{\to}10){\to}10){\to}8{\to}8 \qquad I_0{:}(8{\to}8){\cap}((9{\to}10){\to}9{\to}10)$$

$$S_0{:}((9{\to}10){\to}9{\to}10){\to}((9{\to}10){\to}9){\to}(9{\to}10){\to}10$$

If we define $D\,(x) \to Ap\,(x, x)$, then we can even check that, for example, $D\,(S\,(K_0, S_0, I_0))$ and $D\,(I_0)$ are typeable by $11{\to}11$.

The following lemma will be used in the proofs of the next section.

Lemma 12.4.4 If $PP_{\mathrm{R}_\mathcal{E}}\,(T) = <P, \pi>$, then P is Rank 2 used for $T{:}\pi$, and $\pi \in \mathcal{T}_2$.

Proof: By induction on the definition of $PP_{\mathrm{R}_\mathcal{E}}\,(T)$, using lemma 12.3.4 (ii). □

A direct consequence of this lemma is that if $PP_{\mathrm{R}_\mathcal{E}}\,(T) = <P, \pi>$, then $P \vdash_{\mathrm{R}_\mathcal{E}} T{:}\pi$.

The following lemmas will be needed in the proofs of theorem 12.4.7 and lemma 12.5.3. The first will give a result similar to the one of lemma 8.4.2.4: if a type-chain maps the principal pairs of terms in an application to pairs that allow the application itself to be typed, then these pairs can also be obtained by first performing a unification. The second will generalize this result to arbitrary function applications.

Lemma 12.4.5 Let $\sigma \in \mathcal{T}_2$, and for $i = 1, 2$ $PP_{\mathrm{R}_\mathcal{E}}\,(T_i) = <P_i, \pi_i>$ such that the pairs are disjoint, and let C be a type-chain such that $C\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_1)) = <B_1, \tau{\to}\sigma>$, and $C\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_2)) = <B_2, \tau>$. Then there are type-chains C_g and C' and type $\alpha \in \mathcal{T}_2$ such that

$$PP_{\mathrm{R}_\mathcal{E}}\,(Ap\,(T_1, T_2)) = C_g\,(<\Pi\{P_1, P_2\}, \alpha>), \text{ and}$$
$$C'\,(PP_{\mathrm{R}_\mathcal{E}}\,(Ap\,(T_1, T_2))) = <\Pi\{B_1, B_2\}, \sigma>.$$

Proof: Like the one for lemma 8.4.2.4. □

Lemma 12.4.6 Let $\sigma \in \mathcal{T}_2$, and for every $1 \leq i \leq n$, $PP_{\mathrm{R}_\mathcal{E}}\,(T_i) = <P_i, \pi_i>$, such that the pairs $<P_i, \pi_i>$ and the type $\mathcal{E}\,(F) = \gamma_1{\to}\cdots{\to}\gamma_n{\to}\gamma$ are disjoint, and let C be a type-chain such that

$$C\,(\mathcal{E}\,(F)) = \sigma_1{\to}\cdots{\to}\sigma_n{\to}\sigma \text{ and, for every } 1 \leq i \leq n,\ C\,(<P_i, \pi_i>) = <B_i, \sigma_i>.$$

Then there are type-chains C_g and C_p such that

$$PP_{\mathrm{R}_\mathcal{E}}\,(F\,(T_1, \ldots, T_n)) = C_g\,(<\Pi\{P_1, \ldots, P_n\}, \gamma>), \text{ and}$$

$$C_p\,(PP_{\mathrm{R}_\mathcal{E}}\,(F\,(T_1, \ldots, T_n))) = <\Pi\{B_1, \ldots, B_n\}, \sigma>.$$

Proof: Let, for $1 \leq i \leq n$, $S_i = to\mathcal{T}_\mathrm{C}\,(\pi_i)$. Since for every $1 \leq i \leq n$, $C\,(\pi_i) = \sigma_i \in \mathcal{T}_1$, by lemma 8.4.1.2 (i), for every $1 \leq i \leq n$ there is a $C_i{}'$ such that $C = <S_i> * C_i{}'$. Since the π_i are disjoint, the S_i do not interfere, so, without loss of generality we can assume that there is a C' such that $C = <S_1, \ldots, S_n> * C'$, and for $1 \leq i \leq n$, $C'\,(\gamma_i) = \sigma_i$.

We will finish the proof by showing by induction on i that for every $0 \leq i \leq n$ there are type-chains $C_i, C_1^u, \ldots, C_i^u$ such that $C' = C_1^u * \cdots * C_i^u * C_i$, and for $1 \leq j \leq i$ the C_j^u are not defined on type-variables occurring in $S_k\,(\pi_k)$, with $k > i$.

i) $i = 0$. Take $C_0 = C'$.

ii) Since $C'\,(\gamma_i) = C'\,(S_i\,(\pi_i))$, by induction:

$$C_{i-1}\,(C_1^u * \cdots * C_{i-1}^u\,(\gamma_i)) = C_{i-1}\,(C_1^u * \cdots * C_{i-1}^u\,(S_i\,(\pi_i))) = C_{i-1}\,(S_i\,(\pi_i)).$$

$S_i\,(P_i)$ shares no type-variables with $C_1^u * \cdots * C_{i-1}^u\,(\gamma_i)$, so by lemma 8.4.1.2 (ii) there is a type-chain C_i such that $C_{i-1} = unify_1\,(C_1^u * \cdots * C_{i-1}^u\,(\gamma_i), S_i\,(\pi_i), S_i\,(P_i)) * C_i$. Take $C_i^u = unify_1\,(C_1^u * \cdots * C_{i-1}^u\,(\gamma_i), S_i\,(\pi_i), S_i\,(P_i))$.

Take $C_g = <S_1, \ldots, S_n> * C_1^u * \cdots * C_n^u$, then by definition 12.4.1 (iii) we have

$$PP_{\mathrm{R}_\mathcal{E}}\,(F\,(T_1, \ldots, T_n)) = C_g\,(<\Pi\{P_1, \ldots, P_n\}, \gamma>).$$

Take $C_p = C_n$. □

The following theorem will show that type-chains are sufficient to generate all possible pairs $<B, \sigma>$ for a typeable term T such that B is Rank 2 used for $T{:}\sigma$.

Theorem 12.4.7 If B is Rank 2 used for $T{:}\sigma$, then there are a basis P, type π and a type-chain C such that $PP_{\mathrm{R}_\mathcal{E}}\,(T) = <P, \pi>$, and $C\,(<P, \pi>) = <B, \sigma>$.

Proof: i) $\sigma \in \mathcal{T}_2$. By induction on the structure of T.

a) $T \equiv x$. Then $B = \{x{:}\sigma\}$. Then $\sigma \in \mathcal{T}_\mathrm{C}$, and $PP_{\mathrm{R}_\mathcal{E}}\,(x) = <\{x{:}\varphi\}, \varphi>$. Take $C = <(\varphi \mapsto \sigma)>$.

b) $T \equiv Ap\,(T_1, T_2)$. Then, by definition 12.2.4 (i.a.2), there are $\tau \in \mathcal{T}_1$ and B_1, B_2 such that B_1 is Rank 2 used for $T_1{:}\tau{\to}\sigma$, B_2 is Rank 2 used for $T_2{:}\tau$, and $B = \Pi\{B_1, B_2\}$. By induction for $i = 1, 2$, there are P_i, π_i, and type-chains C_i such that

$$PP_{\mathrm{R}_\mathcal{E}}\,(T_1) = <P_1, \pi_1>,\ C_1\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_1)) = <B_1, \tau{\to}\sigma>, \text{ and}$$
$$PP_{\mathrm{R}_\mathcal{E}}\,(T_2) = <P_2, \pi_2>,\ C_2\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_2)) = <B_2, \tau>.$$

Since the pairs $<P_i, \pi_i>$ are disjoint, the type-chains C_i do not interfere, so

$$C_1 * C_2\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_1)) = <B_1, \tau{\to}\sigma>, \text{ and } C_1 * C_2\,(PP_{\mathrm{R}_\mathcal{E}}\,(T_2)) = <B_2, \tau>.$$

Then, by lemma 12.4.5, there are type-chains C and C_u, and type α such that $PP_{\mathrm{R}_\mathcal{E}}\,(Ap\,(T_1, T_2)) = C_u\,(<\Pi\{P_1, P_2\}, \alpha>)$, and $C\,(PP_{\mathrm{R}_\mathcal{E}}\,(Ap\,(T_1, T_2))) = <B, \sigma>$.

c) $T \equiv F\,(T_1, \ldots, T_n)$. By definition 12.2.4 (i.a.3), there are $B_1, \ldots, B_n$, and $\sigma_1, \ldots, \sigma_n$ such that $B = \Pi\{B_1, \ldots, B_n\}$, and, for every $1 \leq i \leq n$, B_i is Rank 2 used for $T_i{:}\sigma_i$. By definition 12.2.2 (i.d), there is a type-chain C_F such that

$$C_F\,(\mathcal{E}\,(F)) = \sigma_1{\to}\cdots{\to}\sigma_n{\to}\sigma.$$

By induction for $1 \leq i \leq n$, there are $<P_i, \pi_i>$ (disjoint in pairs) and type-chains C_i, such that

$$\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T_i) = \langle P_i, \pi_i \rangle, \text{ and } C_i\,(\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T_i)) = \langle B_i, \sigma_i \rangle.$$

Since the pairs $\langle P_i, \pi_i \rangle$ are disjoint, the type-chains C_i do not interfere. Assume, without loss of generality, that none of the type-variables occurring in $\mathcal{E}(F)$ occur in any of the pairs $\langle P_i, \pi_i \rangle$. Let $C' = C_F * C_1 * \cdots * C_n$. Since, for every $1 \leq i \leq n$, $C'(\langle P_i, \pi_i \rangle) = \langle B_i, \sigma_i \rangle$, and $C'(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$, then, by lemma 12.4.6, there are type-chains C and C_u such that

$$\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(F(T_1, \ldots, T_n)) = C_u\,(\langle \Pi\{P_1, \ldots, P_n\}, \gamma \rangle), \text{ and } C\,(\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(F(T_1, \ldots, T_n))) = \langle B, \sigma \rangle.$$

ii) $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$. By lemma 12.2.4 (i.b), for every $1 \leq i \leq n$ there is a B_i such that B_i is Rank 2 used for $T{:}\sigma_i$, and $B = \Pi\{B_1, \ldots, B_n\}$. Take $\langle B_i', \sigma_i' \rangle$, trivial variants of $\langle B_i, \sigma_i \rangle$ that are disjoint in pairs. Take S such that

$$S\,(\langle \Pi\{B_1', \ldots, B_n'\}, \sigma_1' \cap \cdots \cap \sigma_n' \rangle) = \langle \Pi\{B_1, \ldots, B_n\}, \sigma_1 \cap \cdots \cap \sigma_n \rangle.$$

By induction there are P, π, such that $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P, \pi \rangle$. Let $D = \langle n, P, \pi \rangle$, then

$$D\,(\langle P, \pi \rangle) = \langle \Pi\{P_1, \ldots, P_n\}, \pi_1 \cap \cdots \cap \pi_n \rangle, \text{ with } \mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P_i, \pi_i \rangle.$$

By induction there are type-chains $C_1, \ldots, C_n$ such that

$$\text{for } 1 \leq i \leq n,\ C_i\,(\langle P_i, \pi_i \rangle) = \langle B_i', \sigma_i' \rangle.$$

Since the $\langle P_i, \pi_i \rangle$ and the $\langle B_i', \sigma_i' \rangle$ are disjoint in pairs, the C_i do not interfere. Take $C = \langle D \rangle * C_1 * \cdots * C_n * \langle S \rangle$. □

Theorem 12.4.8 *Principal pair property*.

i) *Soundness.* If $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P, \pi \rangle$, and C a Rank 2 chain such that $C\,(\langle P, \pi \rangle) = \langle B, \sigma \rangle$, then $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$.

ii) *Completeness.* If $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$, then there are P and π, such that $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P, \pi \rangle$, and a Rank 2 chain C such that $C\,(\langle P, \pi \rangle) = \langle B, \sigma \rangle$.

Proof: i) By lemmas 12.2.5 and 12.4.4 and theorem 12.3.4 (i).

ii) If $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$, then there is a B' such that $B \leq_{\mathrm{R}} B'$ and B' is Rank 2 used for $T{:}\sigma$. By theorem 12.4.7 there exist P, π, and a type-chain C' such that $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P, \pi \rangle$, and $C'(\langle P, \pi \rangle) = \langle B', \sigma \rangle$. Take the weakening $W = \langle B', B \rangle$, then $W(\langle B', \sigma \rangle) = \langle B, \sigma \rangle$. Take $C = C' * \langle W \rangle$. □

12.5 *A sufficient condition for subject reduction*

In this section we will formulate a condition typeable rewrite rules should satisfy in order to obtain subject reduction. This condition will be the same as the one formulated for the type assignment system presented in the previous chapter. We will show that this condition is sufficient.

Definition 12.5.1 i) We call a rewrite rule $\mathit{Lhs} \rightarrow \mathit{Rhs}$ *safe* if:

If $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(\mathit{Lhs}) = \langle P, \pi \rangle$, then $P \vdash_{\mathrm{R}_{\mathcal{E}}} \mathit{Rhs}{:}\pi$.

ii) The definition of '*safely Rank 2 typeable with respect to* $\mathcal{E}$' is the same as given in definition 12.2.2 and 12.2.6, by replacing condition (i.a) of definition 12.2.6 by:

If $\langle P, \pi \rangle = \mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(\mathit{Lhs})$, then $P \vdash_{\mathrm{R}_{\mathcal{E}}} \mathit{Rhs}{:}\pi$.

Notice that the notion $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T)$ is defined independently from the definition of typeable rewrite rules. Moreover, since type assignment as defined in this chapter is decidable, this safeness constraint on rewrite rules is decidable too.

Before we come to the proof that the condition is sufficient, we will prove some preliminary results.

The following lemmas will formulate the relation between replacements performed on a term that is the left hand side of a rewrite rule, and possible type assignments for that term. The construction of the proof of lemma 12.5.3 will in fact be one of the motivations for the limitation of the possible left hand sides, as given in definition 9.3.2 (ii.a).

Lemma 12.5.2 If $B \vdash_{\mathrm{R}_{\mathcal{E}}} T{:}\sigma$, and R is a replacement and B' a basis such that for every statement $x{:}\alpha \in B$ $B' \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}\alpha$, then $B' \vdash_{\mathrm{R}_{\mathcal{E}}} T^{\mathrm{R}}{:}\sigma$.

Proof: By easy induction on the structure of T (see the proof of lemma 11.3.4 (ii)). □

Lemma 12.5.3 If $T \in \mathit{LHS}$, for the replacement R there are B and σ such that B is Rank 2 used for $T^{\mathrm{R}}{:}\sigma$, and $\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T) = \langle P, \pi \rangle$, then there is a type-chain C, such that $C\,(\pi) = \sigma$, and, for every statement $x{:}\alpha \in P$, $B \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C\,(\alpha)$.

Proof: By induction on the structure of *LHS*.

i) $T \equiv \mathit{Ap}(T_1, T_2)$. Since $\mathit{Ap}(T_1, T_2)^{\mathrm{R}} = \mathit{Ap}(T_1^{\mathrm{R}}, T_2^{\mathrm{R}})$, by definition 12.2.4 (i.a.2) there are $\tau \in \mathcal{T}_1$, and B_1, B_2 such that B_1 is Rank 2 used for $T_1^{\mathrm{R}}{:}\tau \rightarrow \sigma$, and B_2 is Rank 2 used for $T_2^{\mathrm{R}}{:}\tau$. By theorem 12.4.7 for $i = 1, 2$, there are disjoint $\langle P_i, \pi_i \rangle = \mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(T_i)$. Since $T_1 \in \mathit{LHS}$, by induction there is a type-chain C_1 such that $C_1\,(\pi_1) = \tau \rightarrow \sigma$, and, for every statement $x{:}\alpha \in P_1$, $B_1 \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_1\,(\alpha)$. For T_2 we have either:

a) $T_2 \equiv x$, so $\langle P_2, \pi_2 \rangle = \langle \{x{:}\varphi\}, \varphi \rangle$ for some φ. Take $C_2 = \mathit{unify}_1\,(\tau, \varphi, \{x{:}\varphi\})$, then $\tau = C_2\,(\varphi)$. B_2 is Rank 2 used for $x^{\mathrm{R}}{:}\tau$, so by lemma 12.2.5 (i), for every statement $x{:}\alpha \in P_2$, $B_2 \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_2\,(\alpha)$.

b) $T_2 \in \mathit{LHS}$, so by induction there is a type-chain C_2 such that $C_2\,(\pi_2) = \tau$, and, for every statement $x{:}\alpha \in P_2$, $B_2 \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_2\,(\alpha)$.

Since the pairs $\langle P_i, \pi_i \rangle$ are disjoint, the chains C_i do not interfere. Let

$$C_1 * C_2\,(\langle P_1, \pi_1 \rangle) = \langle B_1', \tau \rightarrow \sigma \rangle \text{ and } C_1 * C_2\,(\langle P_2, \pi_2 \rangle) = \langle B_2', \tau \rangle.$$

Let C_g and π be such that

$$\mathit{PP}_{\mathrm{R}_{\mathcal{E}}}(\mathit{Ap}(T_1, T_2)) = \langle C_g\,(\Pi\{P_1, P_2\}), \pi \rangle,$$

then by lemma 12.4.5 there is a C such that

$$C\,(\langle C_g\,(\Pi\{P_1, P_2\}), \pi \rangle) = \langle \Pi\{B_1', B_2'\}, \sigma \rangle.$$

Also: $\forall\, x{:}\alpha \in P_1\ [\ B_1 \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_1\,(\alpha)\ ]\ \&\ \forall\, x{:}\alpha \in P_2\ [\ B_2 \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_2\,(\alpha)\ ] \Rightarrow$
$\forall\, x{:}\alpha \in \Pi\{P_1, P_2\}\ [\ \Pi\{B_1, B_2\} \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C_g * C\,(\alpha)\ ] \Rightarrow$
$\forall\, x{:}\alpha \in C_g\,(\Pi\{P_1, P_2\})\ [\ \Pi\{B_1, B_2\} \vdash_{\mathrm{R}_{\mathcal{E}}} x^{\mathrm{R}}{:}C\,(\alpha)\ ]$.

ii) $T \equiv F(T_1, \ldots, T_n)$. Since $F(T_1, \ldots, T_n)^{\mathrm{R}} = F(T_1^{\mathrm{R}}, \ldots, T_n^{\mathrm{R}})$, by definition 12.2.4 (i.a.3), there are $B_1, \ldots, B_n, \sigma_1, \ldots, \sigma_n$ such that, for every $1 \leq i \leq n$, B_i is

Rank 2 used for $T_i^{\mathrm{R}}{:}\sigma_i$, and $B = \Pi\{B_1, \ldots, B_n\}$. By definition 12.2.2 (i.d) there is a type-chain C_F such that $C_F(\mathcal{E}(F)) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$.
Let $\mathcal{E}(F) = \gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$. By theorem 12.4.7 and definition 12.4.1 (iii) there is a type-chain C_g and $<P_i, \pi_i>$ ($1 \leq i \leq n$, disjoint in pairs) such that, for every $1 \leq i \leq n$, $PP_{\mathrm{R}_\mathcal{E}}(T_i) = <P_i, \pi_i>$, and $PP_{\mathrm{R}_\mathcal{E}}(F(T_1, \ldots, T_n)) = C_g(<\Pi\{P_1, \ldots, P_n\}, \gamma>)$.
Assume that none of the type-variables occurring in $\gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$ occur in any of the pairs $<P_i, \pi_i>$. Then for every $1 \leq i \leq n$ either:

a) $T_i \equiv x$, so $<P_i, \pi_i> = <\{x{:}\varphi\}, \varphi>$ for some φ. Take $C_i = \mathit{unify}_1(\sigma_i, \varphi, \{x{:}\varphi\})$, then $\sigma_i = C_i(\varphi)$. B_i is Rank 2 used for $x^{\mathrm{R}}{:}\sigma_i$, so, by lemma 12.2.5 (i), for every statement $x{:}\alpha \in P_i$, $B_i \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C_i(\alpha)$.

b) $T_i \in \mathit{LHS}$, so by induction there is a type-chain C_i such that $C_i(\pi_i) = \sigma_i$, and, for every statement $x{:}\alpha \in P_i$, $B_i \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C_i(\alpha)$.

In any case, for every $1 \leq i \leq n$ there is a type-chain C_i such that $C_i(\pi_i) = \sigma_i$, and for every statement $x{:}\alpha \in P_i$, $B_i \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C_i(\alpha)$.
Let $C' = C_F * C_1 * \cdots * C_n$. Since for every $1 \leq i \leq n$, $C'(<P_i, \pi_i>) = <B_i, \sigma_i>$, and

$$C'(\gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma) = \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma,$$

by lemma 12.4.6 there is a type-chain C such that

$$C(PP_{\mathrm{R}_\mathcal{E}}(F(T_1, \ldots, T_n))) = <\Pi\{B_1, \ldots, B_n\}, \sigma>.$$

Also: If for every $1 \leq i \leq n$, $x{:}\alpha \in P_i$, $B_i \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C_i(\alpha)$ then
for every $x{:}\alpha \in \Pi\{P_1, \ldots, P_n\}$, $\Pi\{B_1, \ldots, B_n\} \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C_g * C(\alpha)$
so for every $x{:}\alpha \in C_g(\Pi\{P_1, \ldots, P_n\})$, $\Pi\{B_1, \ldots, B_n\} \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C(\alpha)$. □

In the following theorem, we will prove that our solution is sufficient.

Theorem 12.5.4 *The condition is sufficient.* Let $\mathbf{r}$: $Lhs \rightarrow Rhs$ be a safe rewrite rule. Then for every replacement R, basis B and type μ: if $B \vdash_{\mathrm{R}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$, then $B \vdash_{\mathrm{R}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$.

Proof: (Assume $\mu \in \mathcal{T}_2$.) Let $PP_{\mathrm{R}_\mathcal{E}}(Lhs) = <P, \pi>$. Since $\mathbf{r}$ is safe, $P \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}\pi$.

Suppose R is a replacement such that there are B, μ such that $B \vdash_{\mathrm{R}_\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$. By lemma 12.2.5 (ii) we can assume that B is Rank 2 used for $Lhs^{\mathrm{R}}{:}\mu$, so by lemma 12.5.3 there is a type-chain C such that $C(\pi) = \mu$ & $\forall\, x{:}\alpha \in P$ [$B \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C(\alpha)$].

By lemma 12.3.4 (i) $C(P) \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}C(\pi)$, so $C(P) \vdash_{\mathrm{R}_\mathcal{E}} Rhs{:}\mu$, and

$$\forall\, x{:}\alpha \in P\ [\ B \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}C(\alpha)\ ] \Rightarrow \forall\, x{:}\alpha \in C(P)\ [\ B \vdash_{\mathrm{R}_\mathcal{E}} x^{\mathrm{R}}{:}\alpha\ ].$$

So by lemma 12.5.2 $B \vdash_{\mathrm{R}_\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$. □

In the previous chapter it was also shown that the condition formulated in that chapter is necessary. This result was achieved by extending the set of types with type constants, and for every rewrite rule that was not safe, a specific replacement that gives the counterexample was created. In this construction it was used that every type σ can be inhabited in a trivial way: just pick a constant Q, not already used, and assume that $\mathcal{E}(Q) = \sigma$.

In the notion of type assignment as defined in this chapter we cannot give this construction, because we cannot let every type be trivially inhabited. The environment of this chapter returns types in $\mathcal{T}_2$, and a function symbol F can only have an intersection type $\alpha \cap \beta$ if there exists

a type chain C such that $C(\mathcal{E}(F)) = \alpha \cap \beta$. This means that we cannot show that there is, for example, a function symbol that can be assigned the type $\varphi_0 \cap (\varphi_1 \rightarrow \varphi_0)$.

12.6 *Implementation aspects of Rank 2 type assignment in ATRS's*

The results of this chapter could be used to implement a type-check algorithm for ATRS's. It should be noted that the notion of type assignment as defined here is really a *type-check* system. Take, for example, the rewrite rule

$$I(x) \rightarrow x.$$

The smallest environment possible for this rule maps I to the type $1 \rightarrow 1$. We have shown that all types that can be obtained from this one by substitution or duplication are allowed for the rewrite rule, but the set of types that can be used is larger that just the set obtained by those operations. For example, the rewrite rule can also be typed using the type $1 \cap 2 \rightarrow 1$. To obtain a type-inference algorithm an operation should be inserted that allows of more specific types than generated by substitution and duplication. Take, for example, the rewrite rules:

$$F(x) \rightarrow x$$
$$F(x) \rightarrow Ap(x, x).$$

A type-inference algorithm could, for example, type both alternatives separately and combine the results found. For the first rule it would find $\mathcal{E}(F) = 1 \rightarrow 1$, for the second $\mathcal{E}(F) = (2 \rightarrow 3) \cap 2 \rightarrow 3$. These types cannot be unified using operations defined in this chapter. To obtain the correct type for F, $6 \cap (5 \rightarrow 6) \cap 5 \rightarrow 6$, an operation is needed that inserts extra types in the left hand side of the top arrow type constructor. It is not at all easy to define such an operation that is sound in all cases.

Another thing to notice is that, although type assignment (and type-checking) by the notion of type assignment defined here is decidable, the complexity of type-checking is bigger than for a system based on Curry-types. The biggest problem will arise when checking a type provided for a function symbol. Suppose $Lhs \rightarrow Rhs$ is a rewrite rule. One way to implement type-checking for this rule would be to construct the principal pair $<P, \pi>$ for the term Lhs, and to try and type Rhs using this pair. Let $\sigma_1 \cap \cdots \cap \sigma_n$ be the type assigned to the term-variable x in P. Then, for every occurrence of x in Rhs, some selection of the types in $\sigma_1 \cap \cdots \cap \sigma_n$ should be made. In the worst case, the number of possibilities that must be tried is huge: 2^n. There are some more efficient ways to type-check a rule, but the complexity is still exponential. However, in every day programming life n will rarely be larger than 2.

The concept of overloading in programming languages is normally used to express that different objects (typically procedures) can have the same identifier. (For another approach to overloading, see [Castagna *et al.* '92].) At first sight, it seems to be nothing but a tool to obtain programming convenience, but the implementational aspects of languages with overloading are not at all trivial. In functional programming languages, functions are *first order citizens*, which means that they can be handled as any object, like, for example, numbers. In particular, a function can be passed as an argument to another one, or could constitutes its result. Especially in the first case it can occur that, at compile time, it cannot be decided which of the several function definitions for an overloaded identifier is really needed. If this decision cannot be made, the compiler should generate code that contains all possible functions, and some kind of case-construct that makes it possible to select at runtime which is the code to use. For reasons of efficiency, and to avoid run-time checks on function types, it seems natural to allow of overloaded objects only if at compile time, it can be decided which of the different function definitions is meant, since then the compiler can decide for every occurrence of an overloaded symbol which of the several function definitions should be linked into the object code.

The intersection type constructor is a good candidate to express this kind of overloading (see also [Pierce '91]). It seems natural to say, for example, that the type for addition *Add* is $(\mathsf{int}\to\mathsf{int}\to\mathsf{int})\cap(\mathsf{real}\to\mathsf{real}\to\mathsf{real})$. Incorporating the notion of overloading into a formal system for type assignment as defined in this chapter implies that the restriction on types that can be provided by a Rank 2 environment should be dropped; in such a formalism, types provided by the environment should be in $\mathcal{T}_{\mathrm{R}}$, not just $\mathcal{T}_2$.

Selection of one of the function definitions for an overloaded identifier can be accomplished by defining, similar to definition 12.2.2 (i.d) how a type for a function symbol can be obtained from the one provided by the environment, in the following way:

> If the type assigned to a function node containing F is τ, then there is a $\sigma \in \mathcal{T}_2$ such that $\mathcal{E}(F) \leq_{\mathrm{R}} \sigma$, and a type chain C such that $C(\sigma) = \tau$.

This selection will then be reflected in the way intersection types are unified. Since only *one* of the types in an 'overloaded' type can be used, the unification should try to unify the demanded type with *each individual type* occurring in the provided type. Using this definition, the notion of 'principal pair' becomes slightly more complicated. This is best explained by discussing the implementation of the type-checker that is looking for such a pair. Take the function *foldr* that is defined by

$$\begin{array}{ll} \mathit{foldr}\ f\ i\ [\,] & = i \\ \mathit{foldr}\ f\ i\ (a{:}b) & = f\ a\ (\mathit{foldr}\ f\ i\ b) \end{array}$$

and can be typed by $(1\to2\to2)\to2\to[1]\to2$. If we take the term '*foldr Add* 1 [2, 3, 4]', then it is clear that this term should be typeable by the type int. When constructing the type assignment for this term, the subterm '*foldr Add*' is typed. As such, the type needed for *Add* cannot be uniquely determined for this term: it is the second argument of *foldr* that forces the selection. Since there is a chance of success, the type-checker should postpone the decision to reject the term, and consider both possibilities simultaneously. This means that, formally, the term *foldr Add* has *two* principal types (which is not the same as saying that its principal type is an intersection).

Summary

> *"When you know how to do it, Zuck, it's very easy.*
> *It's like riding a bike."*
> *"No, no, people tend to devalue the sophistication of their own special field.*
> *It's easy only because of all you know."*
>
> – Philip Roth, *The Anatomy Lesson* (1983)

This thesis deals with a number of different notions of type assignment that use intersection types, and it can roughly be divided in three parts. The first part briefly discusses definitions and properties of a number of type assignment systems that were developed in the past by other authors. The second and third part contain results of the author's research; notions of type assignment defined for the lambda calculus are presented in the second part and for Term Rewriting Systems in the third part.

In the first chapter, Curry's Type Assignment System is discussed, the oldest en most elementary type assignment system, in which type assignment is decidable and all typeable terms are strongly normalizable. This system has the principal type property: all possible types for a term can be generated from a specific type, by means of operations chosen from a predefined collection. For Curry's system this collection consists entirely of type substitutions.

In chapter two a number of intersection type assignment systems is discussed. The Coppo-Dezani Type Assignment System (CD) in section 2.1 is a generalization of Curry's system in which it is possible to type term-variables with more than one type. If the set of terms is limited to those from the λI-calculus, type assignment in this system is closed for beta-equality. In section 2.2 three different Coppo-Dezani-Venneri Type Assignment Systems (CDV) are discussed; the main difference between these and the CD-system is the addition of the type constant ω. These systems are closed for beta-equality for the full lambda calculus, and, therefore, type assignment in these systems is undecidable. It is possible to characterize the normalizing terms and the terms that have a head normal form by means of the assignable types. For one of these systems the principal type property is proved; in order to do so, restrictions have to be made. Next to type substitutions the collection of operations allowed for this system contains expansions on types as well.

In section 2.3 the Barendregt-Coppo-Dezani Type Assignment System (BCD) is discussed, a generalization of the CDV-systems in which it is possible to give the same characterization of typeable terms as in the CDV-systems. The generalization made consists of treating '$\cap$' as a normal type constructor and introducing a partial type inclusion relation ($\leq$), and it is made in order to prove completeness of type assignment. The set of typeable terms is exactly the same as in the CDV-systems, only the set of types that can be assigned to a term is significantly bigger. The construction of a filter lambda model and the definition of a map from types to elements of this model (a *simple type interpretation*) make the proof of completeness possible: if the interpretation of the term M is an element of the interpretation of the type σ, then M is typeable with σ. Also, the BCD-system has the principal type property. Next to type substitutions and expansions on types the collection of operations allowed for the BCD-system contains rises as well.

In chapter three two notions of type assignment are discussed, the Milner Type Assignment System and the Mycroft Type Assignment System, that are defined for a primitive applicative (functional) programming language. They differ from Curry's system in the extension of the lambda calculus with new syntactic constructors that enable to express recursion and polymorphism. Both systems have the principal type property. Type assignment in Milner's system is decidable, in Mycroft's system it is not.

The second part of this thesis starts with chapter four, which contains the presentation of the Strict Intersection Type Assignment System. It is a slight restriction of one of the CDV-systems and at the same time a subsystem of the BCD-system: the set of typeable terms is the same as in both the other systems, and typeability in the strict system is therefore

undecidable too. All possible derivations in the BCD-system can be built by first constructing a derivation in the strict system and to use the type inclusion relation $\leq$ afterwards. This system gives rise to the definition of a strict filter lambda model, and completeness of type assignment is proved using an *inference* type interpretation instead of a simple type interpretation. In chapter six it is proved that the strict system too has the principal type property. The collection of operations allowed for the strict system consists of, next to type substitutions and expansions on types, liftings as well, which is a restricted kind of rise.

Chapter five contains the presentation of the Essential Intersection Type Assignment System, a small extension of the strict system and also a restriction of the BCD-system. This notion of type assignment is equivalent to BCD-type assignment, and for this system it is even possible to prove completeness of type assignment using a simple type interpretation. The essential system also has the principal type property; the collection of operations allowed for is the same as for the strict system; only the order in which operations can be used is different.

Chapter seven contains the presentation of a type assignment system that, like the strict system, is a restriction of the BCD-system. The restriction consists of the elimination of the type constant ω, and gives rise to the definition of a filter λI-model. Completeness of type assignment for terms from the λI-calculus is proved using this model, using a simple type interpretation. Since the set of terms from the complete λK-calculus that is typeable in this system is exactly the set of strongly normalizable terms, type assignment in this system is undecidable as well.

In chapter eight a Rank 2 Intersection Type Assignment System is presented, that is based on the CD-system and has a strong similarity with Milner's system. This system too has the principal type property, and the operations allowed of are type substitutions and type duplications, i.e. a simple kind of expansion on types. Type assignment in this system is decidable.

The third part of this thesis starts with the definition of Applicative Term Rewriting Systems in chapter nine, a kind of term rewriting systems that has a special predefined function symbol (Ap). Terms in this class of rewriting systems are not only constructed by means of application, but also by supplying a function symbol that has a certain arity with sufficient arguments.

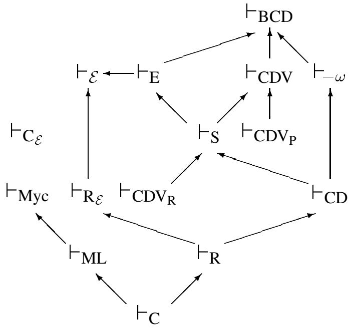
The first notion of type assignment on the tree representation of these rewriting systems is defined in chapter ten, and is based on the Essential Type Assignment System for the lambda calculus and Milner's way of dealing with recursion. The definition of type assignment consists of defining an environment that returns a type for every function symbol, and the formulation of conditions that types assigned to the tree representation of terms and rewrite rules have to meet. Types that can be assigned to function symbols in terms can be obtained from the types supplied by the environment: they can be obtained by application of the operations of type substitutions, expansions on types, or liftings as defined for the strict system and the essential system for the lambda calculus. In general, the notion of type assignment in Term Rewriting Systems does not have the property that types assignable to terms are also assignable to terms that are obtained by applying a rewrite rule (the *subject-reduction property*).

In chapter eleven a notion of type assignment on Left Linear Applicative Term Rewriting Systems is defined in the same way as in chapter ten. This one, however, uses Curry types and Mycroft's way of dealing with recursion and is, therefore, not a restriction of the system in that chapter. This notion of type assignment has the principal type property, and, using the principal type of the left hand side of a rewrite rule, a condition is formulated that is necessary and sufficient to obtain the subject-reduction property.

The thesis is concluded with the presentation of a notion of type assignment on Applicative Term Rewriting Systems that uses Rank 2 intersection types, and is a real restriction of the system defined in chapter ten. This notion of type assignment too has the principal type property, and, using the principal type of the left hand side of a rewrite rule, a condition is formulated that is sufficient to obtain the subject-reduction property.

Comparing the different notions of type assignment

In this thesis fifteen different notions of type assignment are defined. As mentioned in the introduction, there exists a very precise relation between the various systems.



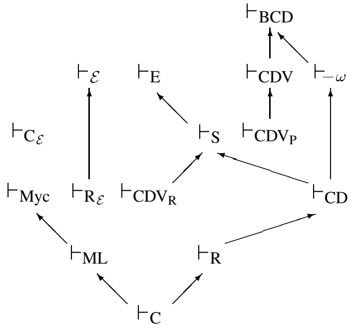
There is an arrow in this picture from one turnstyle to another if the system at the end of the arrow is an extension of the system at the start: if $B \vdash M{:}\sigma$ holds in the system at the start of the arrow, then $B \vdash M{:}\sigma$ holds in the system at the end; the latter M is sometimes obtained from the former one by bracket-abstraction. To summarize the general differences and similarities between the systems, we emphasize the following:

- The set of typeable terms is the same in the systems $\vdash_{\mathrm{CD}}$ and $\vdash_{-\omega}$, and in the systems $\vdash_{\mathrm{CDV_R}}$, $\vdash_{\mathrm{CDV_P}}$, $\vdash_{\mathrm{CDV}}$, $\vdash_{\mathrm{S}}$, $\vdash_{\mathrm{E}}$ and $\vdash_{\mathrm{BCD}}$, although, of course, the set of types assignable to those terms differ.
- Type assignment is decidable in the systems $\vdash_{\mathrm{C}}$, $\vdash_{\mathrm{R}}$, $\vdash_{\mathrm{ML}}$, $\vdash_{\mathrm{C}_{\mathcal{E}}}$, and $\vdash_{\mathrm{R}_{\mathcal{E}}}$, and is undecidable in the systems $\vdash_{\mathrm{CD}}$, $\vdash_{\mathrm{CDV_R}}$, $\vdash_{\mathrm{CDV_P}}$, $\vdash_{\mathrm{CDV}}$, $\vdash_{\mathrm{S}}$, $\vdash_{\mathrm{E}}$, $\vdash_{-\omega}$, $\vdash_{\mathrm{BCD}}$, $\vdash_{\mathrm{Myc}}$, and $\vdash_{\mathcal{E}}$.
- For the systems $\vdash_{\mathrm{C}}$, $\vdash_{\mathrm{R}}$, $\vdash_{\mathrm{CDV_P}}$, $\vdash_{\mathrm{S}}$, $\vdash_{\mathrm{E}}$, $\vdash_{-\omega}$, $\vdash_{\mathrm{BCD}}$, $\vdash_{\mathrm{ML}}$, $\vdash_{\mathrm{Myc}}$, $\vdash_{\mathrm{C}_{\mathcal{E}}}$, and $\vdash_{\mathrm{R}_{\mathcal{E}}}$ the principal type property is proved.

As mentioned in chapters four and five, a type assignment system is called *conservative* over another if the following is true:

> Suppose all types occurring in B and σ are in the set of types of the latter system. Then, $B \vdash M{:}\sigma$ will hold in the former system if and only if $B \vdash M{:}\sigma$ holds in the latter.

Most of the extensions in the picture above are not conservative extensions. In the following picture we will show the true extensions that are not conservative.



We will illustrate this picture by a list of examples. The statements mentioned are derivable in the first system, and not in the second. Let $D = \lambda x.xx$, $I = \lambda x.x$, $S = \lambda xyz.xz(yz)$, $K = \lambda xy.x$, $R =$ Fix $r.\lambda xy.(r\,(r\,y\,(\lambda ab.a))\,x)$, $M =$ (let $x = (\lambda y.y)$ in (xx)) ($\sim DI$).

$\vdash_{\mathrm{BCD}}$ over $\vdash_{\mathrm{CDV}}$:	$I{:}(\sigma\to\tau)\to\sigma\cap\rho\to\tau$
$\vdash_{\mathrm{CDV}}$ over $\vdash_{\mathrm{CDV_P}}$:	$K{:}\sigma\to\tau\to\sigma$
$\vdash_{\mathrm{BCD}}$ over $\vdash_{-\omega}$:	$SK{:}\tau\to\sigma\to\sigma$
$\vdash_{-\omega}$ over $\vdash_{\mathrm{CD}}$:	$I{:}(\sigma\to\tau)\to\sigma\cap\rho\to\tau$
$\vdash_{\mathrm{CD}}$ over $\vdash_{\mathrm{R}}$:	$ID{:}(\sigma\to\tau)\cap\sigma\to\tau$
$\vdash_{\mathrm{R}}$ over $\vdash_{\mathrm{C}}$:	$DI{:}\sigma\to\sigma$
$\vdash_{\mathrm{E}}$ over $\vdash_{\mathrm{S}}$:	$I{:}(\sigma\to\tau)\to\sigma\cap\rho\to\tau$
$\vdash_{\mathrm{S}}$ over $\vdash_{\mathrm{CD}}$:	$K{:}\sigma\to\omega\to\sigma$
$\vdash_{\mathrm{S}}$ over $\vdash_{\mathrm{CDV_R}}$:	$K{:}\sigma\to\tau\to\sigma$
$\vdash_{\mathrm{Myc}}$ over $\vdash_{\mathrm{ML}}$:	$RKI{:}\varphi$
$\vdash_{\mathrm{ML}}$ over $\vdash_{\mathrm{C}}$:	$M{:}\sigma\to\sigma$
$\vdash_{\mathcal{E}}$ over $\vdash_{\mathrm{R}_{\mathcal{E}}}$:	$I_0{:}(\sigma\to\tau)\to\sigma\cap\rho\to\tau$

In the following we will give a list of examples to show the differences between incomparable systems. As before, the statements mentioned are derivable in the first system, and not in the second. Let $E = \lambda xy.xy$, $R'(x,y) \rightarrow R'(R'(y,K_0),x)$, $D'(x) \rightarrow Ap(x,x)$

$\vdash_{\mathrm{ML}}$ versus $\vdash_{\mathrm{R}}$:	$M{:}\sigma\rightarrow\sigma$
$\vdash_{\mathrm{R}}$ versus $\vdash_{\mathrm{ML}}$:	$D{:}(\sigma\rightarrow\tau)\cap\sigma\rightarrow\tau$
$\vdash_{\mathrm{C}_{\mathcal{E}}}$ versus $\vdash_{\mathrm{R}_{\mathcal{E}}}$:	$R'(K_0,I_0){:}\varphi$
$\vdash_{\mathrm{R}_{\mathcal{E}}}$ versus $\vdash_{\mathrm{C}_{\mathcal{E}}}$:	$D'(I_0){:}\sigma\rightarrow\sigma$
$\vdash_{\mathrm{CDV_R}}$ versus $\vdash_{\mathrm{CD}}$:	$SK{:}\tau\rightarrow\sigma\rightarrow\sigma$
$\vdash_{\mathrm{CD}}$ versus $\vdash_{\mathrm{CDV_R}}$:	$E{:}(\sigma\rightarrow\tau)\rightarrow\sigma\cap\rho\rightarrow\tau$
$\vdash_{\mathrm{E}}$ versus $\vdash_{\mathrm{CDV}}$:	$I{:}(\sigma\rightarrow\tau)\rightarrow\sigma\cap\rho\rightarrow\tau$
$\vdash_{\mathrm{CDV}}$ versus $\vdash_{\mathrm{E}}$:	$I{:}\sigma\rightarrow\omega$
$\vdash_{\mathrm{CDV_P}}$ versus $\vdash_{\mathrm{CD}}$:	$K{:}\sigma\rightarrow\omega\rightarrow\sigma$
$\vdash_{\mathrm{CD}}$ versus $\vdash_{\mathrm{CDV_P}}$:	$I{:}(\sigma\rightarrow\tau)\rightarrow\sigma\cap\rho\rightarrow\tau$
$\vdash_{\mathrm{CDV_P}}$ versus $\vdash_{\mathrm{S}}$:	$I{:}\sigma\rightarrow\omega$
$\vdash_{\mathrm{S}}$ versus $\vdash_{\mathrm{CDV_P}}$:	$K{:}\sigma\rightarrow\tau\rightarrow\sigma$

Using Intersection Types for Term Graph Rewriting Systems

As was remarked in the introduction, one of the reasons to use intersection types is that we aim to extend the notion of type assignment defined here to one for Term Graph Rewriting Systems. Those systems, presented in [Barendregt *et al.* '87], are a restricted kind of graph rewriting systems. In general, graph rewriting is defined via presenting graph rewrite rules: each rule consists of a left hand side graph, an optional right hand side, and one or more redirections. The main restriction made from general graph rewriting systems to Term Graph Rewriting Systems is to allow for only *one* redirection in a rewrite rule. A *term graph rewrite rule* is a pair of rooted graphs, and the only rewrite that is permitted by this rule is that of replacing a graph that matches the left hand side graph entirely by a copy of the right hand side in which term-variables are replaced. So this kind of graph rewriting corresponds to term rewriting, but allows of sharing and cyclic structures.

Term graphs can be obtained from terms through lifting them to graphs. This lifting consists of writing terms as trees and of sharing variables that occur more than once in the term that is lifted. Term graph rewrite rules are obtained from term rewrite rules in very much the same way: the left- and right hand side terms of every term rewrite rule are lifted to term graphs, and the nodes that represent variables occurring in both terms are shared. If a variable appears in both the left and right hand side, this operation will generate a connected graph. Of course it is also possible to define term graphs and term graph rewrite rules directly, without first taking a term or term rewrite rule and lifting it.

The extension of the notion of type assignment as presented in this thesis for Applicative Term Rewriting Systems to a system for Term Graph Rewriting Systems is straightforward. Problems only arise when properties of such a notion should be proved, like, for example, the principal type property. Since graphs cannot be defined inductively (not even the subclass of term graphs can), it is not possible to define the principal pair for a graph inductively as done in, for example, chapter twelve. However, there exists an easy, straightforward translation of term graphs to (possibly infinite) trees that is called unraveling, and type assignment on term graphs could be defined via type assignment on the (unravelled) trees the same way as in this thesis. The concept of sharing itself causes no difficulties, since a shared node can be typed in the same way as in the corresponding tree, and when the graph is reconstructed, types assigned to corresponding nodes should be intersected. In an implementation this will imply that the type assignment algorithm need not care about types that are already assigned to a node when it is visited.

The only problem arises when the graph itself is allowed to have a cyclic structure, which causes the unraveling to generate an infinite tree. Then it is possible that the (infinite number of) copies of a node are all typed with different types, thus creating an intersection over an infinite number of types for the type assignment to the term graph. One solution for this problem would be to detect cyclic structures while unraveling. Cyclic nodes could then be typed with, at the most, the same number of types as the number of in-going edges.

Conclusion

There is a great number of intersection type assignment systems defined in this thesis. For the greater part, only the syntactical aspects of type assignment are investigated, like the set of typeable terms, the principal type property, and decidability of type assignment.

We have seen that Curry's Type Assignment System has the principal type property, type assignment is decidable, and all typeable terms are strongly normalizable; unfortunately, however, the set of typeable terms is rather small.

The various intersection type assignment systems (like the Coppo-Dezani system, the three different Coppo-Dezani-Venneri systems, and the Barendregt-Coppo-Dezani system) are all far more general type assignment systems than Curry's system. The set of terms typeable in the latter two with a type different from the type constant ω is just the set of terms having a head normal form, and the set of terms typeable without ω in basis and conclusion is the set of normalizable terms. Moreover, the CDV- and BCD-systems are closed for β-equality. The results of chapter seven show that, if ω is not used at all (like in the CD-system), the set of terms typeable is exactly the set of strongly normalizable terms. We have seen the development of those intersection type assignment systems of which the BCD-system was the final one, and it is also the most frequently used and quoted.

From the functional languages point of view the disadvantages of these intersection systems are, however, great. Type assignment in all those systems is undecidable, and especially the BCD-system is very general: because of the $\leq$-relation on types it is not easy to really understand the relation between all types assignable to a term. The Strict Type Assignment System, which is defined as a restriction of the BCD-system a couple of years after the BCD-paper was published, is in fact closer to the CDV-systems than to the

BCD-system. It is strict in more than one aspect: it is strict in the sense that it has just enough power to type all CDV- and BCD-typeable terms, and is restricted since it is not closed for η-reduction.

The results of chapters four and six actually show that the extension made in the CDV-systems to obtain the BCD-system was to general. To obtain the major results of the BCD-paper, it is sufficient to treat ω not as a type-variable as done in the CDV-systems but as an empty intersection, and to use a relation on types that is induced in a natural way by intersections and by defining ω as the universal type. To prove completeness of type assignment it is sufficient to use the strict filter lambda model (Engeler's model $\mathcal{D}_A$) and the inference type semantics. The results of chapter five for the Essential Type Assignment System show that even if the simple type semantics are preferred, the completeness result can be obtained via a filter lambda model that is defined using a relation on types that is just a small extension of the relation defined for the strict system.

Even the fact that the BCD-system has the principal type property does not speak in its favour, since both the strict system and the essential one have the same property.

Implementation of type inference algorithms using intersection types is complicated. Even implementing a partial type assignment algorithm that is only supposed to terminate on terms that have a normal form is not as straightforward as in other systems; the main source of the problems is the operation of expansion. There are various ways of restricting intersection type assignment systems in order to obtain decidable type assignment. The one studied in this thesis, the Rank 2 system, is the smallest restriction imaginable to have intersection types. It has the following advantages: the complexity of this system is manageable, it has the principal type property, type assignment is decidable, it is close to Milner's notion of type assignment, it is sufficiently powerful to express overloading operators, and it can type a larger class of objects. Therefore, it seems reasonable to use this notion of type assignment in (functional) programming languages.

The notions of type assignment on Applicative Term Rewriting Systems as defined in this thesis use the approach for similar systems for the lambda calculus. They show that it is possible to define such a notion for reduction systems that are more general than that calculus, and that are more close to (functional) programming languages. The Applicative Term Rewriting Systems differ from these languages in that there are almost no restrictions on the structure of possible left hand sides of function definitions. The results of chapter ten show that defining such a notion is straightforward, but that, unfortunately, the subject-reduction property is lost. Since this property is vital for reduction systems, two less general notion of type assignment are defined in chapters eleven and twelve. The first is very close to notions of type assignment used for functional programming languages at this moment and a condition is formulated that typeable rewrite rules should satisfy in order to obtain subject reduction. In these systems type assignment is decidable, both these notions of type assignment have the principal type property, and, using the principal type of the left hand side of a rewrite rule, a condition is formulated that is sufficient to obtain the subject-reduction property.

References

S. van Bakel, S. Smetsers, and S. Brock.

'92 Partial Type Assignment in Left Linear Applicative Term Rewriting Systems. In J.-C. Raoult, editor, *Proceedings of CAAP '92. 17th Colloquim on Trees in Algebra and Programming, Rennes, France*, volume 581 of *Lecture Notes in Computer Science*, pages 300–321. Springer-Verlag, 1992.

S. van Bakel.

'88 Derivations in Type Assignment Systems. Master's thesis, University of Nijmegen, 1988.

'90 A first attempt in polymorphic type assignment using intersection types. The problem of unification. Technical Report 90-9, Department of Computer Science, University of Nijmegen, 1990.

'91 Principal type schemes for the Strict Type Assignment System. Technical Report 91-6, Department of Computer Science, University of Nijmegen, 1991.

'92a Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102:135–163, 1992.

'92b Essential Intersection Type Assignment. Technical Report 92-28, Department of Computer Science, University of Nijmegen, 1992.

'92c Partial Intersection Type Assignment of Rank 2 in Applicative Term Rewriting Systems. Technical Report 92-03, Department of Computer Science, University of Nijmegen, 1992.

'93 Partial Intersection Type Assignment in Applicative Term Rewriting Systems. In *Proceedings of TLCA '93. International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993. To appear.

H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini.

'83 A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep.

'87 Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, volume 259-II of *Lecture Notes in Computer Science*, pages 141–158. Springer-Verlag, 1987.

H. Barendregt.

'84 *The lambda calculus: its syntax and semantics*. North-Holland, Amsterdam, revised edition, 1984.

T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer.

'87 Clean - A Language for Functional Graph Rewriting. In *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, volume 274 of *Lecture Notes in Computer Science*, pages 364–368. Springer-Verlag, 1987.

G. Castagna, G. Ghelli, and G. Longo.

'92 A Calculus for Overloaded Functions with Subtyping. Technical Report LIENS-92-4, Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1992.

M. Coppo and M. Dezani-Ciancaglini.

'80 An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame, Journal of Formal Logic*, 21(4):685–693, 1980.

M. Coppo and P. Giannini.

'92 A complete type inference algorithm for simple intersection types. In J.-C. Raoult, editor, *Proceedings of CAAP '92. 17th Colloquim on Trees in Algebra and Programming, Rennes, France*, volume 581 of *Lecture Notes in Computer Science*, pages 102–123. Springer-Verlag, 1992.

M. Coppo, M. Dezani-Ciancaglini, and B. Venneri.

'80 Principal type schemes and λ-calculus semantics. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry, Essays in combinatory logic, lambda-calculus and formalism*, pages 535–560. Academic press, New York, 1980.

'81 Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo.

'84 Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium 82*, pages 241–262, Amsterdam, 1984.

M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi.

'87 Type Theories, Normal Forms and D_∞-Lambda-Models. *Information and Computation*, 72(2):85–116, 1987.

H.B. Curry and R. Feys.

'58 *Combinatory Logic*. volume 1. North-Holland, Amsterdam, 1958.

H.B. Curry.

'34 Functionality in combinatory logic. In *Proc. Nat. Acad. Sci. U.S.A.*, volume 20, pages 584–590, 1934.

L. Damas and R. Milner.

'82 Principal type-schemes for functional programs. In *Proceedings 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

L.M.M. Damas.

'85 *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, 1985. Thesis CST-33-85.

N. Dershowitz and J.P. Jouannaud.

'90 Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 245–320. North-Holland, 1990.

M. Dezani-Ciancaglini and J.R. Hindley.

'92 Intersection types for combinatory logic. *Theoretical Computer Science*, 100:303–324, 1992.

M. Dezani-Ciancaglini and I. Margaria.

'84 F-semantics for intersection type discipline. In G. R. Kahn, D. B. Macqueen, and G. Plotkin., editors, *Sematics of data types. International symposium Sophia - Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 279–300. Springer-Verlag, 1984.

'86 A characterisation of F-complete type assignments. *Theoretical Computer Science*, 45:121–157, 1986.

M. Dezani-Ciancaglini, F. Honsell, and S. Ronchi della Rocca.

'86 Models for theories of functions strictly depending on all their arguments. Stanford ASL Summer Meeting '85. *Journal of Symbolic Logic*, 51 (2):845–846, 1986. abstract.

E. Engeler.

'81 Algebras and combinators. *Algebra universalis*, 13(3):389–392, 1981.

K. Futatsugi, J. Goguen, J.P. Jouannaud, and J. Meseguer.

'85 Principles of OBJ2. In *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

P. Giannini and S. Ronchi della Rocca.

'88 Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 61–70, 1988.

J.Y. Girard.

'86 The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

R. Hindley and G. Longo.

'80 Lambda calculus models and extensionality. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 26:289–310, 1980.

J.R. Hindley.

'69 The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

'82 The simple semantics for Coppo-Dezani-Sallé type assignment. In M. Dezani and U. Montanari, editors, *International symposium on programming*, volume 137 of *Lecture Notes in Computer Science*, pages 212–226. Springer-Verlag, 1982.

'83 The Completeness Theorem for Typing λ-terms. *Theoretical Computer Science*, 22(1):1–17, 1983.

F. Honsell and S. Ronchi della Rocca.

'84 Models for theories of functions strictly depending on all their arguments. Internal report, Department of Computer Science, Turin, Italy, 1984.

B. Jacobs, I. Margaria, and M. Zacchi.

'92 Filter Models with Polymorphic Types. *Theoretical Computer Science*, 95:143–158, 1992.

A.J. Kfoury and J. Tiuryn.

'89 Typeability in the polymorphic λ-calculus of Rank 2 is decidable. 1989. Preliminary Report.

A.J. Kfoury, J. Tiuryn, and P. Urzyczyn.

'88 A proper extension of ML with an effective type-assignment. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 58–69, San Diego, California, 1988.

'90 ML Typability is Dexptime-Complete. In A. Arnold, editor, *Proceedings of CAAP '90. 15th Colloquim on Trees in Algebra and Programming, Copenhagen, Denmark*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer-Verlag, 1990.

J.W. Klop.

'90 Term Rewriting Systems. Report CS-R9073, Centre for Mathematics and Computer Science, Amsterdam, 1990.

J-L. Krivine.

'90 *Lambda-calcul types et modèles*. Masson, Paris, 1990.

D. Leivant.

'83 Polymorphic Type Inference. In *Proceedings 10th ACM Symposium on Principles of Programming Languages, Austin Texas*, pages 88–98, 1983.

I. Margaria and M. Zacchi.

'90 Principal Typing in a $\forall\cap$-Discipline. Internal report, Dipartimento di Informatica dell' Università de Torino, 1990.

R. Milner.

'78 A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

J.C. Mitchell.

'88 Polymorphic Type Inference and Containment. *Information and Computation*, 76:211–249, 1988.

A. Mycroft.

'84 Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming, Toulouse*, volume 167 of *Lecture Notes*

Computer Science, pages 217–239. Springer-Verlag, 1984.

E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer.

'91 Concurrent Clean. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, volume 506-II of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, 1991.

V. van Oostrom.

'90 Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

F. Pfenning.

'88 Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the 1988 conference on LISP and Functional Programming Languages*, volume 201 of *Lecture Notes in Computer Science*, pages 153–163. Springer-Verlag, 1988.

B.C. Pierce.

'91 *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, 1991. CMU-CS-91-205.

G.D. Plotkin and M.B. Smyth.

'78 The category-theoretic solution of recursive domain equations. DAI Research Report 60, University of Edinburgh, Scotland, 1978.

G. Pottinger.

'80 A type assignment for the strongly normalizable λ-terms. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry, Essays in combinatory logic, lambda-calculus and formalism*, pages 561–577. Academic press, New York, 1980.

J.A. Robinson.

'65 A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

S. Ronchi della Rocca and B. Venneri.

'84 Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

S. Ronchi della Rocca.

'88 Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

P. Sallé.

'78 Une extension de la théorie des types. In G. Ausiello and C. Böhm, editors, *Automata, languages and programming. Fifth Colloquium, Udine, Italy*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410. Springer-Verlag, 1978.

D.A. Turner.

'85 Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

Index

Types

Relations on types (pairs)

Type assignment systems

Notions of bases

Pairs

Operations on types (pairs)

Term Rewriting Notions

Auxiliary notions

Samenvatting

Dit proefschrift behandelt een aantal verschillende noties van typering die gebruik maken van intersectietypes en kan ruwweg worden opgedeeld in drie onderdelen. Het eerste deel behandelt in het kort de definities en eigenschappen van een aantal typeringssystemen die door andere auteurs in het verleden zijn ontwikkeld. Het tweede en derde deel bevatten het resultaat van eigen onderzoek en presenteren noties van typering gedefinieerd voor de lambda calculus in het tweede deel en noties voor termherschrijfsystemen in het derde deel.

In het eerste hoofdstuk wordt Curry's typeringssysteem besproken, het oudste en meest simpele typeringssysteem, waarin typering beslisbaar is en alle typeerbare termen strek normaliserend zijn. Dit systeem heeft de voornaamste typeëigenschap (*principal type property*): alle mogelijke types voor een term kunnen uit een zeker gegeven type worden gegenereerd door middel van operaties gekozen uit een vooraf gedefinieerde collectie. Voor Curry's systeem bestaat deze collectie volledig uit typesubstituties.

In hoofdstuk twee worden een aantal intersectietyperingssystemen besproken. Het Coppo - Dezani systeem (CD) in sectie 2.1 is een generalisatie van Curry's systeem waarin het mogelijk is termvariabelen met meer dan één type te typeren. Wanneer de verzameling termen beperkt wordt tot die uit de λI-calculus, is typering in dit systeem gesloten voor beta-gelijkheid. In sectie 2.2 worden drie verschillende Coppo - Dezani - Venneri systemen (CDV) besproken; het belangrijkste verschil tussen deze en het CD-systeem is de toevoeging van de typeconstante ω. Deze systemen zijn gesloten voor beta-gelijkheid voor de volledige lambda calculus, en daarmee is typering binnen deze systemen onbeslisbaar. Het is mogelijk de normaliserende termen en de termen die een kop-normaal vorm (*head normal form*) hebben, te karakteriseren door middel van de toewijsbare types. Voor één van deze systemen wordt de voornaamste typeëigenschap bewezen; daartoe moeten echter aan het systeem beperkingen worden opgelegd. De collectie van toegestane operaties voor dit systeem bestaat buiten typesubstituties ook uit expansies op types.

In sectie 2.3 wordt het Barendregt - Coppo - Dezani systeem (BCD) besproken, een generalisatie van de CDV-systemen waarin het mogelijk is eenzelfde karakterisering van typeerbare termen te geven. De uitgevoerde generalisatie bestaat uit het behandelen van '$\cap$' als een volwaardige typeconstructor en het introduceren van een partiële type inclusie relatie ($\leq$) en wordt gemaakt teneinde volledigheid van typetoekenning te bewijzen. De verzameling van typeerbare termen is exact gelijk aan die van de CDV-systemen, alleen de verzameling van de types die kunnen worden toegekend aan een term is beduidend groter. De constructie van een filter lambda model en de definitie van een afbeelding van types naar elementen van dit model (een *simpele typeïnterpretatie*) maakt het bewijs voor volledigheid mogelijk: als de interpretatie van de term M een element is van de interpretatie van het type σ, dan is M typeerbaar met σ. Ook het BCD-systeem bezit de voornaamste typeëigenschap. De collectie van toegestane operaties voor het BCD-systeem bestaat buiten typesubstituties en expansies op types ook uit typeverheffingen.

In hoofdstuk drie worden twee noties van typering besproken, het Milner systeem en het Mycroft systeem, die gedefinieerd zijn voor een primitieve applicatieve (functionele) programeertaal. Ze onderscheiden zich van Curry's systeem door de uitbreiding van de lambda calculus met nieuwe syntactische constructoren voor het kunnen uitdrukken van recursie en polymorfie. Beide systemen bezitten de voornaamste typeëigenschap. Typering binnen Milners systeem is beslisbaar, binnen Mycrofts systeem niet.

Het tweede deel van dit proefschrift begint met hoofdstuk vier, dat de presentatie van het stricte intersectietype systeem bevat, welk een lichte beperking vormt van één van de CDV-systemen en daarmee een deelsysteem is van het BCD-systeem: de verzameling van de typeerbare termen is dezelfde als in beide andere systemen en typeerbaarheid in het stricte systeem is daarmee dan ook onbeslisbaar. Alle mogelijke afleidingen binnen het BCD-systeem kunnen worden opgebouwd door eerst een afleiding in het stricte systeem te maken en daarna de type inclusie relatie '$\leq$' te gebruiken. Dit systeem geeft aanleiding tot de definitie van een strict filter lambda model en volledigheid van typetoekenning wordt bewezen door in plaats van een simpele typeïnterpretatie de *inferentie* typeïnterpretatie te gebruiken. In hoofdstuk zes wordt aangetoond dat ook het stricte systeem de voornaamste typeëigenschap bezit. De collectie van toegestane operaties voor het stricte systeem bestaat buiten typesubstituties en expansies op types ook uit typeoptillingen, een beperkte vorm van typeverheffingen.

Hoofdstuk vijf bevat de presentatie van het essentiële intersectie systeem, een kleine uitbreiding van het stricte systeem en eveneens een beperking van het BCD-systeem. Deze notie van typering is equivalent aan BCD-typering en het is mogelijk zelfs voor dit systeem volledigheid van typetoekenning te bewijzen met behulp van een simpele typeïnterpretatie. Eveneens het essentiële systeem bezit de voornaamste typeëigenschap; de collectie van toegestane operaties is dezelfde als voor het stricte systeem, alleen de volgorde waarin operaties mogen worden toegepast verschilt.

Hoofdstuk zeven bevat de presentatie van een typeringssysteem dat, zoals het stricte systeem, een beperking vormt van het BCD-systeem. De beperking bestaat uit het verwijderen van de typeconstante ω, en geeft aanleiding tot de definitie van een filter λI-model. Volledigheid van typetoekenning voor termen uit de λI-calculus wordt met behulp van dit model bewezen, gebruikmakend van een simpele typeïnterpretatie. De verzameling termen uit de volledige λK-calculus die typeerbaar zijn binnen dit systeem is precies de verzameling van de sterk normaliserende termen, en daarmee is ook binnen dit systeem typering onbeslisbaar.

In hoofdstuk acht wordt een rang 2 intersectietyperingssysteem gepresenteerd, dat gebaseerd is op het CD-systeem en sterke overeenkomst vertoont met het Milner systeem. Ook dit systeem heeft de voornaamste typeëigenschap, en de toegestane operaties zijn typesubstituties en typeduplicaties, een eenvoudige vorm van expansie van types. Typering binnen dit systeem is beslisbaar.

Het derde deel van dit proefschrift begint met de definitie van applicatieve termherschrijfsystemen in hoofdstuk negen, een soort termherschrijfsystemen die een speciaal, voorgedefinieerd functiesymbool (Ap) bevatten. Termen in deze klasse van herschrijfsystemen worden niet alleen door middel van applicatie geconstrueerd, maar ook door een functiesymbool met een zekere ariteit van voldoende argumenten te voorzien.

De eerste notie van typering op de boomrepresentatie van deze herschrijfsystemen wordt gedefinieerd in hoofdstuk tien, en is gebaseerd op het essentiële typeringssysteem voor de lambda calculus en de manier van omgaan met recursie zoals gebruikt in Milners systeem. De definitie van typering bestaat uit het geven van een omgeving die voor elk functiesymbool een type levert en het formuleren van condities waaraan types toegekend aan de boomrepresentatie van termen en herschrijfregels moeten voldoen. De types die kunnen worden toegekend aan functiesymbolen in termen kunnen worden verkregen uit de types geleverd door de omgeving: ze kunnen worden verkregen door toepassing van de operaties typesubstituties, expansies op types en typeoptillingen zoals gedefinieerd voor het stricte systeem en het essentiële systeem voor de lambda calculus. In het algemeen bezit de notie van typering in termhershrijf systemen niet de eigenschap dat types toekenbaar aan termen ook toekenbaar zijn aan termen die ontstaan door het toepassen van een herschrijfregel (de *subject-reductie* eigenschap).

In hoofdstuk elf wordt op dezelfde manier als in hoofdstuk tien een notie van typering op links lineaire applicatieve termherschrijfsystemen gedefinieerd. Deze maakt echter gebruik van Curry types en de manier van omgaan met recursie zoals gebruikt in Mycrofts systeem en vormt daarom dan ook geen beperking van het systeem uit dat hoofdstuk. Deze notie van typering bezit de voornaamste typeëigenschap en met behulp van het voornaamste type van de linkerkant van een herschrijfregel wordt een conditie geformuleerd die noodzakelijk en voldoende is voor het verkrijgen van de subject-reductie eigenschap.

Het proefschrift wordt afgesloten met de presentatie van een notie van typering op applicatieve termherschrijfsystemen dat gebruik maakt van rang 2 intersectietypes en een echte

beperking is van het systeem gedefinieerd in hoofdstuk twaalf. Ook deze notie van typering bezit de voornaamste typeëigenschap en met behulp van het voornaamste type van de linkerkant van een herschrijfregel wordt een conditie geformuleerd die voldoende is voor het verkrijgen van de subject-reductie eigenschap.

Curriculum Vitae

15 februari 1958: Geboren te Doetinchem.

1970 – 1977: Atheneum-B aan het Nebo-Mariënbosch-Gabriël College te Nijmegen, afgesloten met het behalen van het diploma in 1977.

1977 – 1984: Studie Wiskunde aan de Katholieke Universiteit van Nijmegen.

1 september 1982 - 29 januari 1988: Studie Informatica aan de Katholieke Universiteit van Nijmegen, afgesloten met het behalen van het doctoraaldiploma.

1979 – 1987: Diverse studentassistentschappen, zowel binnen de studie Wiskunde als binnen de studie Informatica.

1 januari 1986 – 31 juli 1987: In dienst van de afdeling Computer Architectuur onder leiding van professor R. Boute.

15 februari 1988 – 15 juli 1988: Verblijf aan de Universiteit van Turijn, Italië, ondersteund door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (N.W.O.), beschikking NF-68, onder leiding van professor M. Dezani.

1 september 1988 – 31 december 1988: In dienst van de afdeling Computer Architectuur onder leiding van professor R. Boute.

1 januari 1989 – Heden: Als toegevoegd onderzoeker in dienst van de afdeling Theoretische Informatica en Berekeningsmodellen onder leiding van Dr.Ir. R. Plasmeijer, ondersteund door Esprit Basic Research Action 3074 'Semantics and Pragmatics of Generalized Graph Rewriting (Semagraph)' (1 juli 1989 - 31 december 1991), en de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (N.W.O.), beschikking 612-316-027 (1 februari 1990 - 1 februari 1994).

1 mei 1989 – 31 december 1991: In dienst als toegevoegd docent bij de Post-Doctorale Opleiding Informatica.

1 september 1989 – 31 december 1989: In dienst van de afdeling Computer Architectuur onder leiding van professor R. Boute.