

# Reduction Types and Intensionality in the Lambda-Calculus

by

David Amson Wright, BSc(Hons) MSc (Rhodes)

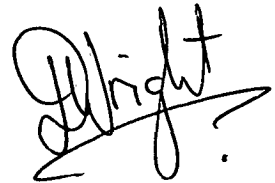
Submitted in fulfilment of the requirements  
for the Degree of  
Doctor of Philosophy

UNIVERSITY OF TASMANIA

September 1992

## Declaration

This thesis contains no material which has been accepted for the award of any other higher degree or graduate diploma in any tertiary institution. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference has been made in the text of the thesis.

A handwritten signature in black ink, appearing to read 'D. Wright', with a stylized flourish at the end.

David A. Wright

*To Merridy*

# Abstract

In this thesis I introduce a new approach to the automated analysis of the reduction behaviour of  $\lambda$ -calculus terms. This new approach improves on earlier analysers in several ways, not least in its treatment of higher-order terms and polymorphism, two notably troublesome issues.

In addition, this thesis introduces a stronger notion of reduction behaviour than *strictness*. This concept, called *strong head neededness*, forms the basis for a new notation for describing the reduction behaviour of terms. This notation is a kind of *type*, elements of which are built using a Boolean algebra of function type constructors. Thus the form of the methodology proposed is that of a *type system*.

Consideration is given to a variety of type assignment systems for the new type system. This supports the hypothesis that the approach proposed is suitable as a framework for building a range of analyses. Having established this framework it is then a matter of engineering to determine the appropriate trade off between information derived and performance achieved.

An investigation is conducted into the formal semantics of all the constructs introduced. In particular, the investigation proves a range of soundness and completeness results. Also examined is the semantics of the new notion of type and the development of a model for reduction types. The model is of interest in its own right, as it gives further insight into the reduction behaviour of  $\lambda$ -terms.

The thesis includes detailed implementations of all the type assignment systems and ascertains the correctness of these implementations.



# Acknowledgements

I would like to thank my supervisor, Clem Baker-Finch, for listening patiently to many partially developed ideas, reading many drafts and lots of discussion about this work. Thank you Clem for all your encouragement.

I thank Geoffrey Burn for always being prepared to give of his especially valuable time. Geoffrey has read many drafts of my work and we have had many discussions about the ideas as described in this thesis. In lots of ways Geoffrey was like a supervisor, even though we were separated by many thousands of miles.

Phil Collier has been a source of much wisdom, especially during some of the darker days. Thank you Phil for stepping in as supervisor when Clem left to take up his new post.

Andrew Partridge has been, and is, an excellent colleague. Andrew, I have enjoyed discussing with you, and learnt much about, dynamic and static techniques for detecting parallelism.

Tony Dekker and I had many useful and enlightening conversations about all manner of things to do with functional programming, types and theory in the earlier days of this project. Also much involved at this time was Ed Kazmierczak whose enthusiasm for the “great works” was an inspiration.

I would like to thank Arthur Sale for providing time to allow me to finish and for good advice.

Thanks to Bernard Gunther for opposing functional programming—for a while!

Without a doubt, I owe the greatest thanks to Merridy for her never ending support and love. I also thank my Mom and Dad for their constant faith in me and their encouragement, even when separated from me by a great distance.

# Contents

<b>Abstract</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Functional Languages and Static Analysis . . . . .	1
1.2 $\lambda$ -Calculus . . . . .	3
1.2.1 $\lambda$ -terms . . . . .	3
1.2.2 The Theory $\lambda$ . . . . .	5
1.2.3 Head Reduction . . . . .	6
1.2.4 Labelled Reduction and Descendants . . . . .	6
1.2.5 Some Common Combinators . . . . .	7
1.3 Curry Types and Type Deduction . . . . .	8
1.3.1 The Curry Type System . . . . .	8
1.4 A Guide to Reading this Thesis . . . . .	9
1.4.1 Plan of this Thesis . . . . .	10
<b>2 Strong Head Neededness and Irrelevance</b>	<b>13</b>
2.1 Strong Head Neededness . . . . .	14
2.2 Variations . . . . .	15
2.3 Irrelevance . . . . .	16
2.4 Properties . . . . .	17
2.5 Discussion . . . . .	20
<b>3 Reduction Types and Type Assignment</b>	<b>22</b>
3.1 Introduction to Reduction Types . . . . .	22
3.1.1 Boolean Algebras . . . . .	25
3.2 Curry-style Type Assignment . . . . .	26
3.2.1 Preliminaries . . . . .	26
3.2.2 The Type Assignment Rules . . . . .	28
3.2.3 Examples . . . . .	30
3.2.4 An Extension of the Curry-style System . . . . .	35
3.2.5 Multiple Occurrences of Term Variables . . . . .	37

3.2.6	Properties of the Curry-style System . . . . .	39
3.2.7	Syntactic Variations . . . . .	47
3.2.8	Discussion . . . . .	48
3.3	LET-Polymorphic Type Assignment . . . . .	49
3.3.1	Generic Types . . . . .	49
3.3.2	The Type Assignment Rules . . . . .	50
3.3.3	Examples . . . . .	53
3.3.4	Properties of the LET-Polymorphic System . . . . .	53
3.3.5	Discussion . . . . .	61
3.4	Intersection-style Type Assignment . . . . .	61
3.4.1	Intersection Boolean Reduction Types . . . . .	63
3.4.2	The Type Assignment Rules . . . . .	65
3.4.3	Extending Intersection-style Inference . . . . .	67
3.4.4	Examples . . . . .	69
3.4.5	Properties of the Intersection-style System . . . . .	70
3.4.6	Discussion . . . . .	79
<b>4</b>	<b>The Semantics of Reduction Types</b>	<b>80</b>
4.1	Models of Terms . . . . .	81
4.1.1	A Model of the $\lambda$ -calculus . . . . .	81
4.1.2	A Semi-Model of the $\lambda$ -calculus . . . . .	83
4.2	Models of Boolean Reduction Types . . . . .	88
4.2.1	What is a Type Interpretation? . . . . .	89
4.2.2	The Semantics of Simple Reduction Types . . . . .	90
4.2.3	The Semantics of LET-Polymorphic Reduction Types . . . . .	92
4.2.4	The Semantics of Intersection Reduction Types . . . . .	92
4.3	The Intensional Applicative Behaviour of $\lambda$ -terms . . . . .	93
4.4	Type Assignment is Sound . . . . .	97
4.4.1	Soundness of $\vdash^C$ . . . . .	97
4.4.2	Soundness of $\vdash^L$ . . . . .	98
4.4.3	Soundness of $\vdash^I$ . . . . .	98
4.5	Type Assignment is Complete . . . . .	98
4.5.1	Completeness of $\vdash^C$ . . . . .	99
4.5.2	Completeness of $\vdash^L$ . . . . .	100
4.5.3	Completeness of $\vdash^I$ . . . . .	101
4.6	Discussion . . . . .	102
<b>5</b>	<b>Implementation</b>	<b>104</b>
5.1	Preliminaries . . . . .	105
5.1.1	Substitutions . . . . .	105
5.1.2	Triples . . . . .	106
5.1.3	Unification of Boolean Rings . . . . .	106

5.2	Implementation of the Curry-style System . . . . .	108
5.2.1	Preliminaries . . . . .	109
5.2.2	Unification of Types . . . . .	109
5.2.3	Type Inference . . . . .	112
5.2.4	Decidability . . . . .	115
5.3	Implementation of the LET-Polymorphic System . . . . .	115
5.3.1	Substitution . . . . .	115
5.3.2	Unification of Types . . . . .	116
5.3.3	Type Inference . . . . .	117
5.3.4	Decidability . . . . .	121
5.4	Implementation of the Intersection-style System . . . . .	121
5.4.1	Substitution . . . . .	121
5.4.2	Normal Forms . . . . .	122
5.4.3	Expansion of a Deduction . . . . .	122
5.4.4	Principal Triples . . . . .	129
5.4.5	Unification of Intersection Types . . . . .	130
5.4.6	The Type Inference Algorithm . . . . .	133
5.4.7	Decidability . . . . .	136
<b>6</b>	<b>Extensions and Future Work</b>	<b>137</b>
6.1	Constants . . . . .	137
6.1.1	Term Constants . . . . .	138
6.1.2	Type Constants . . . . .	138
6.1.3	Type Assignment . . . . .	138
6.1.4	An Example: The Conditional . . . . .	139
6.2	Adding a Fixpoint Constant . . . . .	140
6.3	Data Structures . . . . .	143
6.3.1	A Detailed Example . . . . .	145
6.4	Second-Order Polymorphism . . . . .	147
6.5	Algebraic Reduction Types . . . . .	149
6.5.1	Constants . . . . .	152
6.5.2	Fixpoints and Algebraic Reduction Types . . . . .	153
6.5.3	Data Structures . . . . .	153
6.5.4	Implementation . . . . .	155
6.6	Non-termination . . . . .	156
6.7	A Two Step Strong Head Neededness Algorithm . . . . .	158
<b>7</b>	<b>Conclusion</b>	<b>160</b>
7.1	Related Work . . . . .	161
7.1.1	Analyses based on Types . . . . .	161
7.1.2	Abstract Interpretations not based on Types . . . . .	163
7.1.3	Other Techniques . . . . .	165

7.2	Summary . . . . .	166
7.2.1	Strong Head Neededness . . . . .	166
7.2.2	Boolean Reduction Types and Type Deduction . . . . .	166
7.2.3	Intensional Semantics for Reduction Types . . . . .	167
7.2.4	Implementation of Reduction Type Inference . . . . .	168
7.2.5	Extensions . . . . .	168
<b>A</b>	<b>An Implementation in Orwell . . . . .</b>	<b>170</b>
A.1	Orwell . . . . .	170
A.2	Preliminaries . . . . .	171
A.2.1	Auxiliary functions . . . . .	171
A.2.2	Parsing . . . . .	173
A.2.3	Printing . . . . .	177
A.2.4	Association Lists . . . . .	179
A.3	An Implementation of Boolean Algebra . . . . .	181
A.3.1	Parsing and Printing . . . . .	182
A.3.2	Sum-of-Products Normal Form . . . . .	183
A.3.3	Unification of Boolean Rings . . . . .	184
A.4	Lambda Terms . . . . .	186
A.4.1	Parsing $\lambda$ -terms . . . . .	186
A.4.2	Printing $\lambda$ -terms . . . . .	187
A.5	Intersection Reduction Types . . . . .	187
A.5.1	Parsing Reduction Types . . . . .	188
A.5.2	Simplification of Reduction Types . . . . .	189
A.5.3	Printing Reduction Types . . . . .	190
A.5.4	Substitution for types . . . . .	191
A.5.5	State-based functions . . . . .	191
A.5.6	Expansion . . . . .	193
A.6	Unification . . . . .	195
A.6.1	Guaranteeing Termination . . . . .	196
A.7	Constants . . . . .	197
A.8	Type Inference . . . . .	198
A.9	The Top-level of the Type Inference System . . . . .	199
A.10	Examples . . . . .	200

# List of Figures

1	The Theory $\lambda$ . . . . .	5
2	Curry's system for Type Deduction . . . . .	9
3	A Diagram of the Structure of the Thesis . . . . .	11
4	The Axioms of a Boolean Algebra . . . . .	25
5	The Curry-style Rules for deducing Reduction Types . . . . .	29
6	Extended Curry-style Rules for deducing Reduction Types . . . . .	35
7	The Alternative Curry-style Rules for deducing Reduction Types . . . . .	47
8	The LET-Polymorphic Rules for deducing Reduction Types . . . . .	51
9	The Intersection-style Rules for deducing Reduction Types . . . . .	66
10	The Extended Intersection-style Rules for deducing Reduction Types . . . . .	68
11	The Semantics of Simple Boolean Reduction Types . . . . .	90
12	The Semantics of LET-Polymorphic Reduction Types . . . . .	92
13	The Semantics of Intersection Boolean Reduction Types . . . . .	93
14	The Algorithm for unifying Boolean Arrow Expressions . . . . .	108
15	The Algorithm for unifying Simple Boolean Reduction Types . . . . .	110
16	The Algorithm for solving a set of Equations . . . . .	110
17	The Type Inference Algorithm for Simple Boolean Reduction Types . . . . .	112
18	The Type Inference Algorithm for LET-polymorphic Reduction Types . . . . .	118
19	The Algorithm for unifying Intersection Reduction Types . . . . .	132
20	The Type Inference Algorithm for Intersection Reduction Types . . . . .	134
21	The Delta Rules for Some Term Constants . . . . .	138
22	The Delta Rules for Data Structures . . . . .	143
23	The Second-Order System for deducing Reduction Types . . . . .	148
24	The Rules for deducing Intersection Algebraic Reduction Types . . . . .	151
25	The Rules for Non-termination . . . . .	157
26	A System for Analysing Type Checked $\lambda$ -terms . . . . .	159

# Chapter 1

## Introduction

### 1.1 Functional Languages and Static Analysis

Functional languages are closely related to the  $\lambda$ -calculus<sup>1</sup>—for the most part they may be considered as “syntactically sugared” versions of this calculus. Functional languages may be divided into two classes: *strict* and *non-strict*. Strict languages typically employ a call-by-value left-to-right evaluation strategy for arguments to functions. Non-strict languages employ a graphical form of call-by-name evaluation strategy, in which once the value of an argument is calculated this value replaces the expression representing the argument. This ensures that an argument is reduced at most once in the evaluation of the function call. (This latter strategy is often known as a “lazy” evaluation strategy).

Each of these methods of evaluation has an associated cost. The most serious cost is in the case of strict languages: certain expressions which may reasonably be expected to have a defined value fail to have such a value in this evaluation scheme. This is because under the call-by-value scheme an argument to a function is evaluated *whether or not* it is required by the function. This forces the programmer to bear the additional burden of having always to ensure explicitly the finiteness of any argument passed to a function.<sup>2</sup>

In the case of a non-strict language the cost incurred is in the implementation of the language. Each time the value of an argument is requested in such a language it is necessary to first test whether or not the argument has been evaluated. If it has not been evaluated, then its value must be computed, otherwise its value can be immediately looked up. Thus, if a value is *definitely* required then the call-by-value strategy is more “lazy” than the lazy strategy in the sense that less work is required to implement it.

However, it is possible to have the best of both schemes if the program is

---

<sup>1</sup>The  $\lambda$ -calculus is described in a later section of this chapter.

<sup>2</sup>The case for non-strict languages is argued in greater detail by Turner [65] and Hughes [34].

analysed so as to determine exactly which arguments are required and which are not required. With this information a compiler can *transform* the program to use a *mixed* evaluation strategy: it can use the call-by-name strategy for arguments which are not required and the call-by-value strategy for those that are required.

Unfortunately, things are not quite so simple. The problem of determining which arguments are required in an arbitrary program is reducible to the halting problem. Hence, this task is undecidable in general. The phrase “in general” does offer some hope: can an analysis technique be found which determines a high proportion of the arguments which are definitely required or are definitely not required? Moreover, can such a technique be found which is reasonably economical with time?

In this thesis, I examine the first question afresh. Although good answers to this question have been found,<sup>3</sup> no satisfactory answer to the combination of these questions has been found. My goal has been to introduce and explore a new method for performing such an analysis. This new method returns very precise information about higher-order terms. The major departure from other analysis techniques is the use of a form of *type* to express intensional properties of functions. Traditionally, types have been used to express extensional information about functions: what their inputs are and what their outputs are. My addition to this concept involves types containing extra information about the functions with which they are associated. This additional information is whether or not an argument to a function is actually used *in an essential way* to produce the output of the function.

A further use for an analysis technique such as is described in this thesis is that of allowing the *parallel evaluation* of a program written in a non-strict language. The restriction to only evaluating an argument when it is determined that it is required *during the evaluation of the function call* enforces a sequential evaluation strategy. If it is known that the argument is required before the function call is actually made, then the evaluation of the argument may be rescheduled by the compiler to occur at an earlier time.

There is an alternative method for conducting the parallel evaluation of a program written in a non-strict language, but discussion of this method is delayed until Chapter 2.

In this chapter some preliminary subjects are reviewed which are required in order to understand the rest of the thesis. At the end of the chapter is a guide to reading the thesis.

---

<sup>3</sup>See the section on related work in Chapter 7 for more details.



## 1.2 $\lambda$ -Calculus

This section serves to remind the reader of some well known concepts associated with the  $\lambda$ -calculus. It is recommended that the reader at least briefly peruse this section.

### 1.2.1 $\lambda$ -terms

#### Definition 1.2.1 ( $\lambda$ -terms)

Let  $X = \{v_0, v_1, \dots\}$  be a set of *term variables*, then the set  $\Lambda$  of  $\lambda$ -terms is inductively defined to be the smallest set containing  $X$  which is closed under function application and abstraction, namely

- if  $M \in \Lambda$  and  $N \in \Lambda$  then  $MN \in \Lambda$  and
- if  $x \in X$  and  $N \in \Lambda$  then  $\lambda x.N \in \Lambda$ .

Following the usual conventions, application associates to the left, and the scope of an abstraction is as far to the right as possible. Parentheses will often be omitted where these conventions make clear the intended meaning. Also, a term of the form  $(\lambda x_1.\lambda x_2.\dots\lambda x_n.N)$  will often be written as  $(\lambda x_1 x_2 \dots x_n.N)$ .

#### Definition 1.2.2

In a  $\lambda$ -term  $(\lambda x.N)$  the object  $\lambda x$  is the *binder* of the term, and  $x$  in  $\lambda x$  is the *binding occurrence* of  $x$ . A variable  $x$  occurs *bound* in a term  $M$  if  $M$  has a subterm  $(\lambda x.N)$  and  $x$  occurs in  $N$ , in which case the term  $N$  is the *scope* of this binding occurrence of  $x$ . A variable  $x$  occurs *free* in  $M$  if it is a subterm of  $M$ , and occurs outside the scope of any binding occurrence of  $x$ . The set of all free variables of a term  $M$  is denoted by  $FV(M)$ . The set of all bound variables of a term  $M$  is denoted by  $BV(M)$ .

Barendregt [3] describes the following useful variation on the set of  $\lambda$ -terms:

#### Definition 1.2.3

1. The set of *contexts* is defined as the least set satisfying:
  - $x \in X$  implies  $x$  is a context,
  - $[]$  is a context, and
  - if  $C_1[]$  and  $C_2[]$  are contexts, then so are  $C_1[]C_2[]$  and  $\lambda x.C_1[]$ .
2. If  $C[]$  is a context and  $M \in \Lambda$ , then  $C[M] \in \Lambda$  is that term which results by replacing each occurrence of  $[]$  in  $C[]$  by  $M$ .
3. A *head context* is any context of the form  $\lambda x_1 \dots x_m.[]M_1 \dots M_n$ ,  $m, n \geq 0$ .

By convention terms which are identical modulo change of bound variables are identified. Essentially, it is now the de Bruijn representation (see Barendregt [3], Appendix C) of terms which will be used in the rest of this work, though the above more readable notation for terms will be adhered to in the naive way. The symbol  $\equiv$  will be used to denote syntactic identity, modulo this convention. If  $M \equiv C[N]$ , then write  $N \subseteq M$  and say  $N$  is a *subterm* of  $M$ . Similarly defined  $N$  is a *proper subterm* of  $M$  (notation  $N \subset M$ ), if  $N \subseteq M$ , but  $N \neq M$ .

Substitution on  $\lambda$ -terms can now be defined in a straightforward manner:

1.  $x[x := N] \equiv N$ ,
2.  $y[x := N] \equiv y$  ( $y \neq x$ ),
3.  $(MN)[x := N] \equiv (M[x := N])(N[x := N])$ , and
4.  $(\lambda y.N)[x := N] \equiv \lambda y.N[x := N]$ .

Note that in the last case above no “confusion of bound variables” can occur by the convention concerning bound variables described above.

#### Definition 1.2.4

The rules of  $\lambda$ -reduction are:

- ( $\beta$ )  $(\lambda x.N)M \rightarrow_{\beta} N[x := M]$  and  
 ( $\eta$ ) if  $x \notin \text{FV}(N)$ , then  $(\lambda x.Nx) \rightarrow_{\eta} N$ .

If a  $\lambda$ -term  $M$  has the form of any of the left-hand sides of these rules, then  $M$  is a ( $\beta$  or  $\eta$ )-redex. If a term contains no  $\beta$ -redexes then it is in  $\beta$ -normal form (and similarly for  $\eta$ - and  $\beta\eta$ -normal forms). Let the reflexive, transitive closure of  $\rightarrow_{\beta}$  be denoted by  $\rightarrow_{\beta}^*$  (and similarly for  $\rightarrow_{\eta}$ ), and the reflexive, transitive and symmetric closure of  $\rightarrow_{\beta}$  be denoted by  $=_{\beta}$  (similarly  $=_{\eta}$ ), this relation being called  $\beta$ -conversion.

From now on only  $\beta$ -reduction will be considered unless explicitly indicated otherwise, and so “reduction” will mean  $\beta$ -reduction, “redex” will mean  $\beta$ -redex and “normal form” will mean  $\beta$ -normal form.

The following is standard, see Chapter 3 of Barendregt [3]. A *notion of reduction*,  $R$ , is a binary relation on  $\Lambda$ .

#### Definition 1.2.5

Let  $R$  be a notion of reduction.

1.  $R$  satisfies the *diamond* property if for all  $M$ ,  $M_1$  and  $M_2$ :  $M R M_1$  and  $M R M_2$  implies there exists  $M_3$  such that  $M_1 R M_3$  and  $M_2 R M_3$ .

Axioms	
$M = M$	$(\lambda x.M)N = M[x := N]$
Rules	
$\frac{M = N}{N = M}$	$\frac{M = N \quad N = L}{M = L}$
$\frac{M = N}{MZ = NZ}$	$\frac{M = N}{ZM = ZN}$
$\frac{M = N}{\lambda x.M = \lambda x.N}$	

Figure 1: The Theory  $\lambda$ 

2.  $R$  is Church-Rosser if its compatible<sup>4</sup>, reflexive and transitive closure satisfies the diamond property.

**Theorem 1.2.6**

$\beta$  is Church-Rosser.

**Proof**

This is shown several times in Barendregt [3]. See any of Theorem 3.2.8, Theorem 11.1.10, Corollary 11.2.29 or Corollary 14.2.4 (the simplest).  $\square$

**1.2.2 The Theory  $\lambda$** 

The theory  $\lambda$  is axiomatised in Figure 1. Write  $\lambda \vdash M = N$  if  $M = N$  is a consequent of this deduction system. The following well-known result expresses the relationship between the theory  $\lambda$  and the notion of  $\beta$ -conversion:

**Proposition 1.2.7 (Barendregt [3], Proposition 3.2.1)**

$\lambda \vdash M = N$  iff  $M =_{\beta} N$ .

This Proposition is essential in relating the results of Chapter 2 to those of Chapter 4.

<sup>4</sup>See Definition 3.1.1.(i) of Barendregt [3] for the definition of a *compatible* relation.

### 1.2.3 Head Reduction

The concept of *head normal form* plays a central role in the  $\lambda$ -calculus, in particular, a term has a head normal form iff it is *solvable*, see Barendregt [3] pp41–42 for discussion. Associated with the notion of head normal form is a particular reduction strategy known as *head reduction*. In Chapter 2, head reduction is used to define a property called *strong head neededness*. This property is chosen as a test bed for the idea of using types as the notation for performing analyses of terms proposed by this thesis.

#### Definition 1.2.8

A subterm  $N$  of a term  $M$  is at the *head* of  $M$  if

- $M \equiv N$ , or
- $M \equiv \lambda x.N'$  and  $N$  is at the head of  $N'$ , or
- $M \equiv N_1N_2$  and  $N$  is at the head of  $N_1$ .

Suppose  $M$  is not in normal form. The *leftmost* redex of  $M$  is the redex whose binder is to the left of the binder of every other redex in  $M$ . The leftmost redex,  $R$ , of  $M$  is a *head* redex of  $M$  if  $R$  is at the head of  $M$ .  $M$  is in *head normal form* if it has no head redex. The *head reduction path* of a term  $M$  is a sequence of reduction steps in which every redex which is reduced is a head-redex. Suppose  $M$  has a head redex, then write  $M \rightarrow_h M'$ , if  $M'$  results from  $M$  by contraction of the head redex of  $M$ . Let  $\rightarrow_h$  be the transitive, reflexive closure of the least relation generated by  $\rightarrow_h$ .

The opposite of head reduction is *internal* reduction, i.e.,  $M \rightarrow_i N$  iff no head redex is contracted in this reduction path from  $M$  to  $N$ .

### 1.2.4 Labelled Reduction and Descendants

The following is taken from Klop [40], see also Barendregt [3].

#### Definition 1.2.9

Let  $\mathcal{A}$  be a set of symbols, called *labels*. The labelled  $\lambda$ -terms,  $\Lambda^{\mathcal{A}}$ , are inductively defined by

- $x^a \in \Lambda^{\mathcal{A}}$ , for all  $x \in X$  and  $a \in \mathcal{A}$ ,
- $A, B \in \Lambda^{\mathcal{A}}$  implies  $(AB)^a \in \Lambda^{\mathcal{A}}$ , for all  $a \in \mathcal{A}$ , and
- $A \in \Lambda^{\mathcal{A}}$  implies  $(\lambda x.A)^a \in \Lambda^{\mathcal{A}}$ , for all  $x \in X$  and  $a \in \mathcal{A}$ .

$A \in \Lambda^{\mathcal{A}}$  may also be written as  $M^I$  where  $M \in \Lambda$  is obtained from  $A$  by erasing all the labels and  $I$  is a map from the set of occurrences of subterms of  $M$  to labels such that  $I(M)$  is  $A$ . The map  $I$  is called a *labeling* of  $M$ .

A notion of reduction can be formulated for the labelled terms. Firstly, labelled substitution is defined by

- $x^a[x := B] \equiv B$ ,
- $y^a[x := B] \equiv y, y \neq x$ ,
- $(AA')^a[x := B] \equiv ((A[x := B])(A'[x := B]))^a$ , and
- $(\lambda y.A)^a[x := B] \equiv (\lambda y.A[x := B])^a$ . (By the variable convention it is not necessary to specify  $x \neq y$ ).

Now the reduction relation can be generated from the schema

$$\beta_A : ((\lambda x.A)^a B)^b \rightarrow A[x := B].$$

Suppose  $M \rightarrow_{\beta} N$ , then the *descendants* of some subterm  $M'$  of  $M$  can be found in  $N$  (if any exist), by marking  $M'$  and following it through the reduction from  $M$  to  $N$ . More formally:

### Definition 1.2.10

Suppose  $M \rightarrow_{\beta} N$  by contraction of a redex  $R$ . Let  $I$  be a labeling of  $M$ . Then  $R$  and  $I$  uniquely determine a corresponding labelled reduction,  $M^I \rightarrow_{\beta_A} N^J$ , for some labeling  $J$  of  $N$ .

A labeling is *initial* if the labels it assigns to distinct subterm occurrences of a term are distinct. Let  $I$  be an initial labeling (assume  $\mathcal{A}$  is sufficiently large). Let  $P$  be a subterm occurrence of  $M$  and  $Q$  be a subterm occurrence of  $N$ , then  $Q$  is a *descendent* of  $P$  if  $J(Q) = I(P)$ . This concept generalises to  $\rightarrow_{\beta}$  in the obvious fashion.

Any descendent of a redex,  $R$ , is itself a redex, and is called a *residual*. This is so since if  $R$  is contracted in  $M \rightarrow_{\beta} N$ , then clearly  $R$  has no descendants in  $N$ . A redex  $R$  of a term  $M$  is *erased* in a particular reduction path of  $M$  to a term  $N$  if no residual of  $R$  is contracted on this reduction path and there is no residual of  $R$  in  $N$ .

## 1.2.5 Some Common Combinators

In this subsection names are given to some combinators which are frequently used in this thesis. Firstly, a *combinator* is a closed  $\lambda$ -term. All of the combinators presented here may be found in Barendregt [3] and the reader should refer to that work for further discussion.

Three well known combinators are S, K and I. These have the following definitions:

- $S \equiv \lambda f g x. f x (g x);$
- $K \equiv \lambda x y. x;$  and
- $I \equiv \lambda x. x.$

The simplest example of a function incorporating a self application is  $\omega \equiv \lambda x. x x$ . From this term is constructed the simplest unsolvable term:  $\Omega \equiv \omega \omega$ . An example of an unsolvable term which becomes progressively longer with each reduction step taken is  $\Omega_3 \equiv (\lambda x. x x x) \lambda x. x x x$ .

The two best known fixpoint combinators are  $Y \equiv \lambda f. (\lambda x. f (x x)) \lambda x. f (x x)$  (due to Curry) and  $\Theta \equiv (\lambda x f. f (x x f)) \lambda x f. f (x x f)$  (due to Turing).

## 1.3 Curry Types and Type Deduction

In this section a brief introduction to type deduction systems is given. The type deduction system studied here, the *Curry type system*, is the simplest system for assigning types to (a fragment of) the  $\lambda$ -calculus. A much more complete discussion of this and other type systems is given in Hindley and Seldin [30].

The basic philosophy of Curry's type system is to avoid building the notion of type into the terms themselves. This approach preserves the operator oriented flavour of the  $\lambda$ -calculus, as opposed to a set-oriented view of functions as particular sets of pairs. Thus it is quite possible for an operator such as  $\lambda x. x$  to have many types, rather than requiring an identity for each type. (For more polemic on this point see Barendregt [3] and Hindley and Seldin [30]).

The main goal of this thesis is to extend Curry's style of describing the extensional behaviour of  $\lambda$ -terms to include a description of their reduction behaviour. Thus the question asked is not just "what terms may be accepted as input?", but instead the question is "what terms may be accepted as input and how are they used to compute the result?".

Curry's approach is to specify a *type deduction system* in which types may be assigned to terms according to a logic specified in a natural deduction style (Gentzen [63]). Below is given a description of Curry's system.

### 1.3.1 The Curry Type System

Let  $\tau_v$  be a sufficiently large set of *type variables*. (The initial lowercase Greek letters  $\alpha, \beta, \dots$  are used to range over type variables). The set of Curry types  $T$ , ranged over by  $\sigma$  and  $\tau$ , is inductively defined to be the least set satisfying:

- $\alpha \in \tau_v$  implies  $\alpha \in T$ , and

VAR	$A_x \cup \{x:\sigma\} \vdash x:\sigma$
APP	$\frac{A \vdash N_1:\sigma \rightarrow \tau \quad A \vdash N_2:\sigma}{A \vdash N_1 N_2:\tau}$
ABS	$\frac{A_x \cup \{x:\sigma\} \vdash N:\tau}{A \vdash \lambda x.N:\sigma \rightarrow \tau}$

Figure 2: Curry's system for Type Deduction

- $\sigma, \tau \in T$  implies  $\sigma \rightarrow \tau \in T$ .

Clearly, in order to assign types to  $\lambda$ -terms, types must be assumed for free variables of the term. A *type assumption* for Curry's deduction system is a term of the form  $x:\sigma$ , where  $x$  is a term variable and  $\sigma$  is a Curry type.<sup>5</sup>

Statements of Curry's logic<sup>6</sup> are written as

$$A \vdash M:\tau,$$

where  $A$  is a set of type assumptions in which no term variable occurs more than once,  $M$  is a term and  $\tau$  is a Curry type. Such a statement may be read as " $M$  has type  $\tau$  in context  $A$ ". An assumption set containing no type assumption of the form  $x:\sigma$ , for any type  $\sigma$ , is denoted by  $A_x$ .

Curry's type deduction system is specified by one axiom and two rules, as detailed in Figure 2. Note the correspondence between the structure of  $\lambda$ -terms and the structure of the proofs resulting from the use of this type deduction system. Hindley and Seldin [30] develop this theme further, see Sections 14C, 14D, 15C and 15D of that work.

## 1.4 A Guide to Reading this Thesis

Since this is a fairly long document with many technical results, some advice is appropriate on how to read the report. To start with, all of the current chapter should be read. In Chapter 2 it would be best to read all sections through to the end of Section 2.3, with Section 2.4 being left for later. This will introduce the reader to strong head neededness, but leave the examination

<sup>5</sup>Curry originally wrote assumptions as  $\sigma x$  (see [18]), though the notation chosen here has become widely used in recent times.

<sup>6</sup>The word "logic" is used interchangeably with the phrase "type deduction system".

of the properties of strong head neededness until later. Section 2.5 contains the concluding remarks for this chapter.

In Chapter 3, Section 3.1 introduces and motivates Boolean Reduction Types and so this should be read on a first pass of the report. Then Section 3.2 studies the case of a Curry-style type deduction system for Boolean Reduction Types. Within this section, Section 3.2.6 could be omitted on an initial reading. Sections 3.3 and 3.4, which give two more case studies of type deduction for Boolean Reduction Types, may be left for a later reading.

Within Chapter 4, on the semantics of the various constructs introduced in this thesis, all sections after Section 4.2.2 may be omitted on first reading. These latter sections present the semantics of forms of Boolean Reduction Types which are introduced in the case studies of Sections 3.3 and 3.4; establish a connection between strong head neededness and Boolean Reduction Types and prove the correctness of all the type deduction systems.

Following the theme of concentrating only on the Curry-style system, in Chapter 5 read Sections 5.1 and 5.2.

At this stage the reader will have seen most of the major innovations of this thesis and have witnessed a complete development of the analysis methodology for one particular type system. The only major omission in the reading was a study of the semantic correctness proofs for the simplest system of logic and the semantic connection between strong head neededness, Boolean Reduction Types and the applicative behaviour of terms (Section 4.3). The reader could now go back to Section 2.4 and commence from there a more detailed reading of Chapters 2 through 5. Alternatively, the reader could press on and read Chapter 6. This chapter contains several extensions to the work already presented, though some of these extensions use or discuss material from parts of the report which the reader will have omitted on this first reading.

### 1.4.1 Plan of this Thesis

The thesis is divided into chapters which deal with the following topics:

- strong head neededness,
- Reduction Types and type deduction systems,
- semantics,
- implementation, and
- extensions.

Figure 3 is a graphical picture of the structure of Chapters 2 through 5. In this Figure each number within a box is the number of a section or subsection from



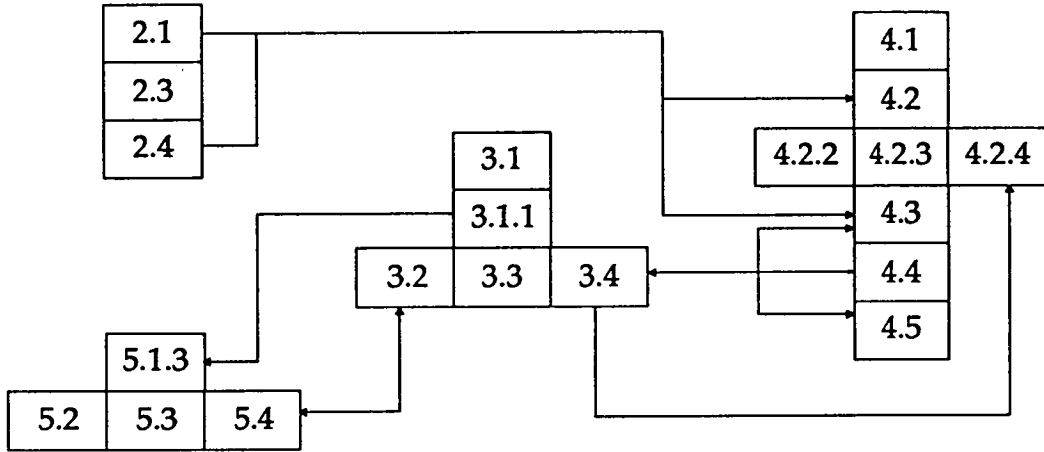


Figure 3: A Diagram of the Structure of the Thesis

the report. A box which is placed directly on top of a second box is meant to be read before the second box. Boxes placed side by side are relatively independent of each other, though for didactic purposes there is a left-to-right preference. Lines joining boxes indicate that these sections are linked by important theorems. In reading the paragraphs below it may prove useful to refer to this Figure.

In Section 2.1 the concept of *strong head neededness* is introduced and motivated. This leads onto Section 2.3 in which a notion which is the opposite of strong head neededness is introduced, namely *irrelevance*.<sup>7</sup> Then Section 2.4 examines these properties in further detail.

In Section 3.1 the concept of *Boolean Reduction Types* is introduced and motivated. Section 3.1.1 contains an overview of Boolean algebras—these are used in the definition of Boolean Reduction Types. Then Sections 3.2 through 3.4 each describe a case study (in some detail) of the use of Boolean Reduction Types in a type deduction system.

Up until this point no formal connection between strong head neededness and Boolean Reduction Types has been established. This is achieved in Chapter 4. Section 4.1 commences this chapter by introducing models of  $\lambda$ -terms. This is followed by Section 4.2 in which models of Boolean Reduction Types are constructed. In this section is first constructed a formal link between the strong head neededness and irrelevance of Chapter 2 and models of terms. Then, in the three concurrent sections 4.2.2 through 4.2.4, semantics are given to the three forms of Boolean Reduction Type introduced in Sections 3.2 through 3.4. Section 4.3 relates the semantics given to Boolean Reduction Types to the concept of strong head neededness, making use of the connection between strong head neededness and models of terms already shown, as well as results from

<sup>7</sup>In Chapter 6 it is shown how these two concepts are really part of a *continuum*.

Section 2.4. Sections 4.4 and 4.5 prove that the type deduction systems described in Sections 3.2 through 3.4 are semantically correct.

With the logics of type deduction for Boolean Reduction Types shown to be correct Chapter 5 proceeds to provide algorithms so as to allow the automatic inference of Boolean Reduction Types for terms. Section 5.1.3 describes an algorithm for solving equations in a Boolean algebra. This is followed, in Sections 5.2 through 5.4, by implementations corresponding to each of the type deduction systems of Sections 3.2 through 3.4. Within each of Sections 5.2, 5.3 and 5.4 the algorithms described are proved to be correct with respect to the appropriate type deduction system. Thus the *semantic* correctness of these algorithms is shown, since it is already known that the type deduction systems are semantically correct.

Chapter 6 (not shown in the Figure) contains a number of extensions to the work introduced in earlier chapters. In particular, analysis techniques are given for certain constructs which commonly occur in programming languages, such as constants and data structures. In addition, some discussion is given of a natural generalisation of Boolean Reduction Types.

Finally, Chapter 7 contains an overview of related work and a summary of the results of the thesis. Indeed, this summary is a good place to start for the reader interested in a slightly more detailed overview of the thesis than has been given in this section.

## Chapter 2

# Strong Head Neededness and Irrelevance

In this Chapter *strong head neededness* is introduced and examined. Strong head neededness is a slight variation on a property commonly explored in other works called *strictness* (defined below). Strong head neededness is similar to the notion of *head neededness* proposed by Barendregt et al [5].

An important reason for considering strong head neededness rather than head neededness or strictness is the parallel reduction strategy known as *speculative evaluation*, see Partridge [53]. This strategy is based on the idea of evaluating arguments to terms even when it is not known whether they are required. Non-strict semantics are preserved by terminating the evaluation of an argument as soon as it becomes known that the argument is not required.

As shown by Proposition 2.1.4 below, an implementation which performs parallel evaluation based on the notion of head neededness or strictness may needlessly evaluate certain sub-expressions of a non-terminating expression. This is because *all* redexes of a non-terminating expression are considered to be head needed and so in a parallel machine all of the redexes will be evaluated. Of course, it is trivial to see that it is semantically safe to perform this extra evaluation, at least for sequential implementations. However, in parallel implementations employing speculative evaluation, the designation of these sub-expressions as required for the computation might cause premature overloading of the machine. Strong head neededness on the other hand, determines exactly which redexes are required—even in a non-terminating term. Hence, this work introduces and examines this sharper notion of neededness.

As an example of this phenomenon, consider the term

$$\lambda x.\Omega.$$

Now head neededness (strictness) determines that the argument to this function is required, whereas strong head neededness determines that it is not

required. If this term is applied to an argument,  $M$ , (suppose computing  $M$  requires a large amount of memory), then head neededness will indicate that the evaluation of  $M$  should commence at the highest priority. Furthermore, if the expression  $(\lambda x.\Omega) M$  occurs in a *speculative* context, then initiating the computation of  $M$  is particularly undesirable. In the case of strong head neededness, the evaluation of  $M$  will proceed at a lower (speculative) priority.<sup>1</sup>

This chapter starts by introducing strong head neededness and comparing it with head neededness (strictness). Then some different ways of expressing the concept are introduced—chiefly to allow the concept to be assigned to non-redex terms. Strong head neededness has a natural dual, namely *irrelevance*, and this is also introduced in this chapter. After this, a section is given which establishes many properties concerning strong head neededness. In particular its relation is explored with reduction and conversion.

## 2.1 Strong Head Neededness

Firstly, the new property:

### Definition 2.1.1 (Strong Head Neededness)

Suppose  $R$  is a redex of  $M$ , then  $R$  is *strongly head needed* in  $M$  if the head reduction path of  $M$  reduces a residual of  $R$ .

In contrast:

### Definition 2.1.2 (Barendregt et al [5])

Suppose  $R$  is a redex of  $M$ , then  $R$  is *head needed* in  $M$  if every reduction path of  $M$  to head normal form reduces a residual of  $R$ .

Terms which cannot be reduced to head normal form are identified with the symbol  $\perp$ . A function  $f$  is *strict* on its argument if  $f\perp = \perp$ . The process of determining whether a function is strict is referred to as *strictness analysis*. Similarly the process of determining whether a redex is head needed is referred to as *head neededness analysis*. By Proposition 5.1 of Barendregt et al [5] (due to H. Mulder), strictness analysis is equivalent to head neededness analysis, since a function is strict on its argument iff the function head needs its argument.

Note that the notion of *strongly head neededness* introduced above is a somewhat sharper notion than that of *head neededness*, as introduced by Barendregt et al [5]. Suppose  $R \in \Lambda$  is a redex, then in the case of a function such as  $\lambda x.\Omega$ , where  $\Omega \equiv (\lambda y.yy)(\lambda y.yy)$ , it is vacuously true that  $R$  in  $(\lambda x.\Omega)R$  is head needed. In contrast,  $R$  is *not* strongly head needed in  $(\lambda x.\Omega)R$ , since  $R$  is

<sup>1</sup>In fact, for this particular example the analysis technique introduced in the following chapter would determine that  $M$  is not required at all.

never reduced in the head reduction path of  $(\lambda x.\Omega)R$ . (The difference between these two concepts arises entirely when the term involved does not have a head normal form).

**Lemma 2.1.3 (Theorem 3.6 (ii) of Barendregt et al [5])**

Let  $R$  be a redex in  $M \in \Lambda$  and  $M$  be solvable, then  $R$  is head needed iff a residual of  $R$  is contracted on the head reduction path of  $M$ .

Now the following is easily shown.

**Proposition 2.1.4**

Let  $M \in \Lambda$  and  $R \subseteq M$  be a redex.

1. If  $M$  is solvable, then  $R$  is strongly head needed in  $M$  iff  $R$  is head needed in  $M$ .
2. If  $M$  is not solvable, then if  $R$  is *any* redex in  $M$ , then  $R$  is a head needed redex in  $M$ .

**Proof**

For part 1 the result follows by the Lemma. Part 2 is trivial.  $\square$

The second part of this proposition demonstrates that every redex of a non-terminating expression is designated as head needed. The same cannot be said for strong head neededness which can distinguish redexes which are required even in a non-terminating expression. This extra information is expected to be important for parallel machines using a speculative evaluation strategy. This is because speculative evaluation is implemented using a *priority scheme* for choosing which redexes to reduce (Partridge [53]). Thus, marking extra redexes of a non-terminating term as demanded will cause these redexes to be given greater or equal preference to other tasks at the same priority level. Naturally this phenomenon should be minimised, and it is strong head neededness which appears to best fit the requirements.

## 2.2 Variations

The following variations on Definition 2.1.1 are sometimes useful. The main utility of these is the ability to talk of the strong head neededness of terms which are not redexes.

**Definition 2.2.1**

Suppose  $R \equiv N_1 N_2$ , then  $N_1$  *strongly head needs its argument*,  $N_2$ , if a descendent of  $N_2$  occurs at the head of some term on the head reduction path of  $R$ .

**Definition 2.2.2**

Let  $N \in \Lambda$ . For an arbitrary  $M \subseteq N$ ,  $M$  is *strongly head needed* in  $N$  if a descendent of  $M$  occurs at the head of some term on the head reduction path of  $N$ .

These definitions of strong head neededness are related as follows:

**Proposition 2.2.3**

Let  $R \in \Lambda$  be a redex and  $M \in \Lambda$ .

1.  $M$  is strongly head needed in  $C[M]^2$  iff  $R$  is strongly head needed in  $C[R]$ , and
2.  $N_1 N_2$  strongly head needs its argument  $N_2$  iff  $N_2$  is strongly head needed in  $N_1 N_2$ .

**Proof**

Note that a residual is just a special kind of descendent, then the results are immediate.  $\square$

## 2.3 Irrelevance

The opposite of strong head neededness is irrelevance:

**Definition 2.3.1**

$M$  is *irrelevant* in  $N$  if  $M$  is not strongly head needed in  $N$ .

**Lemma 2.3.2 (Barendregt [3], Lemma 11.4.6)**

If  $M \rightarrow_\beta N$ , then  $\exists P. M \rightarrow_h P \rightarrow_i N$ .<sup>3</sup>

**Proposition 2.3.3**

$\forall M, N. C[M] = C[N]$  implies  $\forall P. P$  is irrelevant in  $C[P]$ .

**Proof**

By the Church-Rosser Theorem for  $\beta$ -reduction,  $C[M] = C[N]$  implies there exists  $Z \in \Lambda$  such that  $C[M] \rightarrow_\beta Z \leftarrow_\beta C[N]$ . Since this is true for all  $M$  and  $N$ , all descendents of both  $M$  and  $N$  must be erased in each  $Z$ .<sup>4</sup> By the Lemma, there exists  $P \in \Lambda$  such that  $C[M] \rightarrow_h P \rightarrow_i Z$  and again there exists  $P' \in \Lambda$  such that  $C[N] \rightarrow_h P' \rightarrow_i Z$ . Suppose that a residual of  $M$  (and hence of  $N$ ) is reduced on the head reduction path from  $C[M]$  to  $P$  ( $C[N]$  to  $P'$ ), then set  $M \equiv \Omega$  and  $N \equiv \Omega_3$  to arrive at a contradiction. (Since  $\Omega \not\equiv \Omega_3$  and if either occurs at the head of some term then neither may be erased in *any* reduction of that term).  $\square$

<sup>2</sup>Contexts are defined in Chapter 1.

<sup>3</sup> $\rightarrow_i$  is *internal reduction*, as defined in Chapter 1.

<sup>4</sup>The notion of *erasure* is defined in Chapter 1.

## 2.4 Properties

In this section the relations between reduction, conversion and strong head neededness are explored. The work of Barendregt et al [5] serves as a suitable structure upon which to base the following investigation. Some additional results are also developed outside this structure, in particular the preservation of the strong head neededness of a subexpression which is not contracted during a  $\beta$ -conversion (Proposition 2.4.10).

The main results used outside of this present chapter are Proposition 2.4.10, Corollary 2.4.8 and Theorem 2.4.15. The latter theorem establishes the unsurprising result that determining strong head neededness is in general undecidable.

### Definition 2.4.1

A *reduction path* is a (possibly infinite) sequence of terms,  $M_0, M_1, \dots$ , such that  $M_0 \rightarrow_\beta M_1 \rightarrow_\beta \dots$ . Reduction paths will be denoted by the calligraphic letters  $\mathcal{R}, \mathcal{S}$ .

Let  $M \rightarrow_\beta N$  by contraction of a redex  $R \subseteq M$  and  $\mathcal{S} : M \rightarrow_\beta N'$ , then the projection of  $R$  by  $\mathcal{S}$  is obtained by contracting all residuals of  $R$  in  $N'$  after reduction  $\mathcal{S}$  has occurred. This can be generalised to the projection of a reduction path  $\mathcal{R} : M \rightarrow_\beta N_1$  by another path  $\mathcal{S} : M \rightarrow_\beta N_2$  in the obvious manner. In Klop [40] it is shown that as a consequence of the Church-Rosser property for  $\beta$  that the projection of a sequence  $\mathcal{R}$  by a sequence  $\mathcal{S}$  always ends in exactly the same term as the projection of  $\mathcal{S}$  by  $\mathcal{R}$ .

### Lemma 2.4.2 (Barendregt et al [5], Proposition 2.6)

Let  $\mathcal{R} : M \rightarrow_\beta M'$  and  $\mathcal{S} : M \rightarrow_\beta N$  be two reduction paths. Suppose  $R \subseteq M$  is a redex none of whose residuals are contracted in  $\mathcal{S}$  and let  $R' \subseteq M'$  be a residual of  $R$  after the reduction  $\mathcal{R}$ , then no residual of  $R'$  is contracted in the projection of  $\mathcal{R}$  by  $\mathcal{S}$ .

### Theorem 2.4.3

Let  $M \rightarrow_\beta M'$  and let  $S$  be a redex in  $M$  and  $S'$  be a residual of  $S$  in  $M'$ , then  $S$  is not strongly head needed in  $M$  implies  $S'$  is not strongly head needed in  $M'$ . (Equivalently,  $S'$  is strongly head needed in  $M'$  implies  $S$  is strongly head needed in  $M$ ).

### Proof

$S$  not strongly head needed implies that no residual of  $S$  is contracted in the (possibly infinite) head reduction of  $M$ . Then the result follows by iterated use of Lemma 2.4.2.  $\square$

### Theorem 2.4.4

Let  $R$  be a redex in  $M$ , then  $R$  is strongly head needed in  $M$  iff some residual of  $R$  is contracted in the head reduction of  $M$ .

**Proof**

The “if” part is immediate from the definition of strong head neededness. For the “only if” part, the result follows from Theorem 2.4.3.  $\square$

The above result can be summarised as:  $R$  is strongly head needed in  $M$  iff  $R$  is not erased in the head reduction of  $M$  iff  $R$  has a residual in the head reduction of  $M$  which is a head redex.

Let  $S \subseteq M'$  be a redex and  $M'$  results from  $M$  by contraction of a redex  $R$ , then  $S$  is *created* in  $M \rightarrow_\beta M'$  if  $S$  is not a residual of a redex in  $M$ .

**Proposition 2.4.5**

Suppose  $M \rightarrow_\beta M'$  by contraction of a redex  $R$  in  $M$ . If  $Q$  is a strongly head needed redex in  $M'$  created by this contraction, then  $R$  is strongly head needed in  $M$ .

**Proof**

Suppose  $R$  is not strongly head needed in  $M$ . Since  $Q$  is strongly head needed in  $M'$  it has some residual  $Q'$  which is contracted in the head reduction of  $M'$ . Since the head reduction of  $M'$  is the projection of the reduction of  $R$  by the head reduction of  $M$ ,  $Q'$  is a residual of a redex in the head reduction of  $M$ . Then by Proposition 2.8 (ii) of Barendregt et al [5]  $Q$  is not created, which is a contradiction.  $\square$

**Definition 2.4.6**

Let  $\text{SHN}(M) = \{N \in \Lambda \mid M \rightarrow_\beta N; N \text{ is strongly head needed in } M'\}$ .

**Lemma 2.4.7**

$M \rightarrow_\beta N$  implies  $\text{SHN}(M) \supseteq \text{SHN}(N)$ .

**Proof**

The case  $M$  already in head normal form is trivial. Suppose  $M$  not in head normal form.

Consider the case  $M \rightarrow_\beta N$  by contraction of the head redex of  $M$ . Then the result is immediate by definition of  $\text{SHN}(M)$ .

Consider the case  $M \rightarrow_\beta N$  by contraction of a non-head redex of  $M$ . Then the result follows by Proposition 2.4.5.

The result now follows by induction on the reduction path  $M \rightarrow_\beta N$ .  $\square$

**Corollary 2.4.8**

Let  $M \rightarrow_\beta N$ , then if  $R$  is the head redex of  $N$ , then  $R$  is strongly head needed in  $M$ .

**Proof**

By Lemma 2.4.7.  $\square$



**Proposition 2.4.9**

Suppose  $R$  is a strongly head needed redex of  $M$  and suppose  $M \rightarrow_\beta N$  and no residual of  $R$  is contracted in this reduction. Then  $R$  has a strongly head needed residual in  $N$ .

**Proof**

Immediate by Lemma 2.4.7 and Theorem 2.4.4.  $\square$

Write  $M_1 =_\beta M_2$  *without contracting*  $M$  if  $\exists Z \in \Lambda. M_1 \rightarrow_\beta Z, M_2 \rightarrow_\beta Z$  and no residual of  $M$  is contracted in either of these reductions.<sup>5</sup>

**Proposition 2.4.10**

Suppose  $M_1 =_\beta M_2$  without contracting  $M$ , then  $M$  strongly head needed in  $M_1$  implies  $M$  strongly head needed in  $M_2$ .

**Proof**

By definition,  $\exists Z \in \Lambda. M_1 \rightarrow_\beta Z, M_2 \rightarrow_\beta Z$  and no residual of  $M$  is contracted in either of these reductions. Hence, the result follows by application of Proposition 2.4.9 to both reductions.  $\square$

**Lemma 2.4.11**

Suppose  $R$  and  $S$  are redexes such that  $R$  is a subterm of  $S$  and  $S$  is a subterm of  $M$ , then  $R$  is strongly head needed in  $M$  implies  $S$  is strongly head needed in  $M$ .

**Proof**

Suppose  $S$  is not strongly head needed in  $M$ , then no residual of  $S$  is contracted in the head reduction of  $M$ . Since  $S$  is to the left of  $R$  (and the same is true for residuals of  $S$  and  $R$ , see Barendregt et al [5]), it follows that no residual of  $R$  is contracted in the head reduction of  $M$ . By Theorem 2.4.3,  $R$  is not strongly head needed in  $M$ .  $\square$

**Proposition 2.4.12**

Let  $M \rightarrow_\beta N$  by contraction of a redex  $R$  in  $M$ , and  $S$  be a strongly head needed redex in  $M$ . Suppose  $R$  is not strongly head needed, then  $S$  has a strongly head needed residual  $S'$  in  $N$ .

**Proof**

If  $S$  is multiplied by contracting  $R$ , then by Lemma 2.4.11  $R$  is strongly head needed, which is a contradiction. So  $S'$  is the unique residual of  $S$  in  $N$ . By Proposition 2.4.9  $S'$  is strongly head needed.  $\square$

**Proposition 2.4.13**

If  $M =_\beta N$ , then  $R$  strongly head needed in  $MR$  implies  $R$  strongly head needed in  $NR$ .

---

<sup>5</sup>Note that this includes the cases where  $M \subseteq M_1$  and/or  $M \subseteq M_2$ .

**Proof**

By the Church-Rosser Theorem for  $\rightarrow_\beta$ ,  $M \rightarrow_\beta P$  and  $N \rightarrow_\beta P$ , for some  $P \in \Lambda$ . Thus  $MR \rightarrow_\beta PR$  and  $NR \rightarrow_\beta PR$ . By Proposition 2.4.9  $R$  strongly head needed in  $MR$  implies  $R$  strongly head needed in  $PR$  and hence in  $NR$  by Theorem 2.4.3.  $\square$

**Proposition 2.4.14**

If  $M =_h N$ , then  $R$  strongly head needed in  $MR$  implies  $R$  strongly head needed in  $NR$ .

**Proof**

Similar to the proof of Proposition 2.4.13 (simply replace  $\rightarrow_\beta$  and  $\twoheadrightarrow_\beta$  by  $\rightarrow_h$  and  $\twoheadrightarrow_h$ , respectively, and note that  $M \rightarrow_h N$  implies  $M \rightarrow_\beta N$ ).  $\square$

**Theorem 2.4.15**

1. It is undecidable whether a redex in some term is strongly head needed.
2. It is undecidable whether a redex in some term is irrelevant.

**Proof**

1. By the definition of strong head neededness,  $R$  is strongly head needed in  $MR$  iff there exists a head context  $C[]$  such that  $MR \twoheadrightarrow_h C[R]$ , i.e.,  $MR =_h C[R]$ . Now the result follows by Proposition 2.4.14 since determining  $MR =_h C[R]$  is undecidable.

2. By part 1.

$\square$

## 2.5 Discussion

In this Chapter a new notion of subterms required to be contracted in the reduction of a term has been introduced. This new notion has certain advantages over previous ideas in that it gives a finer analysis of the properties of unsolvable terms—an advantage expected to have particular relevance to those parallel systems incorporating speculative evaluation.

A very precise characterisation of the new notion of strong head neededness has also been given, and in particular with respect to  $\beta$ -conversion. This will prove to be of considerable value in Chapter 4.

As in the case of strictness analysis the property that is to be analysed is not in general decidable. Thus, in practice only approximations to the complete information will be *decidably* determinable.

In the following chapter a notation for identifying strongly head needed arguments of (functional) terms is introduced. Also introduced are several *type assignment* systems for deducing elements of this notation for certain terms.

---

(The correctness of these systems is considered in Chapter 4). Clearly, by the above result any system which gives complete information for *all* terms will itself be undecidable. Such a system is presented at end of the following chapter, but first some decidable (restricted) systems are investigated.

# Chapter 3

## Reduction Types and Type Assignment

This chapter introduces an original notation for describing the reduction behaviour of  $\lambda$ -terms. This notation is a form of type and is described in the first of the four parts of this chapter. The remaining sections of the chapter are dedicated to a number of type assignment systems which associate to particular terms a range of possible types. The treatment is semantically informal as Chapter 4 is dedicated to the semantics of all constructs introduced in the current chapter.

Each of the three case studies of type deduction systems for Boolean Reduction Types takes a similar form. First, the particular sets of types are introduced and motivated. Second, the system for type deductions is presented and discussed. Finally, a study is conducted of the relationship between the type deductions systems and  $\beta$ -conversion.

### 3.1 Introduction to Reduction Types

In this section the new notation will be introduced for describing the reduction behaviour of  $\lambda$ -terms. This notation is informally based on the ideas of strong head neededness and irrelevance from Chapter 2. The formal connection between these concepts is investigated in Chapter 4.

Terms which strongly head need their argument (Definition 2.2.1) will be assigned a type of the form  $\sigma \Rightarrow \tau$ , for some  $\sigma$  and  $\tau$ . (Read this type as “ $\sigma$  strongly head needed-to  $\tau$ ”). One example would be the identity function,<sup>1</sup> which may be assigned the type  $\sigma \Rightarrow \sigma$ . Note that this type contains both

---

<sup>1</sup>Functions are identified with the equivalence classes of terms representing them. By abuse of language, terms will themselves be called “functions” on occasion.

strong head neededness information as well as all the information of a conventional type, and so is a true extension of conventional types.

Similarly, functions which make no use of their argument (“are constant on their argument”) will be assigned a type of the form  $\sigma \multimap \tau$ . (Read this type as “ $\sigma$  constant-to  $\tau$ ”). An example of such a function is  $(\lambda x.y)$ . Also  $K \equiv \lambda x.\lambda y.x$  should be assigned the type  $\sigma \Rightarrow \tau \multimap \sigma$ , for some types  $\sigma$  and  $\tau$ .

However, the examples above are of a particularly simple nature—these properties are apparent for the terms above independent of the *context* in which they are used. If the context in which a term occurs is never taken account of, then this results in a substantially less interesting formalism. Thus the context in which a term occurs must also be expressible in the proposed notation. For example, the type  $\sigma \Rightarrow \tau$  will also be assigned to terms which only strongly head need their argument (of type  $\sigma$ ) when the term is applied in sequence to some or all other arguments indicated by the type  $\tau$ .

Consider a function such as  $(\lambda f.fx)$ . This function strongly head needs its argument as  $(\lambda f.fx)M \rightarrow_h Mx$ . In the reduction sequence

$$(\lambda f.fx)(\lambda y.y) \rightarrow_h (\lambda y.y)x \rightarrow_h x,$$

it can be seen that the functional argument to  $(\lambda f.fx)$  strongly head needs *its* argument and therefore the type of  $(\lambda f.fx)$  in *this instance* may be reasonably written as  $(\alpha \Rightarrow \alpha) \Rightarrow \alpha$ . In contrast, in the reduction sequence

$$(\lambda f.fx)(\lambda x.y) \rightarrow_h (\lambda x.y)x \rightarrow_h y,$$

the functional argument of  $(\lambda f.fx)$  has a constant type and so the type of  $(\lambda f.fx)$  in this instance may be written as  $(\alpha \multimap \beta) \Rightarrow \beta$ .

There are two things to note about these two types for  $(\lambda f.fx)$ . Firstly, both of these types for  $(\lambda f.fx)$  tell us that it strongly head needs its argument. Hence, the second function type constructor in both types is the strongly head needed function type constructor. Secondly, both types disagree on the first function type constructor. While it is easy to decide which is correct given the argument to  $(\lambda f.fx)$ , the problem arises as to what type should be given to  $(\lambda f.fx)$  when no argument is present. It is of course still desirable that if an argument is eventually given to  $(\lambda f.fx)$ , then the appropriate one of the two types for  $(\lambda f.fx)$  given above should be derivable. This problem is solved by the introduction of *variable* function type constructors (or “variable arrows”), which are denoted by an arrow with a subscripted number. Thus the type of  $(\lambda f.fx)$  can be written, for any variable function type constructor  $\rightarrow_i$ , where  $i$  is some natural number, as

$$(\alpha \rightarrow_i \beta) \Rightarrow \beta.$$

This type states that

- $(\lambda f.f x)$  is a function,
- that its argument is also a function,
- that this functional argument takes an argument which is possibly independent of its result type,
- that the result type of the functional argument is the same as the result type of  $(\lambda f.f x)$ ,
- that the functional argument to  $(\lambda f.f x)$  may have any strong head needness property, and
- that  $(\lambda f.f x)$  strongly head needs its argument.

$((\alpha \rightarrow_i \beta) \Rightarrow \beta$  may be read “a function which takes a functional argument of type  $\alpha$  variable- $i$ -to  $\beta$  which is strongly head needed-to  $\beta$ ”).

Consider the function  $(\lambda f g x.f(gx))$ . To capture the kind of reduction information implicit in this term, the function type constructors are extended to a *Boolean algebra* of function type constructors. So function type constructors may now be Boolean expressions built from the function type constructors introduced above. Thus the type for this term may be concisely expressed as:

$$(\rho \rightarrow_i \tau) \Rightarrow (\sigma \rightarrow_j \rho) \rightarrow_i \sigma (\rightarrow_i \wedge \rightarrow_j) \tau.$$

Now it can be seen that the type

$$(\rho \Rightarrow \tau) \Rightarrow (\sigma \Rightarrow \rho) \Rightarrow \sigma \Rightarrow \tau$$

may also be assigned to this term. This reduction type says that  $S$  strongly head needs all three of its arguments—a fact which is only true if the first and second arguments to this term are *both* functions which strongly head need their respective arguments. Here the proposed formalism precisely captures the additional requirements of context-sensitivity, as is transparent from the structure of the more general type given to  $(\lambda f g x.f(gx))$  above.

An equivalent statement is true for types of the form  $\sigma \nrightarrow \tau$ . That is, this type will be assigned to a term which is constant on its first argument whenever this is made true by the application of the term to sufficient terms, these terms being of an appropriate type as indicated by  $\sigma \nrightarrow \tau$ . The definition of “sufficient” will depend on the form of the type itself.

Similarly, functions such as  $S \equiv (\lambda f g x.f x(gx))$  have very much more complex reduction behaviours, but these can be concisely expressed using an  $\vee$  operation in conjunction with an  $\wedge$  operation in the reduction type. A suitable type for  $S$  is

$$(\rho \rightarrow_i \sigma \rightarrow_j \tau) \Rightarrow (\rho \rightarrow_k \sigma) \rightarrow_j \rho (\rightarrow_i \vee (\rightarrow_j \wedge \rightarrow_k)) \tau.$$

$$\begin{array}{ll}
x \wedge x = x & x \vee y = y \vee x \\
x \wedge 1 = x & (x \vee y) \vee z = x \vee (y \vee z) \\
(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z) & x \vee x = x \\
x \wedge y = y \wedge x & x \vee 0 = x \\
(x \wedge y) \wedge z = x \wedge (y \wedge z) & x \vee 1 = 1 \\
x \wedge 0 = 0 & \\
\neg 1 = 0 & \\
\neg 0 = 1 & \\
\neg(x \wedge y) = \neg x \vee \neg y & \\
\neg(x \vee y) = \neg x \wedge \neg y &
\end{array}$$

Figure 4: The Axioms of a Boolean Algebra

The nice thing about this notation is the natural way it expresses information about higher-order usage of terms and the equally natural way it informs us about the polymorphic and context-sensitive attributes of a term's strong head neededness information. Indeed, it is conceivable that a most general *Boolean Reduction Type* might be assignable to a term and that this most general type contains *all* the possible behaviours of the term. As will be seen such most general types do exist in all of the type assignment systems that are considered in this thesis (see Chapter 5).

The formal meaning of reduction types is discussed in detail in Chapter 4.

### 3.1.1 Boolean Algebras

From the examples above it can be seen that a *Boolean algebra* of function type constructors is required to concisely describe the dependency relations between subterms of a term. In this Boolean algebra,  $\Rightarrow$  plays the role of the distinguished element 1 of a traditional Boolean algebra and  $\nrightarrow$  plays the role of the 0 element. Although the negation operator,  $\neg$ , has not yet entered into any of the examples, it is useful to include it, as was illustrated in Wright [72, 74], see also Chapter 6.

A Boolean algebra is a set  $B$  containing distinguished elements 0 (the zero element) and 1 (the unit element) under the operations  $\wedge$ ,  $\vee$  and  $\neg$ , where for all  $x, y$  and  $z \in B$  the axioms of Figure 4 are satisfied. Let  $=_{BA}$  denote the “=” operation over Boolean algebras defined in this Figure.

Function type constructors are built from a set of basic function type constructors called *arrows*:

#### Definition 3.1.1 (Arrows)

The set of *arrows* is  $\Delta = \Delta_g \cup \Delta_v$ , where

- the *ground arrows* ( $\Delta_g$ ) are  $\{\Rightarrow, \nrightarrow\}$ , and
- the *variable arrows* ( $\Delta_v$ ), also called *arrow variables*, are  $\{\rightarrow_1, \rightarrow_2, \dots\}$ .

The arrow  $\Rightarrow$  is called the *strong head needed* arrow and the arrow  $\nrightarrow$  is called the *constant* arrow.

Following Martin and Nipkow [45], write  $T(B, \wedge, \vee, \neg)$  for the Boolean algebra over  $B$  generated by  $\wedge$ ,  $\vee$  and  $\neg$ .

### Definition 3.1.2 (Arrow Expressions)

Let  $\nabla$  be a set of arrows, then the Boolean algebra of *arrow expressions over*  $\nabla$  is  $B_\nabla = T(\nabla, \wedge, \vee, \neg)$ . The set of *arrow expressions* is  $B_\Delta$ , the set of *ground arrow expressions* is  $B_{\Delta_g}$  and the set of *variable arrow expressions* is  $B_{\Delta_v}$ .

In the following, arrow expressions of various kinds will be considered modulo  $=_{BA}$  and the letters  $b, b', \dots, b_1, b_2 \dots$  will be used to range over them.

## 3.2 Curry-style Type Assignment

The first type assignment system introduced for Boolean Reduction Types is one in the style of Curry's system for F-deducibility (Curry and Feys [18]). This relatively simple system illustrates some features of the more sophisticated type assignment systems to come. In particular, the treatment of *variable strong head neededness functions* is elucidated, and this treatment will remain essentially unchanged in later type assignment systems.

In the following subsection the set of types for this type deduction system are defined. After this subsection the rules for forming deductions within the Curry-style system are presented and discussed. Then several example deductions are described in detail, these demonstrating the main features of the deduction system and its use.

The next subsection discusses several extensions which arise due to the unique nature of the type deduction systems presented. Some limitations of the system are discussed and some observations are made regarding the relationship of the terms which may be assigned a type in the Curry-style systems for deducing Boolean Reduction Types and certain other sets of terms.

The final major subsection presents an investigation of the relationship between the present type deduction systems and  $\beta$ -reduction and  $\beta$ -expansion. This subsection is of great importance for the following chapter in which the semantics are examined of the various constructs introduced in this chapter.

### 3.2.1 Preliminaries

Firstly, the set of admissible types is defined:



**Definition 3.2.1**

Let  $\nabla$  be a set of arrow expressions, then the set of *Abstract Simple Boolean Reduction Types*,  $T_C^\nabla$ ,<sup>2</sup> is inductively defined to be the least set satisfying:

1.  $\alpha \in \tau_v$  implies  $\alpha \in T_C^\nabla$ , and
2.  $\sigma \in T_C^\nabla$ ,  $\tau \in T_C^\nabla$  and  $b \in \nabla$  implies  $\sigma \mathbin{b} \tau \in T_C^\nabla$ .

By instantiating  $\nabla$  in the above definition many different sets of types may be generated. (Since this work is concerned with strong head neededness information only  $B_\Delta$ ,  $\Delta_g$ ,  $\{\Rightarrow\}$  and  $\{\rightarrow\}$  will be considered as instances of  $\nabla$ ). The set of *Simple Boolean Reduction Types* is  $T_C^{B_\Delta}$ . The set of *ground Boolean Reduction Types* is  $T_C^{\Delta_g}$ . The set of *hereditarily strongly head needed types* is  $T_C^{\{\Rightarrow\}}$  and the set of *hereditarily irrelevant types* is  $T_C^{\{\rightarrow\}}$ . The set of *free type and arrow variables* of a type  $\sigma$ , written  $FV(\sigma)$ , is the set of all of the type and arrow variables occurring in  $\sigma$ .

A *type assumption* is a statement of the form  $x : \tau$ , where  $x \in X$  and  $\tau \in T_C^\nabla$ , for  $\nabla = B_\Delta$ ,  $\Delta_g$ ,  $\{\Rightarrow\}$  or  $\{\rightarrow\}$ . An *assumption set* is simply a set of type assumptions, with the restriction that no term variable occurs more than once in the assumption set. The letter  $A$  (possibly with subscripts) will be used to denote an arbitrary assumption set. Since assumption sets have no more than one occurrence of each term variable they are set-theoretic functions. Thus,  $A(x)$  will denote the type associated with the variable  $x$  in  $A$ . Similarly,  $\text{dom}(A)$  will denote the set of all term variables occurring in  $A$ . Finally, write  $A_x$  for the assumption set equal to  $A$  except that any occurrence of a type assumption containing the variable  $x$  in  $A$  is removed. The set of *free type and arrow variables* of an assumption set  $A$ , written  $FV(A)$ , is  $\bigcup_{x\sigma \in A} FV(\sigma)$ .

A *term variable strong head neededness function* (or, for conciseness, *variable neededness function*) is a function of type  $X \rightarrow B_\nabla$ , for some  $\nabla$ , which denotes whether a term variable is strongly head needed in a term. Furthermore, variable neededness functions must be *total* functions, i.e., every term variable is bound within a variable neededness function. Variable neededness functions are denoted by the letter  $V$  (possibly with subscripts). If  $V$  is a variable neededness function,  $x \in X$  and  $b$  is an arrow expression, then  $V[x := b]$  denotes the variable neededness function which is everywhere identical to  $V$  except at  $x$ , where its value is  $b$ .

Let  $V_\rightarrow$  denote any variable neededness function with the property that  $\forall x \in X. V_\rightarrow(x) =_{B_A} \rightarrow$ . For example, the expression  $x$  should have the variable neededness function  $V_\rightarrow[x := \Rightarrow]$  associated with it since  $x$  is strongly head needed in  $x$ , but for all  $y \neq x$   $y$  is irrelevant in the expression  $x$ . In fact, for any legal derivation (see below) variable neededness functions will be completely determined by the assumption set and term used in the deduction of

<sup>2</sup>The  $C$  is short for Curry.

a type for that term. To avoid confusion with the symbol  $\lambda$  used as the abstractor in  $\lambda$ -terms, functions in the meta-language will be constructed using a meta-abstractor which will be written as  $\underline{\lambda}$ . The set of free type and arrow variables of a variable strong head neededness function  $V$ , written  $FV(V)$ , is  $\bigcup_{x \in X} FV(V(x))$ , where the set of free variables of an arrow expression is all the arrow variables occurring in the arrow expression.

A *typing statement* is a quadruple of an assumption set  $A$ , a variable neededness function  $V$ , a term  $M$ , and a type  $\tau$ . A typing statement will be written as

$$A \vdash_V^* M : \tau,$$

where  $*$  will be a letter denoting a particular deduction system. For example, in this section the deduction system of interest is the Curry-style system and so typing statements will be written as

$$A \vdash_V^C M : \tau.$$

In the statements of lemmata or theorems, unquantified statements of the form:

$$A \vdash_V^C M : \tau,$$

are intended to be interpreted as saying  $\exists A, V, \tau. A \vdash_V^C M : \tau$  is a legal derivation for  $M \in \Lambda$  in the Curry-style type assignment system for Boolean Reduction Types (and similarly for later systems of type assignment).

### 3.2.2 The Type Assignment Rules

The deduction system takes the form of an inference rule or axiom for each kind of  $\lambda$ -term (see Figure 5). This figure in fact defines *four* type deduction systems, depending on whether the set of types is chosen to be  $T_C^{B\Delta}$ ,  $T_C^{\Delta\sigma}$ ,  $T_C^{\{\Rightarrow\}}$  or  $T_C^{\{\rightarrow\}}$ . Since terms are built from application and abstraction of terms and term variables, there are three corresponding rules. The form of each of these rules is now informally described.

#### The VAR Rule

For term variables an assumption about what types they belong to must be made. Each occurrence of the term variable will then be assigned the type assumed for it. Thus, this rule insists that the usage of a term variable is consistent across all its occurrences in a  $\lambda$ -term.

As for the variable neededness function  $V$ , note that in a term consisting of a single term variable, if this variable is abstracted the identity function is obtained. Hence, the variable neededness function should assign the value  $\Rightarrow$

VAR	$A_x \cup \{x : \sigma\} \vdash_{V_{\rightarrow}[x := \Rightarrow]}^C x : \sigma$
APP	$\frac{A \vdash_{V_1}^C N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^C N_2 : \sigma}{A \vdash_V^C N_1 N_2 : \tau}$ $(V = \lambda x. V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := b]}^C N : \tau}{A \vdash_{V[x := \rightarrow]}^C \lambda x. N : \sigma \text{ b } \tau}$

Figure 5: The Curry-style Rules for deducing Reduction Types

to the variable. Since there are no other term variables in the term, all other variables should be assigned the value  $\rightarrow$  by the variable neededness function. Hence, the appropriate variable neededness function is  $V = V_{\rightarrow}[x := \Rightarrow]$ .

### The APP Rule

For the case of an application of two terms, both terms should have legal typing statements derivable for them in order for the APP rule to be applicable. In addition, the term used in a functional manner in the application must have a functional type. Moreover, the argument type of this functional type must also agree with the type of the argument term in the application. The last requirement is that the type assumptions made for free variables in the two terms by their respective typing statements should be the same.

The typing statement for the application of the two terms can then be built from these typing statements. The chief operation here is the construction of the variable neededness function for the combined terms. Any free variable head needed by the functional term in the application will be head needed in the application. Free variables in the argument term will be head needed by the application if the argument term is head needed by the functional term and if the argument term itself head needs them.

### The ABS Rule

Consider the cases where the set of types is  $T_C^{B\Delta}$ ,  $T_C^{\Delta\sigma}$ , or  $T_C^{\{\rightarrow\}}$ . For an abstraction, there is the possibility that the variable being abstracted may not appear

free in the term.<sup>3</sup> In this case, it is clear that the variable neededness function should map the variable to  $\rightarrow$ . (Note how the choice of the variable neededness function in the VAR rule will fulfill this requirement). Of course, the assumption set must still assign a type to the variable even if it does not appear free in the term. Given a legal typing statement which satisfies these requirements, the typing statement for the abstraction can be constructed. The type assigned to the abstraction should be that of a function from the type of the abstraction variable to the type of the expression, with neededness determined by the variable neededness function.

Consider the case where the set of types is  $T_C^{\{\rightarrow\}}$ . An application of the rule for abstraction may only be made if the abstraction variable definitely occurs as a free variable in the term being abstracted. Thus the set of terms for which types can be found in this system would be expected to be a subset of Church's  $\lambda I$ -terms (see Barendregt [3] for a discussion of these terms). (Baker-Finch [2] has investigated the connection between the system  $T_C^{\{\rightarrow\}}$  and Church's  $\lambda I$ -terms in greater detail).

Since an abstraction closes the scope of the abstraction variable, any further abstraction of that variable will result in a function which does not have the variable free within it, thus the variable neededness function should have its entry for the abstraction variable reset to  $\rightarrow$  in the resultant typing statement. In fact, for a valid typing statement derived using a type deduction system the variable neededness function,  $V$ , is completely determined by the structure of the term and the type assumption set,  $A$ , and so is not written as an assumption. To see this, note that the variable neededness function is completely determined for each use of rule VAR. By induction it is straightforward to see that rules APP and ABS also completely determine the variable neededness function.

In the case of the type assumption set, the current entry for the abstraction variable should be deleted and any previous entry for it re-instated. (This is because abstractions may nest the use of term variables and this requires that earlier assumptions for that variable be made valid again).

Note again that at each stage of the deduction of a typing statement the variable neededness function is completely specified.

### 3.2.3 Examples

In this subsection some initial simple examples are presented of deductions using the above type assignment system. It is hoped that these will help the reader to understand how strong head neededness information is derived. All

---

<sup>3</sup>In the case of  $T_C^{\{\rightarrow\}}$  the variable being abstracted may appear free in the term, for example in  $\lambda x.yx$ ,  $y$  may be assumed to have type  $\sigma \rightarrow \tau$ , for some  $\sigma$  and  $\tau$ .

derivations use the system based on  $T_C^{B\Delta}$ , unless stated otherwise.

Let the symbol  $\emptyset$  denote the empty set.

**Example 3.2.2**

Consider  $K \equiv \lambda b.a$ . By rule VAR, under the assumption set  $A \equiv \{a: \sigma, b: \tau\}$ , we obtain:

$$A \vdash_{V_{\rightarrow}[a := \Rightarrow]}^C a: \sigma.$$

Now we take an instance of the ABS rule to obtain:

$$A_b \vdash_{V_{\rightarrow}[a := \Rightarrow]}^C \lambda b.a: \tau \rightarrow \sigma.$$

Note how the variable neededness function is in this example unaffected by the abstraction and how the form of this function provides us with the correct value for the strong head neededness of  $b$  in this term. Finally, a last instance of rule ABS is required:

$$A_{ba} \equiv \emptyset \vdash_{V_{\rightarrow}}^C \lambda a b.a: \sigma \Rightarrow \tau \rightarrow \sigma.$$

The main point to notice here is that since this term is now closed any further abstractions of this term should have the constant arrow assigned as their strong head neededness value, as is indeed satisfied by the variable neededness function  $V_{\rightarrow}$ .

A type for this term is only derivable in the systems based on  $T_C^{B\Delta}$  and  $T_C^{\Delta\sigma}$ .

**Example 3.2.3**

Now consider a term with an explicit higher-order function,  $\lambda x f.f x$ . Choose  $A \equiv \{f: \sigma \rightarrow_i \tau, x: \sigma\}$ , then by rule VAR:

$$A \vdash_{V_{\rightarrow}[x := \Rightarrow]}^C x: \sigma.$$

A further instance of rule VAR is required:

$$A \vdash_{V_{\rightarrow}[f := \Rightarrow]}^C f: \sigma \rightarrow_i \tau.$$

Now rule APP is applicable and we obtain:

$$A \vdash_V^C f x: \tau,$$

where  $V = \lambda y.(V_{\rightarrow}[f := \Rightarrow, x := \rightarrow_i])(y)$ . Note that all term variables other than  $f$  and  $x$  are mapped to  $\rightarrow$  by  $V$  and that this is as we would expect for the term  $f x$ . Now follow two occurrences of rule ABS, with result:

$$A_{fx} \equiv \emptyset \vdash_{V_{\rightarrow}}^C \lambda x f.f x: \sigma \rightarrow_i (\sigma \rightarrow_i \tau) \Rightarrow \tau.$$

This type agrees with our intuition that the strong head neededness of  $x$  in this term is dependent on the strong head neededness of the argument to the

function  $f$ . In this way higher-order functions are dealt with in a simple and effective manner. This naturally has a positive impact on implementations of this system, as will be seen later (Chapter 5 and Appendix A). This should be contrasted with systems based on conventional abstract interpretations in which polymorphic higher-order functions cause some difficulty (see Chapter 7).

A type for this term may be derived in all but the system based on  $T_C^{\{\rightarrow\}}$ . Note that in the systems based on  $T_C^{\Delta_g}$  and  $T_C^{\{\Rightarrow\}}$  the derivations of types for this term involve only ground types. In particular, for  $T_C^{\Delta_g}$  derivations of the following form (for some  $A$ ) are derivable:

$$A \vdash_{V_{\rightarrow}}^C \lambda x f.f x : \sigma \rightarrow (\sigma \rightarrow \tau) \Rightarrow \tau$$

and, for  $T_C^{\Delta_g}$  as well as for  $T_C^{\{\Rightarrow\}}$ , derivations of the form below (for some  $A$ ) are derivable:

$$A \vdash_{V_{\rightarrow}}^C \lambda x f.f x : \sigma \Rightarrow (\sigma \Rightarrow \tau) \Rightarrow \tau.$$

#### Example 3.2.4

Now consider an example in which a variable occurs more than once, say  $\lambda f x.f(f x)$ . This function (the Church numeral  $c_2$ ) has best type  $(\sigma \rightarrow \sigma) \rightarrow \sigma \rightarrow \sigma$ , for some  $\sigma$ , in the system for F-deducibility of Curry and Feys [18]. (We shall see a better type for this term in a later deduction system). In the present system, we can reason as follows. Let  $A \equiv \{f : \sigma \rightarrow; \sigma, x : \sigma\}$ , then by rule VAR:

$$A \vdash_{V_{\rightarrow}[x := \Rightarrow]}^C x : \sigma,$$

and by rule VAR again:

$$A \vdash_{V_{\rightarrow}[f := \Rightarrow]}^C f : \sigma \rightarrow; \sigma.$$

By rule APP:

$$A \vdash_{V_1}^C f x : \sigma,$$

where  $V_1 = \lambda y.(V_{\rightarrow}[f := \Rightarrow])(y)$ . Again by rule APP,

$$A \vdash_{V_2}^C f(f x) : \sigma,$$

where  $V_2 = \lambda y.(V_{\rightarrow}[f := \Rightarrow])(y) \vee (\rightarrow; \wedge V_1(y))$ . In fact  $V_2 = V_1$  since the present system forces us to give identical types to each occurrence of a term variable, even if there is more than one occurrence. Finally after two instances of rule ABS, we obtain

$$A_{xf} \equiv \emptyset \vdash_{V_{\rightarrow}}^C \lambda f x.f(f x) : (\sigma \rightarrow; \sigma) \Rightarrow \sigma \rightarrow; \sigma.$$

A type for this term may be derived in all but the system based on  $T_C^{\{\rightarrow\}}$ .

### Example 3.2.5

As a variation on the previous example, consider  $\lambda f g x. f(gx)$ . Let  $A \equiv \{f : \sigma \rightarrow_i \tau, g : \rho \rightarrow_j \sigma, x : \rho\}$ , then by rule VAR:

$$A \vdash_{V_{\rightarrow}[x:=\Rightarrow]}^C x : \rho,$$

and by rule VAR again:

$$A \vdash_{V_{\rightarrow}[g:=\Rightarrow]}^C g : \rho \rightarrow_j \sigma.$$

By rule APP:

$$A \vdash_{V_1}^C gx : \sigma,$$

where  $V_1 = \lambda y. (V_{\rightarrow}[g := \Rightarrow])(y)$ . Again by rule APP,

$$A \vdash_{V_2}^C f(gx) : \tau,$$

where  $V_2 = \lambda y. (V_{\rightarrow}[f := \Rightarrow])(y) \vee (\rightarrow_i \wedge V_1(y))$ . In contrast to the previous example  $V_2 \neq V_1$  since we can now distinguish between the two applications by the types alone. Finally after three instances of rule ABS, we obtain

$$A_{xgf} \equiv \emptyset \vdash_{V_{\rightarrow}}^C \lambda f g x. f(gx) : (\sigma \rightarrow_i \tau) \Rightarrow (\rho \rightarrow_j \sigma) \rightarrow_i \rho (\rightarrow_i \wedge \rightarrow_j) \tau.$$

This type clearly describes the dependence of  $x$  on both  $f$  and  $g$ .

### Example 3.2.6

As an example which has a logical disjunction of strong head neededness properties in its final type, consider  $\lambda f x. fxx$ . Suppose  $A \equiv \{f : \sigma \rightarrow_i \sigma \rightarrow_j \tau, x : \sigma\}$ , then the following deduction is easily found:

$$A \vdash_{V_{\rightarrow}[x:=\Rightarrow]}^C x : \sigma,$$

and

$$A \vdash_{V_{\rightarrow}[f:=\Rightarrow]}^C f : \sigma \rightarrow_i \sigma \rightarrow_j \tau,$$

both by rule VAR;

$$A \vdash_{V_1}^C fx : \sigma \rightarrow_j \tau,$$

where  $V_1 = \lambda y. (V_{\rightarrow}[f := \Rightarrow])(y)$ , by rule APP;

$$A \vdash_{V_2}^C fxx : \sigma \rightarrow_j \tau,$$

where  $V_2 = \lambda y. V_1(y) \vee (\rightarrow_j \wedge (V_{\rightarrow}[x := \Rightarrow])(y))$ , again by rule APP; hence

$$A_{xf} \equiv \emptyset \vdash_{V_{\rightarrow}}^C \lambda f x. fxx : (\sigma \rightarrow_i \sigma \rightarrow_j \tau) \Rightarrow \sigma (\rightarrow_i \vee \rightarrow_j) \tau,$$

by two instances of rule ABS. Note that this type accurately records our intuition that  $x$  may be strongly head needed through either of its occurrences (consider when this term is applied to  $\lambda ab.a$  or  $\lambda ab.b$ ).

**Example 3.2.7**

Now, an example which involves the use of both  $\vee$  and  $\wedge$ ,  $S \equiv \lambda fgx.fx(gx)$ . For this term let  $A \equiv \{f: \rho \rightarrow_i \sigma \rightarrow_j \tau, g: \rho \rightarrow_k \sigma, x: \rho\}$ , then we derive some instances of the VAR rule:

$$A \vdash_{V_{\rightarrow}[f := \Rightarrow]}^C f: \rho \rightarrow_i \sigma \rightarrow_j \tau,$$

$$A \vdash_{V_{\rightarrow}[g := \Rightarrow]}^C g: \rho \rightarrow_k \sigma$$

and

$$A \vdash_{V_{\rightarrow}[x := \Rightarrow]}^C x: \rho.$$

Now we can make three uses of instances of rule APP:

$$A \vdash_{V_1}^C fx: \sigma \rightarrow_j \tau,$$

where  $V_1 = \lambda y.(V_{\rightarrow}[f := \Rightarrow])(y)$ ,

$$A \vdash_{V_2}^C gx: \sigma,$$

where  $V_2 = \lambda y.(V_{\rightarrow}[g := \Rightarrow])(y)$  and

$$A \vdash_{V_3}^C fx(gx): \tau,$$

where  $V_3 = \lambda y.V_1(y) \vee (\rightarrow_j \wedge V_2(y))$ . At this point one can trace through what the strong head neededness values for  $f$ ,  $g$  and  $x$  are in  $V_3$  and so after three uses of the ABS rule it is not too surprising that we obtain:

$$\emptyset \vdash_{V_{\rightarrow}}^C S: (\rho \rightarrow_i \sigma \rightarrow_j \tau) \Rightarrow (\rho \rightarrow_k \sigma) \rightarrow_j \rho (\rightarrow_i \vee (\rightarrow_j \wedge \rightarrow_k)) \tau.$$

Here we see that the neededness of  $g$  is dependent on whether  $f$  requires its second argument, whereas  $x$  is dependent on whether  $f$  needs its first argument or whether  $f$  needs its second argument and  $g$  needs its sole argument.

Note that nowhere has any use of the negation operator ( $\neg$ ) been made in these examples and the type assignment system. In fact no use in this thesis will be made of this operator for any system involving only pure  $\lambda$ -terms. However, when typical constants are added to the sets of terms and types it turns out that such an operator is of great utility (though clearly not essential), see Chapter 6 as well as Wright [72, 74].



VAR	$A_x \cup \{x : \sigma\} \vdash_{V \rightarrow [x := \Rightarrow]}^C x : \sigma$
APP	$\frac{A \vdash_{V_1}^C N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^C N_2 : \sigma}{A \vdash_V^C N_1 N_2 : \tau}$ $(V = \underline{\lambda}x.V_1(x) \vee (\text{b} \wedge V_2(x)))$
APP- $\rightarrow$	$\frac{A \vdash_{V_1}^C N_1 : \sigma \rightarrow \tau \quad A' \vdash_{V_2}^C N_2 : \sigma'}{A \vdash_{V_1}^C N_1 N_2 : \tau}$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := \text{b}]}^C N : \tau}{A \vdash_{V[x := \rightarrow]}^C \underline{\lambda}x.N : \sigma \text{ b } \tau}$

Figure 6: Extended Curry-style Rules for deducing Reduction Types

### 3.2.4 An Extension of the Curry-style System

In Figure 6 a simple extension of the Curry-style system is presented. This extension makes direct use of the intensional information available in Boolean Reduction types to support an additional form of application. In this rule, named APP- $\rightarrow$  in the Figure, advantage is taken of the fact that any term which is constant on its argument will, by definition, make no use of that argument in computing the result of the expression. Note that this rule (APP- $\rightarrow$ ) would only be expressible in a simplistic way in a conventional system not based on Boolean Reduction Types (by restricting consideration to terms of the form  $(\underline{\lambda}x.M)N$ , where  $x \notin \text{FV}(M)$ ).

That the APP- $\rightarrow$  rule is indeed a degenerate case of the APP rule (*as far as strong head neededness information is concerned*) can be seen by examining the way the variable neededness function is constructed from its antecedents in the APP rule. Clearly, in the construction of

$$V = \underline{\lambda}x.V_1(x) \vee (\rightarrow \wedge V_2(x)),$$

the value of  $V_2$  is irrelevant and  $V$  is the same as  $V_1$ .

#### Definition 3.2.8

If a type  $\sigma \rightarrow \tau$  is deduced for a term  $P$ , then by abuse of language it will be said that  $Q$  (and all its subterms) is *irrelevant* to the deduction of a type for the term  $PQ$ .

Similarly, a deduction of  $A' \vdash_V^C N : \sigma'$  is an *irrelevant subdeduction* of a

deduction

$$\frac{A \vdash_V^C M : \sigma \multimap \tau \quad A' \vdash_V^C N : \sigma'}{A \vdash_V^C MN : \tau}.$$

(In the case that rule APP is used it will also be true that  $A = A'$  and  $\sigma = \sigma'$ ).

The APP- $\multimap$  rule fundamentally extends the system presented in that it allows the typing of (irrelevant) self-application. Consider:

$$\{x : \sigma \multimap \tau\} \vdash_{V \multimap [x := \Rightarrow]}^C x : \sigma \multimap \tau$$

and

$$\{x : \rho\} \vdash_{V \multimap [x := \Rightarrow]}^C x : \rho,$$

for  $\rho$ ,  $\sigma$  and  $\tau$  arbitrary types. Then rule APP- $\multimap$  is applicable and so:

$$\{x : \sigma \multimap \tau\} \vdash_{V \multimap [x := \Rightarrow]}^C xx : \tau.$$

Similarly, the term  $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$  can now be assigned any type of the form:

$$(\rho \multimap \sigma) \Rightarrow \tau,$$

for  $\rho$ ,  $\sigma$  and  $\tau$  arbitrary types. Note that any typable term of the form  $YM$  is always normalisable (but clearly is not strongly normalising), since  $YM =_\beta M(YM)$  and  $YM$  is an irrelevant argument to  $M$ .

An alternative would be to specify rule APP- $\multimap$  in the form:

$$\frac{A \vdash_{V_1}^C N_1 : \sigma \multimap \tau}{A \vdash_{V_1}^C N_1 N_2 : \tau},$$

where  $N_2 \in \Lambda$ . This would allow terms such as  $(\lambda x.y)((\lambda z.zz)(\lambda z.zz))$  to be typed in the system. In contrast, the form of the rule originally given requires that *some* type be assignable to the term under an arbitrary assumption set. (Clearly, no type can be assigned to  $(\lambda z.zz)(\lambda z.zz)$  in the present system including rule APP- $\multimap$ ).

In the rest of the thesis, it is the system of Figure 5 which will be referred to as the ‘‘Curry-style system’’. However, in the investigation in this section the rule APP- $\multimap$  of Figure 6 *will* be considered as part of the system unless otherwise specified. The main reason for including this rule is that in its presence a full strength result can be established concerning  $\beta$ -expansion (see Theorem 3.2.24).

Finally, note that the rule APP- $\multimap$  is not definable for the system based on  $T_C^{\{\Rightarrow\}}$ , and so all results concerning this rule are implicitly only statements concerning the other three systems.

### 3.2.5 Multiple Occurrences of Term Variables

The type assignment systems of Figure 5 have the property that, like Curry's system of F-deducibility, every variable bound by a  $\lambda$ -abstraction must have a single type. However, since Boolean Reduction Types contain information about how a function will use its argument, the type assignment system of Figure 5 is *more* restrictive than F-deducibility<sup>4</sup> (in the sense of the class of  $\lambda$ -terms which may be assigned a type). Note that this is still an elegant result as the present type assignment system has exactly the same constraints as the system of F-deducibility—it is just that far more information is contained about  $\lambda$ -terms in a Boolean Reduction Type when compared to the conventional types of F-deducibility. The following example is useful for illustrating this behaviour.

#### Example 3.2.9

Consider the term  $\lambda f.g(f(\lambda x.x))(f(\lambda b.a))$ . In the system of F-deducibility a type for this term is  $((\rho \rightarrow \rho) \rightarrow \sigma) \rightarrow \tau$ , for some  $\rho$ ,  $\sigma$  and  $\tau$ , under the assumptions  $a: \rho$  and  $g: \sigma \rightarrow \sigma \rightarrow \tau$ .

In the present system we may deduce the following type for the subterm  $\lambda x.x$  of this term:

$$A \vdash_{V_{\rightarrow}}^C \lambda x.x: \sigma \Rightarrow \sigma.$$

(For any  $A$  and  $\sigma$ ). In contrast, we deduce:

$$A'_a \cup \{a: \tau\} \vdash_{V_{\rightarrow}[a:=\Rightarrow]}^C \lambda b.a: \tau' \dashv\rightarrow \tau,$$

for any  $\tau$ ,  $\tau'$  and  $A'$ . Now, even if we choose  $\tau = \tau' = \sigma$  and  $A = A'_a \cup \{a: \tau\}$ , we cannot find a single type to assume for  $f$  such that we can deduce a type for the term  $\lambda f.g(f(\lambda x.x))(f(\lambda b.a))$ . The problem is that the arrows assigned to the arguments of the two occurrences of  $f$  differ and are irreconcilable. Thus this example proves that the system presented here can only deduce types for a *subset* of those terms which can be assigned a type by the system of F-deducibility (in that system the typable terms are known as the *simply-typable terms*). Later in this chapter systems will be investigated which alleviate this restriction.

Note that, under the assumption that  $f$  does not require its argument, the above term may be assigned a type in the systems of Figure 6. (Simply by making use of an instance of APP- $\dashv\rightarrow$  in typing the subterm  $f(\lambda x.x)$ ).

To summarise, the only terms for which a type may be deduced in the present system are those whose every variable (both free and bound, as appropriate) is used in both an extensionally *and* intensionally identical fashion. This should be contrasted with systems such as the  $\lambda I$ -calculus (in which at least

<sup>4</sup>This is the system of Curry type deduction, as described in Chapter 1.

one occurrence of a variable must occur in a term if that variable is abstracted), and sets of *linear* terms (in which exactly one occurrence of a variable must occur in a term). It is clear that the set of linear terms is contained within the set of  $\lambda I$ -terms, as well as the set of terms derivable in the present system when based upon  $T_C^{\{\Rightarrow\}}$  (see Baker-Finch [2] for a more detailed examination of this point). So in the present system all linear terms are typable, but not all  $\lambda I$ -terms (for example  $\lambda x.xx$  is a  $\lambda I$ -term which is not typable in the system excluding rule APP- $\rightarrow$ ). On the other hand there are typable terms in the present system (except when based upon  $T_C^{\{\Rightarrow\}}$ ) which are not  $\lambda I$ -terms (for example,  $K \equiv \lambda xy.x$ ).

**Definition 3.2.10**

1. The set of  $\lambda I$ -terms,  $\Lambda I$ , are inductively defined to be the least set satisfying:
  - $x \in X$  implies  $x \in \Lambda I$ ,
  - $M, N \in \Lambda I$  implies  $MN \in \Lambda I$ , and
  - $M \in \Lambda I$  and  $x \in \text{FV}(M)$  implies  $\lambda x.M \in \Lambda I$ .
2. The set of *linear*  $\lambda$ -terms,  $\Lambda L$ , are inductively defined to be the least set satisfying:
  - $x \in X$  implies  $x \in \Lambda L$ ,
  - $M, N \in \Lambda L$  and  $\text{FV}(M) \cap \text{FV}(N) = \emptyset$  implies  $MN \in \Lambda L$ , and
  - $M \in \Lambda L$  and  $x \in \text{FV}(M)$  implies  $\lambda x.M \in \Lambda L$ .

Let  $\Lambda T_C^\nabla$  denote the set of terms for which a type may be found in the Curry-style system when the set of types used is  $T_C^\nabla$ . Then the following relationships ensue:

1.  $\Lambda L \subset \Lambda T_C^{\{\Rightarrow\}} \subset \Lambda T_C^{\Delta_g} \subset \Lambda T_C^{B_\Delta} \subset \Lambda$ , and
2.  $\Lambda L \subset \Lambda T_C^{\{\Rightarrow\}} \subset \Lambda I \subset \Lambda$ .

(See the proposition below).

**Proposition 3.2.11**

1.  $M \in \Lambda L$  implies  $M$  is typable in the Curry-style system  $\Lambda T_C^{\{\Rightarrow\}}$ .
2. If  $M \in \Lambda$  can be assigned a type in the Curry-style system without containing any irrelevant sub-deduction, then  $M \in \Lambda I$ .

**Proof**

1. Easy induction on  $M \in \Lambda L$ . (In the case of an application we can use the *Strengthening Lemma*, Lemma 3.2.18, which is proved later in this section, to derive deductions for the two components of the application such that their assumption sets are disjoint. Then use Lemma 3.2.17 to find the same types for them under a common assumption set. Finally, apply rule APP).
2. Again, a straightforward induction since if  $M \equiv \lambda x.N$ , then  $x \in \text{FV}(N)$ , as required.

□

The key point is that the richness of the basic framework presented thus far in the thesis allows the *incorporation of more sophisticated logical systems of type assignment in a straightforward manner*. This is demonstrated in later sections of this chapter in which a progression of increasingly powerful type assignment systems using this framework are presented.

### 3.2.6 Properties of the Curry-style System

In this subsection some fundamental properties of the Curry-style deduction systems for Boolean Reduction Types are examined. Naturally, the focus is on reduction and its relationship to the deduction system. Later the relation between reduction and Boolean Reduction Types themselves will be considered (Chapter 4).

The main results established are that the type deduction system always respects  $\beta$ -contraction (Theorem 3.2.21) and that in certain situations it respects  $\beta$ -expansion (Theorems 3.2.23 and 3.2.24). This means that if a type and variable neededness function is derived for a certain term, then every  $\beta$ -contractum of that term can have assigned to it the same type and variable neededness function. In certain situations, as described by the Theorems themselves, the same is true for  $\beta$ -expandums of a term.

Note that if a result is proved for  $T_C^\nabla$  or the set of types is not specified, then the intended meaning of this is that the result holds for all four systems of Curry-style deduction. A property is *generic* for a deduction system based on  $T_C^\nabla$  if it holds for all subsystems of  $T_C^\nabla$ . This genericity is of great help in simplifying the analysis. Fortunately, there are only a few places in the following investigation which are not generic.

Firstly, for a given term and assumption set, if two deductions give identical types, then their variable neededness functions must be equivalent.

**Lemma 3.2.12**

Suppose  $\exists A, \sigma, V_1, V_2. A \vdash_{V_1}^C M : \sigma$  and  $A \vdash_{V_2}^C M : \sigma$ , then  $\forall x \in X. V_1(x) =_{\text{BA}} V_2(x)$ .

**Proof**

By induction on the structure of  $M$ .

The interesting case is when  $M$  is an application-term,  $M_1 M_2$ . In this case the result follows from the induction hypotheses (for each combination of rules used) and the definition of  $=_{BA}$ .  $\square$

The following three Lemmas are *trivial* in the current system, though they will not be in the study of the system which includes type intersection (which appears later in this chapter). These Lemmas are included here for consistency with the latter system. The first is essentially the inverse of the APP rule.

**Lemma 3.2.13**

If  $A \vdash_V^C MN : \tau$  and rule APP is the final rule used in this deduction, then  $\exists \sigma \in T_C^\nabla, b \in \nabla, V', V''.(A \vdash_{V'}^C M : \sigma \text{ b } \tau, A \vdash_{V''}^C N : \sigma \text{ and } \forall x \in X. V = \lambda x.V'(x) \vee (b \wedge V''(x)))$ .

**Proof**

By rule APP, since this was the final rule used in the derivation of  $A \vdash_V^C MN : \tau$ .  $\square$

The inverse of rule APP- $\rightarrow$ :

**Lemma 3.2.14**

If  $A \vdash_V^C MN : \tau$  and rule APP- $\rightarrow$  is the final rule applied in this deduction, then there exists  $\sigma, \sigma' \in T_C^\nabla$  and assumption set  $A'$  such that  $(A \vdash_{V'}^C M : \sigma \rightarrow \tau$  and  $A' \vdash_{V''}^C N : \sigma')$ .

**Proof**

By rule APP- $\rightarrow$ .  $\square$

Similarly, the inverse of the ABS rule:

**Lemma 3.2.15**

$A \vdash_V^C \lambda x.M : \sigma \text{ b } \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_{V[x:=b]}^C M : \tau$ .

**Proof**

By rule ABS, since a type can only be deduced for  $\lambda x.M$  by using this rule.  $\square$

Some more general statements about the APP rule can be made:

**Corollary 3.2.16**

Let  $M \equiv NN_1 \dots N_n$  where  $N$  is not an application term and  $n \geq 0$ . Suppose  $A \vdash_V^C M : \tau$ , without using rule APP- $\rightarrow$ , then  $\exists \rho_i \in T_C^\nabla, b_i \in \nabla$  and  $V_i$  ( $1 \leq i \leq n$ ):

1.  $A \vdash_{V_i}^C N_i : \rho_i, A \vdash_{V'}^C N : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$  and  $V = \lambda x.V'(x) \vee (V_{i=1}^n(b_i \wedge V_i(x)))$ ,

2. if  $N \equiv x$ , then  $A = A'_x \cup \{x : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau\}$  and  $V' = V_{\rightarrow}[x := \Rightarrow]$ , and
3. if  $N \equiv \lambda x.N'$ , then  $A_x \cup \{x : \rho_1\} \vdash_{V'[x:=b_1]}^C N' : \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$  and  $V'(x) = \rightarrow$ .

**Proof**

1. By repeated use of Lemma 3.2.13.
2. By part 1 and rule VAR, since this is the only rule which can be used to find a type for  $x$ .
3. By part 1 and rule ABS, since this is the only rule which can be used to find a type for  $\lambda x.N'$ .

□

It turns out that the following rules for *strengthening* and *weakening* are derivable in the present system:

$\text{STR} \quad \frac{A_x \cup \{x : \sigma\} \vdash_V^C M : \tau}{A_x \vdash_V^C M : \tau} \quad (x \notin \text{FV}(M))$ $\text{WEAK} \quad \frac{A \vdash_V^C M : \tau}{A_x \cup \{x : \sigma\} \vdash_V^C M : \tau} \quad (x \notin \text{FV}(M))$
--

This is demonstrated by the following two lemmas:

**Lemma 3.2.17**

Suppose  $x \notin \text{FV}(M)$ , then  $A \vdash_V^C M : \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_V^C M : \tau$ .

**Proof**

Straitforward induction over the deduction of  $A \vdash_V^C M : \tau$ . □

**Lemma 3.2.18**

If  $A_x \cup \{x : \sigma\} \vdash_V^C M : \tau$  and  $x \notin \text{FV}(M)$ , then  $A_x \vdash_V^C M : \tau$ .

**Proof**

By induction over the deduction of  $A \vdash_V^C M : \tau$ . Use Lemma 3.2.17 for the case of  $M$  an abstraction term. □

The above properties are used to establish several results in the rest of this section.

Now it is time to consider the effect of term substitution (see Chapter 1). The primary interest in the following two lemmas is type preservation, *i.e.*,

when a term is transformed by substitution, what conditions need to be imposed in order to preserve its type? Firstly, the effect of undoing a substitution is examined:

**Lemma 3.2.19**

Suppose  $A \vdash_V^C M[x := N] : \tau$ , only the type  $\sigma$  is assigned to  $N$  in this deduction and  $A \vdash_V^C N : \sigma$ , then there exist  $b$  and  $V''$  such that:

- $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^C M : \tau$ , and
- $V = \lambda y. V''(y) \vee (b \wedge V'(y))$ .

**Proof**

By induction on  $M[x := N]$ .

$x[x := N] \equiv N$  In this case  $\sigma = \tau$  and  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^C x : \tau$ , where  $V'' =_{BA} V_{\rightarrow}$  and  $b =_{BA} \Rightarrow$ . Now,

$$\begin{aligned} V &= \lambda y. V'(y) \\ &= \lambda y. V_{\rightarrow}(y) \vee (\Rightarrow \wedge V'(y)) \\ &= \lambda y. V''(y) \vee (b \wedge V'(y)), \end{aligned}$$

as required.

$z[x := N] \equiv z$  ( $z \neq x$ ) It is immediate that

$$A \vdash_{V_{\rightarrow}[z:=\Rightarrow]}^C z : \tau,$$

so  $V'' =_{BA} V_{\rightarrow}[z := \Rightarrow]$  and  $b =_{BA} \rightarrow$ . Now, for any  $V'$ ,

$$\begin{aligned} V &= \lambda y. (V_{\rightarrow}[z := \Rightarrow])(y) \vee (\rightarrow \wedge V'(y)) \\ &= \lambda y. V''(y) \vee (b \wedge V'(y)). \end{aligned}$$

$N_1 N_2[x := N] (\equiv (N_1[x := N])(N_2[x := N]))$  and the last rule used is rule APP. By Lemma 3.2.13,

- $A \vdash_{V_1}^C N_1[x := N] : \rho \text{ b' } \tau$ , and
- $A \vdash_{V_2}^C N_2[x := N] : \rho$ ,

where  $V = \lambda y. V_1(y) \vee (b' \wedge V_2(y))$ . By the induction hypotheses

- $A_x \cup \{x : \sigma\} \vdash_{V_1''[x:=b_1]}^C N_1 : \rho \text{ b' } \tau$ , and
- $A_x \cup \{x : \sigma\} \vdash_{V_2''[x:=b_2]}^C N_2 : \rho$ ,



where  $V_1(y) =_{\text{BA}} V_1''(y) \vee (b_1 \wedge V'(y))$  and  $V_2(y) =_{\text{BA}} V_2''(y) \vee (b_2 \wedge V'(y))$ . By APP,  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^C N_1 N_2 : \tau$ , where  $V''(y) =_{\text{BA}} V_1''(y) \vee (b' \wedge V_2''(y))$  and  $b =_{\text{BA}} b_1 \vee (b' \wedge b_2)$ . As is conventional, assume that  $\wedge$  binds tighter than  $\vee$  and also temporarily replace  $\wedge$  by juxtaposition, then by rule APP,

$$\begin{aligned} V &= \underline{\lambda}y. (V_1''(y) \vee b_1 V'(y)) \vee (b' (V_2''(y) \vee b_2 V'(y))) \\ &= \underline{\lambda}y. V_1''(y) \vee b_1 V'(y) \vee b' (V_2''(y) \vee b_2 V'(y)) \\ &= \underline{\lambda}y. V_1''(y) \vee b_1 V'(y) \vee b' V_2''(y) \vee b' b_2 V'(y) \\ &= \underline{\lambda}y. V_1''(y) \vee b' V_2''(y) \vee b_1 V'(y) \vee b' b_2 V'(y) \\ &= \underline{\lambda}y. (V_1''(y) \vee b' V_2''(y)) \vee (b_1 \vee b' b_2) V'(y) \\ &= \underline{\lambda}y. V''(y) \vee b V'(y). \end{aligned}$$

$N_1 N_2[x := N] (\equiv (N_1[x := N])(N_2[x := N]))$  and the last rule used is rule APP- $\rightarrow$ . Then the result follows easily by Lemma 3.2.14, rule APP- $\rightarrow$  and the induction hypotheses.

$(\lambda z. N')[x := N]$  The case  $z \equiv x$  is not allowed by the variable convention. Suppose  $z \not\equiv x$ , then  $(\lambda z. N')[x := N] \equiv \lambda z. (N'[x := N])$ . Since  $A \vdash_V^C (\lambda z. N')[x := N] : \tau$  is deduced using rule ABS,  $\tau = \rho b' \sigma'$ , for some  $\rho$ ,  $b'$  and  $\sigma'$ . By Lemma 3.2.15,

$$A_z \cup \{z : \rho\} \vdash_{V_1[z:=b']}^C N'[x := N] : \sigma',$$

where  $V_1[z := b'] = \underline{\lambda}y. V_1''(y)[z := b'] \vee (b_1 \wedge V'(y))$ . (Note that by the variable convention,  $z \notin \text{FV}(N)$  hence  $V'(z) =_{\text{BA}} \rightarrow$ ). By induction hypothesis,

$$A_{zx} \cup \{z : \rho, x : \sigma\} \vdash_{V_1''[x:=b_1][z:=b']}^C N' : \sigma',$$

then by ABS,

$$A_x \cup \{x : \sigma\} \vdash_{V_1''[x:=b_1][z:=\rightarrow]}^C \lambda z. N' : \rho b' \sigma'.$$

Now,

$$\begin{aligned} V &= \underline{\lambda}y. V_1[z := \rightarrow](y) \\ &= \underline{\lambda}y. V_1''[z := \rightarrow](y) \vee (b_1 \wedge V'(y)), \end{aligned}$$

as required.

□

Now the effect of performing a substitution:

**Lemma 3.2.20**

If  $A_x \cup \{x : \sigma\} \vdash_{V' \setminus [x := b]}^C M : \tau$ ,  $A \vdash_{V''}^C N : \sigma$ ,  $x \notin \text{FV}(N)$  and  $V'(x) = \rightarrow$ , then

- $A \vdash_V^C M[x := N] : \tau$ , and
- $V = \lambda y. V'(y) \vee (b \wedge V''(y))$ .

**Proof**

By induction on  $M$ .

$M \equiv x$  In this case  $M[x := N] \equiv N$ .

1.  $A_x \cup \{x : \sigma\} \vdash_{V_{\rightarrow}[x := \Rightarrow]}^C x : \tau$ ,  $\sigma = \tau$ ,  $V' = V_{\rightarrow}$  and  $b =_{\text{BA}} \Rightarrow$ .
2.  $A \vdash_V^C x[x := N] : \tau$ .

$M \equiv z$ ,  $z \neq x$  In this case  $M[x := N] \equiv M$ .

1.  $A_x \cup \{x : \sigma\} \vdash_{V_{\rightarrow}[z := \Rightarrow]}^C z : \tau$ ,  $V' = V_{\rightarrow}[z := \Rightarrow]$  and  $b =_{\text{BA}} \rightarrow$ .
2. By Lemma 3.2.17,  $A \vdash_{V_{\rightarrow}[z := \Rightarrow]}^C z : \tau$ .
3.  $A \vdash_V^C z[x := N] : \tau$ .

$M \equiv N_1 N_2$  and rule APP is the final rule used. In this case  $M[x := N] \equiv (N_1[x := N])(N_2[x := N])$ .

1. By Lemma 3.2.13,  $\exists \sigma' \in T_C^\nabla. A \vdash_{V_1'}^C N_1 : \sigma' b' \tau$ ,  $A \vdash_{V_2'}^C N_2 : \sigma'$  and  $(V'[x := b])(y) = V_1'(y) \vee (b' \wedge V_2'(y))$ .
2. By the induction hypotheses,  $A \vdash_{V_1''}^C N_1[x := N] : \sigma' b' \tau$ , where  $V_1'''(y) = V_1'(y) \vee (V_1'(x) \wedge V''(y))$  and  $A \vdash_{V_2''}^C N_2[x := N] : \sigma'$ , where  $V_2'''(y) = V_2'(y) \vee (V_2'(x) \wedge V''(y))$ .
3. 
$$\begin{aligned} & V_1'''(y) \vee (b' \wedge V_2'''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (V_1'(x) \wedge V''(y)) \vee (b' \wedge V_2'(y)) \vee \\ & \quad (b' \wedge V_2'(x) \wedge V''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (b' \wedge V_2'(y)) \vee (V_1'(x) \vee \\ & \quad (b' \wedge V_2'(x))) \wedge V''(y) \\ &=_{\text{BA}} (V'[x := b])(y) \vee (b \wedge V''(y)). \end{aligned}$$
4. Finally, by APP and the definition of substitution for terms it follows that  $A \vdash_V^C (N_1 N_2)[x := N] : \tau$ .

$M \equiv N_1 N_2$  and rule APP- $\rightarrow$  is the final rule used. Straightforward.

$M \equiv \lambda z. M'$  The case  $z \equiv x$  is not interesting. Suppose  $z \neq x$ :

1. By Lemma 3.2.15,  $\tau = \sigma_1 \mathbf{b}' \sigma_2$ , for some  $\sigma_i$  and  $\mathbf{b}'$ ,  $A_{xz} \cup \{x : \sigma, z : \sigma_1\} \vdash_{V'[x:=b, z:=b']}^C M' : \sigma_2$ .
2. By the induction hypothesis,  $A_z \cup \{z : \sigma_1\} \vdash_{V''}^C M'[x := N] : \sigma_2$ , where  $V'''(y) = (V'[z := b'])(y) \vee (b \wedge V''(y))$ .
3. By ABS,  $A \vdash_V^C (\lambda y. M')[x := N] : \sigma_1 \mathbf{b}' \sigma_2$ .

□

Reassuringly, any type assigned to a redex term by the present type assignment system may also be assigned to the (top-level)  $\beta$ -contractum of that term.

### Theorem 3.2.21

If  $A \vdash_V^C (\lambda x. M)N : \tau$ , then there exists a deduction of  $A \vdash_V^C M[x := N] : \tau$ .

#### Proof

Choose  $x \notin \text{FV}(N)$ . By Lemma 3.2.13,  $\exists \sigma \in T_C^\nabla, \mathbf{b} \in \nabla, V', V''. A \vdash_V^C \lambda x. M : \sigma \mathbf{b} \tau, A \vdash_{V''}^C N : \sigma$  and  $V = \lambda y. V'(y) \vee (b \wedge V''(y))$ . By Lemma 3.2.15,  $A_x \cup \{x : \sigma\} \vdash_{V'[x:=b]}^C M : \tau$ . By rule ABS  $V'(x) =_{\text{BA}} \tau$ . Therefore Lemma 3.2.20 applies and so  $A \vdash_V^C M[x := N] : \tau$ , as required. □

The following Corollary is also known as the “Subject-Reduction Theorem”, see Curry and Feys [18].

### Corollary 3.2.22

Let  $M \rightarrow_\beta N$ , then  $A \vdash_V^C M : \tau$  implies  $A \vdash_V^C N : \tau$ .

#### Proof

By iterated use of Theorem 3.2.21 along the reduction path from  $M$  to  $N$ . □

Curry’s system of F-deducibility is *not* complete with respect to  $\beta$ -expansion ([18]). It is not too surprising then that the current system should also suffer this limitation. However, the result does hold in certain special cases, as is now demonstrated by the following two Theorems.

Firstly, the case in which the abstraction variable does occur free in the term:

### Theorem 3.2.23

If  $A \vdash_V^C M[x := N] : \tau$  and  $x \in \text{FV}(M)$ , and suppose only  $\sigma$  is assigned to relevant occurrences<sup>5</sup> of  $N$  in the deduction of  $A \vdash_V^C M[x := N] : \tau$ , then if  $A \vdash_V^C N : \sigma$ , then  $A \vdash_V^C (\lambda x. M)N : \tau$ .

<sup>5</sup>See Definition 3.2.8. This Theorem is made more general by only requiring relevant occurrences of  $N$  to meet this condition.

**Proof**

1. By Lemma 3.2.19, there exist  $b$  and  $V''$  such that  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^C M : \tau$  and  $V = \underline{\lambda}y.V''(y) \vee (b \wedge V'(y))$ .
2. By ABS,  $A \vdash_{V''[x:=\rightarrow]}^C \lambda x.M : \sigma \text{ b } \tau$ .
3.  $V''[x := \rightarrow] =_{\text{BA}} V''$ , since  $x \notin \text{FV}(M[x := N])$ .
4. By APP,  $A \vdash_V^C (\lambda x.M)N : \tau$ .

□

Secondly, the case in which the abstraction variable does not occur free in the term:

**Theorem 3.2.24**

If  $A \vdash_V^C M[x := N] : \tau$ ,  $x \notin \text{FV}(M)$  then  $A \vdash_V^C (\lambda x.M)N : \tau$ .

**Proof**

By rule APP- $\rightarrow$ . The theorem also holds for the system without APP- $\rightarrow$ , provided that  $A \vdash_V^C N : \sigma$ , as follows:

1.  $M[x := N] \equiv M$ , so  $A \vdash_V^C M : \tau$ .
2. By Lemma 3.2.17,  $A_x \cup \{x : \sigma\} \vdash_V^C M : \tau$ .
3. By ABS,  $A \vdash_{V[x:=\rightarrow]}^C \lambda x.M : \sigma \text{ b } \tau$ .
4.  $x \notin \text{FV}(M)$  implies  $V(x) =_{\text{BA}} \rightarrow$ .
5. By APP,  $A \vdash_{V'''}^C (\lambda x.M)N : \tau$  where  $V''' = \underline{\lambda}y.V(y) \vee (V(x) \wedge V'(y)) = \underline{\lambda}y.V(y) = V$ , as required.

□

Finally,  $\eta$ -contraction preserves types:

**Theorem 3.2.25**

If  $x \notin \text{FV}(M)$  and  $A \vdash_V^C \lambda x.Mx : \sigma_1 \text{ b } \sigma_2$ , then  $A \vdash_V^C M : \sigma_1 \text{ b } \sigma_2$ .

**Proof**

By induction on the derivation of  $A \vdash_V^C \lambda x.Mx : \sigma_1 \text{ b } \sigma_2$ .

1. By Lemma 3.2.15,  $A_x \cup \{x : \sigma_1\} \vdash_{V[x:=b]}^C Mx : \sigma_2$ .
2. By Lemma 3.2.13,  $A_x \cup \{x : \sigma_1\} \vdash_V^C M : \sigma_1 \text{ b } \sigma_2$  and by rule VAR,  $A_x \cup \{x : \sigma_1\} \vdash_{V[x:=\Rightarrow]}^C x : \sigma_1$ .
3. By Lemma 3.2.18,  $A \vdash_V^C M : \sigma_1 \text{ b } \sigma_2$ .

□

VAR	$(A_{\rightarrow})_x \cup \{x : (\Rightarrow, \sigma)\} \vdash^C x : \sigma$
APP	$\frac{A_1 \vdash^C N_1 : \sigma \text{ b } \tau \quad A_2 \vdash^C N_2 : \sigma}{A_1 \oplus_b A_2 \vdash^C N_1 N_2 : \tau}$
ABS	$\frac{A_x \cup \{x : (\text{b}, \sigma)\} \vdash^C N : \tau}{A \vdash^C \lambda x. N : \sigma \text{ b } \tau}$

Figure 7: The Alternative Curry-style Rules for deducing Reduction Types

### 3.2.7 Syntactic Variations

As a matter of syntax, two variations on the above formalism for type statements are now discussed.<sup>6</sup> The first is merely writing the variable strong head neededness function as a pair with the type deduced, *i.e.*, typing statements are written as:

$$A \vdash^C M : (V, \tau).$$

This option was not followed here as it is somewhat more cumbersome to write.

The second variation is somewhat more substantial, but has the disadvantage of complicating the form of some of the proofs in the last section, hence it too is not further pursued.

Let an assumption set,  $A$ , be a set of triples of the form

$$x : (\text{b}, \tau),$$

where  $\text{b}$  is an arrow expression and  $\tau$  is a type, such that no term variable  $x$  occurs more than once in  $A$ . Let  $A_{\rightarrow}$  be any assumption set such that  $\forall x. x : (\text{b}, \tau) \in A$  implies  $\text{b} =_{\text{BA}} \rightarrow$ . Two assumption sets  $A_1$  and  $A_2$  are *compatible* if  $\forall x. x : (\text{b}_1, \sigma) \in A_1$  iff  $x : (\text{b}_2, \sigma) \in A_2$ . Suppose  $A_1$  and  $A_2$  are two compatible assumption sets, then let  $A_1 \oplus_b A_2 = \{x : (\text{b}_1 \vee (\text{b} \wedge \text{b}_2), \sigma) \mid x : (\text{b}_1, \sigma) \in A_1; x : (\text{b}_2, \sigma) \in A_2\}$ . In Figure 7 is defined the alternative type assignment system.

It is easy to show by induction that both these systems are equivalent<sup>7</sup> to the earlier definition of the  $\vdash^C$  system.

<sup>6</sup>This section is intended to motivate the chosen syntax for typing statements.

<sup>7</sup>In the sense that each term that is typable has the same set of types assignable to it in these systems.

### 3.2.8 Discussion

So far in this chapter several new things have been introduced:

- a new formalism based on types has been introduced which describes the reduction properties of  $\lambda$ -terms;
- deduction systems in the style of Curry have been presented which allow the inference of elements of this new formalism for a given  $\lambda$ -term (the correctness of these systems is considered later);
- use has been made of the intensional information now available to derive a new deduction rule called APP- $\rightarrow$ . This rule extends the basic system to allow the application of functions to irrelevant arguments while still preserving the intuitive correctness of typed terms; and
- an entirely new syntactic way of categorising  $\lambda$ -terms has resulted from the new formalism, namely that the type assignment system of this section allows the identification of those  $\lambda$ -terms which are constructed from variables which are used in both an intensionally and extensionally identical manner. This categorisation forms an interesting contrast with some other well known categorisations ( $\Lambda L$ ,  $\Lambda I$  and simply-typable terms).

In addition, the type assignment system has been shown to possess certain important properties:

- the strong head neededness of free variables of a term are constant for a given type assumption set, type deduced and term;
- both strengthening and weakening rules are derivable from the type assignment system; and
- the set of deductions for a term are a subset of the deductions for all  $\beta$ - and  $\eta$ -contractums of that term.

On the other hand, certain limitations of the type assignment system have also been revealed:

- terms containing variables used in an explicitly higher-order manner and in which the uses of these variables are necessarily intensionally different are not typable; and, as a related point,
- $\beta$ -expansions of terms can only be assigned the same type (under identical assumption sets) if the *term* being abstracted is used in an intensionally as well as extensionally congruent manner (this is intuitively clear from the previous point if one considers that the term being abstracted will be replaced by a variable). The only exception to this is the degenerate case of rule APP- $\rightarrow$ .

In the following sections of this chapter increasingly powerful deduction systems are described. Similar investigations to the current one are conducted for each of these systems. Investigation of the semantic correctness of these systems is delayed until Chapter 4.

### 3.3 LET-Polymorphic Type Assignment

The next type assignment system introduced for Boolean Reduction Types is one in the style of Milner's system for ML (Milner [47]). In this system certain term variables may be used polymorphically, *i.e.*, be given more than one type, as long as all the types satisfy a certain property. In Milner's system the term variables which may be used polymorphically are those which are bound by a special kind of term, the LET clause. This kind of term has the form:

$$\text{LET } x = N \text{ IN } M.$$

As Milner states this is not strictly necessary as any LET term can be simply converted to the application of an abstraction term to an argument. So the above term would be translated to

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

However, for the purposes of type assignment a term of the form  $\text{LET } x = N \text{ IN } M$ , and hence of the form  $(\lambda x.M)N$ , is treated as if it were the term  $M[x := N]$ . Naturally, this is only possible in those situations in which the argument to the abstraction is explicitly available.

In this section the additional complication of extending the set of terms to include a LET expression is avoided in precisely the straightforward manner suggested by Milner. This has the advantage of making the various systems for type assignment more easily comparable.

LET-polymorphism is a restricted form of *parametric* polymorphism (as is often cited, this was first formalised by Girard [25] and independently by Reynolds [57]). In the present section, the restricted form of parametric polymorphism embodied by the LET-polymorphism approach is extended to include (a restricted form of) parametric *intensional* polymorphism. Full parametric polymorphism requires explicit and unrestricted use of universally quantified types.

#### 3.3.1 Generic Types

##### Definition 3.3.1

The set of *Abstract LET-Polymorphic Boolean Reduction Types*,  $T_L^\nabla$ , is inductively defined to be the least set satisfying:

1.  $\sigma \in T_C^\nabla$  implies  $\sigma \in T_L^\nabla$ ,
2.  $\alpha \in \tau_v, \sigma \in T_L^\nabla$  implies  $\forall \alpha. \sigma \in T_L^\nabla$ , and
3.  $\rightarrow_i \in \Delta_v, \sigma \in T_L^\nabla$  implies  $\forall \rightarrow_i. \sigma \in T_L^\nabla$ .

Quantified types will be called *generic* and ordinary types will be called *non-generic*. Note that the definition constrains all quantifiers to occur only at the top level of a generic type.

Types which may be generic will be underlined from now on to distinguish them from types which are definitely non-generic.

It will be useful to denote by the metavariables  $\delta, \delta_1, \delta_2, \dots$  either an arrow variable or a type variable. Thus  $\forall \delta. \sigma$  will stand for either  $\forall \rightarrow_i. \sigma$  or  $\forall \alpha. \sigma$ . Also  $\forall \delta_1. \dots \forall \delta_n. \sigma$  will often be denoted by  $\forall \underline{\delta}. \sigma$ . Similarly, it will occasionally be useful to use the metavariables  $\Delta, \Delta_1, \Delta_2, \dots$  to stand for both arrow expressions and non-generic types.

A type assumption in the LET-polymorphic system is the same as for the Curry-style system except that the types assumed for variables may now be generic types as well as ordinary types.

Term variable strong head neededness functions are exactly the same as described in the Curry-style type assignment system.

As before, the set of *free variables* of a non-generic type  $\sigma$ , written  $FV(\sigma)$ , is the set of all the type and arrow variables occurring in  $\sigma$ . The set of *free variables* of a generic type  $\underline{\sigma} = \forall \delta_1. \forall \delta_2. \dots \forall \delta_n. \tau$ , written  $FV(\underline{\sigma})$ , is  $FV(\tau) - \{\delta_1, \delta_2, \dots, \delta_n\}$ . The set of free variables of an assumption set  $A$ , written  $FV(A)$ , is  $\bigcup_{x \underline{\sigma} \in A} FV(\underline{\sigma})$ . Similarly, the set of free variables of a variable strong head neededness function  $V$ , written  $FV(V)$ , is  $\bigcup_{x \in X} FV(V(x))$ , where the free variables of an arrow expression are all the arrow variables occurring in the arrow expression.<sup>8</sup>

### Definition 3.3.2

Let  $\Delta_1, \dots, \Delta_n$  be as defined above. Let  $\tau$  be a type and let  $\sigma$  be a type identical to  $\tau$  except that no bound variable within  $\sigma$  is contained in  $\bigcup_{i=1}^n FV(\Delta_i)$ . Let  $\tau[\delta_1, \dots, \delta_n := \Delta_1, \dots, \Delta_n]$  be the type identical to  $\sigma$  except that each occurrence of  $\delta_i$  in  $\sigma$  is replaced by  $\Delta_i$ . Then write  $\forall \delta_1, \dots, \delta_n. \tau \preceq \tau'$  if  $\tau' = \tau[\delta_1, \dots, \delta_n := \Delta_1, \dots, \Delta_n]$  and  $\{\delta_1, \dots, \delta_n\} \cap FV(\Delta_1, \dots, \Delta_n) = \emptyset$ .

## 3.3.2 The Type Assignment Rules

The deduction system again takes the form of an inference rule or axiom for each kind of  $\lambda$ -term (see Figure 8). There are now three kinds of application:

<sup>8</sup>In the type assignment systems described in this thesis there are no bound arrow variables in any variable neededness function.



VAR	$A_x \cup \{x : \underline{\sigma}\} \vdash_{V \mapsto [x := \Rightarrow]}^L x : \tau \quad (\underline{\sigma} \preceq \tau)$
APP	$\frac{A \vdash_{V_1}^L N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^L N_2 : \sigma}{A \vdash_V^L N_1 N_2 : \tau} \quad (V = \underline{\lambda}x.V_1(x) \vee (b \wedge V_2(x)))$
APP- $\forall$	$\frac{A_x \cup \{x : \underline{\sigma}\} \vdash_{V_1}^L N_1 : \tau \quad A \vdash_{V_2}^L N_2 : \tau'}{A \vdash_V^L (\underline{\lambda}x.N_1)N_2 : \tau} \quad (\underline{\sigma} = \text{gen}(A, V_2, \tau'))$ $(V = \underline{\lambda}y.V_1(y) \vee (V_1(x) \wedge V_2(y)))$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := b]}^L N : \tau}{A \vdash_{V[x := \mapsto]}^L \underline{\lambda}x.N : \sigma \text{ b } \tau}$

Figure 8: The LET-Polymorphic Rules for deducing Reduction Types

- *polymorphic* application (rule APP- $\forall$ ),
- *irrelevant* application (rule APP- $\rightarrow$ ), (not detailed in the Figure), and
- *ordinary* application (rule APP).

These rules are described below.

### The VAR Rule

There are two changes to this rule when compared with the Curry-style system:

1. the assumption for a term variable may be a generic type, and
2. the type given the term must be an instance of the generic type assumed, where the type assigned by the rule is non-generic.

The point of this is that each occurrence of a term variable with a generic type may be used in multiple intensionally and extensionally different contexts.

### The APP Rule

This rule is the same as that of the Curry-style system. Note that all types not in the assumption set are required to be non-generic. Also note that by using this rule only when the term  $N_1$  is not an abstraction term, the maximum amount of polymorphism may be achieved. The rule for polymorphic

application (described below) will then be used whenever  $N_1$  is an abstraction term. Of course, this is not enforced and so derivations with less general types may also be derived.

### The APP- $\rightarrow$ Rule

This rule is unchanged from the Curry-style system. See the comments in that section for discussion of this rule. For completeness, the rule is given below.

$$\text{APP-}\rightarrow \quad \frac{A \vdash_{V_1}^L N_1 : \sigma \rightarrow \tau \quad A' \vdash_{V_2}^L N_2 : \sigma'}{A \vdash_{V_1}^L N_1 N_2 : \tau}$$

### The APP- $\forall$ Rule

The APP- $\forall$  rule is the second place in which polymorphism is added to the system. This new rule effectively encompasses the logic of both the ABS and APP rules, with the key addition of a generic type for the term variable being abstracted. The restriction on this generic type is that it must be a *generalisation* of the non-generic type assigned to the argument term. By *generalisation* is meant the following (see Wright [72, 74] and, in the case of ordinary types, Milner [47]):

#### Definition 3.3.3

$\text{gen}(A, V, \tau) = \forall \delta_1. \forall \delta_2. \dots \forall \delta_n. \tau$ , where  $\{\delta_1, \delta_2, \dots, \delta_n\} = (\text{FV}(\tau) - \text{FV}(A)) - \text{FV}(V)$ .

The idea behind this definition is to make the resulting type as generic as is possible without causing names free in other contexts to become erroneously bound.

Also notice that  $V(x) = V_1(x) \vee (V_1(x) \wedge V_2(x)) =_{\text{BA}} V_1(x)$ , as expected, since  $(\lambda x. N_1)x \equiv N_1$ .<sup>9</sup>

### The ABS Rule

The abstraction rule is completely unchanged from that occurring in the Curry-style type assignment system. Note that the type assumed for the variable being abstracted must be non-generic, as in the Curry-style system.

Finally, note that at each stage of the deduction of a typing statement in the LET-polymorphic system the types assigned to terms are non-generic. All the genericity occurs in the assumptions of the deduction.

<sup>9</sup>See Moral 2.1.14 of Barendregt [3] (and the preceding discussion) for a justification of the use of  $\equiv$  in this case.

### 3.3.3 Examples

Unlike the Curry-style type assignment system of the previous section, the LET-polymorphic style type assignment system can be used to find types for *certain* terms which use variables in intensionally different ways—those for which the term to be substituted for them is explicitly present. A good example is the type assignment for the term

$$(\lambda f.g(f \text{ I } x)(f (\lambda x.y) z))(\lambda ab.ab).$$

In this term it is required that  $f$  be assigned first a type of the form:

$$(\sigma \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \sigma,$$

and then of the form:

$$(\sigma' \nrightarrow \sigma') \Rightarrow \sigma' \nrightarrow \sigma',$$

for some  $\sigma, \sigma'$ . Given that the argument to be substituted for  $f$  is explicitly known (*i.e.*, rule APP- $\forall$  is applicable) and is  $(\lambda ab.ab)$ , the present system can satisfy these constraints.

Note that the term

$$\lambda f.g(f \text{ I } x)(f (\lambda x.y) z),$$

cannot be assigned a term in the LET-polymorphic style system.

It is clear that every term for which a type can be assigned using the Curry-style system can be assigned a type using the current system—the Curry-style system is a strict subset of the current system as every rule of the former system is present in or is subsumed by a rule of the LET-polymorphic system. Thus there is little point in repeating the examples of the previous section. The proposition below proves that the systems  $\vdash^L$  are conservative extensions of the systems  $\vdash^C$ .

#### Proposition 3.3.4

If  $A \vdash_V^C M : \tau$ , then there is a deduction of  $A \vdash_V^L M : \tau$  which has precisely the same structure as the derivation of  $A \vdash_V^C M : \tau$ .

#### Proof

Induction on  $A \vdash_V^L M : \tau$ .  $\square$

### 3.3.4 Properties of the LET-Polymorphic System

In this subsection a similar investigation to that for the Curry-style system is undertaken. Many of the differences between the systems turn out to be of a somewhat subtle nature, so it is worthwhile repeating the format of the previous investigation in some detail. For example, in analysing the effect of

term substitution some more general statements may be made because of the presence of rule APP- $\forall$ , but on the other hand the proofs have this extra case to consider. Similarly, the results concerning  $\beta$ -expansion are more general than for the Curry-style system.

In the LET-Polymorphic system, the reversal of the APP rule still applies, but now there are three kinds of application and so there are three corresponding lemmas. Again, these reversal results are trivial for these systems, but are included for completeness.

Firstly, ordinary application.

**Lemma 3.3.5**

If  $A \vdash_V^L MN : \tau$  and rule APP is the final rule applied in this deduction, then  $\exists \sigma \in T_L^\nabla, b \in \nabla. (A \vdash_V^L M : \sigma \text{ b } \tau, A \vdash_V^L N : \sigma \text{ and } \forall x \in X. V = \lambda x. V'(x) \vee (b \wedge V''(x)))$ .

**Proof**

By rule APP.  $\square$

Secondly, irrelevant application.

**Lemma 3.3.6**

If  $A \vdash_V^L MN : \tau$  and rule APP- $\rightarrow$  is the final rule applied in this deduction, then  $\exists \sigma, \sigma' \in T_L^\nabla$ , and an assumption set  $A'$  such that  $A \vdash_V^L M : \sigma \rightarrow \tau$  and  $A' \vdash_V^L N : \sigma'$ .

**Proof**

By rule APP- $\rightarrow$ .  $\square$

Thirdly, polymorphic application.

**Lemma 3.3.7**

If  $A \vdash_V^L MN : \tau$  and rule APP- $\forall$  is the final rule applied in this deduction, then  $M$  is an abstraction term, say  $M \equiv \lambda x. N'$  and  $\exists \sigma \in T_L^\nabla, b \in \nabla. (A_x \cup \{x : \underline{\sigma}\} \vdash_{V[x:=b]}^L N' : \tau, A \vdash_V^L N : \sigma, \underline{\sigma} = \text{gen}(A, V'', \sigma) \text{ and } \forall x \in X. V = \lambda x. V'(x) \vee (b \wedge V''(x)))$ .

**Proof**

By rule APP- $\forall$ .  $\square$

Since the abstraction rule is unchanged from the Curry-style system, the following result carries through unchanged as well.

**Lemma 3.3.8**

$A \vdash_V^L \lambda x. M : \sigma \text{ b } \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_{V[x:=b]}^L M : \tau$ .

**Proof**

By rule ABS, since such a type can only be deduced for  $\lambda x.M$  by using this rule.  $\square$

**Lemma 3.3.9**

Suppose  $A \vdash_{V_1}^L M : \sigma$  and  $A \vdash_{V_2}^L M : \sigma$ , then  $\forall x \in X. V_1(x) =_{BA} V_2(x)$ .

**Proof**

By induction of the structure of  $M$ .

If  $M \equiv x$ , then by rule VAR  $\forall y \in X. V_1(y) =_{BA} V_2(y) [x := \Rightarrow](y) =_{BA} V_2(y)$ .

If  $M \equiv \lambda x.N$ , then by the induction hypothesis for  $N$  and by rule ABS.

The tedious case is when  $M$  is an application-term,  $M_1 M_2$ . In this case the result again follows from the induction hypotheses, by definition of  $=_{BA}$  and by the type deduction rules for application terms (for the nine *significant* cases).  $\square$

The usual rules for *strengthening* and *weakening* are derivable in the present system:

$\text{STR} \quad \frac{A_x \cup \{x : \underline{\sigma}\} \vdash_V^L M : \tau}{A_x \vdash_V^L M : \tau} \quad (x \notin \text{FV}(M))$
$\text{WEAK} \quad \frac{A \vdash_V^L M : \tau}{A_x \cup \{x : \underline{\sigma}\} \vdash_V^L M : \tau} \quad (x \notin \text{FV}(M))$

This is demonstrated by the following two lemmas:

**Lemma 3.3.10**

Suppose  $x \notin \text{FV}(M)$ , then  $A \vdash_V^L M : \tau$  implies  $A_x \cup \{x : \underline{\sigma}\} \vdash_V^L M : \tau$ .

**Proof**

Straightforward induction over the deduction of  $A \vdash_V^L M : \tau$ .  $\square$

**Lemma 3.3.11**

If  $A_x \cup \{x : \underline{\sigma}\} \vdash_V^L M : \tau$  and  $x \notin \text{FV}(M)$ , then  $A_x \vdash_V^L M : \tau$ .

**Proof**

By induction over the deduction of  $A \vdash_V^L M : \tau$ , using Lemma 3.3.10 for the case of  $M$  an abstraction term.  $\square$

Now consider the effect of term substitution on type assignment. The following two lemmas are analogous to Lemma 3.2.19 and Lemma 3.2.20, but differ in both their expression and details of their proof. In the LET-polymorphic system certain variables may have multiple types assigned to them and this is

the principle advantages of the LET-polymorphic system over the Curry-style system.

**Lemma 3.3.12**

Suppose  $A \vdash_V^L M[x := N] : \tau$ , the types  $\rho_1, \dots, \rho_n, \sigma \preceq \rho_i$ , are assigned to  $N$  in this deduction and  $A \vdash_V^L N : \rho_i$ , ( $i \in \{1, \dots, n\}$ ), then there exists  $b$  and  $V''$  such that:

- $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^L M : \tau$ , and
- $\forall y \in X. V(y) =_{BA} V''(y) \vee (b \wedge V'(y))$ .

**Proof**

By induction on  $M[x := N]$ .

$x[x := N] \equiv N$  In this case  $M[x := N] \equiv N$ , so  $\sigma \preceq \tau \in \{\rho_i\}$  and  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^L x : \tau$ , where  $V'' =_{BA} V_{\rightarrow}$  and  $b =_{BA} \Rightarrow$ . Now,

$$\begin{aligned} V(y) &=_{BA} V'(y) \\ &=_{BA} V_{\rightarrow}(y) \vee (\Rightarrow \wedge V'(y)) \\ &=_{BA} V''(y) \vee (b \wedge V'(y)), \end{aligned}$$

as required.

$z[x := N] \equiv z$  ( $z \neq x$ ) It is immediate that

$$A \vdash_{V_{\rightarrow}[z:=\Rightarrow]}^L z : \tau,$$

so  $V'' =_{BA} V_{\rightarrow}[z := \Rightarrow]$  and  $b =_{BA} \rightarrow$ . Now, for any  $V'$ ,

$$\begin{aligned} V(y) &=_{BA} (V_{\rightarrow}[z := \Rightarrow])(y) \vee (\rightarrow \wedge V'(y)) \\ &=_{BA} V''(y) \vee (b \wedge V'(y)). \end{aligned}$$

Then the result follows immediately or by using Lemma 3.3.10.

$N_1 N_2[x := N] \equiv (N_1[x := N])(N_2[x := N])$  and rule APP is used as the final step in the deduction. By Lemma 3.3.5:

- $A \vdash_{V_1}^L N_1[x := N] : \rho \text{ b } \tau$ , and
- $A \vdash_{V_2}^L N_2[x := N] : \rho$ ,

where  $V(y) =_{BA} V_1(y) \vee (b' \wedge V_2(y))$ . By the induction hypotheses,

- $A_x \cup \{x : \sigma\} \vdash_{V_1''[x:=b_1]}^L N_1 : \rho \text{ b } \tau$ , and
- $A_x \cup \{x : \sigma\} \vdash_{V_2''[x:=b_2]}^L N_2 : \rho$ ,

where  $V_1(y) =_{\text{BA}} V_1''(y) \vee (b_1 \wedge V'(y))$  and  $V_2(y) =_{\text{BA}} V_2''(y) \vee (b_2 \wedge V'(y))$ . By APP,  $A_x \cup \{x : \underline{\sigma}\} \vdash_{V''[x:=b]}^L N_1 N_2 : \tau$ , where  $V''(y) =_{\text{BA}} V_1''(y) \vee (b' \wedge V_2''(y))$  and  $b =_{\text{BA}} b_1 \vee (b' \wedge b_2)$ . By APP,

$$\begin{aligned}
 V(y) &=_{\text{BA}} V_1(y) \vee (b' V_2(y)) \\
 &=_{\text{BA}} (V_1''(y) \vee b_1 V'(y)) \vee (b' (V_2''(y) \vee b_2 V'(y))) \\
 &=_{\text{BA}} V_1''(y) \vee b_1 V'(y) \vee b' (V_2''(y) \vee b_2 V'(y)) \\
 &=_{\text{BA}} V_1''(y) \vee b_1 V'(y) \vee b' V_2''(y) \vee b' b_2 V'(y) \\
 &=_{\text{BA}} V_1''(y) \vee b' V_2''(y) \vee b_1 V'(y) \vee b' b_2 V'(y) \\
 &=_{\text{BA}} (V_1''(y) \vee b' V_2''(y)) \vee (b_1 \vee b' b_2) V'(y) \\
 &=_{\text{BA}} V''(y) \vee b V'(y).
 \end{aligned}$$

$N_1 N_2[x := N] \equiv (N_1[x := N])(N_2[x := N])$  and rule APP- $\rightarrow$  is used as the final step in the deduction. Straightforward by the induction hypotheses, Lemma 3.3.6 and rule APP- $\rightarrow$ .

$((\lambda y. N_1) N_2)[x := N]$  and rule APP- $\forall$  is the final step of the deduction. By the variable convention  $x \neq y$ , so  $((\lambda y. N_1) N_2)[x := N] \equiv (\lambda y. N_1[x := N])(N_2[x := N])$ . Now by Lemma 3.3.7  $\exists \rho \in T_L^\nabla. \underline{\rho} = \text{gen}(A, V, \rho)$  and:

- $A \cup \{y : \underline{\rho}\} \vdash_{V_1[y:=b]}^L N_1[x := N] : \tau$ , and
- $A \vdash_{V_2}^L N_2[x := N] : \rho$ ,

where  $V(z) =_{\text{BA}} V_1(z) \vee (b' \wedge V_2(z))$ . By the induction hypotheses,

- $A_{xy} \cup \{x : \underline{\sigma}, y : \underline{\rho}\} \vdash_{V_1''[x:=b_1]}^L N_1 : \tau$ , and
- $A_{xy} \cup \{x : \underline{\sigma}\} \vdash_{V_2''[x:=b_2]}^L N_2 : \rho$ ,

where  $V_1(y) =_{\text{BA}} V_1''(y) \vee (b_1 \wedge V'(y))$  and  $V_2(y) =_{\text{BA}} V_2''(y) \vee (b_2 \wedge V'(y))$ . By APP- $\forall$  and Lemma 3.3.10,  $A_x \cup \{x : \underline{\sigma}\} \vdash_{V''[x:=b]}^L (\lambda y. N_1) N_2 : \tau$ , where  $V''(y) =_{\text{BA}} V_1''(y) \vee (b' \wedge V_2''(y))$  and  $b =_{\text{BA}} b_1 \vee (b' \wedge b_2)$ . As is conventional, assume that  $\wedge$  binds tighter than  $\vee$  and also temporarily replace  $\wedge$  by juxtaposition, then by APP,

$$\begin{aligned}
 V(y) &=_{\text{BA}} (V_1''(y) \vee b_1 V'(y)) \vee (b' (V_2''(y) \vee b_2 V'(y))) \\
 &=_{\text{BA}} V_1''(y) \vee b_1 V'(y) \vee b' (V_2''(y) \vee b_2 V'(y)) \\
 &=_{\text{BA}} V_1''(y) \vee b_1 V'(y) \vee b' V_2''(y) \vee b' b_2 V'(y) \\
 &=_{\text{BA}} V_1''(y) \vee b' V_2''(y) \vee b_1 V'(y) \vee b' b_2 V'(y) \\
 &=_{\text{BA}} (V_1''(y) \vee b' V_2''(y)) \vee (b_1 \vee b' b_2) V'(y) \\
 &=_{\text{BA}} V''(y) \vee b V'(y).
 \end{aligned}$$

$(\lambda z.N')[x := N] \equiv \lambda z.(N'[x := N])$  Since  $A \vdash_V^L (\lambda z.N')[x := N] : \tau$  is deduced using rule ABS,  $\tau = \rho \mathbf{b}' \sigma'$ , for some  $\rho, \mathbf{b}'$  and  $\sigma'$ . By Lemma 3.3.8,

$$A_{zx} \cup \{z : \rho, x : \underline{\sigma}\} \vdash_{V_1''[x := \mathbf{b}_1][z := \mathbf{b}']}^L N' : \sigma',$$

then by ABS,

$$A_x \cup \{x : \underline{\sigma}\} \vdash_{V_1''[x := \mathbf{b}_1][z := \dashv]}^L \lambda z.N' : \rho \mathbf{b}' \sigma'.$$

By the induction hypothesis,

$$A_z \cup \{z : \rho\} \vdash_{V_1[z := \mathbf{b}']}^L N'[x := N] : \sigma',$$

where  $V_1[z := \mathbf{b}'](y) =_{\text{BA}} V_1''(y)[z := \mathbf{b}'] \vee (\mathbf{b}_1 \wedge V'(y))$ . (Note that by the variable convention,  $z \notin \text{FV}(N)$  hence  $V'(z) =_{\text{BA}} \dashv$ , see Barendregt [3]). Now,

$$\begin{aligned} V(y) &=_{\text{BA}} V_1[z := \dashv](y) \\ &=_{\text{BA}} V_1''[z := \dashv](y) \vee (\mathbf{b}_1 \wedge V'(y)), \end{aligned}$$

as required.

□

### Lemma 3.3.13

If  $A_x \cup \{x : \underline{\sigma}\} \vdash_{V'[x := \mathbf{b}]}^L M : \tau$ , the types  $\rho_1, \dots, \rho_n$  ( $n \geq 0$  and  $\underline{\sigma} \preceq \rho_i$ ) are assigned to  $N$  in this deduction,  $A \vdash_V^L N : \rho_i$ , for  $i \in \{1, \dots, n\}$ ,  $x \notin \text{FV}(N)$  and  $V'(x) = \dashv$ , then

- $A \vdash_V^L M[x := N] : \tau$ , and
- $\forall y \in X. V(y) =_{\text{BA}} V'(y) \vee (\mathbf{b} \wedge V''(y))$ .

### Proof

By induction on  $M$ .

$M \equiv x$  In this case  $M[x := N] \equiv N$ . Then  $A_x \cup \{x : \underline{\sigma}\} \vdash_{V_{\dashv}[x := \Rightarrow]}^L x : \tau, \underline{\sigma} \preceq \tau$ ,  $V' = V_{\dashv}$  and  $\mathbf{b} =_{\text{BA}} \Rightarrow$ , thus  $A \vdash_V^L x[x := N] : \tau$ , by Lemmas 3.3.10 and 3.3.11.

$M \equiv z, z \neq x$  In this case  $M[x := N] \equiv M$ .

1.  $A_x \cup \{x : \underline{\sigma}\} \vdash_{V_{\dashv}[z := \Rightarrow]}^L z : \tau, V' = V_{\dashv}[z := \Rightarrow]$  and  $\mathbf{b} =_{\text{BA}} \dashv$ .
2. By Lemma 3.3.10 and Lemma 3.3.11,  $A \vdash_{V_{\dashv}[z := \Rightarrow]}^L z : \tau$ , i.e.,  $A \vdash_V^L z[x := N] : \tau$ .



$M \equiv N_1 N_2$  and rule APP is the final rule used. By definition of substitution  $M[x := N] \equiv (N_1[x := N])(N_2[x := N])$ .

1. By Lemma 3.4.12,  $\exists \sigma' \in T_L^\nabla. A \vdash_{V_1'}^L N_1 : \sigma' b' \tau$ ,  $A \vdash_{V_2'}^L N_2 : \sigma'$  and  $(V'[x := b])(y) = V_1'(y) \vee (b' \wedge V_2'(y))$ .
2. By the induction hypotheses,  $A \vdash_{V_1''}^L N_1[x := N] : \sigma' b' \tau$ , where  $V_1''(y) = V_1'(y) \vee (V_1'(x) \wedge V''(y))$  and  $A \vdash_{V_2''}^L N_2[x := N] : \sigma'$ , where  $V_2''(y) = V_2'(y) \vee (V_2'(x) \wedge V''(y))$ .
3. 
$$\begin{aligned} & V_1''(y) \vee (b' \wedge V_2''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (V_1'(x) \wedge V''(y)) \vee (b' \wedge V_2'(y)) \vee \\ & \quad (b' \wedge V_2'(x) \wedge V''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (b' \wedge V_2'(y)) \vee (V_1'(x) \vee \\ & \quad (b' \wedge V_2'(x))) \wedge V''(y) \\ &=_{\text{BA}} (V'[x := b])(y) \vee (b \wedge V''(y)). \end{aligned}$$
4. By APP,  $A \vdash_V^L (N_1 N_2)[x := N] : \tau$ .

$M \equiv N_1 N_2$  and rule APP $\rightarrow$  is the final rule used. By definition of substitution  $M[x := N] \equiv (N_1[x := N])(N_2[x := N])$ . Then the result follows from Lemma 3.3.6, rule APP $\rightarrow$  and the induction hypotheses.

$M \equiv (\lambda z. N_1) N_2$  and rule APP $\forall$  is the final rule used. By definition of substitution  $M[x := N] \equiv (\lambda z. N_1[x := N])(N_2[x := N])$ . By Lemma 3.3.7, rule APP $\forall$  and the induction hypotheses.

$M \equiv \lambda z. M'$  In this case  $M[x := N] \equiv \lambda z. M'[x := N]$ .

1. By Lemma 3.3.8  $\tau = \sigma_1 b' \sigma_2$ , for some  $\sigma_i$  and  $b'$  and  $A_{xz} \cup \{x : \sigma, z : \sigma_1\} \vdash_{V'[x := b, z := b']}^L M' : \sigma_2$ .
2. By the induction hypothesis,  $A_z \cup \{z : \sigma_1\} \vdash_{V''}^L M'[x := N] : \sigma_2$ , where  $V''(y) = (V'[z := b'])(y) \vee (b \wedge V''(y))$ .
3. By ABS,  $A \vdash_V^L (\lambda y. M')[x := N] : \sigma_1 b' \sigma_2$ .

□

As with the previous type assignment systems, the set of types given a term is a (not necessarily proper) subset of the set of types assignable to the (top-level)  $\beta$ -contractum of the term.

### Theorem 3.3.14

If  $A \vdash_V^L (\lambda x. M) N : \tau$ , then  $A \vdash_V^L M[x := N] : \tau$ .

**Proof**

By Lemmas 3.3.5, 3.3.6 (in the presence of rule APP- $\rightarrow$ ), 3.3.7, 3.3.8 and 3.3.13. (See the proof of Theorem 3.2.21 for a more detailed but substantially similar proof).  $\square$

From the above the following subject-reduction result is immediate.

**Corollary 3.3.15**

Let  $M \rightarrow_\beta N$ , then  $A \vdash_V^L M : \tau$  implies  $A \vdash_V^L N : \tau$ .

**Proof**

By iterated use of Theorem 3.3.14 along the reduction path from  $M$  to  $N$ .  $\square$

A result which does not cause any surprise is that the LET-Polymorphic system presented here is not complete with respect to  $\beta$ -expansion. As in the Curry-style system, the result does hold in certain special cases, as is again demonstrated. Note that multiple types may be assigned to  $N$  in the following Theorem (provided they are all instances of a common generic type). This should be contrasted with the situation for the Curry-style system of the previous section (see Theorem 3.2.23). In that system a single type,  $\sigma$ , had to be assigned to each relevant occurrence of the term being abstracted,  $N$ .

**Theorem 3.3.16**

If  $A \vdash_V^L M[x := N] : \tau$  and  $x \in \text{FV}(M)$ , and suppose  $\rho_1, \dots, \rho_n$ ,  $n \geq 1$ , is assigned to relevant occurrences of  $N$  in the deduction of  $A \vdash_V^L M[x := N] : \tau$ , then if  $A \vdash_V^L N : \sigma$  and  $\text{gen}(A, V', \sigma) \preceq \rho_i$ , ( $i \in \{1, \dots, n\}$ ), then  $A \vdash_V^L (\lambda x.M)N : \tau$ .

**Proof**

By Lemma 3.3.12,  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^L M : \tau$  and  $\forall y \in X. V(y) =_{\text{BA}} V''(y) \vee (b \wedge V'(y))$ . The case  $n = 0$  is not allowed by statement of the Theorem.

For the case  $n = 1$  the result may hold in two ways. Firstly, if  $n = 1$  and  $\sigma = \rho_1$ , then by ABS,  $A \vdash_{V''[x:=\rightarrow]}^L \lambda x.M : \sigma \text{ b } \tau$  and  $V''[x := \rightarrow] =_{\text{BA}} V''$ , since  $x \notin \text{FV}(M[x := N])$ . Then by APP,  $A \vdash_V^L (\lambda x.M)N : \tau$ . Secondly, if  $n \geq 1$ , then use rule APP- $\forall$ .  $\square$

**Theorem 3.3.17**

If  $A \vdash_V^L M[x := N] : \tau$  and  $x \notin \text{FV}(M)$ , then  $A \vdash_V^L (\lambda x.M)N : \tau$ .

**Proof**

By rule APP- $\rightarrow$ . (This Theorem also holds for the systems without rule APP- $\rightarrow$ , but only if  $A \vdash_V^L N : \sigma$ , for some  $\sigma$ ).  $\square$

Finally,  $\eta$ -contraction holds for these type assignment systems.

**Theorem 3.3.18**

If  $x \notin \text{FV}(M)$  and  $A \vdash_V^L \lambda x. Mx : \tau$ , then  $A \vdash_V^L M : \tau$ .

**Proof**

By induction on  $A \vdash_V^L \lambda x. Mx : \tau$ . The interesting case is where the deduction ends in a use of the ABS rule:

1.  $\tau = \sigma_1 \text{ b } \sigma_2$ .
2. By ABS,  $A_x \cup \{x : \sigma_1\} \vdash_{V[x:=b]}^L Mx : \sigma_2$ .
3. By Lemma 3.3.5,  $\exists \sigma_3 \in T_L^\nabla. A_x \cup \{x : \sigma_1\} \vdash_V^L M : \sigma_3 \text{ b } \sigma_2$  and  $A_x \cup \{x : \sigma_1\} \vdash_{V[x:=\Rightarrow]}^L x : \sigma_3$ .
4.  $\sigma_1 = \sigma_3$ , hence  $\sigma_3 \text{ b } \sigma_2 = \sigma_1 \text{ b } \sigma_2$ .
5. By Lemma 3.3.10,  $A \vdash_V^L M : \sigma_1 \text{ b } \sigma_2$ .

□

**3.3.5 Discussion**

In this section the limited form of parametric polymorphism as embodied by the LET-polymorphic type system has been modified to use Boolean Reduction Types and extended to allow intensional parametric polymorphism. The system presented shares the same limitation as the conventional system of LET-polymorphism, namely that parametric polymorphism may only be used where the argument to a function is explicitly known. However, the system presented shares with the conventional LET-polymorphic system certain advantages when it comes to implementation (see Chapter 5).

Note that in exactly the same way that certain simply-typed terms were not typable in the Curry-style system of the previous section, these terms are not typable in the LET-polymorphic style system of this section. Thus the adoption of limited parametric polymorphism is not of great pragmatic interest.

The second-order system of polymorphism of Girard [25] and Reynolds [57] is a natural generalisation of the system of LET-polymorphism. Extending the second-order system to Boolean Reduction Types is straightforward given the above treatment of the LET-polymorphic system (see Chapter 6).

**3.4 Intersection-style Type Assignment**

In this section a system capable of assigning Reduction Types to every  $\lambda$ -term is introduced. This system shares most of its structure with the system for assigning Intersection Types to arbitrary  $\lambda$ -terms (see [14, 16, 4, 12]).

Since universal quantification is most naturally interpreted as intersection (more generally as some notion of *meet*), it will become clear that the system for assigning Intersection Boolean Reduction Types is a generalisation of the systems previously considered. As shall be seen in Chapter 5 and Appendix A this generality results in a semi-decidable system.

Consider a function such as  $(\lambda f.f x)$ . This function was previously given the type  $(\sigma \rightarrow_i \tau) \Rightarrow \tau$ , for some types  $\sigma$  and  $\tau$  and for some arrow variable  $\rightarrow_i$ . Thus, as previously argued, we would expect that  $(\lambda f.f x)$  might reasonably be assigned *both* the type  $(\sigma \Rightarrow \tau) \Rightarrow \tau$  and the type  $(\sigma \nrightarrow \tau) \Rightarrow \tau$ . In fact, we might consider the type  $(\sigma \rightarrow_i \tau) \Rightarrow \tau$  as merely a kind of *shorthand* for the type  $((\sigma \Rightarrow \tau) \Rightarrow \tau) \cap ((\sigma \nrightarrow \tau) \Rightarrow \tau)$ .

As another example, consider the function  $(\lambda f g x.f(gx))$ . Earlier, we assigned the following type to this term:

$$(\rho \rightarrow_1 \tau) \Rightarrow (\sigma \rightarrow_2 \rho) \rightarrow_1 \sigma (\rightarrow_1 \wedge \rightarrow_2) \tau.$$

Taking a cue from the previous example, we can expand out this type as follows:

$$\begin{aligned} & ((\rho \nrightarrow \tau) \Rightarrow (\sigma \nrightarrow \rho) \nrightarrow \sigma \nrightarrow \tau) \cap \\ & ((\rho \nrightarrow \tau) \Rightarrow (\sigma \Rightarrow \rho) \nrightarrow \sigma \nrightarrow \tau) \cap \\ & ((\rho \Rightarrow \tau) \Rightarrow (\sigma \nrightarrow \rho) \Rightarrow \sigma \nrightarrow \tau) \cap \\ & ((\rho \Rightarrow \tau) \Rightarrow (\sigma \Rightarrow \rho) \Rightarrow \sigma \Rightarrow \tau). \end{aligned}$$

Clearly, the more concise form using Boolean function type constructors has a readability advantage! Of course, in the earlier type assignment systems it was more than just an advantage in readability that was gained through the use of these Boolean expressions. However, it intuitively appears that by including an intersection operator on types that one can express all that was expressed in earlier type assignment systems. This is indeed so, as will be shown in both this section and the following chapter. It should be noted that for complexity reasons it is essential that the length of types be minimised using arrow expressions when an implementation is considered (Chapter 5).

However, the intersection operator is much more useful than merely as a way of expanding out Boolean arrow expressions: consider for example the term  $(\lambda f g x y.f(gx)(gy))$ . The issue here is that the variable  $g$  occurs twice in the term. The best type that can be given to this term *requires* the explicit use of the intersection operator:

$$\begin{aligned} & (\rho_1 \rightarrow_1 \rho_2 \rightarrow_2 \rho_3) \Rightarrow \\ & ((\sigma \rightarrow_3 \rho_1) \cap (\tau \rightarrow_4 \rho_2)) (\rightarrow_1 \vee \rightarrow_2) \sigma (\rightarrow_1 \wedge \rightarrow_3) \tau (\rightarrow_2 \wedge \rightarrow_4) \rho_3. \end{aligned}$$

(A precisely analogous use of the intersection operator must be used in the Intersection Type Discipline in order to obtain the best type for this term, see [12, 16]).

### 3.4.1 Intersection Boolean Reduction Types

The following definition introduces the sets of Intersection Boolean Reduction Types.

#### Definition 3.4.1

The set of *Abstract Intersection Boolean Reduction Types*,  $T_I^\nabla$ , is inductively defined to be the least set satisfying:

1.  $\alpha \in \tau_v$  implies  $\alpha \in T_I^\nabla$ ,
2.  $\sigma \in T_I^\nabla, \tau \in T_I^\nabla$  and  $b \in \nabla$  implies  $\sigma b \tau \in T_I^\nabla$ ,
3.  $\sigma \in T_I^\nabla$  and  $\tau \in T_I^\nabla$  implies  $\sigma \cap \tau \in T_I^\nabla$ , and
4.  $\omega \in T_I^\nabla$ .

The type constructor  $\cap$  in this definition may be thought of as greatest lower bound or intersection. The type constant  $\omega$  can be thought of as the set of all terms. The set of *Intersection Boolean Reduction Types* is  $T_I^{B\Delta}$ . The set of *ground Boolean Reduction Types* is  $T_I^{\Delta_g}$ . The set of *hereditarily strongly head needed types* is  $T_I^{\{\Rightarrow\}}$  and the set of *hereditarily irrelevant types* is  $T_I^{\{\Leftarrow\}}$ .

Type assumptions and type assumption sets are as before, except that the types are from  $T_I^\nabla$ . Variable head neededness functions are precisely as before.

Continuing in the footsteps of Barendregt et al [4], it is natural to introduce an ordering on these sets of types which intuitively corresponds to a notion of subset.

#### Definition 3.4.2

1. The relation  $\leq$  on  $T_I^\nabla$  is inductively defined to be the least relation satisfying:
  - (a)  $\tau \leq \omega$ ,
  - (b)  $\omega \leq \omega b \omega$ ,
  - (c)  $\tau \leq \tau$ ,
  - (d)  $\tau \leq \tau \cap \tau$ ,
  - (e)  $\sigma \cap \tau \leq \sigma, \sigma \cap \tau \leq \tau$ ,
  - (f)  $(\sigma b \rho) \cap (\sigma b \tau) \leq \sigma b (\rho \cap \tau)$ ,
  - (g)  $\sigma \leq \tau \leq \rho$  implies  $\sigma \leq \rho$ ,
  - (h)  $\sigma \leq \sigma', \tau \leq \tau'$  implies  $\sigma \cap \tau \leq \sigma' \cap \tau'$ ,
  - (i)  $\sigma \leq \sigma', \tau \leq \tau'$  implies  $\sigma' b \tau \leq \sigma b \tau'$ .

2.  $\sigma = \tau$  iff  $\sigma \leq \tau \leq \sigma$ .

As stated in Barendregt et al [4], the pre-order  $\leq$  defined above is a partial order when the set of types,  $T_I^\nabla$ , is factored by  $=$ . In the following it will be assumed that  $T_I^\nabla$  is indeed factored by  $=$ .

Unless specifically noted, the results in the rest of this section apply equally to systems built from any of the four sets of types defined above. The following fact states some easily shown equivalences amongst types.

**Fact 3.4.3**

$\omega = \sigma \mathbin{\text{b}} \omega$ ,  $(\sigma \mathbin{\text{b}} \rho) \cap (\sigma \mathbin{\text{b}} \tau) = \sigma \mathbin{\text{b}} (\rho \cap \tau)$  and  $\rho \cap (\sigma \cap \tau) = (\rho \cap \sigma) \cap \tau$ .

**Lemma 3.4.4**

$\sigma \mathbin{\text{b}} \tau = \omega$  iff  $\tau = \omega$ .

**Proof**

As in Barendregt et al [4], Lemma 2.4.(i), let  $\Omega \subseteq T_I^\nabla$  such that  $\Omega$  is the least set satisfying:

- $\omega \in \Omega$ ,
- $\rho \in \Omega$  implies  $\sigma \mathbin{\text{b}} \rho \in \Omega$  (arbitrary  $\sigma$  and  $\mathbin{\text{b}}$ ), and
- $\rho, \sigma \in \Omega$  implies  $\rho \cap \sigma \in \Omega$ .

Then  $\sigma \in \Omega$  implies  $\sigma = \omega$ . Now it is easily established by induction on  $\leq$  that  $\Omega$  is closed under  $\leq$ , hence  $\sigma \in \Omega$  iff  $\sigma = \omega$ . Then  $\sigma \mathbin{\text{b}} \tau = \omega$  iff  $\sigma \mathbin{\text{b}} \tau \in \Omega$  iff  $\tau = \omega$ , as required.  $\square$

**Lemma 3.4.5**

Suppose

$$\bigcap_{i=1}^m (\theta_i \mathbin{\text{b}} \rho_i) \cap \bigcap_{i=1}^n \alpha_i \leq \bigcap_{i=1}^k (\sigma_i \mathbin{\text{b}} \tau_i) \cap \bigcap_{i=1}^l \beta_i,$$

where for no  $i$  is  $\tau_i = \omega$ , then for all  $j \in \{1, \dots, k\}$  there are  $i_1, \dots, i_p \in \{1, \dots, m\}$  such that

$$\bigcap_{h=1}^p (\theta_{i_h} \mathbin{\text{b}} \rho_{i_h}) \leq \sigma_j \mathbin{\text{b}} \tau_j,$$

and  $\{\beta_1, \dots, \beta_l\} \subseteq \{\alpha_1, \dots, \alpha_n\}$ .

**Proof**

By induction on  $\leq$ . (See Barendregt et al [4], Lemma 2.4.(ii)).

$\square$

By the above Lemma and the definition of  $\leq$  the following is immediately true. Suppose  $\bigcap_{i=1}^n (\sigma_i \text{ b } \tau_i) \leq \sigma \text{ b } \tau$  and  $\tau \neq \omega$ , then there are  $i_1, \dots, i_k \in \{1, \dots, n\}$  such that  $\sigma \leq \bigcap_{j=1}^k \sigma_{i_j}$  and  $\bigcap_{j=1}^k \tau_{i_j} \leq \tau$ .

### 3.4.2 The Type Assignment Rules

The type assignment systems are presented in Figure 9. The VAR, APP and ABS rules carry through without change from the Curry-style type assignment systems. To allow reasoning with intersection types two new rules are introduced, MEET and LEQ. The rule LEQ allows the natural ordering on intersection types introduced above to be used in the type assignment system. MEET expresses the expected property of intersection. Note that a rule such as SPLIT (below) is not required as it is directly derivable from rule LEQ.

$$\text{SPLIT} \quad \frac{A \vdash_V^I M : \sigma_1 \cap \sigma_2}{A \vdash_V^I M : \sigma_i} \quad (i \in \{1, 2\})$$

Unlike the standard system for intersection type assignment, there is no explicit rule allowing the type  $\omega$  to be assigned to terms (see discussion below). The reason for this difference is that it is not obvious what variable neededness function should be assigned to such a rule. *However, the rule LEQ still allows us to find a type for any  $\lambda$ -term.*

#### Definition 3.4.6

Given two assumption sets,  $A_1$  and  $A_2$ , then for each  $x$ ,  $A_1 \text{ m } A_2$  is the assumption set defined by:

$$A_1 \text{ m } A_2(x) = \begin{cases} A_1(x), & x \in \text{dom}(A_1) - \text{dom}(A_2) \\ A_2(x), & x \in \text{dom}(A_2) - \text{dom}(A_1) \\ A_1(x) \cap A_2(x), & x \in \text{dom}(A_1) \cap \text{dom}(A_2) \end{cases}$$

#### Lemma 3.4.7

If  $A \vdash_V^I M : \tau$ , then, for any  $A'$ ,  $A \text{ m } A' \vdash_V^I M : \tau$ .

#### Proof

Easy induction on the form of the deduction of  $A \vdash_V^I M : \tau$ . In particular, consider the case when the final rule used is rule ABS (so  $M \equiv \lambda x. N$ , for some  $N$ ). Then the induction hypothesis for this case is  $(A \text{ m } A')_x \cup \{x : \sigma\} \vdash_V^I [x := b] N : \tau$  and the result follows.  $\square$

Every  $\lambda$ -term possesses a type assignment using the above type assignment system, for some set of assumptions  $A$  of types for type variables.

VAR	$A_x \cup \{x:\sigma\} \vdash_{V \multimap [x:=\Rightarrow]}^I x:\sigma$
APP	$\frac{A \vdash_{V_1}^I N_1:\sigma \text{ b } \tau \quad A \vdash_{V_2}^I N_2:\sigma}{A \vdash_V^I N_1 N_2:\tau}$ $(V = \underline{\lambda}x.V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x:\sigma\} \vdash_{V[x:=b]}^I N:\tau}{A \vdash_{V[x:=\Rightarrow]}^I \underline{\lambda}x.N:\sigma \text{ b } \tau}$
MEET	$\frac{A \vdash_V^I N:\sigma \quad A \vdash_V^I N:\tau}{A \vdash_V^I N:\sigma \cap \tau}$
LEQ	$\frac{A \vdash_V^I N:\sigma \quad \sigma \leq \tau}{A \vdash_V^I N:\tau}$

Figure 9: The Intersection-style Rules for deducing Reduction Types

**Theorem 3.4.8**

$\forall M \in \Lambda. \exists A, \tau, V. A \vdash_V^I M:\tau.$

**Proof**

By induction on  $M$ . The interesting case is application. Suppose  $M \equiv N_1 N_2$ , then from the induction hypotheses

$$\exists A_1, \sigma_1, V_1. A_1 \vdash_{V_1}^I N_1:\sigma_1$$

and

$$\exists A_2, \sigma_2, V_2. A_2 \vdash_{V_2}^I N_2:\sigma_2,$$

then by Lemma 3.4.7 it is clear that  $A_1 \sqcap A_2 \vdash_{V_1}^I N_1:\sigma_1$  and  $A_1 \sqcap A_2 \vdash_{V_2}^I N_2:\sigma_2$  are both derivable type assignments for  $N_1$  and  $N_2$  (respectively). Now form the deduction:

$$\frac{\frac{A_1 \sqcap A_2 \vdash_{V_1}^I N_1:\sigma_1 \quad \sigma_1 \leq \omega \text{ b } \omega}{A_1 \sqcap A_2 \vdash_{V_1}^I N_1:\omega \text{ b } \omega} \quad \frac{A_1 \sqcap A_2 \vdash_{V_2}^I N_2:\sigma_2 \quad \sigma_2 \leq \omega}{A_1 \sqcap A_2 \vdash_{V_2}^I N_2:\omega}}{A_1 \sqcap A_2 \vdash_V^I N_1 N_2:\omega}$$

where  $V = \underline{\lambda}x.V_1(x) \vee (b \wedge V_2(x)). \quad \square$

As is to be hoped given the intuitive interpretation of the symbol  $\omega$  (see Chapter 4 for a formal interpretation), every term may be assigned this type



by the type assignment system.

**Corollary 3.4.9**

$\forall M \in \Lambda. \exists A, V. A \vdash_V^I M : \omega.$

**Proof**

Immediate by Theorem 3.4.8 and rule LEQ.  $\square$

### 3.4.3 Extending Intersection-style Inference

Consider the following rule:

$$\text{OMEGA} \quad \forall A, V. A \vdash_V^I M : \omega$$

Suppose that  $x \notin \text{FV}(F)$  and  $\emptyset \vdash_{V_{\omega}}^I F : \omega \Rightarrow \sigma$ , for some  $\sigma \neq \omega$  (for example, choose  $F \equiv \lambda z. yz$ ,  $y \neq x$  and assume  $y : \omega \Rightarrow \sigma$ ). Then using the OMEGA rule it is easy to show  $\{x : \omega\} \vdash_{V_{\omega}}^I x : \omega$  and hence by APP and ABS:  $\emptyset \vdash_{V_{\omega}}^I \lambda x. Fx : \omega \multimap \sigma$ . This type of deduction is certainly erroneous if we wish to have types preserved under  $\eta$ -reduction.

Another rule which might be considered is:

$$\frac{M \text{ is unsolvable}}{A \vdash_{V_{\omega}}^I M : \omega}$$

Note that this rule is *not* derivable in the system excluding rule APP- $\multimap$ , since an arbitrary assumption set is allowed by the rule. (Consider the term  $K\Omega(xx)$  which is clearly unsolvable, then this rule allows  $\{x : \alpha\} \vdash_{V_{\omega}}^I K\Omega(xx) : \omega$ ).

In Figure 10 the now usual extension of the system is proposed—the APP- $\multimap$  rule has been added whose main purpose is to deal with the situation in which an unneeded argument is supplied to a function. Notice that this rule is very naturally expressed using the type constant  $\omega$ . As will be shown, the lack of any rule analogous to the  $(\omega)$  rule of the intersection type discipline weakens the  $\vdash^I$  system with respect to invariance under  $\beta$ -conversion of terms. (The standard system for intersection types has this property, see Coppo and Cardone [12], for example). The APP- $\multimap$  rule regains some of this power by simulating the  $(\omega)$  rule in the particular context of a function which does not require its argument. The “protection” afforded by restricting use of the  $(\omega)$  rule to this context ensures the preservation of the validity of the strong head neededness information deduced.

As with earlier systems, it will be the system excluding the rule APP- $\multimap$  that will in general be referred to in the rest of this work, except where otherwise noted. However, the current section always takes this rule into consideration since it allows the establishment of a stronger result concerning  $\beta$ -expansion.

VAR	$A_x \cup \{x:\sigma\} \vdash_{V \rightarrow [x:=\Rightarrow]}^I x:\sigma$
APP	$\frac{A \vdash_{V_1}^I N_1:\sigma \text{ b } \tau \quad A \vdash_{V_2}^I N_2:\sigma}{A \vdash_V^I N_1 N_2:\tau}$ $(V = \underline{\lambda}x.V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x:\sigma\} \vdash_{V[x:=b]}^I N:\tau}{A \vdash_{V[x:=\neg]}^I \lambda x.N:\sigma \text{ b } \tau}$
MEET	$\frac{A \vdash_V^I N:\sigma \quad A \vdash_V^I N:\tau}{A \vdash_V^I N:\sigma \cap \tau}$
LEQ	$\frac{A \vdash_V^I N:\sigma \quad \sigma \leq \tau}{A \vdash_V^I N:\tau}$
APP $\rightarrow$	$\frac{A \vdash_V^I N_1:\omega \rightarrow \tau \quad A' \vdash_{V'}^I N_2:\omega}{A \vdash_V^I N_1 N_2:\tau}$

Figure 10: The Extended Intersection-style Rules for deducing Reduction Types

### 3.4.4 Examples

As an example of the use of this system for type assignment of Boolean Reduction Types, consider the deduction of a type for the term  $Twice \equiv \lambda f x. f(fx)$ . (This term is also known as the Church numeral  $c_2$ ). The following typing statement is deduced given  $f: (\alpha \rightarrow_1 \beta) \cap (\beta \rightarrow_2 \gamma)$  and  $x: \alpha$ :

$$\emptyset \vdash_{V_{\omega}}^I Twice: ((\alpha \rightarrow_1 \beta) \cap (\beta \rightarrow_2 \gamma)) \Rightarrow \alpha (\rightarrow_1 \wedge \rightarrow_2) \gamma.$$

(See Cardone and Coppo [12] for a derivation of a type for  $Twice$  in the Intersection Type Discipline). The following are instances of the above type for  $Twice$ :

$$((\alpha \rightarrow_1 \beta) \cap (\beta \rightarrow_1 \gamma)) \Rightarrow \alpha \Rightarrow \gamma,$$

$$(\alpha \rightarrow_1 \alpha) \Rightarrow \alpha \rightarrow_1 \alpha,$$

and

$$((\alpha \rightarrow_1 \beta) \cap (\beta (\neg \rightarrow_1) \gamma)) \Rightarrow \alpha \nrightarrow \gamma.$$

In Chapter 4 the term  $Twice$  is indeed interpreted as an element of these types.

Consider the example from the previous section on LET-polymorphic style type assignment:

$$\lambda f. g(f \text{ I } x)(f (\lambda x. y) z).$$

In the LET-polymorphic system no type could be found for this term unless its argument was known. In contrast, the following deduction holds in the intersection-style system. Suppose  $g: \sigma_1 \rightarrow_i \sigma_2 \rightarrow_j \sigma_3$ ,  $x: \rho_1$ ,  $z: \rho_2$  and  $f: ((\rho_1 \Rightarrow \rho_1) \rightarrow_k \rho_1 \rightarrow_l \sigma_1) \cap ((\rho_2 \nrightarrow \tau_2) \rightarrow_m \rho_2 \rightarrow_n \sigma_2)$  are assumptions in some assumption set  $A$ . Then the LEQ rule may be used after each instance of a use of the VAR rule for  $f$  in the term to obtain deductions of

$$f: (\rho_1 \Rightarrow \rho_1) \rightarrow_k \rho_1 \rightarrow_l \sigma_1$$

and

$$f: (\rho_2 \nrightarrow \tau_2) \rightarrow_m \rho_2 \rightarrow_n \sigma_2,$$

then the rest is clear: there exists a deduction for this term in the intersection-style system (apart from the trivial one assigning  $\omega$  to it, see Theorem 3.4.8).

Another example is the term  $\lambda x. xx$ . A type deducible for this term is  $((\alpha \rightarrow_1 \beta) \cap \alpha) \Rightarrow \beta$ . This tells us that  $x$  may behave as both a function and an argument, as well as saying that the argument to  $\lambda x. xx$  is strongly head needed, independently of whether or not the argument requires *its* argument. From this it follows that the term  $\Omega \equiv (\lambda x. xx)(\lambda x. xx)$  only has type  $\omega$ . (This

type for  $\Omega$  is derived in a similar fashion to the proof for the case of application in Theorem 3.4.8 above).

Consider the term  $Y \equiv \lambda f. \omega_f \omega_f$ , where  $\omega_f \equiv \lambda x. f(xx)$ . This term is a fixed point combinator introduced by Curry, see [3, 30]. Suppose that  $x: \omega \rightarrow_1 \alpha$  and  $f: \omega \rightarrow_1 \alpha$ , then using rule LEQ, we can deduce  $xx: \omega$  and so  $\omega_f: (\omega \rightarrow_1 \alpha) \rightarrow_1 \alpha$ . Similarly, Suppose that  $x: \omega$ , then it is easy to show that  $\omega_f: \omega \rightarrow_1 \alpha$ . Thus,  $\omega_f \omega_f: \alpha$  and  $Y: (\omega \rightarrow_1 \alpha) \Rightarrow \alpha$ , as expected.

Finally, consider Turing's fixed point combinator  $\Theta \equiv AA$ , where  $A \equiv \lambda x f. f(xxf)$ . Suppose  $f: \omega \rightarrow_1 \alpha$ , then we can deduce

$$A: (\omega \rightarrow_1 (\omega \rightarrow_1 \beta) \Rightarrow \beta) \rightarrow_1 (\omega \rightarrow_1 \alpha) \Rightarrow \alpha$$

and  $A: \omega \rightarrow_1 (\omega \rightarrow_1 \beta) \Rightarrow \beta$ . Thus,  $\Theta \equiv AA: (\omega \rightarrow_1 \alpha) \Rightarrow \alpha$ . This is the same as the type deduced for  $Y$  above.

### 3.4.5 Properties of the Intersection-style System

Note that the sets of derivations in the present type assignment systems are constrained by more than just the absence of an  $(\omega)$  rule, when compared with the Intersection Type Discipline. This is because the MEET rule is more demanding than the  $(\cap I)$  rule of the Intersection Type Discipline.<sup>10</sup> A good example to think about is the various derivations for the term  $\lambda f. fx$ . In particular, for this term we can show

$$\{x: \sigma\} \vdash_{V_\omega}^I \lambda f. fx: (\sigma \multimap \tau) \Rightarrow \tau$$

and

$$\{x: \sigma\} \vdash_{V_\omega[x:=\Rightarrow]}^I \lambda f. fx: (\sigma \Rightarrow \tau) \Rightarrow \tau.$$

So we cannot use the MEET rule with these two derivations, though in the Intersection Type Discipline one cannot distinguish between these and so all such derivations can be used as antecedents to the  $(\cap I)$  rule. Note that the MEET rule is applicable to all *closed* terms (under the same assumption set), as these terms all have the variable strong head neededness function  $V_\omega$  assigned to them. As well, the MEET rule is always applicable to term variables (when assigned a type under the same assumption set). The consequences of this weakness of the MEET rule are further discussed below.

<sup>10</sup>The  $(\cap I)$  rule of the Intersection Type Discipline has the following form:

$$\frac{A \vdash M: \sigma_1 \quad A \vdash M: \sigma_2}{A \vdash M: \sigma_1 \cap \sigma_2}$$

Firstly, for a given term and assumption set, if two deductions give identical types, then their variable neededness functions must be equivalent.

**Lemma 3.4.10**

Suppose  $A \vdash_{V_1}^I M : \sigma$  and  $A \vdash_{V_2}^I M : \sigma$ , then  $\forall x \in X. V_1(x) =_{BA} V_2(x)$ .

**Proof**

By induction on the structure of  $M$ .  $\square$

**Lemma 3.4.11**

If  $A \vdash_V^I M : \tau$  is derived from  $A \vdash_V^I M : \sigma_i$  ( $i = 1, \dots, n$ ) using only rules MEET and LEQ, then  $\bigcap_{i=1}^n \sigma_i \leq \tau$ .

**Proof**

Straightforward induction on  $A \vdash_V^I M : \tau$ .  $\square$

**Lemma 3.4.12**

If  $A \vdash_V^I MN : \tau$  and this is derived using rule APP, possibly followed by some instances of LEQ and MEET, then  $\exists \sigma \in T_I^\nabla, b \in \nabla. (A \vdash_V^I M : \sigma \text{ b } \tau, A \vdash_{V''}^I N : \sigma \text{ and } \forall x \in X. V = \lambda x. V'(x) \vee (b \wedge V''(x)))$ .

**Proof**

By induction on the structure of  $A \vdash_V^I MN : \tau$ . The proof is similar to the proof of Lemma 2.8.(i) of Barendregt et al [4]. As in their proof, the interesting case is that of rule MEET. The essential step is:

$$\begin{aligned} (\sigma_1 \text{ b } \tau_1) \cap (\sigma_2 \text{ b } \tau_2) &\leq ((\sigma_1 \cap \sigma_2) \text{ b } \tau_1) \cap ((\sigma_1 \cap \sigma_2) \text{ b } \tau_2) \\ &\leq (\sigma_1 \cap \sigma_2) \text{ b } (\tau_1 \cap \tau_2). \end{aligned}$$

$\square$

The following lemma is not applicable to the system based on  $T_C^{\{\Rightarrow\}}$  because rule APP- $\rightarrow$  is not definable in that system.

**Lemma 3.4.13**

If  $A \vdash_V^I MN : \tau$  and this is derived using rule APP- $\rightarrow$ , possibly followed by some instances of LEQ and MEET, then  $\exists A' \in \{X \times T_I^\nabla\}. (A \vdash_V^I M : \omega \rightarrow \tau \text{ and } A' \vdash_{V''}^I N : \omega). (\nabla \text{ not } \{\Rightarrow\})$

**Proof**

By induction on  $A \vdash_V^I MN : \tau$ . The induction cases are when the deduction ends in instances of either MEET or LEQ, the basis is when the deduction ends in an instance APP- $\rightarrow$ .  $\square$

**Lemma 3.4.14**

$A \vdash_V^I \lambda x. M : \sigma \text{ b } \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_{V[x:=b]}^I M : \tau$ .

**Proof**

By Lemma 3.4.5 and Lemma 3.4.11 (similar to the proof of Lemma 2.8.(iii) in Barendregt et al [4]).  $\square$

**Lemma 3.4.15**

Suppose  $\forall \sigma, \tau \in T_I^\nabla. A_x \cup \{x : \sigma\} \vdash_V^I M : \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_V^I N : \tau$ , then  $\forall \rho \in T_I^\nabla. A \vdash_{V[x:=\rightarrow]}^I \lambda x. M : \rho$  implies  $A \vdash_{V[x:=\rightarrow]}^I \lambda x. N : \rho$ .

**Proof**

Suppose that  $A \vdash_{V[x:=\rightarrow]}^I \lambda x. M : \rho$  is derived using rule ABS. By Lemma 3.4.14,  $A_x \cup \{x : \sigma\} \vdash_V^I M : \tau$ , and from the assumption we have  $A_x \cup \{x : \sigma\} \vdash_V^I N : \tau$ . Then the result follows from rule ABS. The other cases (MEET, LEQ) follow by induction.  $\square$

**Corollary 3.4.16**

Let  $M \equiv NN_1 \dots N_n$  where  $N$  is not an application term and  $n \geq 0$ . Suppose  $A \vdash_V^I M : \tau$  without using rule APP- $\rightarrow$ , then  $\exists \rho_i \in T_I^\nabla$  and  $V_i$  ( $1 \leq i \leq n$ ):

1.  $A \vdash_{V_i}^I N_i : \rho_i$ ,  $A \vdash_V^I N : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$  and  $V = \underline{\lambda}x. V'(x) \vee (\bigvee_{i=1}^n (b_i \wedge V_i(x)))$ ,
2. if  $N \equiv x$ , then  $A = A'_x \cup \{x : \mu\}$ , where either

$$\mu = \sigma_1 b_1 \sigma_2 b_2 \dots b_{n-1} \sigma_n b_n \tau'$$

or

$$\mu = (\sigma_1 b_1 \sigma_2 b_2 \dots b_{n-1} \sigma_n b_n \tau') \cap \sigma,$$

for some  $\sigma, \sigma_i$  such that  $\sigma_i \leq \rho_i$  and  $\tau' \leq \tau$ ,  $V' = V_\rightarrow[x := \Rightarrow]$ , and

3. if  $N \equiv \lambda x. N'$ , then  $A_x \cup \{x : \mu\} \vdash_{V'[x:=b_1]}^I N' : \nu$ , where  $\rho_1 \leq \mu$  and  $\nu \leq \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$ .

**Proof**

1. By iterated application of Lemma 3.4.12.
2. Since  $A \vdash_V^I x : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$  is derived using only rules VAR, MEET and LEQ and by Lemma 3.4.11.
3. Since  $A \vdash_V^I \lambda x. N' : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau$  must be derived using only rules ABS, MEET and LEQ and by Lemma 3.4.14.

$\square$

**Lemma 3.4.17**

If  $A \vdash_V^I \lambda x.M : \tau$ , and  $\tau \neq \omega$ , then  $\tau = \tau_1 \cap \tau_2 \cap \dots \cap \tau_n$  ( $n \geq 1$ ), where  $\tau_i = \mu_i b_i \nu_i$  ( $1 \leq i \leq n$ ).

**Proof**

Each deduction of a type for  $\lambda x.M$  must end with a use of the ABS rule followed by zero or more uses of MEET and/or LEQ. Note that both MEET and LEQ cannot alter the number of arrow expressions in a type if the case  $\sigma \leq \omega$  is disallowed.  $\square$

The rules of strengthening and weakening are also derivable in the system for deducing Intersection Boolean Reduction Types. These rules may be written as usual:

STR	$\frac{A_x \cup \{x : \sigma\} \vdash_V^I M : \tau}{A_x \vdash_V^I M : \tau} \quad (x \notin \text{FV}(M))$
WEAK	$\frac{A \vdash_V^I M : \tau}{A_x \cup \{x : \sigma\} \vdash_V^I M : \tau} \quad (x \notin \text{FV}(M))$

This is verified by the following pair of Lemmas:

**Lemma 3.4.18**

Suppose  $x \notin \text{FV}(M)$ , then  $A \vdash_V^I M : \tau$  implies  $A_x \cup \{x : \sigma\} \vdash_V^I M : \tau$ .

**Proof**

Straightforward induction over the deduction of  $A \vdash_V^I M : \tau$ .  $\square$

**Lemma 3.4.19**

If  $A \vdash_V^I M : \tau$  and  $x \notin \text{FV}(M)$ , then  $A_x \vdash_V^I M : \tau$ .

**Proof**

Straightforward induction over the deduction of  $A \vdash_V^I M : \tau$ .  $\square$

All variable strong head neededness functions derivable by the type assignment system share a common structure. This can be expressed using the notion of substitution as defined on  $\lambda$ -terms earlier.

**Lemma 3.4.20**

Suppose  $A \vdash_V^I M[x := N] : \tau$ , the types  $\sigma_1, \dots, \sigma_n$  are assigned to  $N$  in this deduction and  $A \vdash_V^I N : \sigma$ , where  $\sigma = \sigma_1 \cap \dots \cap \sigma_n$ , then,

- $A_x \cup \{x : \sigma\} \vdash_V^I M[x := b] : \tau$ , and
- $\forall y \in X. V(y) =_{\text{BA}} V''(y) \vee (b \wedge V'(y))$ .

**Proof**

By induction on  $M[x := N]$ .

$x[x := N] \equiv N$  In this case  $M[x := N] \equiv N$ , so  $\sigma_1 \leq \tau, \dots, \sigma_n \leq \tau$  and  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^I x : \tau$ , where  $V'' =_{BA} V_{\rightarrow}$  and  $b =_{BA} \Rightarrow$ . Now,

$$\begin{aligned} V(y) &=_{BA} V'(y) \\ &=_{BA} V_{\rightarrow}(y) \vee (\Rightarrow \wedge V'(y)) \\ &=_{BA} V''(y) \vee (b \wedge V'(y)), \end{aligned}$$

as required.

$z[x := N] \equiv z$  ( $z \neq x$ ) It is immediate that

$$A \vdash_{V_{\rightarrow}[z:=\Rightarrow]}^I z : \tau,$$

so  $V'' =_{BA} V_{\rightarrow}[z := \Rightarrow]$  and  $b =_{BA} \rightarrow$ . Now, for any  $V'$ ,

$$\begin{aligned} V(y) &=_{BA} (V_{\rightarrow}[z := \Rightarrow])(y) \vee (\rightarrow \wedge V'(y)) \\ &=_{BA} V''(y) \vee (b \wedge V'(y)). \end{aligned}$$

$N_1 N_2[x := N] \equiv (N_1[x := N])(N_2[x := N])$  If  $A \vdash_V^I N_1 N_2[x := N] : \tau$  is deduced using rules LEQ or MEET, then the result holds in these cases by induction. Assume that  $A \vdash_V^I N_1 N_2[x := N] : \tau$  is deduced using rule APP. By Lemma 3.4.12:

- $A \vdash_{V_1}^I N_1[x := N] : \rho \ b' \ \tau$ , and
- $A \vdash_{V_2}^I N_2[x := N] : \rho$ ,

where  $V(y) =_{BA} V_1(y) \vee (b' \wedge V_2(y))$ . By the induction hypotheses:

- $A_x \cup \{x : \sigma\} \vdash_{V_1''[x:=b_1]}^I N_1 : \rho \ b' \ \tau$ , and
- $A_x \cup \{x : \sigma\} \vdash_{V_2''[x:=b_2]}^I N_2 : \rho$ ,

where  $V_1(y) =_{BA} V_1''(y) \vee (b_1 \wedge V'(y))$  and  $V_2(y) =_{BA} V_2''(y) \vee (b_2 \wedge V'(y))$ . By APP,  $A_x \cup \{x : \sigma\} \vdash_{V''[x:=b]}^I N_1 N_2 : \tau$ , where  $V''(y) =_{BA} V_1''(y) \vee (b' \wedge V_2''(y))$  and  $b =_{BA} b_1 \vee (b' \wedge b_2)$ . By APP,

$$\begin{aligned} V(y) &=_{BA} (V_1''(y) \vee b_1 V'(y)) \vee (b' (V_2''(y) \vee b_2 V'(y))) \\ &=_{BA} V_1''(y) \vee b_1 V'(y) \vee b' (V_2''(y) \vee b_2 V'(y)) \\ &=_{BA} V_1''(y) \vee b_1 V'(y) \vee b' V_2''(y) \vee b' b_2 V'(y) \\ &=_{BA} V_1''(y) \vee b' V_2''(y) \vee b_1 V'(y) \vee b' b_2 V'(y) \\ &=_{BA} (V_1''(y) \vee b' V_2''(y)) \vee (b_1 \vee b' b_2) V'(y) \\ &=_{BA} V''(y) \vee b V'(y). \end{aligned}$$



$N_1 N_2[x := N] \equiv (N_1[x := N])(N_2[x := N])$  and rule APP- $\rightarrow$  is used as the final step in the deduction. Straightforward by the induction hypotheses, Lemma 3.4.13 and rule APP- $\rightarrow$ .

$(\lambda z.N')[x := N] \equiv \lambda z.(N'[x := N])$  If  $A \vdash_V^I (\lambda z.N')[x := N] : \tau$  is deduced using rules LEQ or MEET, then the result holds in these cases by induction. Assume that  $A \vdash_V^I (\lambda z.N')[x := N] : \tau$  is deduced using rule ABS, then  $\tau = \rho \mathbf{b}' \sigma'$ , for some  $\rho$ ,  $\mathbf{b}'$  and  $\sigma'$ . By induction hypothesis,

$$A_{zx} \cup \{z : \rho, x : \sigma\} \vdash_{V_1'[x := \mathbf{b}_1][z := \mathbf{b}']}^I N' : \sigma',$$

then by ABS,

$$A_x \cup \{x : \sigma\} \vdash_{V_1''[x := \mathbf{b}_1][z := \rightarrow]}^I \lambda z.N' : \rho \mathbf{b}' \sigma'.$$

Also by induction hypothesis,

$$A_z \cup \{z : \rho\} \vdash_{V_1[z := \mathbf{b}']}^I N'[x := N] : \sigma',$$

where  $V_1[z := \mathbf{b}'](y) =_{\text{BA}} V_1''(y)[z := \mathbf{b}'] \vee (\mathbf{b}_1 \wedge V'(y))$ . (Note that by the variable convention,  $z \notin \text{FV}(N)$  hence  $V'(z) =_{\text{BA}} \rightarrow$ , see Barendregt [3]). Now,

$$\begin{aligned} V(y) &=_{\text{BA}} V_1[z := \rightarrow](y) \\ &=_{\text{BA}} V_1''[z := \rightarrow](y) \vee (\mathbf{b}_1 \wedge V'(y)), \end{aligned}$$

as required.

□

#### Lemma 3.4.21

If  $A_x \cup \{x : \sigma\} \vdash_{V'[x := \mathbf{b}]}^I M : \tau$ ,  $A \vdash_V^I N : \sigma$ ,  $x \notin \text{FV}(N)$  and  $V'(x) = \rightarrow$ , then

- $A \vdash_V^I M[x := N] : \tau$ , and
- $\forall y \in X. V(y) =_{\text{BA}} V'(y) \vee (\mathbf{b} \wedge V''(y))$ .

#### Proof

By induction on  $M$ .

$M \equiv x$  In this case  $M[x := N] \equiv N$ .

1.  $A_x \cup \{x : \sigma\} \vdash_{V_{\rightarrow}[x := \rightarrow]}^I x : \tau$ ,  $\sigma \leq \tau$ ,  $V' = V_{\rightarrow}$  and  $\mathbf{b} =_{\text{BA}} \Rightarrow$ .
2. By LEQ,  $A \vdash_V^I N : \tau$ .

3.  $A \vdash_V^I x[x := N] : \tau$ .

$M \equiv z, z \neq x$  In this case  $M[x := N] \equiv M$ .

1.  $A_x \cup \{x : \sigma\} \vdash_{V_{\rightarrow}[z := \Rightarrow]}^I z : \tau, V' = V_{\rightarrow}[z := \Rightarrow]$  and  $\mathbf{b} =_{\text{BA}} \dashv$ .
2. By Lemmas 3.4.18, 3.4.18 and 3.4.19,  $A \vdash_{V_{\rightarrow}[z := \Rightarrow]}^I z : \tau$ .
3.  $A \vdash_V^I z[x := N] : \tau$ .

$M \equiv N_1 N_2$  In this case  $M[x := N] \equiv (N_1[x := N])(N_2[x := N])$ . The cases MEET and LEQ are straightforward. Suppose that APP is the final rule used in the deduction.

1. By Lemma 3.4.12,  $\exists \sigma' \in T_I^\nabla. A \vdash_{V_1'}^I N_1 : \sigma' \mathbf{b}' \tau, A \vdash_{V_2'}^I N_2 : \sigma'$  and  $(V'[x := \mathbf{b}])(y) = V_1'(y) \vee (\mathbf{b}' \wedge V_2'(y))$ .
2. By the induction hypotheses,  $A \vdash_{V_1''}^I N_1[x := N] : \sigma' \mathbf{b}' \tau$ , where  $V_1''(y) = V_1'(y) \vee (V_1'(x) \wedge V''(y))$  and  $A \vdash_{V_2''}^I N_2[x := N] : \sigma'$ , where  $V_2''(y) = V_2'(y) \vee (V_2'(x) \wedge V''(y))$ .
3. 
$$\begin{aligned} & V_1''(y) \vee (\mathbf{b}' \wedge V_2''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (V_1'(x) \wedge V''(y)) \vee (\mathbf{b}' \wedge V_2'(y)) \vee \\ & \quad (\mathbf{b}' \wedge V_2'(x) \wedge V''(y)) \\ &=_{\text{BA}} V_1'(y) \vee (\mathbf{b}' \wedge V_2'(y)) \vee (V_1'(x) \vee \\ & \quad (\mathbf{b}' \wedge V_2'(x))) \wedge V''(y) \\ &=_{\text{BA}} (V'[x := \mathbf{b}])(y) \vee (\mathbf{b} \wedge V''(y)). \end{aligned}$$
4. By APP,  $A \vdash_V^I (N_1 N_2)[x := N] : \tau$ .

$M \equiv N_1 N_2$  and rule APP- $\dashv$  is used to make this deduction. As before, the cases LEQ and MEET follow easily. Suppose APP- $\dashv$  is the final rule used.

By definition of substitution  $M[x := N] \equiv (N_1[x := N])(N_2[x := N])$ . Then the result follows from Lemma 3.4.13, rule APP- $\dashv$  and the induction hypotheses.

$M \equiv \lambda z. M'$  If  $A_x \cup \{x : \sigma\} \vdash_{V'[x := \mathbf{b}]}^I \lambda z. M' : \tau$  is derived using either of MEET or LEQ, then the result follows easily from the induction hypotheses. The interesting case is ABS:

1. By Lemma 3.4.14,  $\tau = \sigma_1 \mathbf{b}' \sigma_2$ , for some  $\sigma_i$  and  $\mathbf{b}'$  and  $A_{xz} \cup \{x : \sigma, z : \sigma_1\} \vdash_{V''[x := \mathbf{b}, z := \mathbf{b}]}^I M' : \sigma_2$ .
2. By the induction hypothesis,  $A_z \cup \{z : \sigma_1\} \vdash_{V'''}^I M'[x := N] : \sigma_2$ , where  $V'''(y) = (V''[z := \mathbf{b}'])(y) \vee (\mathbf{b} \wedge V''(y))$ .
3. By ABS,  $A \vdash_V^I (\lambda y. M')[x := N] : \sigma_1 \mathbf{b}' \sigma_2$ .

□

### Contraction and Expansion

One of the nice properties of the Intersection Type Discipline is that the set of types deducible for a term is invariant under  $\beta$ -convertibility. In this section this property is considered for the present deduction systems for Reduction Types. It turns out that the property does *not* hold for these deduction systems, though some weaker results do hold. The main problems are the absence of an  $\omega$  axiom (partly compensated for by the introduction of the APP- $\rightarrow$  rule) and the relative weakness of the MEET rule. Thus, for the extended system incorporating the APP- $\rightarrow$  rule the main restriction is the form of the MEET rule.

Types are invariant under  $\beta$  contraction. (The Subject-Reduction property, see Curry and Feys [18]).

#### Theorem 3.4.22

If  $A \vdash_V^I (\lambda x.M)N : \tau$ , then  $A \vdash_V^I M[x := N] : \tau$ .

#### Proof

By induction on  $A \vdash_V^I (\lambda x.M)N : \tau$ , using Lemma 3.4.21. (See the proof of Theorem 3.2.21 for a more detailed but substantially similar proof).  $\square$

The above result is generalised to full  $\beta$ -reduction in the following corollary.

#### Corollary 3.4.23

Let  $M \rightarrow_\beta N$ , then  $A \vdash_V^I M : \tau$  implies  $A \vdash_V^I N : \tau$ .

#### Proof

By iterated use of Theorem 3.4.22 along the reduction path from  $M$  to  $N$ .  $\square$

Now consider  $\beta$ -expansion. Firstly, the case of non-trivial  $\beta$ -expansion. In this case, types are invariant under the restriction that the subterm being abstracted has a single variable strong head neededness function which is assigned to every relevant occurrence of the subterm.

#### Theorem 3.4.24

If  $A \vdash_V^I M[x := N] : \tau$  and  $x \in \text{FV}(M)$ , and suppose  $\sigma_1, \dots, \sigma_n$  are assigned to relevant occurrences of  $N$  in the deduction of  $A \vdash_V^I M[x := N] : \tau$ , then if  $A \vdash_V^I N : \sigma_i$  ( $\forall i \in \{1, \dots, n\}$ ), then  $A \vdash_V^I (\lambda x.M)N : \tau$ .

#### Proof

1. Let  $\sigma_{i_1}, \dots, \sigma_{i_p}$  be the types assigned to relevant occurrences of  $N$  in the deduction of  $A \vdash_V^I M[x := N] : \tau$ , for  $i_1, \dots, i_p \in \{1, \dots, n\}$ , then by MEET,  $A \vdash_V^I N : \sigma_{i_1} \cap \dots \cap \sigma_{i_p}$  ( $p \leq n$ ).
2. By Lemma 3.4.20,  $A_x \cup \{x : \sigma_{i_1} \cap \dots \cap \sigma_{i_p}\} \vdash_{V''[x:=b]}^I M : \tau$  and  $\forall y \in X. V(y) =_{\text{BA}} V''(y) \vee (b \wedge V''(y))$ .

3. By ABS,  $A \vdash_{V''[x := \rightarrow]}^I \lambda x.M : (\sigma_{i_1} \cap \dots \cap \sigma_{i_p}) \multimap \tau$ .
4.  $V''[x := \rightarrow] =_{BA} V''$ , since  $x \notin \text{FV}(M[x := N])$ .
5. By APP,  $A \vdash_V^I (\lambda x.M)N : \tau$ .

□

Note that there are many cases where the conditions on  $N$  in this lemma are trivially satisfied. For example, if  $N$  is a closed  $\lambda$ -term or a variable.

Also the following weak result may be established for the second case of  $\beta$ -expansion under  $\vdash^I$  *excluding* the rule APP- $\rightarrow$ . This result is weak since, in the absence of APP- $\rightarrow$ , the type assignment system does not have an equivalent to the  $\omega$  rule of the Intersection Type Discipline (see Cardone and Coppo [12] for a proof of this case in the Intersection Type Discipline, which depends on the presence of the  $\omega$  rule).

**Theorem 3.4.25**

If  $A \vdash_V^I M[x := N] : \tau$  and  $x \notin \text{FV}(M)$ , then, for some  $A'$ ,  $A \multimap A' \vdash_V^I (\lambda x.M)N : \tau$ .

**Proof**

1.  $M[x := N] \equiv M$ , so  $A \vdash_V^I M : \tau$ .
2. By Lemma 3.4.18,  $A_x \cup \{x : \omega\} \vdash_V^I M : \tau$ .
3. By ABS,  $A \vdash_V^I \lambda x.M : \omega \multimap \tau$ .
4. By Corollary 3.4.9,  $A' \vdash_{V'}^I N : \omega$  for some  $A'$  and  $V'$ .
5.  $A \multimap A' \vdash_V^I \lambda x.M : \omega \multimap \tau$ .
6. By APP,  $A \multimap A' \vdash_V^I (\lambda x.M)N : \tau$ .

□

In the second case of  $\beta$ -expansion under the system including rule APP- $\rightarrow$ , a full strength result follows.

**Theorem 3.4.26**

If  $A \vdash_V^I M[x := N] : \tau$  and  $x \notin \text{FV}(M)$ , then,  $A \vdash_V^I (\lambda x.M)N : \tau$ .

**Proof**

1.  $M[x := N] \equiv M$ , so  $A \vdash_V^I M : \tau$ .
2. By Lemma 3.4.18,  $A_x \cup \{x : \omega\} \vdash_V^I M : \tau$ .
3. By ABS,  $A \vdash_V^I \lambda x.M : \omega \multimap \tau$ .

4. By Corollary 3.4.9,  $A' \vdash_V^I N : \omega$  for some  $A'$  and  $V'$ .

5. By APP- $\rightarrow$ ,  $A \vdash_V^I (\lambda x.M)N : \tau$ .

□

The type assignment system behaves well under  $\eta$  contraction.

**Theorem 3.4.27**

If  $x \notin \text{FV}(M)$  and  $A \vdash_V^I \lambda x.Mx : \tau$ , then  $A \vdash_V^I M : \tau$ .

**Proof**

By induction on  $A \vdash_V^I \lambda x.Mx : \tau$ . The interesting case is where the deduction ends in a use of the ABS rule:

1.  $\tau = \sigma_1 \text{ b } \sigma_2$ .

2. By ABS,  $A_x \cup \{x : \sigma_1\} \vdash_{V[x:=b]}^I Mx : \sigma_2$ .

3. By Lemma 3.4.12,  $\exists \sigma_3 \in T_I^\nabla. A_x \cup \{x : \sigma_1\} \vdash_V^I M : \sigma_3 \text{ b } \sigma_2$  and  $A_x \cup \{x : \sigma_1\} \vdash_{V \multimap [x:=\Rightarrow]}^I x : \sigma_3$ .

4.  $\sigma_1 \leq \sigma_3$ , hence  $\sigma_3 \text{ b } \sigma_2 \leq \sigma_1 \text{ b } \sigma_2$ .

5. By LEQ,  $A_x \cup \{x : \sigma_1\} \vdash_V^I M : \sigma_1 \text{ b } \sigma_2$ .

6. By Lemma 3.4.18,  $A \vdash_V^I M : \sigma_1 \text{ b } \sigma_2$ .

□

### 3.4.6 Discussion

This is the most powerful of the type assignment systems considered. Naturally, this results in a system which is only semi-decidable when implemented, as will be apparent in Chapter 5. On the other hand, the MEET rule is not sufficiently powerful. In particular, identical terms assigned types under equivalent assumption sets may not be applicable as antecedents to this rule if their variable neededness functions differ. What is needed to correct this deficiency is a mechanism to allow the intersection of deductions so as to preserve variable strong head neededness functions. If a variable which is being abstracted has such a tagged value in the variable neededness function then this will result in a tagged arrow appearing in the constructed type and a special mechanism is needed to resolve which deduction is being used in all contexts in which this abstraction occurs. This extension is left for future work.

# Chapter 4

## The Semantics of Reduction Types

This chapter presents semantics for the various sets of Reduction Types considered in the previous chapter. This allows the corresponding type assignment systems to be rigorously tested, the meaning of types to be formalised and classes of terms with like behaviour to be explicitly associated with a reduction type.

Both a traditional approach based on a model of the  $\lambda$ -calculus and a non-traditional one based on a *semi-model* of the  $\lambda$ -calculus are investigated. This latter approach has recently been introduced as a mechanism for conducting a semantic analysis of Curry's original system of F-deducibility (Plotkin [56]). The attraction of this approach in the current context is that it is based on reduction rather than conversion. In this chapter the soundness of all of the type assignment systems will be verified with respect to models of  $\lambda$ , and the soundness and completeness of these systems with respect to semi-models of  $\lambda$ .

The unique aspect of the semantics for Reduction Types presented here is the need to look at the behaviour of a function as it is transformed by applying it to a sequence of arguments. Both the number of arguments and the properties of these arguments are then used to collect together functions into "types" of functions (i.e., the interpretation of types). Thus the semantics introduced in this thesis for Boolean Reduction Types is *context-sensitive*, a fact which makes them of particular interest.

In what follows I have chosen to concentrate on syntactical models. There is a long tradition of using such models in typing contexts (see, for example, Hindley [29], Barendregt et al [4], Mitchell [49], Coppo and Cardone [13], Plotkin [56]), and they are quite capable of elucidating the salient points of the semantics for this thesis.

The chapter commences by introducing two forms of models for terms. The first is the ordinary  $\lambda$ -model and the second Plotkin's semi- $\lambda$ -model. A term

model is described as a concrete instance of a  $\lambda$ -model. A term-semi-model is given as a particular instance of a semi- $\lambda$ -model. These models are useful when the completeness of the type deduction logics is being examined.

In addition, a novel filter semi- $\lambda$ -model is constructed. Filter models are used by Barendregt et al [4] to establish the soundness and completeness of the Intersection Type Discipline. Advantage can be taken of this method in the current approach, as is demonstrated in this chapter.

The next step is to decide what is the meaning of a Boolean Reduction Type. The first step towards deciding this is to provide a semantic analogue for strong head neededness and for irrelevance. Once this has been done, some requirements are specified that a model for Boolean Reduction Types must satisfy. Next, three constructions are given which are shown to be models for  $T_C^\nabla$ ,  $T_L^\nabla$  and  $T_I^\nabla$ , respectively.

One step remains before the correctness of the type deduction systems can be shown. This is the establishment of a connection between (interpretations of)  $\lambda$ -terms and (interpretations of) Boolean Reduction Types. This is expedited by the fact that equality in models of terms is a model for  $\beta$ -conversion, and similarly for reduction and the ordering relation in semi- $\lambda$ -models (Theorems 4.1.2, 4.1.3 and 4.1.5, 4.1.7). This allows the understanding of strong head neededness gathered in Chapter 2 to be brought into play.

With this background, the chapter concludes by proving the correctness of the type deduction systems of Chapter 3.

## 4.1 Models of Terms

### 4.1.1 A Model of the $\lambda$ -calculus

The following definition of  $\lambda$ -model is very syntactic—it would be as easy to base the rest of the Chapter on, say, a Scott-Meyer style definition of  $\lambda$ -model (see Hindley and Seldin [30]), but the definition below is preferred because it allows an easy comparison with the original definition of  $\lambda$ -semi-model. (Presumably the definition of  $\lambda$ -semi-model can also be made less syntactic, though Plotkin [56] does not do this).

Any model of the  $\lambda$ -calculus is a triple consisting of a non-empty set  $D$  called the *domain*, a map  $\bullet: D \times D \rightarrow D$  called *application* (i.e.,  $\langle D, \bullet \rangle$  is an applicative structure) and another map  $\llbracket \cdot \rrbracket$ , which assigns each term  $M \in \Lambda$  to an element  $\llbracket M \rrbracket_\mu \in D$ , for an environment  $\mu: X \rightarrow D$ . Furthermore, the map  $\llbracket \cdot \rrbracket$ , must satisfy, for each  $M \in \Lambda$  and each environment  $\mu: X \rightarrow D$ , the following list of properties (Definition 11.3 of [30]):

1.  $\llbracket x \rrbracket_\mu = \mu(x)$ ,
2.  $\llbracket MN \rrbracket_\mu = \llbracket M \rrbracket_\mu \bullet \llbracket N \rrbracket_\mu$ ,

3.  $\llbracket \lambda x.M \rrbracket_\mu \bullet d = \llbracket M \rrbracket_{\mu[x:=d]}$ , for every  $d \in D$ ,
4.  $\llbracket M \rrbracket_\mu = \llbracket M \rrbracket_{\mu'}$ , if  $\llbracket x \rrbracket_\mu = \llbracket x \rrbracket_{\mu'}$  for all  $x \in \text{FV}(M)$ ,
5.  $\llbracket \lambda x.M \rrbracket_\mu = \llbracket \lambda y.M[x := y] \rrbracket_\mu$ , if  $y \notin \text{FV}(M)$ , (note that the variable convention of Barendregt [3] subsumes this condition in the syntax of  $\lambda$ -terms—effectively this convention results in  $\lambda$ -terms being merely “syntactic sugar” for de Bruijn terms), and
6.  $\llbracket \lambda x.M \rrbracket_\mu = \llbracket \lambda x.N \rrbracket_\mu$ , if  $\forall d \in D. \llbracket M \rrbracket_{\mu[x:=d]} = \llbracket N \rrbracket_{\mu[x:=d]}$ .

### The Term Model

The *term model* is  $\mathcal{M} = \langle D, \bullet, \llbracket \cdot \rrbracket \rangle$  where

$$D = \{[M] \mid M \in \Lambda\},$$

and

$$[M] = \{N \mid \lambda \vdash M = N\}.$$

Let  $[x_1 \cdots x_j := N_1 \cdots N_j]$  denote simultaneous substitution of the  $\lambda$ -terms  $N_1$  to  $N_j$  for the term variables  $x_1$  to  $x_j$ , respectively. Let  $\text{Env}$  denote  $X \rightarrow D$ , then define

$$\begin{aligned} \llbracket \cdot \rrbracket : \Lambda &\rightarrow \text{Env} \rightarrow D \\ \llbracket M \rrbracket_\mu &= [M[x_1 \cdots x_j := N_1 \cdots N_j]], \end{aligned}$$

for  $\mu = [x_1 \cdots x_j := [N_1] \cdots [N_j]]$ . The *canonical* environment is  $\mu_0$ , where  $\mu_0(x) = [x]$ . Finally, let  $[M] \bullet [N] = [MN]$ .

The following results are well known (see Hindley and Seldin [30] or Barendregt [3]).

#### Theorem 4.1.1

$\langle D, \bullet, \llbracket \cdot \rrbracket \rangle$  is a  $\lambda$ -model.

#### Theorem 4.1.2

If  $M =_\beta N$ , then, for any environment  $\mu$ ,  $\llbracket M \rrbracket_\mu = \llbracket N \rrbracket_\mu$ .

#### Theorem 4.1.3

$M =_\beta N$  if in all models and for every environment  $\mu$ ,  $\llbracket M \rrbracket_\mu = \llbracket N \rrbracket_\mu$ .

The main use for the above definition and model construction is in the lead up to and proof of the semantic completeness of the various type assignment systems.



### 4.1.2 A Semi-Model of the $\lambda$ -calculus

Plotkin [56] has introduced the notion of *semi-model* in order to conduct a semantic analysis of Curry's original system of F-deducibility [18]. The key step in this work is the emphasis on *reduction* rather than conversion. In the light of the results of Chapter 3 on expansion and contraction for the various type assignment systems, this approach of Plotkin's has an obvious appeal in order to analyse the semantics of the present work. In particular, since the type deductions all respect  $\beta$ -contraction, but  $\beta$ -expansion in only certain special situations (as detailed in Chapter 3), Plotkin's work is the appropriate forum for establishing the completeness of these systems with respect to  $\beta$ -contraction. Of course, the expectation is that soundness would also hold for  $\lambda$ -models, and this is indeed true as is demonstrated in this chapter. Thus this section presents a semantics based on the notion of semi-model introduced by Plotkin [56]. Later, a *filter semi-model* suitable for Boolean Reduction types will be introduced, as well as a conventional *term semi-model* after Plotkin [56].

A  $\lambda\beta$ -semi-model of the  $\lambda$ -calculus is a triple,  $\langle D, \bullet, \llbracket \cdot \rrbracket \cdot \rangle$ , where  $D$  is a partial order,  $\bullet: D \times D \rightarrow D$  is a map called *application* and  $\llbracket \cdot \rrbracket \cdot$  is another map which assigns each term  $M \in \Lambda$  to an element  $\llbracket M \rrbracket_\mu \in D$ , for an environment  $\mu: X \rightarrow D$ . The (partial) ordering on  $D$  will be written as  $\leq$  and for two environments  $\mu$  and  $\mu'$  write  $\mu \leq \mu'$  iff  $\forall x \in X. \mu(x) \leq \mu'(x)$ . Write  $\llbracket M \rrbracket_\mu = \llbracket M' \rrbracket_\mu$  iff  $\llbracket M \rrbracket_\mu \leq \llbracket M' \rrbracket_\mu$  and  $\llbracket M' \rrbracket_\mu \leq \llbracket M \rrbracket_\mu$ . Furthermore, the map  $\llbracket \cdot \rrbracket \cdot$  must satisfy, for each  $M \in \Lambda$  and each environment  $\mu: X \rightarrow D$ , the following list of properties (Plotkin [56]):

1.  $\llbracket x \rrbracket_\mu = \mu(x)$ ,
2.  $\llbracket MN \rrbracket_\mu = \llbracket M \rrbracket_\mu \bullet \llbracket N \rrbracket_\mu$ ,
3.  $\llbracket \lambda x. M \rrbracket_\mu \bullet d \leq \llbracket M \rrbracket_{\mu[x:=d]}$ , for every  $d \in D$ ,
4. if  $\llbracket x \rrbracket_\mu = \llbracket x \rrbracket_{\mu'}$  for all  $x \in \text{FV}(M)$ , then  $\llbracket M \rrbracket_\mu = \llbracket M \rrbracket_{\mu'}$ ,
5. if  $y \notin \text{FV}(M)$ , then  $\llbracket M[x := y] \rrbracket_\mu = \llbracket M \rrbracket_{\mu[x:=\mu(y)]}$ , and
6. if  $\forall d \in D. \llbracket M \rrbracket_{\mu[x:=d]} \leq \llbracket N \rrbracket_{\mu[x:=d]}$ , then  $\llbracket \lambda x. M \rrbracket_\mu \leq \llbracket \lambda x. N \rrbracket_\mu$ .

When compared with the conditions for the interpretation function of a  $\lambda$ -model, the above conditions differ in that equality is required in all conditions for the  $\lambda$ -model and condition 5 is a consequence. However, the following more general substitution principle still holds:

**Lemma 4.1.4 (Plotkin [56])**

$$\llbracket M[x := N] \rrbracket_\mu = \llbracket M \rrbracket_{\mu[x:=\llbracket N \rrbracket_\mu]}.$$

Given the usual definition of  $\beta$ -reduction, it is easy to show:

**Theorem 4.1.5 (Plotkin [56])**

If  $M \rightarrow_\beta N$ , then, for any environment  $\mu$ ,  $\llbracket M \rrbracket_\mu \leq \llbracket N \rrbracket_\mu$

Thus semi-models do indeed model the process of reduction (not conversion) and so are a good benchmark upon which to judge the correctness of a predictor of reduction behaviour of terms such as is proposed in this thesis. Plotkin's semi-models allow the type deduction systems to be shown to give *complete* information with respect to reduction.

Plotkin extends the notion of semi-model to  *$\beta\eta$ -semi-models* by adding the following condition to the interpretation map  $\llbracket \cdot \rrbracket$ :

7. if  $x \notin \text{FV}(M)$ , then  $\llbracket \lambda x. Mx \rrbracket_\mu \leq \llbracket M \rrbracket_\mu$ .

**The Term Semi-Model**

The *term semi-model* is  $\mathcal{M} = \langle D, \bullet, \llbracket \cdot \rrbracket \rangle$  where

$$D = \{[M] \mid M \in \Lambda\},$$

$$[M] = \{N \mid M \rightarrow_\beta N \leftarrow_\beta M\},$$

and

$$[M] \bullet [N] = [MN].$$

The ordering for this model is  $[M] \leq [N]$  iff  $M \rightarrow_\beta N$ . Let  $\mathbf{Env}$  denote  $X \rightarrow D$ , then  $\llbracket \cdot \rrbracket$  is defined as before:

$$\begin{aligned} \llbracket \cdot \rrbracket &: \Lambda \rightarrow \mathbf{Env} \rightarrow D \\ \llbracket M \rrbracket_\mu &= [M[x_1 \cdots x_j := N_1 \cdots N_j]], \end{aligned}$$

for  $\mu = [x_1 \cdots x_j := [N_1] \cdots [N_j]]$ . The *canonical* environment is  $\mu_0(x) = [x]$ , as before.

As an example, in the term-model  $[\mathbf{KI}\Omega] = [\mathbf{I}]$  whereas in the term-semi-model  $[\mathbf{KI}\Omega] \leq [\mathbf{I}]$ , but *not*  $[\mathbf{I}] \leq [\mathbf{KI}\Omega]$ .

**Theorem 4.1.6 (Plotkin [56])**

Let  $\langle D, \bullet, \llbracket \cdot \rrbracket \rangle$  be as defined above, then it is a semi- $\lambda$ -model.

Plotkin uses the above definition of term semi-models to establish the following *completeness* theorem:

**Theorem 4.1.7**

$M \rightarrow_\beta N$  if in all semi-models and for every environment  $\mu$ ,  $\llbracket M \rrbracket_\mu \leq \llbracket N \rrbracket_\mu$ .

A similar result is obtained by Plotkin for  $\beta\eta$ -reduction by considering  $\beta\eta$ -term semi-models.

### A Filter Semi-Model of Terms

The idea of a *filter* model is to use the set of type expressions themselves to form the domain of the model or semi-model. However, the set of types cannot be used to construct a filter  $\lambda$ -model as  $\beta$ -convertible terms do not necessarily have the same set of types assignable to them (Chapter 3). Instead, a filter semi- $\lambda$ -model may be constructed for the purpose of providing an alternative validation of the completeness of the system with respect to reduction.

Here, a filter semi-model is only considered for the set of Intersection Reduction Types,  $T_I^\nabla$ .

#### Definition 4.1.8

A *filter* is a subset  $d$  of the set of types,  $T_I^\nabla$  such that

- $\omega \in d$ ,
- $\sigma, \tau \in d$  implies  $\sigma \cap \tau \in d$ ,
- $\sigma \in d$  and  $\sigma \leq \tau$  implies  $\tau \in d$ .

#### Lemma 4.1.9

Let  $A$  be an assumption set, then  $A \vdash_{V \multimap}^I [x := \Rightarrow] x : \sigma$  iff  $\sigma$  is in the filter generated by  $\tau$ , where  $x : \tau \in A$ .<sup>1</sup>

#### Proof

By rules MEET and LEQ and induction on the derivation of  $A \vdash_{V \multimap}^I [x := \Rightarrow] x : \sigma$ .  
□

The powerset of a set  $D$  is written here as  $\mathcal{P}(D)$ . Let  $D = \mathcal{P}(T_I^\nabla)$  where  $\leq$  is taken to be the subset relation. Several different notions of semantic application immediately arise, *viz.*, *strict* application, *constant* application and *ordinary* application. These can be defined as follows: for each arrow expression  $b$  and each  $d, d' \in D$ , define

$$d \bullet_b d' = \{\tau \mid \exists \sigma \in d'. \sigma \vdash_b \tau \in d\}.$$

Note the finer structure which Boolean Reduction Types impose on functions is clearly evidenced.

Let  $\mathcal{F} = \{d \in D \mid d \text{ is a filter}\}$ .

#### Lemma 4.1.10

Let  $d, d' \in \mathcal{F}$  and let  $b$  be an arrow expression. If  $d \bullet_b d' \neq \emptyset$ , then  $d \bullet_b d' \in \mathcal{F}$ .

<sup>1</sup>Recall that assumption sets were restricted to contain at most one entry for each term variable.

**Proof**

We need to show that  $\omega \in d \bullet_b d'$ ,  $\sigma, \tau \in d \bullet_b d'$  implies  $\sigma \cap \tau \in d \bullet_b d'$  and  $\sigma \in d \bullet_b d'$  and  $\sigma \leq \tau$  implies  $\tau \in d \bullet_b d'$ .

1. Let  $\sigma \in d'$  and  $\sigma \mathbin{\text{b}} \tau \in d$ , then  $\sigma \mathbin{\text{b}} \omega \in d$  (since  $d$  is a filter), hence  $\omega \in d \bullet_b d'$ .
2. Suppose  $\sigma, \tau \in d \bullet_b d'$ , then for some types  $\sigma_1$  and  $\sigma_2$ ,  $\sigma_1 \in d'$ ,  $\sigma_1 \mathbin{\text{b}} \sigma \in d$  and  $\sigma_2 \in d'$ ,  $\sigma_2 \mathbin{\text{b}} \tau \in d$ , hence  $\sigma_1 \cap \sigma_2 \in d'$  and  $(\sigma_1 \mathbin{\text{b}} \sigma) \cap (\sigma_2 \mathbin{\text{b}} \tau) \in d$ . Thus,  $(\sigma_1 \mathbin{\text{b}} \sigma) \cap (\sigma_2 \mathbin{\text{b}} \tau) \leq (\sigma_1 \cap \sigma_2) \mathbin{\text{b}} (\sigma \cap \tau) \in d$ , hence  $\sigma \cap \tau \in d \bullet_b d'$ .
3. Let  $\sigma \in d \bullet_b d'$  and  $\sigma \leq \tau$ , then  $\exists \sigma'. \sigma' \in d'$  and  $\sigma' \mathbin{\text{b}} \sigma \in d$ , but  $d$  is a filter, hence  $\sigma' \mathbin{\text{b}} \tau \in d$  which implies  $\tau \in d \bullet_b d'$ .

□

**Definition 4.1.11**

Let  $\mu \in X \rightarrow \mathcal{F}$ , then define  $A_\mu = \bigcup_{x \in X} \{x : \bigcap_{\sigma \in \mu(x)} \sigma\}$ . For each  $M \in \Lambda$ , let  $V_{M,\mu}$  be a variable strong head neededness function such that  $\exists \tau. A_\mu \vdash_{V_{M,\mu}}^I M : \tau$  and  $\nexists \sigma < \tau, V.A_\mu \vdash_V^I M : \sigma$ . The interpretation maps for the filter semi- $\lambda$ -models are then defined to be  $\llbracket M \rrbracket_\mu^\nabla = \{\sigma \in T_I^\nabla \mid A_\mu \vdash_{V_{M,\mu}}^I M : \sigma\}$ , for  $V_{M,\mu}$  as specified above.

Note that for each  $M$  and  $\mu$ ,  $V_{M,\mu}$  may *not* be unique. This is because given a particular type assignment system and set of types, there are three degrees of freedom (modulo  $=_{BA}$ ) in determining  $V$ , namely  $A$ ,  $M$  and  $\tau$ , see Lemma 3.4.10. Of these, the above definition does not uniquely fix  $\tau$ .

**Lemma 4.1.12**

Let  $\mu \in X \rightarrow \mathcal{F}$ ,  $M \in \Lambda$  and  $\tau \in T_I^\nabla$ . If  $\exists V.A_\mu \vdash_V^I M : \tau$ , then  $\exists V_{M,\mu}. A_\mu \vdash_{V_{M,\mu}}^I M : \tau$ .

**Proof**

If  $\nexists \sigma < \tau, V'. A_\mu \vdash_{V'}^I M : \sigma$ , then choose  $V_{M,\mu} = V$ . Otherwise, if  $\exists \sigma < \tau, V'. A_\mu \vdash_{V'}^I M : \sigma$ , then choose  $\sigma$  to be a least such type and set  $V_{M,\mu} = V'$ , finally, use rule LEQ to derive  $A_\mu \vdash_{V_{M,\mu}}^I M : \tau$ . □

**Lemma 4.1.13**

Let  $\mu \in X \rightarrow \mathcal{F}$  and for each  $M \in \Lambda$ , let  $V_{M,\mu}$  be as specified above, then  $\forall M \in \Lambda. \llbracket M \rrbracket_\mu^\nabla \in \mathcal{F}$ .

**Proof**

Rule LEQ ensures that  $\llbracket M \rrbracket_\mu^\nabla$  is closed under  $\leq$  and contains  $\omega$ . Also, rule MEET is applicable as all elements of  $\llbracket M \rrbracket_\mu^\nabla$  are derived using  $V_{M,\mu}$ , and so  $\llbracket M \rrbracket_\mu^\nabla$  is closed under  $\cap$ . □

**Definition 4.1.14**

For each  $\nabla$ , define  $d \bullet_{\nabla} d' = \bigcup_{b \in \nabla} d \bullet_b d'$ .

Note that because the intersection type assignment systems are not closed under  $\beta$ -conversion the following is restricted to  $\lambda$ -semi-models.

**Theorem 4.1.15**

$\langle \mathcal{F}, \bullet_{\nabla}, [\![\cdot]\!]^{\nabla} \rangle$  are semi- $\lambda$ -models.

**Proof**

Clearly,  $(\mathcal{F}, \leq)$  is a partial order (see the definition of a filter). Then all that is required is that we check the six conditions on the map  $[\![\cdot]\!]^{\nabla}$ , for each  $\nabla$ .

1.  $\tau \in [\![x]\!]_{\mu}^{\nabla}$  iff  $A_{\mu} \vdash_{V_{\mu}[x:=\Rightarrow]}^I x : \tau$  which implies  $\tau \in \mu(x)$  by Lemma 4.1.9.  
Note also that  $\tau \in \mu(x)$  implies  $A_{\mu} \vdash_{V_{\mu}[x:=\Rightarrow]}^I x : \tau$ , by definition of  $A_{\mu}$ .
2.  $\tau \in [\![MN]\!]_{\mu}^{\nabla}$   
iff  $\exists V_{MN,\mu}. A_{\mu} \vdash_{V_{MN,\mu}}^I MN : \tau$   
iff  $\exists \sigma, b \in \nabla, V_{M,\mu}, V_{N,\mu}. A_{\mu} \vdash_{V_{M,\mu}}^I M : \sigma \text{ b } \tau$  and  $A_{\mu} \vdash_{V_{N,\mu}}^I N : \sigma$   
{by Lemma 3.4.12 and Lemma 4.1.12}  
iff  $\exists \sigma, b \in \nabla. \sigma \text{ b } \tau \in [\![M]\!]_{\mu}^{\nabla}$  and  $\sigma \in [\![N]\!]_{\mu}^{\nabla}$   
iff  $\exists b \in \nabla. \tau \in [\![M]\!]_{\mu}^{\nabla} \bullet_b [\![N]\!]_{\mu}^{\nabla}$   
iff  $\tau \in \bigcup_{b \in \nabla} [\![M]\!]_{\mu}^{\nabla} \bullet_b [\![N]\!]_{\mu}^{\nabla}$   
iff  $\tau \in [\![M]\!]_{\mu}^{\nabla} \bullet_{\nabla} [\![N]\!]_{\mu}^{\nabla}$ .
3.  $\tau \in [\![\lambda x.M]\!]_{\mu}^{\nabla} \bullet_{\nabla} [\![N]\!]_{\mu}^{\nabla}$   
iff  $\exists b \in \nabla. \tau \in [\![\lambda x.M]\!]_{\mu}^{\nabla} \bullet_b [\![N]\!]_{\mu}^{\nabla}$   
iff  $\exists b \in \nabla, \sigma. \sigma \text{ b } \tau \in [\![\lambda x.M]\!]_{\mu}^{\nabla}$  and  $\sigma \in [\![N]\!]_{\mu}^{\nabla}$   
iff  $\exists b \in \nabla, \sigma, V_{\lambda x.M,\mu}, V_{N,\mu}. A_{\mu} \vdash_{V_{\lambda x.M,\mu}}^I \lambda x.M : \sigma \text{ b } \tau$  and  $A_{\mu} \vdash_{V_{N,\mu}}^I N : \sigma$   
iff  $\exists b \in \nabla, V_{(\lambda x.M)N,\mu}. A_{\mu} \vdash_{V_{(\lambda x.M)N,\mu}}^I (\lambda x.M)N : \tau$   
{by rule APP and Lemma 4.1.12}  
implies  $\exists V_{M[x:=N],\mu}. A_{\mu} \vdash_{V_{(\lambda x.M)N,\mu}}^I M[x := N] : \tau$   
{this last step is what restricts the Theorem to semi- $\lambda$ -models}  
iff  $\tau \in [\![M]\!]_{\mu[x:=d]}^{\nabla}$ , by part 5 and Lemma 4.1.4.
4. By Lemma 3.4.19.
5. Immediate from the variable convention for  $\lambda$ -terms.
6. Immediate from Lemma 3.4.15.

□

## 4.2 Models of Boolean Reduction Types

The following definitions provide semantic analogues of the syntactic concepts of *strong head neededness* and *irrelevance*, both introduced in Chapter 2.

For strong head neededness the corresponding semantic notions can profitably be expressed as follows.

### Definition 4.2.1

For  $\lambda$ -semi-models, write  $\llbracket M \rrbracket_\mu$  is strongly head needed in  $\llbracket N \rrbracket_\mu$  if

$$\llbracket N \rrbracket_\mu \leq \llbracket \lambda x_1 \dots x_m. MN_1 \dots N_n \rrbracket_\mu,$$

where  $m, n \geq 0$ . In the case of  $\lambda$ -models, the condition is:

$$\llbracket N \rrbracket_\mu = \llbracket \lambda x_1 \dots x_m. MN_1 \dots N_n \rrbracket_\mu,$$

where  $m, n \geq 0$ .

For irrelevance the situation is equally simple as by the Church Rosser result for  $\beta$ -reduction this reduction schema may be used to analyse convertibility (both  $\beta$ -conversion and those notions of convertibility induced by the respective models). The following definition suits both  $\lambda$ -models and  $\lambda$ -semi-models.

### Definition 4.2.2

Write  $d_0$  is *irrelevant* in  $d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n$  if for all  $d'_0$ ,

$$d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n = d \bullet d'_0 \bullet d_1 \bullet \dots \bullet d_n.$$

Note that by the Church-Rosser result for  $\beta$  and Theorems 4.1.2 and 4.1.5,  $d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n = d \bullet d'_0 \bullet d_1 \bullet \dots \bullet d_n$  implies  $\exists d'. d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n \leq d' \geq d \bullet d'_0 \bullet d_1 \bullet \dots \bullet d_n$ .

### Proposition 4.2.3

$\llbracket M \rrbracket_\mu$  strongly head needed in  $\llbracket N \rrbracket_\mu$  iff  $M$  strongly head needed in  $N$ .

#### Proof

Consider the  $\lambda$ -semi-models. Firstly, suppose  $\llbracket M \rrbracket_\mu$  strongly head needed in  $\llbracket N \rrbracket_\mu$ . Then

$$\llbracket N \rrbracket_\mu \leq \llbracket \lambda x_1 \dots x_m. MN_1 \dots N_n \rrbracket_\mu,$$

hence  $N \rightarrow_\beta \lambda x_1 \dots x_m. MN_1 \dots N_n$ . The result follows by Corollary 2.4.8.

Secondly, suppose  $M$  strongly head needed in  $N$ . Then the result is immediate by Theorem 4.1.5 and by choice of the head reduction of  $N$ .

The case for  $\lambda$ -models is similar, using Theorem 4.1.2.  $\square$

### Proposition 4.2.4

$\llbracket M \rrbracket_\mu$  is irrelevant in  $\llbracket NMN_1 \dots N_n \rrbracket_\mu$  iff  $M$  is irrelevant in  $NMN_1 \dots N_n$ .

**Proof**

Use Theorem 4.1.7 (Theorem 4.1.3), then Proposition 2.3.3 and lastly Theorem 4.1.5 (Theorem 4.1.2).  $\square$

**4.2.1 What is a Type Interpretation?**

A type interpretation is a pair

$$\mathcal{T}_y = \langle \text{Ty}, \llbracket \cdot \rrbracket \cdot \rangle,$$

where  $\text{Ty}$  is a set of subsets of the domain of the chosen (semi-) model of terms which satisfies certain closure properties (see below) and  $\llbracket \cdot \rrbracket$  is an interpretation function from types and type environments to elements of  $\text{Ty}$  which satisfies certain conditions (see below).

If  $\text{Ty}$  is a partial order, then elements of  $\text{Ty}$  will be required to be upper-closed. This ensures that properties are inherited upwards, a condition which is needed by the interpretation of any type system closed under  $\beta$ -contraction, as are the three case studies of the last chapter.

The interpretation of types is given modulo the interpretation of type variables, and the letter  $\nu$  is used to range over type environments. The interpretation function must satisfy the following conditions:

(Arrow 1) let  $d \in \llbracket \sigma \text{ b } \tau \rrbracket_\nu$ , then  $d \bullet \llbracket \sigma \rrbracket_\nu \subseteq \llbracket \tau \rrbracket_\nu$ ,

(Arrow 2) suppose  $d \in \llbracket \sigma \rrbracket_\nu$  implies  $\llbracket M \rrbracket_{\rho[x:=d]} \in \llbracket \tau \rrbracket_\nu$ , then  $\llbracket \lambda x.M \rrbracket_\rho \in \llbracket \sigma \text{ b } \tau \rrbracket_\nu$ , for some  $b \in \nabla$ , and

(Arrow 3) let  $d \in \llbracket \sigma_0 \text{ b}_0 \sigma_1 \text{ b}_1 \dots \sigma_n \text{ b}_n \alpha \rrbracket_\nu$ , then  $\forall d_i \in \llbracket \sigma_i \rrbracket_\nu$

1. if  $\text{b}_0 = \text{BA} \Rightarrow$ , then  $d_0$  is strongly head needed in  $d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n$ , and
2. if  $\text{b}_0 = \text{BA} \nrightarrow$ , then  $d_0$  is irrelevant in  $d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n$ .

**Definition 4.2.5**

Let  $\rho$  and  $\nu$  be as above. The notion of *semantic satisfaction* is defined in the following manner:

1. a statement  $M : \sigma$  is *satisfied* by  $\rho, \nu$ , model  $\mathcal{M}$  and type interpretation  $\mathcal{T}$ , notation  $\rho, \nu, \mathcal{M}, \mathcal{T}_y \models M : \sigma$ , if  $\llbracket M \rrbracket_\rho \in \llbracket \sigma \rrbracket_\nu$ ,
2. an assumption set  $A$  is *satisfied* by  $\rho, \nu$ , model  $\mathcal{M}$  and type interpretation  $\mathcal{T}$ , notation  $\rho, \nu, \mathcal{M}, \mathcal{T}_y \models A$ , if for each  $x : \sigma \in A$ ,  $\rho, \nu, \mathcal{M}, \mathcal{T}_y \models x : \sigma$ , and
3. lastly, write  $A \models_\nu M : \sigma$  if  $\rho, \mathcal{M}, \mathcal{T}_y \models A$  implies  $\rho, \nu, \mathcal{M}, \mathcal{T}_y \models M : \sigma$  and  $\forall x \in X. \rho, \nu, \mathcal{M}, \mathcal{T}_y \models \lambda x.M : A(x) \vee (x) \sigma$  to mean that whenever  $\rho, \nu$ , model  $\mathcal{M}$  and type interpretation  $\mathcal{T}$  satisfy  $A$ , then they also satisfy both  $M : \sigma$  and  $\forall x \in X. \lambda x.M : A(x) \vee (x) \sigma$ .

$$\begin{aligned}
& \mathcal{S}[\cdot] : T_C^\nabla \rightarrow \mathbf{TEnv} \rightarrow \mathcal{P}(D) \\
& \mathcal{S}[\sigma]_\nu = \bigcap_{\sigma_i \in G_C(\sigma)} \mathcal{G}[\sigma_i]_\nu \\
\\
& \mathcal{G}[\cdot] : T_C^{\Delta_g} \rightarrow \mathbf{TEnv} \rightarrow \mathcal{P}(D) \\
& \mathcal{G}[\alpha]_\nu = \nu(\alpha) \\
& \mathcal{G}[\sigma_0 \Rightarrow \sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu \\
& \quad = \{d \in D \mid \forall d_i \in \mathcal{G}[\sigma_i]_\nu, 0 \leq i \leq n. \\
& \quad \quad d_0 \text{ strongly head needed in } d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n; \\
& \quad \quad d \bullet d_0 \in \mathcal{G}[\sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu\} \\
& \mathcal{G}[\sigma_0 \nrightarrow \sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu \\
& \quad = \{d \in D \mid \forall d_i \in \mathcal{G}[\sigma_i]_\nu, 0 \leq i \leq n. \\
& \quad \quad d_0 \text{ irrelevant in } d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n; \\
& \quad \quad d \bullet d_0 \in \mathcal{G}[\sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu\}
\end{aligned}$$

Figure 11: The Semantics of Simple Boolean Reduction Types

Let  $D$  be the domain of a model or semi-model of the lambda-calculus, then write  $\mathbf{TEnv}$  for  $\tau_\nu \rightarrow \mathcal{P}(D)$ , the type of type variable environments.

### 4.2.2 The Semantics of Simple Reduction Types

Let the set of types be  $T_C^\nabla$ .

#### Definition 4.2.6

Write  $\sigma \sqsubseteq_C \tau$  if there is a substitution,  $R$ , of arrow expressions ( $\in \nabla$ ) for arrow variables homomorphically extendible such that  $\tau = R(\sigma)$ . Let  $G_C(\sigma) = \{\tau \mid \sigma \sqsubseteq_C \tau; \tau \in T_C^g\}$ .

Figure 11 contains the semantics of all forms of Simple Boolean Reduction Types. (Not all cases of this definition are applicable to the interpretation of sets of types such as  $T_C^{\{\Rightarrow\}}$  and  $T_C^{\{\nrightarrow\}}$ ).

The definition defines that the meaning of an arrow variable is the (universal) quantification over all its instances (effectively  $\Rightarrow$  and  $\nrightarrow$ )—hence in all the type assignment systems of Chapter 3 the use of arrow variables is simply a mechanism for introducing intersection types, albeit often in a greatly restricted form.

Consider the term  $\lambda x f.f x$ , which can be assigned the type

$$\alpha \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \beta.$$



(Consider the case when the identity function is substituted for  $f$ ). However, for any term  $N$ ,  $\lambda x f.f x$  does *not* strongly head need its argument as  $(\lambda x f.f x) N \rightarrow_{\beta} \lambda f.f N$ , which is in head normal form. Thus it is not sufficient to determine the strong head neededness of an argument on a part of the type of a function—instead the *whole* type must be taken into account, that is, all arguments of the correct type must be given to the function (in the specified order) in order to make this determination. Thus, since  $(\lambda x f.f x) N (\lambda y.y) \rightarrow_{\beta} N$ ,  $\alpha \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \beta$  is a correct type for  $\lambda x f.f x$ . (Of course, we should allow *any* argument of type  $\alpha \Rightarrow \beta$  in making this determination, as is done in Figure 11).

Note that we would also wish  $\lambda x f.f x$  to have the type

$$\alpha \multimap (\alpha \multimap \beta) \Rightarrow \beta.$$

And by inspection of Figure 11 this is indeed the case. This duality of behaviour for  $\lambda x f.f x$  precisely expresses the context-sensitivity of strong head neededness information. Thus a better type assignment for  $\lambda x f.f x$  is

$$\alpha \rightarrow_i (\alpha \rightarrow_i \beta) \Rightarrow \beta,$$

for some arrow variable  $\rightarrow_i$ . Again this is true by inspection of Figure 11, where

$$\begin{aligned} \mathcal{S}[\alpha \rightarrow_i (\alpha \rightarrow_i \beta) \Rightarrow \beta]_{\nu} \\ = \mathcal{G}[\alpha \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \beta]_{\nu} \cap \mathcal{G}[\alpha \multimap (\alpha \multimap \beta) \Rightarrow \beta]_{\nu}. \end{aligned}$$

Let  $\text{Ty} = \mathcal{P}(D)$ , where  $D$  is the domain of a (semi-) model of terms, and  $\mathcal{S}[\cdot]_{\nu}$  be as defined in Figure 11, then the following Theorem holds immediately.

#### Theorem 4.2.7

$\langle \text{Ty}, \mathcal{S}[\cdot]_{\nu} \rangle$  is a type interpretation.

#### Proof

Clearly,  $\text{Ty}$  is upper-closed for semi-models by Theorem 3.2.21 (models do not have a partial order defined on them).

By considering both of the cases for functional types it is clear that condition **Arrow 1** on the interpretation function is satisfied.

Consider **Arrow 2**. Suppose that for all  $d \in \llbracket \sigma \rrbracket_{\nu}$  we have  $\llbracket M \rrbracket_{\mu[x:=d]} \in \llbracket \tau \rrbracket_{\nu}$ . Then in particular  $\llbracket M \rrbracket_{\mu} \in \llbracket \tau \rrbracket_{\nu}$  (just choose  $d = \llbracket x \rrbracket_{\mu}$ ), hence  $\llbracket \lambda x.M \rrbracket_{\mu} \in \llbracket \sigma \multimap \tau \rrbracket_{\nu}$ , for some arrow expression  $b$ .

As for **Arrow 1**, it is clear that condition **Arrow 3** is satisfied by definition in the respective cases of the definition of  $(\mathcal{S}[\cdot]_{\nu})$ .

□

$$\begin{array}{l} \mathcal{L}[\cdot]: T_L^\nabla \rightarrow \mathbf{TEnv} \rightarrow \mathcal{P}(D) \\ \mathcal{L}[\sigma]_\nu = \bigcap_{\sigma \preceq \tau} \mathcal{S}[\tau]_\nu \end{array}$$

Figure 12: The Semantics of LET-Polymorphic Reduction Types

### 4.2.3 The Semantics of LET-Polymorphic Reduction Types

Let the set of types be  $T_L^\nabla$ .

Figure 12 contains the semantics of all forms of LET-Polymorphic Reduction Types. (Not all cases of this definition are applicable to the interpretation of sets of types such as  $T_L^{\{\Rightarrow\}}$  and  $T_L^{\{\Leftarrow\}}$ ).

Note that the restricted form of quantification in  $T_L^\nabla$  makes this naive interpretation reasonable (types such as  $(\forall \alpha. \alpha) \vdash \sigma$  are not admitted). For a full second-order system the domain would need to be restricted to sets of *ideals* (MacQueen et al [44]) or an approach such as that of Bruce and Meyer [7] would need to be followed. (See also Mitchell [49]).

#### Lemma 4.2.8

$\sigma \preceq \tau$  implies  $\mathcal{L}[\sigma]_\nu \subseteq \mathcal{L}[\tau]_\nu$ .

#### Proof

Immediate by definition of  $\mathcal{L}[\sigma]_\nu$ .  $\square$

Let  $\mathbf{Ty} = \mathcal{P}(D)$ , where  $D$  is the domain of a (semi-) model of terms, and  $\mathcal{L}[\cdot]$  be as defined in Figure 12, then the following Theorem clearly follows.

#### Theorem 4.2.9

$\langle \mathbf{Ty}, \mathcal{L}[\cdot] \rangle$  is a type interpretation.

### 4.2.4 The Semantics of Intersection Reduction Types

Let the sets of types be  $T_I^\nabla$ .

#### Definition 4.2.10

Write  $\sigma \preceq_I \tau$  if there is a substitution,  $R$ , of arrow expressions ( $\in \nabla$ ) for arrow variables homomorphically extendible such that  $\tau = R(\sigma)$ . Let  $G_I(\sigma) = \{\tau \mid \sigma \preceq_I \tau; \tau \in T_I^{\Delta_g}\}$ .

Figure 13 contains the semantics of all forms of Intersection Boolean Reduction Types. Note that this covers all cases as  $\rho \vdash (\sigma \cap \tau) = (\rho \vdash \sigma) \cap (\rho \vdash \tau)$ .

$$\begin{aligned}
& \mathcal{I}[\cdot] : T_I^\nabla \rightarrow \mathbf{TEnv} \rightarrow \mathcal{P}(D) \\
& \mathcal{I}[\sigma]_\nu = \bigcap_{\sigma_i \in G_I(\sigma)} \mathcal{G}_I[\sigma_i]_\nu \\
\\
& \mathcal{G}_I[\cdot] : T_C^{\Delta_g} \rightarrow \mathbf{TEnv} \rightarrow \mathcal{P}(D) \\
& \mathcal{G}_I[\omega]_\nu = D \\
& \mathcal{G}_I[\sigma \cap \tau]_\nu = \mathcal{G}_I[\sigma]_\nu \cap \mathcal{G}_I[\tau]_\nu \\
& \mathcal{G}_I[\alpha]_\nu = \nu(\alpha) \\
& \mathcal{G}_I[\sigma_0 \Rightarrow \sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu \\
& \quad = \{d \in D \mid \forall d_i \in \mathcal{G}_I[\sigma_i]_\nu, 0 \leq i \leq n. \\
& \quad \quad d_0 \text{ strongly head needed in } d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n; \\
& \quad \quad d \bullet d_0 \in \mathcal{G}_I[\sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu\} \\
& \mathcal{G}_I[\sigma_0 \nrightarrow \sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu \\
& \quad = \{d \in D \mid \forall d_i \in \mathcal{G}_I[\sigma_i]_\nu, 0 \leq i \leq n. \\
& \quad \quad d_0 \text{ irrelevant in } d \bullet d_0 \bullet d_1 \bullet \dots \bullet d_n; \\
& \quad \quad d \bullet d_0 \in \mathcal{G}_I[\sigma_1 \mathbf{b}_1 \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha]_\nu\}
\end{aligned}$$

Figure 13: The Semantics of Intersection Boolean Reduction Types

(Note that this semantics cannot be defined in terms of the  $\mathcal{S}[\cdot]$  semantics since types of the form  $(\rho \cap \sigma) \mathbf{b} \tau$  are permitted).

**Lemma 4.2.11**

$\sigma \leq \tau$  implies  $\forall \nu. \mathcal{I}[\sigma]_\nu \subseteq \mathcal{I}[\tau]_\nu$ .

**Proof**

By straightforward induction over the definition of  $\leq$ .  $\square$

Let  $\text{Ty} = \mathcal{P}(D)$ , where  $D$  is the domain of a (semi-) model of terms, and  $\mathcal{I}[\cdot]$  be as defined in Figure 13, then it is clear that the following Theorem holds.

**Theorem 4.2.12**

$\langle \text{Ty}, \mathcal{I}[\cdot] \rangle$  is a type interpretation.

## 4.3 The Intensional Applicative Behaviour of $\lambda$ -terms

This section explores some basic properties that the above semantics possesses, while laying the ground work for the major results of this chapter, which are

presented in the following sections. The major result of this section is the establishment of a connection between the interpretation of application of terms and the interpretation of functional types. This is summarised by Theorem 4.3.8 at the end of this section which encapsulates this behaviour in a simple Boolean formula. From this result will flow the soundness of the APP rules of the various type deduction logics. The lemmata which follow are all required in order to establish Theorem 4.3.8.

The results contained in this subsection are generic in the sense that they are applicable to all the sets of types that have been defined. (Remember that  $T_I^\nabla$  is considered modulo  $=$  and that  $\rho \mathbf{b} (\sigma \cap \tau) = (\rho \mathbf{b} \sigma) \cap (\rho \mathbf{b} \tau)$ ). They are also dealt with in a reasonably model independent fashion, except where noted.

The following Lemma is very similar to the results of Hindley [29] as part of his proofs establishing soundness (see for example his Soundness Theorem for quotient-set semantics).

#### Lemma 4.3.1

Let  $\llbracket M \rrbracket_\mu \in \mathcal{S}[\tau]_\nu$  and  $\llbracket x \rrbracket_\mu \in \mathcal{S}[\sigma]_\nu$ , then  $\exists \mathbf{b} . \llbracket \lambda x . M \rrbracket_\mu \in \mathcal{S}[\sigma \mathbf{b} \tau]_\nu$ .

#### Proof

Consider the case of semi-models. For all  $d \in \mathcal{S}[\sigma]_\nu$ ,  $\llbracket \lambda x . M \rrbracket_\mu \bullet d \leq \llbracket M \rrbracket_{\mu[x:=d]} \in \mathcal{S}[\tau]_\nu$ , hence  $\llbracket \lambda x . M \rrbracket_\mu$  maps  $\mathcal{S}[\sigma]_\nu$  into  $\mathcal{S}[\tau]_\nu$ , as required.

This lemma follows in a similar fashion for models (replace  $\leq$  by  $=$ ).  $\square$

#### Lemma 4.3.2

For any environments  $\mu$  and  $\nu$ , term  $M \in \Lambda$  and term variable  $x$ , if  $\llbracket M \rrbracket_\mu \in \llbracket \tau \rrbracket_\nu$ ,  $\llbracket x \rrbracket_\mu \in \llbracket \sigma \rrbracket_\nu$  and  $x \notin \text{FV}(M)$ , then  $\llbracket \lambda x . M \rrbracket_\mu \in \llbracket \sigma \multimap \tau \rrbracket_\nu$ .

#### Proof

$x \notin \text{FV}(M)$  implies  $\forall N, N'. (\lambda x . M)N = M = (\lambda x . M)N'$ , i.e.,  $\forall N, N'. C[N] = C[N']$ , where  $C[] = (\lambda x . M)[\ ]$ . Therefore, by Proposition 2.3.3,  $\forall P. P$  is irrelevant in  $C[P]$ . By Proposition 4.2.4,  $\forall \mu, P. \llbracket P \rrbracket_\mu$  is irrelevant in  $\llbracket C[P] \rrbracket_\mu$ .

The result now holds by combining Lemma 4.3.1 with the above fact, i.e.,  $\llbracket \lambda x . M \rrbracket_\mu \in \llbracket \sigma \multimap \tau \rrbracket_\nu$ .  $\square$

As already explained in the previous section, it is necessary that all the arguments specified by the type of a function be given to it in order to determine whether or not it requires a particular argument. The following syntactic definition is therefore useful:

#### Definition 4.3.3

The *argument types* of a type  $\sigma_1 \mathbf{b} \sigma_2 \dots \sigma_n \mathbf{b}_n \alpha$ ,  $n \geq 0$ , are  $\sigma_1, \sigma_2, \dots, \sigma_n$ .

#### Lemma 4.3.4

Suppose  $\llbracket \lambda x . N_1 \rrbracket_\mu \in \llbracket \sigma \Rightarrow \tau' \mathbf{b} \tau \rrbracket_\nu$  and  $\llbracket N_2 \rrbracket_\mu \in \llbracket \tau' \rrbracket_\nu$ , then  $\llbracket \lambda x . N_1 N_2 \rrbracket_\mu \in \llbracket \sigma \Rightarrow \tau \rrbracket_\nu$ .

**Proof**

Let the argument types of  $\tau$  be  $\sigma_1, \sigma_2, \dots, \sigma_n$ , let  $P_i \in d_i \in \llbracket \sigma_i \rrbracket_\nu$ , for all  $i \in \{1, \dots, n\}$  and  $\llbracket N \rrbracket_\mu = d_\sigma \in \llbracket \sigma \rrbracket_\nu$ . Let  $C[] = []P_1 \dots P_n$ , now by Proposition 4.2.3  $N$  strongly head needed in  $C[(\lambda x.N_1)NN_2]$ , but

$$\begin{aligned} & C[(\lambda x.N_1)NN_2] \\ &= C[(\lambda xy.N_1y)NN_2] \text{ \{for } x \neq y \text{ and } y \notin \text{FV}(N_1N)\} } \\ &= C[(\lambda yx.N_1y)N_2N] \\ &= C[(\lambda x.N_1N_2)N]. \end{aligned}$$

Thus by Proposition 2.4.10  $N$  strongly head needed in  $C[(\lambda x.N_1N_2)N]$ . Hence, by Proposition 4.2.3,  $d_\sigma$  strongly head needed in  $\llbracket \lambda x.N_1N_2 \rrbracket_\mu \bullet d_\sigma \bullet d_1 \bullet \dots \bullet d_n$ .

The result follows from the above and Lemma 4.3.1.  $\square$

**Lemma 4.3.5**

Suppose

$$\llbracket \lambda x.N_1 \rrbracket_\mu \in \llbracket \sigma \multimap \tau' \Rightarrow \tau \rrbracket_\nu$$

and

$$\llbracket \lambda x.N_2 \rrbracket_\mu \in \llbracket \sigma \Rightarrow \tau' \rrbracket_\nu,$$

then  $\llbracket \lambda x.N_1N_2 \rrbracket_\mu \in \llbracket \sigma \Rightarrow \tau \rrbracket_\nu$ .

**Proof**

Let the argument types of  $\tau$  be  $\sigma_1, \sigma_2, \dots, \sigma_n$ , let  $P_i \in d_i \in \llbracket \sigma_i \rrbracket_\nu$ , for all  $i \in \{1, \dots, n\}$ ,  $\llbracket N \rrbracket_\mu = d_\sigma \in \llbracket \sigma \rrbracket_\nu$  and let  $C[] = []P_1 \dots P_n$ . Now by Proposition 4.2.3 we can reason as follows.  $C[(\lambda x.N_1N_2)N] = C[(\lambda x.N_1)N((\lambda x.N_2)N)]$  (without contracting  $N$ ), now since  $C[(\lambda x.N_1)N((\lambda x.N_2)N)]$  strongly head needs  $(\lambda x.N_2)N$ ,  $C[(\lambda x.N_1)N((\lambda x.N_2)N)] \rightarrow_\beta C'[(\lambda x.N_2)N]$ , where  $C'[]$  is a head context.<sup>2</sup> Similarly,  $C'[(\lambda x.N_2)N] \rightarrow_\beta C''[N]$ , where  $C''[]$  is a head context, since  $N$  is strongly head needed in  $C'[(\lambda x.N_2)N]$  (because  $C'[]$  is a head context).

The result can now be seen to hold by Corollary 2.4.8 followed by Proposition 2.4.10 and then Proposition 4.2.3 and finally by combining this with Lemma 4.3.1.  $\square$

**Lemma 4.3.6**

Suppose  $\llbracket \lambda x.N_1 \rrbracket_\mu \in \llbracket \sigma \multimap \tau' \multimap \tau \rrbracket_\nu$  and  $\llbracket N_2 \rrbracket_\mu \in \llbracket \tau' \rrbracket_\nu$ , then  $\llbracket \lambda x.N_1N_2 \rrbracket_\mu \in \llbracket \sigma \multimap \tau \rrbracket_\nu$ .

**Proof**

Let the argument types of  $\tau$  be  $\sigma_1, \sigma_2, \dots, \sigma_n$ , let  $P_i \in d_i \in \llbracket \sigma_i \rrbracket_\nu$ , for all

<sup>2</sup>As was defined in Chapter 1, a *head context* is any context of the form  $\lambda x_1 \dots x_m. []M_1 \dots M_n$ , where  $m, n \geq 0$ .

$i \in \{1, \dots, n\}$ ,  $\llbracket N \rrbracket_\mu = d_\sigma \in \llbracket \sigma \rrbracket_\nu$  and let  $C[] = []P_1 \dots P_n$ .

$$\begin{aligned}
 & C[(\lambda x.N_1 N_2)N] \\
 &= C[(\lambda x.N_1)N((\lambda x.N_2)N)] \\
 & \quad \{ \text{without contracting } N \} \\
 &= C[(\lambda x.N_1)N'((\lambda x.N_2)N')] \\
 & \quad \{ \text{since } N \text{ and } (\lambda x.N_2)N \text{ are irrelevant in } C[(\lambda x.N_1)N((\lambda x.N_2)N)] \} \\
 &= C[(\lambda x.N_1 N_2)N'].
 \end{aligned}$$

The result now holds by Proposition 2.3.3 and Proposition 4.2.4 and finally by combining this with Lemma 4.3.1.  $\square$

#### Lemma 4.3.7

Suppose

$$\llbracket \lambda x.N_1 \rrbracket_\mu \in \llbracket \sigma \multimap \tau' \Rightarrow \tau \rrbracket_\nu$$

and  $\llbracket \lambda x.N_2 \rrbracket_\mu \in \llbracket \sigma \multimap \tau' \rrbracket_\nu$ , then

$$\llbracket \lambda x.N_1 N_2 \rrbracket_\mu \in \llbracket \sigma \multimap \tau \rrbracket_\nu.$$

#### Proof

Let the argument types of  $\tau$  be  $\sigma_1, \sigma_2, \dots, \sigma_n$ , let  $P_i \in d_i \in \llbracket \sigma_i \rrbracket_\nu$ , for all  $i \in \{1, \dots, n\}$ ,  $\llbracket N \rrbracket_\mu = d_\sigma \in \llbracket \sigma \rrbracket_\nu$  and let  $C[] = []P_1 \dots P_n$ .

$$\begin{aligned}
 & C[(\lambda x.N_1 N_2)N] \\
 &= C[(\lambda x.N_1)N((\lambda x.N_2)N)] \\
 & \quad \{ \text{without contracting } N \} \\
 &= C[(\lambda x.N_1)N'((\lambda x.N_2)N)] \\
 & \quad \{ \text{since } N \text{ is irrelevant in } C[(\lambda x.N_1)N((\lambda x.N_2)N)] \} \\
 &= C'[(\lambda x.N_2)N] \\
 & \quad \{ \text{where } C'[] \text{ is a head context, and since } (\lambda x.N_2)N \} \\
 & \quad \{ \text{is strongly head needed in } C[(\lambda x.N_1)N'((\lambda x.N_2)N)] \} \\
 &= C'[(\lambda x.N_2)N'] \\
 & \quad \{ \text{since } N \text{ is irrelevant in } C'[(\lambda x.N_2)N] \} \\
 &= C[(\lambda x.N_1)N'((\lambda x.N_2)N')] \\
 &= C[(\lambda x.N_1 N_2)N'].
 \end{aligned}$$

The result now holds by Proposition 2.3.3 and Proposition 4.2.4 and finally by combining this with Lemma 4.3.1.  $\square$

The intensional applicative behaviour of  $\lambda$ -terms is summarised as follows:

#### Theorem 4.3.8

Suppose

$$\llbracket \lambda x.N_1 \rrbracket_\mu \in \llbracket \sigma b_1 \tau' b_2 \tau \rrbracket_\nu$$

and

$$\llbracket \lambda x. N_2 \rrbracket_\mu \in \llbracket \sigma b_3 \tau' \rrbracket_\nu,$$

then

$$\llbracket \lambda x. N_1 N_2 \rrbracket_\mu \in \llbracket \sigma (b_1 \vee (b_2 \wedge b_3)) \tau \rrbracket_\nu.$$

**Proof**

By Lemmas 4.3.4, 4.3.5, 4.3.6 and 4.3.7. (If  $b_1 =_{BA} \Rightarrow$ , then use Lemma 4.3.4, else if  $b_2 =_{BA} \rightarrow$ , then use Lemma 4.3.6, else if  $b_3 =_{BA} \Rightarrow$ , then use Lemma 4.3.5 else use Lemma 4.3.7. This chain of reasoning is summarised by the above Boolean formula.)  $\square$

## 4.4 Type Assignment is Sound

In this section the soundness of each of the type assignment systems of Chapter 3 is proved. The results in this section apply to both  $\lambda$ -models and semi- $\lambda$ -models and are independent of the selection of a particular concrete instance of these models (for example, term  $\lambda$ -model or term semi- $\lambda$ -model). This independence is not an uncommon situation, see, for example, Hindley [29] or Plotkin [56].

### 4.4.1 Soundness of $\vdash^C$

Let the sets of types be  $T_C^\nabla$ . Establishing the soundness of these type assignment systems is now straightforward, as follows:

**Theorem 4.4.1 (Soundness)**

$A \vdash_V^C M : \tau$  implies  $A \models_V M : \tau$ .

**Proof**

By induction on the derivation  $A \vdash_V^C M : \tau$ .

**VAR** By Lemma 4.3.2,  $\forall y \in X. y \neq x$  and  $y : \sigma \in A$  implies  $\llbracket \lambda y. x \rrbracket_\mu \in \mathcal{S}[\sigma \rightarrow \tau]_\nu$ . Also,  $\llbracket \lambda x. x \rrbracket_\mu \in \mathcal{S}[\tau \Rightarrow \tau]_\nu$ . So,  $A \cup \{x : \tau\} \vdash_{V \rightarrow [x := \Rightarrow]}^C x : \tau$  implies  $A \cup \{x : \tau\} \models_{V \rightarrow [x := \Rightarrow]} x : \tau$ , as required.

**APP** By the induction hypotheses:

- $A \models_{V_1} N_1 : \sigma b \tau$ , and
- $A \models_{V_2} N_2 : \sigma$ .

The result follows immediately from Theorem 4.3.8 and the definition of semantic application for terms ( $\bullet$ ).

**ABS** By the induction hypothesis,  $A_x \cup \{x : \sigma\} \models_{V[x:=b]} N : \tau$ . Then it is immediate by definition of  $\models$  and using Lemma 4.3.2 that  $A \models_{V[x:=a]} \lambda x.N : \sigma \multimap \tau$ , as required.

□

#### 4.4.2 Soundness of $\vdash^L$

Let the sets of types be  $T_L^\nabla$ .

##### Theorem 4.4.2 (Soundness)

$A \vdash_V^L M : \tau$  implies  $A \models_V M : \tau$ .

##### Proof

The interesting cases are when the deduction of  $A \vdash_V^L M : \tau$  ends in a use of rule VAR or rule APP- $\forall$ .

For rule VAR, the result follows as for the Curry-style system and by Lemma 4.2.8.

For rule APP- $\forall$ , the result follows by the induction hypotheses and Theorem 4.3.8. □

#### 4.4.3 Soundness of $\vdash^I$

Let the sets of types be  $T_I^\nabla$ .

##### Theorem 4.4.3 (Soundness)

$A \vdash_V^I M : \tau$  implies  $A \models_V M : \tau$ .

##### Proof

By induction on the derivation of  $A \vdash_V^I M : \tau$ . Consider the last step in this derivation: if this last step is one of VAR, APP or ABS, then the result follows in an identical manner to the soundness of the Curry-style system for this case.

For the case of rule MEET, the result follows immediately by the induction hypotheses and property of  $\cap$ .

For rule LEQ, the result follows by induction and Lemma 4.2.11. □

### 4.5 Type Assignment is Complete

Type assignment is *not* complete with respect to  $\beta$ -conversion. This is because the type assignment systems examined here are not closed under  $\beta$ -conversion. This situation can be easily remedied by postulating a rule to close type assignment of terms under this relation. (After the manner of Hindley [29]). However, I have chosen not to do so and instead follow the method of Plotkin



and only validate the completeness of the type assignment systems with respect to  $\beta$ -contraction. (Since the primary interest of this thesis is in reduction and types).

### 4.5.1 Completeness of $\vdash^C$

Let the set of types be  $T_C^\nabla$ . Define  $\mu_0(x) = [x]$ , then clearly  $\llbracket M \rrbracket_{\mu_0} = [M]$ .

#### Lemma 4.5.1

Let  $A$  be an assumption set and for each  $M \in \Lambda$  if a deduction of a type for  $M$  under assumptions  $A$  exists, then for *each such*  $M$  let  $V_M$  be a strong head neededness function such that  $\exists \sigma. A \vdash_{V_M}^C M : \sigma$ . Let  $\mathcal{A}$  be the assumption set which extends  $A$  such that  $A \subseteq \mathcal{A}$  and such that each type  $\sigma$  is assigned to an infinity of term variables such that no term variable appears more than once in  $\mathcal{A}$ . Consider the term-semi-model, then there is a  $V_M$  such that if  $\nu_0(\alpha) = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^C M : \alpha\}$ , then  $\mathcal{S}[\sigma]_{\nu_0} = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^C M : \sigma\}$ .

#### Proof

By induction on  $\sigma$ .

$\sigma \equiv \alpha$  Immediate by definition of  $\nu_0$ .

$\sigma \equiv \sigma_0 \Rightarrow \sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha$  In the first case,  $\mathcal{A} \vdash_{V_M}^C M : \sigma$  implies  $\mathcal{A} \models_{V_M} M : \sigma$ , hence  $\llbracket M \rrbracket_{\mu_0} \in \mathcal{S}[\sigma]_{\nu_0}$ , which implies  $[M] \in \mathcal{S}[\sigma]_{\nu_0}$ . Thus,  $\{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^C M : \sigma\} \subseteq \mathcal{S}[\sigma]_{\nu_0}$ .

In the second case,  $\llbracket M \rrbracket_{\mu_0} \in \mathcal{S}[\sigma]_{\nu_0}$ . Hence, for all  $\llbracket N_i \rrbracket_{\mu_0} \in \mathcal{S}[\sigma_i]_{\nu_0}$  we have  $\llbracket N_0 \rrbracket_{\mu_0}$  strongly head needed in  $\llbracket M \rrbracket_{\mu_0} \bullet \llbracket N_0 \rrbracket_{\mu_0} \bullet \dots \bullet \llbracket N_n \rrbracket_{\mu_0}$  and  $\llbracket M \rrbracket_{\mu_0} \bullet \llbracket N_0 \rrbracket_{\mu_0} \in \mathcal{S}[\sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha]_{\nu_0}$ . By the induction hypotheses

$$\exists V_{MN_0}. \mathcal{A} \vdash_{V_{MN_0}}^C MN_0 : \sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha$$

and

$$\exists V_{N_i}. \mathcal{A} \vdash_{V_{N_i}}^C N_i : \sigma_i.$$

In particular,  $\exists V_{N_0}. \mathcal{A} \vdash_{V_{N_0}}^C N_0 : \sigma_0$ . Hence we may construct  $V_M$  in the usual way by rule APP:

$$\exists b_0. \mathcal{A} \vdash_{V_M}^C M : \sigma_0 b_0 \sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha.$$

Moreover, by Proposition 4.2.3  $b_0 =_{BA} \Rightarrow$ , thus

$$\mathcal{A} \vdash_{V_M}^C M : \sigma_0 \Rightarrow \sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha,$$

as required. Hence,  $\mathcal{S}[\sigma]_{\nu_0} \subseteq \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^C M : \sigma\}$

$\sigma \equiv \sigma_0 \multimap \sigma_1 b_1 \sigma_2 \dots \sigma_n b_n \alpha$  Similar to the previous case.

□

### Theorem 4.5.2

$A \models_V M : \tau$  implies  $A \vdash_V^C M : \tau$ .

#### Proof

Choose the term-semi-model and let  $\nu_0$  be defined as in the Lemma. Now, since  $A \models_V M : \tau$  implies  $\llbracket M \rrbracket_{\mu_0} \in \mathcal{S}[\tau]_{\nu_0}$ ,  $\llbracket M \rrbracket_{\mu_0} = [M] \in \mathcal{S}[\tau]_{\nu_0} = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^C M : \tau\}$ , by the Lemma, thus establishing that there is a deduction of  $\tau$  for  $M$  under assumptions  $\mathcal{A}$ . Moreover,  $\forall x \in X. \llbracket \lambda x. M \rrbracket_{\mu_0} \in \mathcal{S}[\mathcal{A}(x) V(x) \tau]_{\nu_0}$ , thus by the Lemma this establishes that there is a deduction of  $\mathcal{A}(x) V(x) \tau$  for  $\lambda x. M$  for each  $x$ , and hence by rule ABS and Lemma 3.2.12 we may take  $V_M =_{BA} V$ . These together with Lemma 3.2.18 imply  $A \vdash_V^C M : \tau$  as required.

□

Note that the above proof fails for  $\lambda$ -models since in  $[M]$  there may be terms which cannot be assigned the type  $\sigma$  by  $\vdash^C$ . A good example is the following: in the term  $\lambda$ -model  $[\mathbf{KI}\Omega] = [\mathbf{I}]$ , but in the term semi- $\lambda$ -model  $[\mathbf{KI}\Omega] < [\mathbf{I}]$ . For this example we have  $\forall A, \sigma. A \vdash_{V_\infty}^C \mathbf{I} : \sigma \Rightarrow \sigma$ , but  $\nexists A, \sigma. A \vdash_{V_\infty}^C \mathbf{KI}\Omega : \sigma$ . Hence, in the  $\lambda$ -model, equal elements of the model may not all belong to the interpretation of a type. This is not true for semi- $\lambda$ -models, as proved by the above theorem and demonstrated by this example, in which  $[\mathbf{KI}\Omega]$  and  $[\mathbf{I}]$  are *not* equal.

### 4.5.2 Completeness of $\vdash^L$

Let the set of types be  $T_L^\nabla$ . Let  $V_M$  and  $\mathcal{A}$  be as before and let  $\nu_0(\alpha) = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^L M : \alpha\}$ .

Note that as per the convention in Chapter 3, a type written  $\tau$  is constrained to be *non-generic* (types which may be generic are always written  $\underline{\tau}$ ).

#### Lemma 4.5.3

Consider the term-semi-model, then  $\mathcal{L}[\sigma]_{\nu_0} = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^L M : \sigma\}$ .

#### Proof

By induction on  $\sigma$ . Similar to Lemma 4.5.1 since only non-generic types may be deduced for terms in  $\vdash^L$ . □

### Theorem 4.5.4

$A \models_V M : \tau$  implies  $A \vdash_V^L M : \tau$ .

#### Proof

Similar to Theorem 4.5.2 □

### 4.5.3 Completeness of $\vdash^I$

Let the set of types be  $T_I^\nabla$ .

#### Completeness using the Term Semi-Model

Let  $V_M$  and  $\mathcal{A}$  be as before and let  $\nu_0(\alpha) = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^I M : \alpha\}$ .

##### Lemma 4.5.5

Consider the term-semi-model, then  $\mathcal{I}[\sigma]_{\nu_0} = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^I M : \sigma\}$ .

##### Proof

By induction on  $\sigma$ . Similar to earlier proofs, with the case  $\sigma = \omega$  following by Corollary 3.4.9. The case  $\sigma = \sigma_1 \cap \sigma_2$  also follows by the induction hypotheses and the definition of  $\mathcal{I}[\cdot]$ , using rule MEET which is applicable since the variable head neededness function is the same in each case.  $\square$

Although every  $\lambda$ -term can be assigned a type by the type assignment systems  $T_I^{B\Delta}$  and  $T_I^{\Delta g}$ , not all terms can be assigned a type for a *particular* assumption set and free variable strong head neededness function. In contrast, the  $\omega$  rule of the standard intersection type discipline allows one to assign the type  $\omega$  to every term, *irrespective of the type assumptions*. Thus the completeness proof presented here is modulo this restriction.

##### Theorem 4.5.6

Suppose  $\tau \neq \omega$ , then  $A \models_V M : \tau$  implies  $A \vdash_V^I M : \tau$ .

##### Proof

Similar to earlier proofs of completeness for the other type assignment systems, as follows. Choose the term-semi-model and let  $\nu_0$  be defined as in the Lemma. Now,  $A \models_V M : \tau$  implies  $\llbracket M \rrbracket_{\mu_0} \in \mathcal{I}[\tau]_{\nu_0}$ ,  $\llbracket M \rrbracket_{\mu_0} \in \mathcal{I}[\tau]_{\nu_0} = \{\llbracket M \rrbracket_{\mu_0} \mid \mathcal{A} \vdash_{V_M}^I M : \tau\}$ . Moreover,  $\forall x \in X. \llbracket \lambda x. M \rrbracket_{\mu_0} \in \mathcal{I}[\mathcal{A}(x) V(x) \tau]_{\nu_0}$ , thus by the Lemma this establishes that there is a deduction of  $\mathcal{A}(x) V(x) \tau$  for  $\lambda x. M$  for each  $x$ , and hence by rule ABS and Lemma 3.4.10 we may take  $V_M =_{BA} V$ . By the Lemma above and Lemma 3.4.19  $A \vdash_V^I M : \tau$ , as required.  $\square$

#### Completeness using the Filter Semi-Model

Let  $A$  be an assumption set. Let  $\mu_A(x)$  be the filter generated by  $\sigma$ , where  $x : \sigma \in A^3$ , and  $\nu_A(\alpha) = \{\llbracket M \rrbracket_{\mu_A}^\nabla \mid \alpha \in \llbracket M \rrbracket_{\mu_A}^\nabla\}$ .

##### Lemma 4.5.7

Choose the filter semi-model, then  $\mathcal{I}[\sigma]_{\nu_A}^\nabla = \{\llbracket M \rrbracket_{\mu_A}^\nabla \mid \sigma \in \llbracket M \rrbracket_{\mu_A}^\nabla\}$ .

<sup>3</sup>Recall that there may be at most one entry for each  $x \in X$  in each assumption set.

**Proof**

By induction on  $\sigma$ .

$\sigma \equiv \alpha$  Immediate by definition of  $\nu_A$ .

$\sigma \equiv \omega$  Immediate since for every  $M \in \Lambda$ ,  $\llbracket M \rrbracket_{\mu_A}^\nabla$  is a filter.

$\sigma \equiv \sigma_1 \cap \sigma_2$  Straightforward by induction, as follows:

$$\begin{aligned} \mathcal{I}[\sigma_1 \cap \sigma_2]_{\nu_A}^\nabla &= \mathcal{I}[\sigma_1]_{\nu_A}^\nabla \cap \mathcal{I}[\sigma_2]_{\nu_A}^\nabla \\ &= \{ \llbracket M \rrbracket_{\mu_A}^\nabla \mid \sigma_1 \in \llbracket M \rrbracket_{\mu_A}^\nabla \} \cap \{ \llbracket M \rrbracket_{\mu_A}^\nabla \mid \sigma_2 \in \llbracket M \rrbracket_{\mu_A}^\nabla \} \\ &\quad \{ \text{by the induction hypotheses} \} \\ &= \{ \llbracket M \rrbracket_{\mu_A}^\nabla \mid \sigma_1 \cap \sigma_2 \in \llbracket M \rrbracket_{\mu_A}^\nabla \} \\ &\quad \{ \text{since } \llbracket M \rrbracket_{\mu_A}^\nabla \text{ is a filter.} \} \end{aligned}$$

$\sigma \equiv \sigma_0 \Rightarrow \sigma_1 \text{ b}_1 \dots \sigma_n \text{ b}_n \alpha$  Firstly, suppose  $\llbracket M \rrbracket_{\mu_A}^\nabla \in \mathcal{I}[\sigma]_{\nu_A}^\nabla$ . By the induction hypotheses,  $\llbracket N_i \rrbracket_{\mu_A}^\nabla \in \mathcal{I}[\sigma_i]_{\nu_A}^\nabla$ ,  $i \in \{0, \dots, n\}$  implies  $\sigma_i \in \llbracket N_i \rrbracket_{\mu_A}^\nabla$ . By the remaining induction hypothesis (for  $\sigma_1 \text{ b}_1 \sigma_2 \dots \sigma_n \text{ b}_n \alpha$ ) and by  $\bullet_\nabla$ ,  $\alpha \in \llbracket M N_1 \dots N_n \rrbracket_{\mu_A}^\nabla$ . Then the result follows from Corollary 3.4.16, part 1.

Secondly, let  $V_{M, \mu_A}$  be as previously defined and suppose  $\sigma \in \llbracket M \rrbracket_{\mu_A}^\nabla$ . Then the result follows since  $\llbracket M \rrbracket_{\mu_A}^\nabla = \{ \tau \mid A_{\mu_A} \vdash_{V_{M, \mu_A}}^I M : \tau \}$  hence  $\sigma \in \llbracket M \rrbracket_{\mu_A}^\nabla$  implies  $A_{\mu_A} \vdash_{V_{M, \mu_A}}^I M : \sigma$ .

$\sigma \equiv \sigma_0 \nrightarrow \sigma_1 \text{ b}_1 \dots \sigma_n \text{ b}_n \alpha$  Similar to the previous case.

□

**Theorem 4.5.8**

Suppose  $\tau \neq \omega$ , then  $A \models_V M : \tau$  implies  $A \vdash_V^I M : \tau$ .

**Proof**

Choose the filter semi-model,  $\mu_A$  and  $\nu_A$ , then  $\llbracket M \rrbracket_{\mu_A}^\nabla \in \mathcal{I}[\tau]_{\nu_A}^\nabla$ ,  $\mathcal{I}[\tau]_{\nu_A}^\nabla = \{ \llbracket N \rrbracket_{\mu_A}^\nabla \mid \tau \in \llbracket N \rrbracket_{\mu_A}^\nabla \}$ , by the Lemma, and  $\llbracket M \rrbracket_{\mu_A}^\nabla = \{ \sigma \mid A_{\mu_A} \vdash_{V_{M, \mu_A}}^I M : \sigma \}$  implies  $A \vdash_V^I M : \tau$  (since for any  $\sigma \in T_I^\nabla$ ,  $\sigma = \bigcap_{\sigma \leq \tau} \sigma$  and by Lemma 3.4.19), as required. □

## 4.6 Discussion

This chapter has established a formal connection between Chapters 2 and 3. This was achieved by firstly giving a semantic equivalent of strong head needness (and irrelevance), and then showing the semantic relationship between the interpretation of Boolean Reduction Types and the applicative behaviour of  $\lambda$ -terms. Along the way, *context-sensitive* semantics were developed for the various sets of Boolean Reduction Types.

---

This chapter has also shown that all the type deduction logics are semantically correct with respect to semi- $\lambda$ -models and semantically sound with respect to  $\lambda$ -models.

# Chapter 5

## Implementation

This chapter gives algorithms which correspond to each of the type assignment systems of Chapter 3. Each of the implementations is correct (*i.e.*, sound and complete) with respect to its deduction system. Of course, since these type assignment systems have been shown to be *semantically* correct (up to reduction), this implies the *semantic* correctness of each of the implementations.

Another consequence of the faithfulness of these algorithms to the dictum of the type assignment systems is that any decidability results for these algorithms will be a direct result of the power of the respective deduction system. Hence, for the intersection-style system it is apparent that its implementation is semi-decidable, since this system can distinguish between solvable and unsolvable terms.

An important step in each implementation is the satisfaction of the APP rule. In general, this rule demands that the argument type of the function match the actual type of the argument. Moreover, deductions must be made with identical assumption sets. Both of these problems may be resolved with a *unification* procedure, as is usual for type inference algorithms. However, one important difference with the current work is that since types contain arrow expressions it is necessary to perform unification of these Boolean expressions.

This chapter contains four sections. The first is some material common to all of the implementations. After this first section a single section is devoted to each implementation. The first section discusses substitution and unification of Boolean expressions.

Within each of the sections which describes an implementation a common pattern emerges. Firstly any operations required are introduced and proven correct. These operations are substitution and, in the case of the final implementation, expansion.

The next step is the description of the unification algorithm and proof of its correctness. Fortunately, the first two implementations can share the same unification algorithm. Finally, constructions and correctness proofs are given

for the algorithms which compute a (most general) type for each admissible term.

## 5.1 Preliminaries

### Definition 5.1.1

Write  $\alpha \in \tau$  whenever  $\alpha \in \text{Vars}(\tau)$ .

#### 5.1.1 Substitutions

Let  $\nabla$  be  $\mathcal{B}_\Delta$ ,  $\Delta_g$ ,  $\{\Rightarrow\}$  or  $\{\rightarrow\}$  and let  $T$  be  $T_C^\nabla$ ,  $T_L^\nabla$  or  $T_I^\nabla$  for each possible  $\nabla$ .

### Definition 5.1.2

Let  $\text{Id}$  stand for a generic identity function. A *substitution* is a pair  $(R_T : \tau_v \rightarrow T, R_\nabla : \Delta_v \rightarrow \nabla)$ . Substitutions will usually be denoted by the letter  $S$  (possibly primed or subscripted). However, substitutions of the form  $(\text{Id}, R_\nabla)$ , for any  $R_\nabla : \Delta_v \rightarrow \nabla$ , will often be denoted by the letter  $R$  (possibly primed or subscripted).

The *application* of a substitution to a type or an arrow is defined as follows. Let  $S$  be a substitution  $(R_T, R_\nabla)$  for some  $R_T$  and  $R_\nabla$ ,  $\tau \in T$  and  $b \in \nabla$ , then write

- $S(\tau)$  for  $R_T; R_\nabla(\tau)$  and
- $S(b)$  for  $R_\nabla(b)$ ,

where  $R_T$  and  $R_\nabla$  are homomorphically extended to types, arrows and types and arrows, respectively, in the obvious fashion.

As usual, the left-to-right composition of substitutions  $S$  and  $S'$  will be written  $S; S'$ , that is,  $S; S'(\tau) = S'(S(\tau))$  and  $S; S'(b) = S'(S(b))$ . Let  $\langle A, V, \sigma \rangle$  be a triple of an assumption set  $A$ , a variable strong head neededness function  $V$  and a type  $\sigma$ , then write  $S(\langle A, V, \sigma \rangle)$  for  $\langle S(A), S(V), S(\sigma) \rangle$ .

An ordering on substitution can be defined with least element  $(\text{Id}, \text{Id})$  (which is often denoted by  $\text{Id}$ ):

$$S \leq S' \text{ if } \exists S''. S' = S; S''.$$

Let  $S = (R_T, R_\nabla)$ , then write  $S[\alpha := \tau]$  for the substitution  $(R_T[\alpha := \tau], R_\nabla)$ , where  $R_T[\alpha := \tau]$  is the substitution which differs from  $R_T$  only at the point  $\alpha$  at which point its value is  $\tau$ .

Similarly, write  $S[\rightarrow_i := b]$  for the substitution  $(R_T[\rightarrow_i := b], R_\nabla[\rightarrow_i := b])$ .  $R_T[\rightarrow_i := b]$  is the substitution identical to  $R_T$  except for occurrences of  $\rightarrow_i$  in the range of  $R_T$  which are replaced by  $b$ .  $R_\nabla[\rightarrow_i := b]$  is the substitution which differs from  $R_\nabla$  only at the point  $\rightarrow_i$  at which point its value is  $b$ .

### 5.1.2 Triples

#### Definition 5.1.3

A triple  $\langle A, V, \tau \rangle$  is *suitable* for  $M \in \Lambda$  if  $A \vdash_V^* M : \tau$ , where  $\vdash^*$  is one of  $\vdash^C$ ,  $\vdash^L$  or  $\vdash^I$ .

Write  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$  if  $\exists S. \langle A', V', \tau' \rangle = S(\langle A, V, \tau \rangle)$ . Let  $\mathcal{ST} = \{ \langle A, V, \tau \rangle \mid \exists M \in \Lambda. \langle A, V, \tau \rangle \text{ is suitable for } M \}$ .

#### Definition 5.1.4

Let  $A \vdash_V^* M : \tau$ , where  $\vdash^*$  is one of  $\vdash^C$ ,  $\vdash^L$  or  $\vdash^I$ .  $\tau$  is a *principal type* for  $M$  if  $A' \vdash_V^* M : \sigma$  implies there exists a substitution  $S$  such that  $\sigma = S(\tau)$  and  $A' = S(A)$ .

Similarly, a triple  $\langle A, V, \tau \rangle$  is a *principal triple* for  $M$  if it is a suitable triple for  $M$  and if  $\langle A', V', \tau' \rangle$  is a suitable triple for  $M$ , then  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$ .

### 5.1.3 Unification of Boolean Rings

Since the types used in this thesis have Boolean expressions of arrows in them it will be necessary to perform unification of these Boolean expressions while doing type inference. Fortunately, results from Unification Theory are available which show that this task is decidable.

Hsiang [31] attributes the following theorem to Stone [62]:

#### Theorem 5.1.5 (Stone's Theorem)

1. Every Boolean ring is commutative (both  $+$  and  $*$ ).
2. Every Boolean ring is nilpotent with respect to  $+$ .
3. Let  $(B, +, *, 0, 1)$  be a Boolean ring with unit 1. Introduce the following operators:

$$x \vee y = x + y + x * y,$$

$$x \wedge y = x * y,$$

$$\neg x = x + 1,$$

then  $(B, \vee, \wedge, \neg, 0, 1)$  is a Boolean algebra. Conversely, given a Boolean algebra  $(B, \vee, \wedge, \neg, 0, 1)$ , define:

$$a + b = (a \wedge \neg b) \vee (\neg a \wedge b),$$

$$a * b = a \wedge b,$$

then  $(B, +, *, 0, 1)$  is a Boolean ring.



As a notational convenience, note that a Boolean expression of  $n$  variables ( $n \geq 0$ ),  $b$ , may be written as a function,  $f(x_1, \dots, x_n)$ , of its variables.

From Martin and Nipkow [45] comes the useful result that unification of expressions in a Boolean ring is unitary and decidable, that is, there exists an effective algorithm for determining the most general unifier of two Boolean expressions (if such a unifier exists at all), and furthermore this most general unifier is unique.

The key step to the above result is the transformation of the equations defining a Boolean ring into a canonical rewrite system:

$$\begin{aligned} 0 + x &\rightarrow x \\ x + x &\rightarrow 0 \\ 0 * x &\rightarrow 0 \\ 1 * x &\rightarrow x \\ x * x &\rightarrow x \\ x * (y + z) &\rightarrow x * y + x * z. \end{aligned}$$

(Martin and Nipkow [45] attribute these rules to Hsiang and Dershowitz—they are also mechanically derivable by using the well-known Knuth-Bendix procedure extended with associative-commutative unification).

By using these rewrite rules any Boolean ring expression<sup>1</sup> may be translated into normal form (known as *polynomial normal form*). Note that any equation of the form  $f(\bar{x}) = g(\bar{x})$  may be rewritten as  $f(\bar{x}) + g(\bar{x}) = 0$ .

There are two methods (of equal time complexity) for solving an equation between two expressions of a Boolean ring. The first algorithm, cited by Martin and Nipkow [45] as first formulated by Boole [6], is the method of *successive variable elimination* which is based on the observation that  $f(\bar{x}) = 0$  is true whenever  $f(0, x_2, \dots, x_n) * f(1, x_2, \dots, x_n) = 0$  is true.

The second algorithm is due to Löwenheim [43] (as cited by Martin and Nipkow [45]). This method proceeds by employing a separate algorithm to find a particular solution to the equation and then “plugging” this solution into a universal formula to obtain a most general solution.

The algorithm for Boolean unification based on Boole’s method of variable elimination is given in Figure 14. The algorithm presented in Figure 14 makes use of a Boolean algebra. A Boolean ring is used in Appendix A.

### Theorem 5.1.6

Either algorithm BUNIFY succeeds with the most general unifier of its arguments, or it reports failure.

<sup>1</sup>and hence any Boolean algebra expression by Stone’s Theorem.

```

BUNIFY( $b_1, b_2$ ) = BUNIFY'( $((b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2))$ )
BUNIFY'( $f(x_1, \dots, x_n)$ )
  = if  $n = 0$ 
    then if  $f(x_1, \dots, x_n) =_{BA} \rightarrow$  then Id
    else  $\perp$ 
  else let  $G = \text{BUNIFY}'(f(\rightarrow, x_2, \dots, x_n) \wedge f(\Rightarrow, x_2, \dots, x_n))$ 
    in  $G[x_1 := ((\neg f(\Rightarrow, G(x_2), \dots, G(x_n))) \wedge x_1) \vee$ 
       $f(\rightarrow, G(x_2), \dots, G(x_n))]$ 

```

Figure 14: The Algorithm for unifying Boolean Arrow Expressions

**Proof**

See Martin and Nipkow [45].  $\square$

Note this algorithm is a form of search procedure and, as reported in [45], its worst case complexity is  $O(l * 2^n)$ , where  $l$  is the length (number of symbols) of the expression being solved and  $n$  is the number of variables in the expression. Some improvement can be obtained by the application of simplification steps after each variable elimination step takes place. (The complexity of the method based on Löwenheim's method turns out to be identical, see Martin and Nipkow [45]).

Note that this result may not be as bad as it seems since the length and number of variables in an individual arrow expression in a particular reduction type is typically very short, and there is no negative interaction amongst arrow expressions in a single reduction type.

## 5.2 Implementation of the Curry-style System

In this section the development of an algorithm is conducted which corresponds<sup>2</sup> to the simplest of the deduction systems of Chapter 3 (summarised in Figure 5 of Chapter 3). There are two main components to this implementation: firstly, a function which returns the unifying substitution of two  $T_C^\forall$  types ( $U_C$ ) and secondly, a function ( $TYPE_C$ ) which recursively descends through the term for which a type is sought, calling  $U_C$  (indirectly, see below) where necessary.

<sup>2</sup>In a formal sense, see the correctness proofs at the end of this section.

### 5.2.1 Preliminaries

The implementation applies substitutions to derivations in order to compute instances of these derivations. These instances are then used to allow the application of terms. Thus the preservation of deductions under substitution must be shown before the implementation can proceed.

#### Lemma 5.2.1

Let  $S$  be a substitution, then  $A \vdash_V^C M : \tau$  implies  $S(A) \vdash_{S(V)}^C M : S(\tau)$ .

#### Proof

By induction on the structure of  $A \vdash_V^C M : \tau$ .

All cases (VAR, APP, ABS) follow easily. (The result also holds for APP- $\rightarrow$ , though no use is made of this fact).  $\square$

### 5.2.2 Unification of Types

Now the procedure for unifying types can be presented. The function  $U_C : T_C^\nabla \times T_C^\nabla \rightarrow T_C^\nabla \rightarrow T_C^\nabla$  is defined in Figure 15.

As can readily be seen,  $U_C$  is defined by case analysis on the pair of types which are its argument. In the case of a type variable the unifying substitution is simply the identity function with the exception that the type variable,  $\alpha$ , is mapped to the second argument. Note the inclusion of a test to prevent circular (infinite) unifications. In this last case the algorithm is undefined.

The interesting step is that of unifying two function types. In this case not just the argument and result types must be unified, but the arrows as well. For this the algorithm of Figure 14 is used. The unification of the argument and result types is computed by the mutually recursive (with this function) function  $\text{UNIFY}_C$ . Also note that the order in which the arrow unification is performed is unimportant so long as if it is performed before the call to  $\text{UNIFY}_C$ , then the substitution  $R$  must be applied to  $\sigma_i$  and  $\tau_i$  ( $i \in \{1, 2\}$ ). (To avoid this complication it is written afterwards here).

#### Definition 5.2.2

1. Pairs of the form  $(\sigma, \tau)$ , where  $\sigma, \tau \in T_C^\nabla$ , will be referred to as *equations*.
2. A substitution  $S$  *solves* a set of equations  $E$  if  $\forall (\sigma, \tau) \in E. S(\sigma) = S(\tau)$ .

The function  $\text{UNIFY}_C : \{T_C^\nabla \times T_C^\nabla\} \rightarrow T_C^\nabla \rightarrow T_C^\nabla$  is defined by Figure 16. This function simply computes the in-order composition of the substitutions required to unify each component of the set, applying each such solution to the rest of the set as  $\text{UNIFY}_C$  proceeds.

1.  $U_C(\alpha, \tau) = \text{if } \alpha \in \tau \text{ and } \alpha \neq \tau \text{ then } \perp$   
 $\text{else Id}[\alpha := \tau].$
2.  $U_C(\tau, \alpha) = U_C(\alpha, \tau).$
3.  $U_C(\sigma_1 \text{ b}_1 \sigma_2, \tau_1 \text{ b}_2 \tau_2) = \text{let } S = \text{UNIFY}_C(\{(\sigma_1, \tau_1), (\sigma_2, \tau_2)\}) \text{ in}$   
 $\text{let } R = \text{BUNIFY}(S(\text{b}_1), S(\text{b}_2)) \text{ in}$   
 $S; R.$

Figure 15: The Algorithm for unifying Simple Boolean Reduction Types

1.  $\text{UNIFY}_C(\{(\sigma, \tau)\} \cup E') = S; \text{UNIFY}_C(S(E'))$   
 $\text{where } S = U_C(\sigma, \tau).$
2.  $\text{UNIFY}_C(\emptyset) = \text{Id}.$

Figure 16: The Algorithm for solving a set of Equations

### Correctness

Now the correctness of these two algorithms can be immediately established. For this to be done, both the soundness and the completeness of the algorithms must be shown, whenever their results are defined.

#### Theorem 5.2.3 (Soundness of $U_C$ )

If  $S = U_C(\sigma, \tau)$  is defined, then  $S(\sigma) = S(\tau)$ .

#### Proof

By induction on  $\sigma$  and  $\tau$ .

In the cases  $U_C(\alpha, \tau)$  and  $U_C(\tau, \alpha)$  the result follows immediately since the statement of the theorem guarantees that if  $\alpha \neq \tau$ , then  $\alpha \notin \tau$ .

In the case  $U_C(\sigma_1 \text{ b}_1 \sigma_2, \tau_1 \text{ b}_2 \tau_2)$ , by the induction hypotheses for  $U_C(\sigma_1, \tau_1)$  and  $U_C(S_1(\sigma_2), S_1(\tau_2))$  (which are applicable as both expressions must be defined since the overall expression is defined),  $S_1(\sigma_1) = S_1(\tau_1)$  and  $S_1; S_2(\sigma_2) = S_1; S_2(\tau_2)$ , now  $S = S_1; S_2$ , hence  $S(\sigma_i) = S(\tau_i)$ , where  $i \in \{1, 2\}$ . By Theorem 5.1.6:

$$S; R(\text{b}_1) =_{\text{BA}} S; R(\text{b}_2),$$

hence  $S; R(\sigma_i) = S; R(\tau_i)$  ( $i \in \{1, 2\}$ ), so

$$S; R(\sigma_1 \text{ b}_1 \sigma_2) = S; R(\tau_1 \text{ b}_2 \tau_2).$$

□

**Theorem 5.2.4 (Completeness of  $U_C$ )**

1. If  $S'(\sigma) = S'(\tau)$ , then  $U_C(\sigma, \tau)$  is defined.
2. If  $S'(\sigma) = S'(\tau)$  and  $S = U_C(\sigma, \tau)$ , then  $S \leq S'$ .

**Proof**

By induction on  $\sigma$  and  $\tau$ .

In the case  $\sigma$  is a type variable  $\alpha$ , since  $S'(\sigma) = S'(\tau)$  it must be true that  $\alpha \notin \tau$ , hence  $U_C(\alpha, \tau) = \text{Id}[\alpha := \tau]$ , which is defined. For part 2 of the theorem, note that

$$\begin{aligned}
 S' &= S^*[\alpha := \tau] \\
 &= \text{Id}[\alpha := \tau]; S^*[\alpha := \alpha] \\
 &= \text{Id}[\alpha := \tau]; S'',
 \end{aligned}$$

where  $S'' = S^*[\alpha := \alpha]$ . Hence  $S \leq S'$ .

The case  $\tau$  a type variable is similar.

The case  $\sigma$  and  $\tau$  both arrow (functional) types follows directly from Theorems 5.2.6 and 5.1.6 and construction since the result of  $U_C$  in this case is merely the composition of the substitutions returned by calls on these two algorithms (preserving the order of application of the substitutions). □

**Theorem 5.2.5 (Soundness of  $\text{UNIFY}_C$ )**

If  $S = \text{UNIFY}_C(E)$  is defined, then  $S$  solves  $E$ .

**Proof**

By induction on the cardinality of  $E$ .

The case when  $E$  is the empty set is trivially satisfied since every substitution solves the empty set.

In the case that  $E = \{(\sigma, \tau)\} \cup E'$ ,  $S(\sigma) = S(\tau)$  by Theorem 5.2.3 (which is applicable as  $\text{UNIFY}_C(E)$  defined implies  $U_C(\sigma, \tau)$  is defined), then the result follows from the induction hypotheses. □

**Theorem 5.2.6 (Completeness of  $\text{UNIFY}_C$ )**

Let  $E$  be a set of equations and  $S'$  be a substitution such that  $S'$  solves  $E$ , then

1.  $\text{UNIFY}_C(E)$  is defined, and
2. if  $S = \text{UNIFY}_C(E)$ , then  $S \leq S'$ .

**Proof**

By induction on the cardinality of  $E$ .

The case  $E$  is the empty set is trivially satisfied.

The case  $E = \{(\sigma, \tau)\} \cup E'$  follows from Theorem 5.2.4 and the induction hypothesis. □

1.  $\text{TYPE}_C(x) = \langle \{x : \alpha\}, V_{\rightarrow}[x := \Rightarrow], \alpha \rangle.$
2.  $\text{TYPE}_C(\lambda x.M) = \text{let } \langle A', V', \tau' \rangle = \text{TYPE}_C(M) \text{ in}$   
 $\quad \text{let } \sigma = \text{if } x \in \text{dom } A' \text{ then } A'(x)$   
 $\quad \quad \text{else } \alpha$   
 $\quad \text{in}$   
 $\quad \langle A'_x, V'[x := \rightarrow], \sigma V(x) \tau' \rangle.$
3.  $\text{TYPE}_C(M_1 M_2) = \text{let } \langle A_1, V_1, \tau_1 \rangle = \text{TYPE}_C(M_1) \text{ in}$   
 $\quad \text{let } \langle A_2, V_2, \tau_2 \rangle = \text{TYPE}_C(M_2) \text{ in}$   
 $\quad \text{let } S = \text{UNIFY}_C(\{(\tau_1, \tau_2 \rightarrow_i \alpha)\} \cup$   
 $\quad \quad \bigcup_{x \in \text{dom } A_1 \cap \text{dom } A_2} \{(A_1(x), A_2(x))\})$   
 $\quad \text{in}$   
 $\quad \langle \lambda x. \text{if } x \in \text{dom } A_1 \text{ then } S(A_1(x))$   
 $\quad \quad \text{else } S(A_2(x)),$   
 $\quad \quad \lambda x. S(V_1(x) \vee (\rightarrow_i \wedge V_2(x))),$   
 $\quad \quad S(\alpha) \rangle.$

Figure 17: The Type Inference Algorithm for Simple Boolean Reduction Types

### 5.2.3 Type Inference

Now the function for inferring Simple Boolean Reduction Types can be constructed in terms of the unification functions. Let  $\alpha$  be a new type variable and  $\rightarrow_i$  be a new arrow variable (*i.e.*, these variables are used nowhere else in each case of the function), then  $\text{TYPE}_C : \Lambda \rightarrow \{X \times T_C^\nabla\} \times (X \rightarrow B_\Delta) \times T_C^\nabla$  is defined by Figure 17.

#### Correctness

The soundness of  $\text{TYPE}_C$  requires that for a given  $\lambda$ -term the result that it produces be derivable in the Curry-style type assignment system of Chapter 3, and this can be immediately established:

#### Theorem 5.2.7 (Soundness of $\text{TYPE}_C$ )

If  $\langle A, V, \tau \rangle = \text{TYPE}_C(M)$  is defined, then  $A \vdash_V^C M : \tau$ .

#### Proof

By induction on the structure of  $M$ .

If  $M \equiv x$ , then the result follows immediately by rule VAR.

If  $M \equiv \lambda x.N$ , then by the induction hypothesis  $\langle A', V', \tau' \rangle = \text{TYPE}_C(N)$  implies

$$A' \vdash_{V'}^C N : \tau'.$$

The result then follows (by construction) from rule ABS.

If  $M \equiv M_1 M_2$ , then by the induction hypotheses  $\langle A_1, V_1, \tau_1 \rangle = \text{TYPE}_C(M_1)$  and  $\langle A_2, V_2, \tau_2 \rangle = \text{TYPE}_C(M_2)$  implies

$$A_1 \vdash_{V_1}^C M_1 : \tau_1$$

and

$$A_2 \vdash_{V_2}^C M_2 : \tau_2,$$

respectively. By the statement of the theorem  $\text{TYPE}_C$  is defined, so the following is also defined:

$$S = \text{UNIFY}_C(\{(\tau_1, \tau_2 \rightarrow_i \alpha)\} \cup \bigcup_{x \in \text{dom} A_1 \cap \text{dom} A_2} \{(A_1(x), A_2(x))\}).$$

Now, by Theorem 5.2.5,

$$S(\tau_1) = S(\tau_2 \rightarrow_i \alpha)$$

and

$$\forall x \in \text{dom} A_1 \cap \text{dom} A_2. S(A_1(x)) = S(A_2(x)).$$

By Lemma 5.2.1 and, if necessary, Lemma 3.2.17:

$$S(A) \vdash_{S(V_1)}^C M_1 : S(\tau_1)$$

and

$$S(A) \vdash_{S(V_2)}^C M_2 : S(\tau_2),$$

where  $A = A_1 \cup \{x : \sigma \mid x : \sigma \in A_2; x \notin \text{dom} A_1\}$ .

Now the result follows from rule APP and the construction of  $\langle A, V, \tau \rangle$ .  $\square$

To show that  $\text{TYPE}_C$  is complete with respect to the Curry-style type assignment system of Chapter 3, some subsidiary results are required. In particular we need to know when the unification algorithms,  $U_C$  and  $\text{UNIFY}_C$ , are defined:

#### Lemma 5.2.8

If  $\tau_1 \leq \tau$ ,  $\tau_2 \leq \tau$  and  $\tau_1$  and  $\tau_2$  have no variables in common, then  $U_C(\tau_1, \tau_2)$  is defined.

**Proof**

Straightforward induction on the structure of  $\tau_1$  and  $\tau_2$ . The key fact is that if  $\tau_i$  is a type variable, then  $\tau_i \notin \tau_j$ , where  $i \neq j$ ,  $i, j \in \{1, 2\}$ .  $\square$

**Lemma 5.2.9**

Let  $E$  be a set of equations. If  $\forall(\tau_1, \tau_2) \in E. \exists \tau \in T_C^\nabla. \tau_1 \leq \tau, \tau_2 \leq \tau$  and  $\tau_1$  and  $\tau_2$  have no variables in common, then  $\text{UNIFY}_C(E)$  is defined.

**Proof**

Straightforward induction on the cardinality of  $E$  using Lemma 5.2.8.  $\square$

Note that the completeness theorem for  $\text{TYPE}_C$  presented below entails the existence of principal triples for the Curry-style type assignment system (i.e., every typable term has a principal triple and  $\text{TYPE}_C$  applied to a term returns the principal triple of that term).

**Theorem 5.2.10 (Completeness of  $\text{TYPE}_C$ )**

If for some  $A', V'$  and  $\tau'$  there is a deduction of  $A' \vdash_{V'}^C M : \tau'$  for  $M$ , then  $\text{TYPE}_C(M)$  is defined,  $\langle A, V, \tau \rangle = \text{TYPE}_C(M)$  and  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$ .

**Proof**

By induction on the structure of  $M$ .

If  $M \equiv x$ , then clearly  $\text{TYPE}_C(M)$  is defined and  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$ .

If  $M \equiv \lambda x. N$ , then the deduction of  $A' \vdash_{V'}^C M : \tau'$  must end in a use of the ABS rule whose antecedent has the form:

$$A'_x \cup \{x : \sigma\} \vdash_{V'[x := b']}^C N : \tau'',$$

for some types  $\sigma', \tau''$  and arrow expression  $b'$  such that  $\tau' = \sigma' b' \tau''$ . By the induction hypothesis  $\text{TYPE}_C(N)$  is defined and

$$\langle A_x \cup \{x : \sigma\}, V[x := b], \mu \rangle = \text{TYPE}_C(N),$$

where  $\langle A_x \cup \{x : \sigma\}, V[x := b], \mu \rangle \leq \langle A'_x \cup \{x : \sigma'\}, V'[x := b'], \tau'' \rangle$ . The result follows immediately by construction of  $\langle A, V, \tau \rangle$ .

If  $M \equiv M_1 M_2$ , then the deduction of  $A' \vdash_{V'}^C M : \tau'$  must end in a use of the APP rule whose antecedents have the form:

$$A' \vdash_{V'_1}^C M_1 : \sigma' b' \tau'$$

and

$$A' \vdash_{V'_2}^C M_2 : \sigma',$$

where  $V'_1, V'_2$  and  $b'$  are such that  $V = \lambda x. V'_1(x) \vee (b' \wedge V'_2(x))$ . By the induction hypotheses both  $\text{TYPE}_C(M_1)$  and  $\text{TYPE}_C(M_2)$  are defined and

$$\langle A_1, V_1, \tau_1 \rangle \leq \langle A', V'_1, \sigma' b' \tau' \rangle$$



and

$$\langle A_2, V_2, \tau_2 \rangle \leq \langle A', V'_2, \sigma' \rangle.$$

Then Lemma 5.2.9 applies by construction of  $\text{TYPE}_C$  and so by Theorem 5.2.5,  $S$  solves<sup>3</sup>

$$\{(\tau_1, \tau_1 \rightarrow_i \alpha)\} \cup \bigcup_{x \in \text{dom } A_1 \cap \text{dom } A_2} \{(A_1(x), A_2(x))\},$$

hence Theorem 5.2.6 is now applicable and the result follows by construction and using Lemma 5.2.1.

□

### 5.2.4 Decidability

Since the algorithm for unification of arrow expressions is decidable the following argument shows that unification of simple reduction types is decidable:

1. for the case of one of the types a type variable the algorithm is either undefined (due to the presence of a circular equation) or is a substitution of the other type for the type variable, and
2. for the case of both types functions the result follows immediately from the induction hypotheses and the decidability of unification of arrow expressions (Theorem 5.1.6).

## 5.3 Implementation of the LET-Polymorphic System

In this section the implementation of the LET-polymorphic type assignment system as an inference procedure is explored. The pattern that will be followed here is modelled after that of the previous section. The implementation turns out to be a relatively straightforward extension of the implementation for the Curry-style system.

### 5.3.1 Substitution

The application of a substitution to a generic type  $\underline{\sigma}$  is defined as follows:

- $S(\forall \alpha. \underline{\sigma}') \equiv \forall \alpha. S[\alpha := \alpha](\underline{\sigma}')$ ,
- $S(\forall \rightarrow_i. \underline{\sigma}') \equiv \forall \rightarrow_i. S[\rightarrow_i := \rightarrow_i](\underline{\sigma}')$  and

---

<sup>3</sup> $\tau_1$  is unified with  $\tau_1 \rightarrow_i \alpha$  as specified in the definition of  $\text{TYPE}_C$ .

- $S(\tau)$  is as before.

In order to avoid capture of bound variables, a similar convention is adopted as to that which was adopted for  $\lambda$ -terms, namely that bound variables are renamed such that they do not occur in the free variables of the range of any substitution applied to the generic type. This convention allows the establishment of the following Lemmas.

**Lemma 5.3.1**

$\underline{\sigma} \preceq \tau$  implies  $S(\underline{\sigma}) \preceq S(\tau)$ .

**Proof**

If  $\underline{\sigma}$  is non-generic, then the result follows since  $\tau = S(\underline{\sigma})$ .

Otherwise, suppose  $\underline{\sigma} \equiv \forall \delta_1, \dots, \delta_n. \tau'$ , then  $\tau \equiv S'(\tau')$  where  $\text{dom } S' = \{\delta_1, \dots, \delta_n\}$ , by definition of  $\preceq$ . Hence,

$$\begin{aligned} S(\tau) &\equiv (S'; S)(\tau') \\ &\equiv (S[\bar{\delta} := \underline{\delta}]; S')(\tau'), \end{aligned}$$

which implies  $S(\underline{\sigma}) \equiv \forall \delta_1, \dots, \delta_n. S[\bar{\delta} := \underline{\delta}](\tau') \preceq S(\tau)$ , as required.  $\square$

**Lemma 5.3.2**

Let  $\underline{\sigma} = \text{gen}((A, V, \tau))$ , then  $S(\underline{\sigma}) = \text{gen}((S(A), S(V), S(\tau)))$ .<sup>4</sup>

**Proof**

Straightforward by the convention for bound variables.  $\square$

**Lemma 5.3.3**

Let  $S$  be a substitution, then  $A \vdash_V^L M : \tau$  implies  $S(A) \vdash_{S(V)}^L M : S(\tau)$ .

**Proof**

By induction on the structure of  $A \vdash_V^L M : \tau$ .

For rule VAR use Lemma 5.3.1. For rule APP- $\forall$  use Lemma 5.3.2. All other cases (APP, ABS, APP- $\rightarrow$ ) are straightforward.  $\square$

## 5.3.2 Unification of Types

No new unification procedure is required as the APP rule is unchanged from the APP rule of the Curry-style system. (i.e. the set of types applicable in the APP rule is unchanged).

---

<sup>4</sup>gen() is defined in Chapter 3.

### 5.3.3 Type Inference

The algorithm presented below is closer to that normally used in actual implementations than the one presented for the Curry-style system since the information about assumption sets is shared between recursive calls to the algorithm (see the case of application below). This necessitates the return of a substitution as part of the result of the algorithm, but does avoid the need to use the unification procedure to unify assumption sets.

Let  $\alpha$  be a new type variable and  $\rightarrow_i$  be a new arrow variable, then  $\text{TYPE}_L$  is defined by Figure 18.

#### Correctness

##### Theorem 5.3.4 (Soundness of $\text{TYPE}_L$ )

Let  $A$  be some assumption set and  $M \in \Lambda$ . If  $\langle A', V, \tau \rangle = \text{TYPE}_L(A, M)$  is defined, then  $A' \vdash_V^L M : \tau$ .

#### Proof

By induction on the structure of  $M$ .

If  $M \equiv x$ , then the result follows immediately by rule VAR.

If  $M \equiv \lambda x. N$ , then by the induction hypothesis  $\langle A', V', \tau' \rangle = \text{TYPE}_L(A_x \cup x : \sigma, N)$  implies

$$A' \vdash_{V'}^L N : \tau'.$$

The result then follows (by construction) from rule ABS.

If  $M \equiv M_1 M_2$ , where  $M_1$  is a variable or application term, then by the induction hypotheses  $\langle A_1, V_1, \tau_1, S_1 \rangle = \text{TYPE}_L(A, M_1)$  and  $\langle A_2, V_2, \tau_2, S_2 \rangle = \text{TYPE}_L(A_1, M_2)$  implies

$$A_1 \vdash_{V_1}^L M_1 : \tau_1$$

and

$$A_2 \vdash_{V_2}^L M_2 : \tau_2,$$

respectively. By the statement of the theorem  $\text{TYPE}_L$  is defined, so the following is also defined:

$$S = \text{UC}(S_2(\tau_1), \tau_2 \rightarrow_i \alpha).$$

Now, by Theorem 5.2.3,

$$S(S_2(\tau_1)) = S(\tau_2 \rightarrow_i \alpha)$$

By Lemma 5.3.3 and, if necessary, Lemma 3.3.10:

$$S(A) \vdash_{S(S_2(V_1))}^L M_1 : S(S_2(\tau_1))$$

1.  $\text{TYPE}_L(A, x) = \text{let } \langle S, \sigma, A' \rangle =$   
     if  $x : \forall \delta_1, \dots, \delta_n. \tau \in A$  then  
         let  $S$  be a fresh renaming of  $\delta_1, \dots, \delta_n$  in  
          $\langle S, S(\tau), A \rangle$   
     else  $\langle \text{Id}, \alpha, A \cup \{x : \alpha\} \rangle$   
     in  
      $\langle \{x : \sigma\}, V_{\rightarrow}[x := \Rightarrow], \sigma, S \rangle$ .
2.  $\text{TYPE}_L(A, \lambda x. M) = \text{let } \langle A', V, \tau, S \rangle = \text{TYPE}_L(A_x \cup \{x : \alpha\}, M)$  in  
      $\langle A'_x \cup \{x : S(A(x))\}, V[x := \rightarrow], S(\alpha) V(x) \tau, S \rangle$ .
3. Let  $M_1$  be an application term or a term variable, then  
      $\text{TYPE}_L(A, M_1 M_2) = \text{let } \langle A_1, V_1, \tau_1, S_1 \rangle = \text{TYPE}_L(A, M_1)$  in  
     let  $\langle A_2, V_2, \tau_2, S_2 \rangle = \text{TYPE}_L(A_1, M_2)$  in  
     let  $S = \cup_C(S_2(\tau_1), \tau_2 \rightarrow; \alpha)$  in  
      $\langle S(A_2),$   
      $\lambda x. S(S_2(V_1(x)) \vee (\rightarrow; \wedge V_2(x))),$   
      $S(\alpha),$   
      $S_1; S_2; S \rangle$ .
4.  $\text{TYPE}_L(A, (\lambda x. M) N) = \text{let } \langle A_1, V_1, \tau_1, S_1 \rangle = \text{TYPE}_L(A, N)$  in  
     let  $\underline{\sigma} = \text{gen}(A_1, V_1, \tau_1)$  in  
     let  $\langle A_2, V_2, \tau_2, S_2 \rangle = \text{TYPE}_L((A_1)_x \cup \{x : \underline{\sigma}\}, M)$   
     in  
      $\langle (A_2)_x \cup \{x : S_2(A_1(x))\},$   
      $\lambda y. V_2[x := \rightarrow](y) \vee (V_2(x) \wedge S_2(V_1(y))),$   
      $\tau_2,$   
      $S_1; S_2 \rangle$ .

Figure 18: The Type Inference Algorithm for LET-polymorphic Reduction Types

and

$$S(A) \vdash_{S(V_2)}^L M_2 : S(\tau_2),$$

where  $A = S_2(A_1) \cup \{x : \sigma \mid x : \sigma \in A_2; x \notin \text{dom} A_1\}$ . Now the result follows from rule APP and by construction.

If  $M \equiv (\lambda x. P)Q$ , then by induction hypothesis

$$\langle A_1, V_1, \tau_1, S_1 \rangle = \text{TYPE}_L(A, Q)$$

implies

$$A_1 \vdash_{V_1}^L Q : \tau_1,$$

and  $\langle A_2, V_2, \tau_2, S_2 \rangle = \text{TYPE}_L((A_1)_x \cup \{x : \underline{\sigma}\}, P)$  implies

$$A_2 \vdash_{V_2}^L P : \tau_2.$$

Then the result follows by Lemma 5.3.3, Lemma 3.3.10 and construction.  $\square$

A principal triple for a term  $M$  is the most general type, variable neededness function and type assumption set that can be derived for  $M$  in a particular type assignment system:

### Definition 5.3.5

Let  $*$  be  $C$ ,  $L$  or  $I$ .  $\langle A, V, \tau \rangle$  is a *principal triple* for  $M$  if

1.  $A \vdash_V^* M : \tau$ , and
2. if  $A' \vdash_{V'}^* M : \tau'$ , then  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$ .

Note that the following Theorem entails the existence of principal triples for all typable terms.

### Theorem 5.3.6 (Completeness of $\text{TYPE}_L$ )

If for some  $A'$ ,  $V'$  and  $\tau'$  there is a deduction of  $A' \vdash_{V'}^L M : \tau'$  for  $M$ , then  $\text{TYPE}_L(A', M)$  is defined,  $\langle A, V, \tau, S \rangle = \text{TYPE}_L(A', M)$  and  $\langle A, V, \tau \rangle \leq \langle A', V', \tau' \rangle$ .

### Proof

By induction on the structure of  $M$ .

If  $M \equiv x$ , then clearly  $\text{TYPE}_L(A', M)$  is defined and  $\langle A', V_\infty[x := \Rightarrow], \tau \rangle \leq \langle A', V_\infty[x := \Rightarrow], \tau' \rangle$ , where  $\tau = \alpha$  if  $x \notin A'$ , else  $x : \underline{\sigma} \in A'$  and  $\underline{\sigma} \leq \tau \leq \tau'$ .

If  $M \equiv \lambda x. N$ , then the deduction of  $A' \vdash_{V'}^L M : \tau'$  must end in a use of the ABS rule whose antecedent has the form:

$$A'_x \cup \{x : \sigma\} \vdash_{V'[x := b']}^L N : \tau'',$$

for some types  $\sigma'$ ,  $\tau''$  and arrow expression  $b'$  such that  $\tau' = \sigma' b' \tau''$ . The result follows immediately by the induction hypothesis and by construction of  $\langle A, V, \tau, S \rangle$ .

If  $M \equiv M_1 M_2$ ,  $M_1$  not an abstraction term, and the deduction of  $A' \vdash_V^L M : \tau'$  ends in a use of the APP rule whose antecedents have the form:

$$A' \vdash_{V_1'}^L M_1 : \sigma' b' \tau'$$

and

$$A' \vdash_{V_2'}^L M_2 : \sigma',$$

where  $V_1'$ ,  $V_2'$  and  $b'$  are such that  $V = \lambda x. V_1'(x) \vee (b' \wedge V_2'(x))$ , then by the induction hypotheses both  $\text{TYPE}_L(A', M_1)$  and  $\text{TYPE}_L(A_1, M_2)$  are defined and

$$\langle A_1, V_1, \tau_1 \rangle \leq \langle A', V_1', \sigma' b' \tau' \rangle,$$

and

$$\langle A_2, V_2, \tau_2 \rangle \leq \langle A', V_2', \sigma' \rangle.$$

Then Lemma 5.2.8 applies and by Theorem 5.2.3,  $S$  solves

$$\{(S_2(\tau_1), \tau_2 \rightarrow; \alpha)\},$$

hence, by application of Theorem 5.2.4, the result can be obtained by using Lemma 5.3.3 and by construction.

If  $M \equiv (\lambda x. M_1) M_2$ , and the deduction of  $A' \vdash_V^L M : \tau'$  ends in a use of the APP- $\forall$  rule whose antecedents have the form:

$$A'_x \cup \{x : \underline{\sigma'}\} \vdash_{V_1'[x := b']}^L M_1 : \tau'$$

and

$$A' \vdash_{V_2'}^L M_2 : \sigma',$$

where  $V_1'$ ,  $V_2'$  and  $b'$  are such that  $V = \lambda y. V_1'(y) \vee (b' \wedge V_2'(y))$ , then by the induction hypotheses both  $\text{TYPE}_L(A, M_1)$  and  $\text{TYPE}_L(A_1, M_2)$  are defined and

$$\langle A_1, V_1, \tau_1 \rangle \leq \langle A'_x \cup \{x : \underline{\sigma'}\}, V_1'[x := b'], \tau' \rangle,$$

and

$$\langle A_2, V_2, \tau_2 \rangle \leq \langle A', V_2', \sigma' \rangle.$$

The result follows by construction and Lemma 5.3.3.

If  $M \equiv (\lambda x. P)Q$  and the deduction of a type for  $M$  ends in a use of rule APP, then by Lemmas 3.3.5 and 3.3.8 followed by the induction hypothesis

applied to  $P$  and  $Q$ , the result holds in a similar fashion to the previous case.  $\square$

### 5.3.4 Decidability

It is immediate from the fact that the unification procedure is decidable (see argument in previous section) that the above algorithm is a decision procedure for LET-polymorphically typable terms.

## 5.4 Implementation of the Intersection-style System

The implementation of the intersection-style system is by far the most complex of the implementations conducted. The work of Ronchi della Rocca and Venneri [59] and Ronchi della Rocca [58] has provided an excellent platform upon which to base the current section. However, there are some significant differences, in particular

- Boolean Reduction Types must be integrated into the implementation framework—some care is needed with the details of this, and
- the lack of a (derived)  $\beta$ -conversion rule means that a more detailed analysis of the operation of expansion (see below) is required in order to show that deductions are closed under this notion.

In Appendix A I have included an *Orwell* script which encodes the implementation described in this section in a form suitable for execution on a computer.

### 5.4.1 Substitution

As is now usual, we need to establish that deductions are closed under the operation of substitution.

#### Lemma 5.4.1

If  $\sigma \leq \tau$ , then  $S(\sigma) \leq S(\tau)$ .

#### Proof

By induction on the definition of  $\leq$ .  $\square$

#### Lemma 5.4.2

$A \vdash_V^I M : \tau$  implies  $S(A) \vdash_{S(V)}^I M : S(\tau)$ .

**Proof**

By induction on  $M$  using Lemma 5.4.1.  $\square$

**5.4.2 Normal Forms**

It will prove useful to prove many results with respect to a restricted set of terms as a preliminary step. Since the type assignment systems do not have an  $(\omega)$  rule the set of *normal forms* is used. This is in contrast with the treatments in Ronchi della Rocca and Venneri [59] and Ronchi della Rocca [58] in which the notion of *approximate normal form* is employed. (This latter notion is cited by Barendregt [3] as originating in Wadsworth [69] and has close connections with Böhm trees, see Barendregt [3], definition 14.3.5.ii).

**Definition 5.4.3**

The set of *normal forms*,  $\Lambda^{\text{NF}}$ , is inductively defined to be the least set satisfying:

- $x \in X$  implies  $x \in \Lambda^{\text{NF}}$ ,
- $x \in X, N_i \in \Lambda^{\text{NF}} (1 \leq i \leq n)$  implies  $xN_1 \dots N_n \in \Lambda^{\text{NF}}$ ,
- $x \in X, N \in \Lambda^{\text{NF}}$  implies  $\lambda x.N \in \Lambda^{\text{NF}}$ .

Clearly, the type assignment system for normal forms is just the same as the system for intersection-style Boolean Reduction Types as introduced in Chapter 3.

**5.4.3 Expansion of a Deduction**

The operation of expansion is introduced by Coppo, Dezani-Ciancaglini and Venneri [16]. (Our formulation and notation are different). This operation duplicates a subtree of a deduction and adds an instance of rule MEET to connect the duplicated subtrees. Some time is now spent investigating this operation with the intention of showing that deductions are closed under this operation.

An i-type is any type of the form  $\sigma \cap \tau$ . The concept of “type membership” needs to be extended, as follows:

**Definition 5.4.4**

Write  $\sigma \in \tau$  iff  $\tau = C[\sigma]$ , for some context  $C[\cdot]$  where  $\sigma$  is not an i-type. Define  $!\tau = \{\sigma \mid \sigma \in \tau\}$ . Let  $B$  be a set of types, then  $!B = \bigcup_{\tau \in B} !\tau$ .

**Definition 5.4.5**

The *expansion context* for  $\tau$  with respect to set of types  $B$ ,  $E_\tau^B$ , is defined to be the least set such that:



- $!\tau \subseteq E_\tau^B$ , and
- $\sigma \in E_\tau^B$ ,  $\rho \vdash \sigma \in !B$  implies  $!(\rho \vdash \sigma) \subseteq E_\tau^B$ .

**Definition 5.4.6**

Let  $B$  be a set of types containing  $\sigma$ , then the *expansion* of  $\sigma$  with respect to  $\tau$  and  $B$ ,  $e_\tau^B(\sigma)$ , is defined as follows. Let  $\rho_i$  ( $i \in \{1, \dots, n\}$ ,  $n \geq 0$ ) be the largest disjoint types<sup>5</sup> in  $E_\tau^B$  such that  $\rho_i \in \sigma$ . Let  $S_1$  and  $S_2$  be disjoint renamings of the type variables in  $E_\tau^B$  such that the ranges of  $S_1$  and  $S_2$  are disjoint from the type variables in  $E_\tau^B$ . Let  $\rho'_i = S_1(\rho_i)$  and  $\rho''_i = S_2(\rho_i)$ . Form  $e_\tau^B(\sigma)$  by replacing each  $\rho_i$  in  $\sigma$  by  $\rho'_i \cap \rho''_i$ . Write  $e_\tau^B(A)$  for

$$\{x : e_\tau^B(\sigma) \mid x : \sigma \in A\}$$

and write  $e_\tau^B(\langle A, V, \sigma \rangle)$  for

$$\langle e_\tau^B(A), V, e_\tau^B(\sigma) \rangle.$$

The following is an immediate consequence of Definition 5.4.6.

**Fact 5.4.7**

$$e_\rho^B(\sigma_1) \cap e_\rho^B(\sigma_2) = e_\rho^B(\sigma_1 \cap \sigma_2).$$

**Lemma 5.4.8**

Let  $B$  and  $B'$  be such that  $\sigma \in B$  implies  $\exists \tau \in B'. \sigma \in \tau$ . Let  $\{\rho_1, \dots, \rho_n\} = E_\rho^{B'}$  and  $\bar{\rho} = \bigcap_{i=1}^n \rho_i$ , then  $E_\rho^B = E_\rho^{B'}$  ( $\forall \mu \in !B'. e_\rho^B(\mu) = e_\rho^{B'}(\mu)$ ).

**Proof**

Immediate from Definition 5.4.5 (Definition 5.4.6).  $\square$

**Proposition 5.4.9**

Suppose  $\sigma \leq \tau$  ( $\sigma, \tau \neq \omega$ ) and  $\sigma, \tau \in B$ , then

1.  $\tau \in E_\rho^B$  implies  $\exists \sigma' \leq \tau. \sigma' \in E_\rho^B$  and  $\sigma = \sigma'$  or  $\exists \sigma''. \sigma = \sigma' \cap \sigma''$ , and
2.  $\tau \notin E_\rho^B$  implies  $\sigma \notin E_\rho^B$ .

**Proof**

1. If  $\tau = \alpha$  is a type variable then  $\sigma = \alpha$  or  $\sigma = \alpha \cap \sigma''$ . In this case, choose  $\sigma' = \alpha$ . Otherwise,

$$\begin{aligned} \tau &= \left( \bigcap_{i=1}^m \delta_i \vdash \epsilon_i \right) \cap \left( \bigcap_{i=1}^h \alpha_i \right), \\ \sigma &= \left( \bigcap_{i=1}^n \mu_i \vdash \nu_i \right) \cap \left( \bigcap_{i=1}^k \beta_i \right). \end{aligned}$$

<sup>5</sup>A type  $\sigma$  is larger than another type  $\tau$  if  $\tau \in \sigma$ .

Let  $j \in \{1, \dots, m\}$ . By Lemma 3.4.5,  $\exists j_1 \dots j_p. \bigcap_{i=1}^p \mu_{j_i} \geq \delta_j$ ,  $\bigcap_{i=1}^p \nu_{j_i} \leq \epsilon_j$ ,  $\bigcap_{i=1}^h \alpha_i \leq \bigcap_{i=1}^k \beta_i$  and  $\forall i (1 \leq i \leq p). b_j =_{BA} b_{j_i}$ . Let  $B_j = B \cup \{\epsilon_j\}$  and  $\bar{\rho}$  be constructed from  $E_\rho^B$  as in Lemma 5.4.8, then by the induction hypothesis:

$$\begin{aligned} \tau \in E_\rho^B & \text{ implies } \beta_j \in E_\rho^B \\ & \text{ implies } \beta_j \in E_{\bar{\rho}}^{B_j} \\ & \text{ implies } \exists i_1, \dots, i_r (\{i_1, \dots, i_r\} \subseteq \{j_1, \dots, j_p\}). \\ & \quad \nu_{i_1} \cap \dots \cap \nu_{i_r} \leq \beta_j \end{aligned}$$

and  $\forall s \in \{i_1, \dots, i_r\}$ :

$$\begin{aligned} \nu_{i_1} \cap \dots \cap \nu_{i_r} \in E_{\bar{\rho}}^{B_j} & \text{ implies } \mu_s \overset{!}{b}_s \nu_s \in E_{\bar{\rho}}^{B_j} \\ & \text{ implies } \mu_s \overset{!}{b}_s \nu_s \in E_\rho^B. \end{aligned}$$

Also,  $\tau \in E_\rho^B$  implies  $\alpha_i \in E_\rho^B$  ( $1 \leq i \leq h$ ). Let  $\{p_1, \dots, p_v\}$  be the set of indexes obtained by union of  $\{i_1, \dots, i_r\}$  for every  $j$ , then

$$\sigma' = \left( \bigcap_{i=1}^v (\mu_{p_i} \overset{!}{b}_{p_i} \nu_{p_i}) \right) \cap \left( \bigcap_{i=1}^h \alpha_i \right).$$

Clearly,  $\sigma' \leq \tau$ .

2. This is immediate if  $\tau = \alpha$ . Otherwise, the result follows since  $\exists !\sigma' \subset !\tau. \sigma' \notin E_\rho^B$ .

□

#### Lemma 5.4.10

Suppose  $\sigma \leq \tau$  ( $\sigma, \tau \neq \omega$ ) and  $\sigma, \tau \in B$ , then  $e_\rho^B(\sigma) \leq e_\rho^B(\tau)$ .

#### Proof

Immediate by Property 5.4.9 if  $\tau \in E_\rho^B$ . Otherwise, if  $\tau = \alpha$ , then  $\sigma = \alpha$  or  $\sigma = \alpha \cap \sigma'$  and the result follows. Otherwise,

$$\begin{aligned} \tau &= \left( \bigcap_{i=1}^m \delta_i \overset{!}{b}_i \epsilon_i \right) \cap \left( \bigcap_{i=1}^h \alpha_i \right), \\ \sigma &= \left( \bigcap_{i=1}^n \mu_i \overset{!}{b}_i \nu_i \right) \cap \left( \bigcap_{i=1}^k \beta_i \right). \end{aligned}$$

Suppose  $m = 1$  and  $h = 0$ , i.e.,  $\tau = \delta_1 \overset{!}{b}_1 \epsilon_1$ . By Lemma 3.4.5,

$$\exists j_1 \dots j_p. \bigcap_{i=1}^p \mu_{j_i} \geq \delta_1,$$

and

$$\bigcap_{i=1}^p \nu_{j_i} \leq \epsilon_1$$

and  $\forall i (1 \leq i \leq p). b_1 =_{BA} b_{j_i}$ . Let  $\bar{\rho}$  be constructed from  $E_{\bar{\rho}}^B$  as in Lemma 5.4.8, then

1.  $e_{\bar{\rho}}^B(\delta_1) \leq e_{\bar{\rho}}^B(\mu_{j_i})$ , by induction hypothesis if  $\mu_{j_i} \notin E_{\bar{\rho}}^{B \cup \{\delta_1, \mu_{j_i}\}} = E_{\bar{\rho}}^B$ , otherwise by Property 5.4.9. Thus,  $e_{\bar{\rho}}^B(\delta_1) \leq \bigcap_{i=1}^n e_{\bar{\rho}}^B(\mu_{j_i})$ ,
2.  $\bigcap_{i=1}^n e_{\bar{\rho}}^B(\nu_{j_i}) \leq e_{\bar{\rho}}^B(\epsilon_1)$ , by induction hypothesis.

Thus,

$$\begin{aligned} e_{\bar{\rho}}^B(\delta_1 \mathbin{b}_1 \epsilon_1) &= e_{\bar{\rho}}^B(\delta_1) \mathbin{b}_1 e_{\bar{\rho}}^B(\epsilon_1) \geq \left( \bigcap_{i=1}^p e_{\bar{\rho}}^B(\mu_{j_i}) \right) \mathbin{b}_1 \left( \bigcap_{i=1}^p e_{\bar{\rho}}^B(\nu_{j_i}) \right) \\ &\geq \bigcap_{i=1}^p e_{\bar{\rho}}^B(\mu_{j_i}) \mathbin{b}_{j_i} e_{\bar{\rho}}^B(\nu_{j_i}). \end{aligned}$$

For all  $i$  ( $1 \leq i \leq p$ ),

$$e_{\bar{\rho}}^B(\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i}) \leq e_{\bar{\rho}}^B(\mu_{j_i}) \mathbin{b}_{j_i} e_{\bar{\rho}}^B(\nu_{j_i}),$$

since  $\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i} \in E_{\bar{\rho}}^B$  implies  $e_{\bar{\rho}}^B(\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i}) = e_{\bar{\rho}}^B(\mu_{j_i}) \mathbin{b}_{j_i} e_{\bar{\rho}}^B(\nu_{j_i})$ , and  $\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i} \notin E_{\bar{\rho}}^B$  implies  $e_{\bar{\rho}}^B(\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i}) < e_{\bar{\rho}}^B(\mu_{j_i}) \mathbin{b}_{j_i} e_{\bar{\rho}}^B(\nu_{j_i})$ . Thus,

$$e_{\bar{\rho}}^B(\tau) \geq \bigcap_{i=1}^p e_{\bar{\rho}}^B(\mu_{j_i} \mathbin{b}_{j_i} \nu_{j_i}) \geq e_{\bar{\rho}}^B(\sigma).$$

Now the result for this case follows from Lemma 5.4.8.

The case  $m > 1, h > 0$  follows by an easy induction.  $\square$

Write  $\bar{A}$  for  $\{\sigma \mid x: \sigma \in A\}$ .

#### Lemma 5.4.11

Let  $N \in \Lambda^{\text{NF}}$ ,  $A \vdash_V^I N: \tau$ ,  $B = \bar{A} \cup \{\tau\}$  and  $\tau \in E_{\bar{\rho}}^B$ , then there exist  $A'. A_1, A_2, \tau_1, \tau_2$  such that  $e_{\bar{\rho}}^B(A) = A' \supseteq A_1 \mathbin{\frown} A_2$ ,  $e_{\bar{\rho}}^B(\tau) = \tau_1 \cap \tau_2$  and  $A_i \vdash_V^I N: \tau_i$ , for  $i \in \{1, 2\}$ .

#### Proof

By induction on  $N$ . Suppose  $N \equiv x$ , then, for some  $\sigma \in T_I^\nabla$ ,  $x: \sigma \in A$ ,  $\sigma \leq \tau$  and  $\sigma, \tau \in B$ . Clearly  $A_i = \{x: \sigma_i\}$  ( $i \in \{1, 2\}$ ), where  $\sigma_i$  is a trivial variant of  $\sigma$ .

If  $N \equiv \lambda x. N'$ , then by Lemma 3.4.17  $\tau = \tau_1 \cap \tau_2 \cap \dots \cap \tau_n$  ( $n \geq 1$ ), where  $\tau_i = \mu_i \mathbin{b}_i \nu_i$  ( $1 \leq i \leq n$ ). By induction on  $n$ . If  $n = 1$ , then by

Lemma 3.4.14  $A \vdash_V^I \lambda x. N' : \mu_1 \mathbf{b}_1 \nu_1$  implies  $A_x \cup \{x : \mu_1\} \vdash_{V[x:=b_1]}^I N : \nu_1$ . Let  $B' = \bar{A}_x \cup \{\mu_1, \nu_1\}$  and  $\bar{\rho}$  be constructed from  $E_\rho^B$  as in Lemma 5.4.8. The induction hypothesis applies and so  $e_{\bar{\rho}}^{B'}(A_x \cup \{x : \mu_1\}) = A'' \supseteq A_1'' \cap A_2''$  and  $e_{\bar{\rho}}^{B'}(\nu_1) = \nu_1' \cap \nu_1''$ . Then the result follows from ABS and Lemma 5.4.8. The generalisation to  $n > 1$  is straightforward by Fact 5.4.7.

If  $N \equiv x N_1 N_2 \dots N_n$ , then by Corollary 3.4.16:

- $A \vdash_{V_i}^I N_i : \rho_i, (1 \leq i \leq n),$
- $A \vdash_V^I x : \rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \sigma,$  and
- $V = \lambda x. V'(x) \vee (\bigvee_{i=1}^n (b_i \wedge V_i(x))),$

where  $\sigma \leq \tau$ . Let  $B_i = \bar{A} \cup \{\rho_i\}$ ,  $B' = \bar{A} \cup \{\rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \tau\}$  and let  $\bar{\rho}$  be constructed from  $E_\rho^{B'}$ , as in Lemma 5.4.8. Then by the induction hypotheses:

- $e_{\bar{\rho}}^{B'}(A) = A' \supseteq A_1 \cap A_2$  and

$$e_{\bar{\rho}}^{B'}(\rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \tau) = \rho_1^i \mathbf{b}_1 \rho_2^i \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n^i \mathbf{b}_n \tau^i,$$

for  $i \in \{1, 2\}$ , and

- $e_{\bar{\rho}}^{B_i}(\rho_i) = \rho_i^j$  and  $A_j \vdash_{V_i}^I N_i : \rho_i^j$ , for  $1 \leq i \leq n$  and  $j \in \{1, 2\}$ .

Note that  $e_{\bar{\rho}}^{B'}(A) = e_{\bar{\rho}}^{B_i}(A)$ . By Lemma 5.4.8 and since  $\tau \in E_\rho^B$ ,

$$e_{\bar{\rho}}^{B'}(A) \vdash_{V_i}^I N_i : e_{\bar{\rho}}^{B_i}(\rho_i),$$

and hence

$$e_{\bar{\rho}}^{B'}(A) \vdash_V^I N : e_{\bar{\rho}}^{B'}(\tau).$$

Note that either

$$x : \rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \sigma \in A$$

or

$$x : (\rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \sigma) \cap \sigma' \in A,$$

for some  $\sigma'$ , thus by Property 5.4.9,  $\tau \in E_\rho^B$  implies

$$\rho_1 \mathbf{b}_1 \rho_2 \mathbf{b}_2 \dots \mathbf{b}_{n-1} \rho_n \mathbf{b}_n \sigma \in E_\rho^B$$

and so  $E_\rho^B = E_\rho^{B'}$ . Then the result follows from the ABS rule.  $\square$

**Lemma 5.4.12**

Suppose  $N \in \Lambda^{\text{NF}}$ ,  $A \vdash_V^I N : \tau$  and  $B = \bar{A} \cup \{\tau\}$ , then  $e_\rho^B(A) \vdash_V^I N : e_\rho^B(\tau)$ .

**Proof**

If  $\tau \in E_\rho^B$ , then the result holds by Lemma 5.4.11. Otherwise, by induction on  $N$ .

If  $N \equiv x$ , then  $x : \sigma \in A$ ,  $\sigma \leq \tau$  and  $\sigma, \tau \in B$ . The result follows immediately by Lemma 5.4.10.

If  $N \equiv \lambda x.N'$ , then by Lemma 3.4.17  $\tau = \tau_1 \cap \tau_2 \cap \dots \cap \tau_n$  ( $n \geq 1$ ), where  $\tau_i = \mu_i b_i \nu_i$  ( $1 \leq i \leq n$ ). By induction on  $n$ . If  $n = 1$ , then by Lemma 3.4.14  $A \vdash_V^I \lambda x.N' : \mu_1 b_1 \nu_1$  implies  $A_x \cup \{x : \mu_1\} \vdash_{V[x:=b_1]}^I N' : \nu_1$ . Let  $B' = \bar{A}_x \cup \{\mu_1, \nu_1\}$  and  $\bar{\rho}$  be constructed from  $E_\rho^B$  as in Lemma 5.4.8. By the induction hypothesis  $e_{\bar{\rho}}^{B'}(A_x \cup \{x : \mu_1\}) \vdash_{V[x:=b_1]}^I N' : e_{\bar{\rho}}^{B'}(\nu_1)$ . By rule ABS,  $e_{\bar{\rho}}^{B'}(A) \vdash_V^I \lambda x.N' : e_{\bar{\rho}}^{B'}(\mu_1) b e_{\bar{\rho}}^{B'}(\nu_1)$ . Since  $\tau \notin E_\rho^B$ ,  $e_{\bar{\rho}}^{B'}(\mu_1) b e_{\bar{\rho}}^{B'}(\nu_1) = e_{\bar{\rho}}^{B'}(\mu_1 b \nu_1)$ , and so  $e_{\bar{\rho}}^{B'}(A) \vdash_V^I \lambda x.N' : e_{\bar{\rho}}^{B'}(\mu_1 b \nu_1)$ . The result follows from Lemma 5.4.8. The case  $n > 1$  is straightforward by induction.

If  $N \equiv xN_1N_2\dots N_n$ , then by Corollary 3.4.16:  $A \vdash_{V_i}^I N_i : \rho_i$ , ( $1 \leq i \leq n$ ),  $A \vdash_V^I x : \rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \sigma$ , and  $V = \underline{\lambda}x.V'(x) \vee (\bigvee_{i=1}^n (b_i \wedge V_i(x)))$ , where  $\sigma \leq \tau$ . Let  $B_i = \bar{A} \cup \{\rho_i\}$  and  $B' = \bar{A} \cup \{\rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau\}$ , then by the induction hypotheses:

- $e_\rho^{B'}(A) \vdash_V^I x : e_\rho^{B'}(\rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau)$ , and
- $e_\rho^{B_i}(A) \vdash_{V_i}^I N_i : e_\rho^{B_i}(\rho_i)$ , for  $1 \leq i \leq n$ .

Let  $\bar{\rho}$  be constructed from  $E_\rho^B$ , as in Lemma 5.4.8. Since  $\tau \notin E_\rho^B$ ,

$$\rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau \notin E_\rho^{B'}$$

and  $E_\rho^B = E_\rho^{B'} = E_\rho^{B_i}$ , using Lemma 5.4.8. Thus,

- $e_\rho^B(A) \vdash_V^I x : e_\rho^B(\rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau)$ , and
- $e_\rho^B(A) \vdash_{V_i}^I N_i : e_\rho^B(\rho_i)$ , for  $1 \leq i \leq n$ ,

and  $e_\rho^B(\rho_1 b_1 \rho_2 b_2 \dots b_{n-1} \rho_n b_n \tau) = e_\rho^B(\rho_1) b_1 e_\rho^B(\rho_2) b_2 \dots b_{n-1} e_\rho^B(\rho_n) b_n e_\rho^B(\tau)$ . Then,  $e_\rho^B(A) \vdash_V^I N : e_\rho^B(\tau)$  by repeated use of rule APP.

□

The following Lemma provides useful insight into how deductions may be structured in the Intersection-based systems.

**Lemma 5.4.13**

Suppose  $A_x \cup \{x : \sigma\} \vdash_{V_1}^I P : \tau$ ,  $x \in \text{FV}(P)$  and  $A \vdash_{V_2}^I Q : \sigma$  where  $x \notin \text{FV}(Q)$ , then there is a deduction of  $A \vdash_V^I P[x := Q] : \tau$ , with  $V = \underline{\lambda}y.V_1(y) \vee (V_2(x) \wedge$

$V_2(y))$ , (such a deduction exists by Theorem 3.4.22) in which each occurrence of a deduction of a type for  $Q$  either has the form  $A \vdash_{V_2}^I Q : \sigma$  or  $Q$  is irrelevant to the deduction.

### Proof

By induction on the structure of  $A_x \cup \{x : \sigma\} \vdash_{V_1}^I P : \tau$ . Since the cases for either of rule LEQ or MEET are easy by induction only the structure of  $P$  need be considered.

In the case  $P \equiv x$ , the result holds immediately ( $V =_{BA} V_2, \sigma \leq \tau$ ).

The case  $P \equiv y$ , for  $y \neq x$ , is not allowed by statement of the lemma.

In the case  $P \equiv P_1 P_2$ , and rule APP is used to deduce a type for  $P$ , by Lemma 3.4.12  $\exists \sigma' \in T_I^\nabla, b \in \nabla. (A \vdash_{V'}^I P_1 : \sigma' \text{ b } \tau, A \vdash_{V''}^I P_2 : \sigma' \text{ and } V_1 = \lambda x. V'(x) \vee (b \wedge V''(x)))$ . From the definition of substitution for terms it holds that  $(P_1 P_2)[x := Q] \equiv (P_1[x := Q])(P_2[x := Q])$ . Hence the induction hypothesis applies to the deductions of  $P_1[x := Q]$  and  $P_2[x := Q]$ , namely that the deductions of types for  $Q$  in these terms have the form  $A \vdash_{V_2}^I Q : \sigma$ . Then the result follows by rule APP and the definition of substitution.

In the case  $P \equiv P_1 P_2$ , and rule APP $\rightarrow$  is used to deduce a type for  $P$ , the result follows in a similar fashion except that occurrences of  $Q$  in  $P_2[x := Q]$  are irrelevant in the deduction of  $P[x := Q]$ , as required.

The case  $P \equiv \lambda y. N$ ,  $y \neq x$ , follows in a similar manner to the case of rule APP (only one instance of the induction hypothesis).

The case  $P \equiv \lambda x. N$  is not allowed by statement of the lemma. (And the convention for bound variables).  $\square$

### Lemma 5.4.14

If  $A \vdash_V^I (\lambda x. P)Q : \tau$  and  $e_\rho^B(A) \vdash_V^I P[x := Q] : e_\rho^B(\tau)$ , where  $\bar{A} \cup \{\tau\} \subseteq B$ , then  $e_\rho^B(A) \vdash_V^I (\lambda x. P)Q : e_\rho^B(\tau)$ .

### Proof

If  $x \notin \text{FV}(P)$ , then the result holds immediately by Theorem 3.4.26.

Otherwise, by Theorem 3.4.22,  $A \vdash_V^I (\lambda x. P)Q : \tau$  implies  $A \vdash_V^I P[x := Q] : \tau$ .

If rule APP is used to deduce a type for  $(\lambda x. P)Q$ , then by Lemma 3.4.12,

$$\exists \sigma \in T_I^\nabla, b \in \nabla. (A \vdash_{V'}^I \lambda x. P : \sigma \text{ b } \tau,$$

$$A \vdash_{V''}^I Q : \sigma$$

and  $V = \lambda y. V'(y) \vee (b \wedge V''(y))$ . So Lemma 5.4.13 applies and there is a deduction for  $A \vdash_V^I P[x := Q] : \tau$  such that each occurrence of  $Q$  either has a deduction of the form  $A \vdash_{V''}^I Q : \sigma$  or is irrelevant. Then for this deduction in particular, by statement of the lemma,  $e_\rho^B(A) \vdash_{V''}^I Q : e_\rho^B(\sigma)$ . Then

Theorem 3.4.24 applies (Theorem 3.4.25 or, if rule APP- $\rightarrow$  is admitted, Theorem 3.4.26 (directly applicable) for irrelevant occurrences of  $Q$ ) and the result follows.

In the case that rule APP- $\rightarrow$  is used to deduce a type for  $(\lambda x.P)Q$ , by Lemma 3.4.13,  $\exists A' \in \{X \times T_I^\nabla\}. (A \vdash_V^I, \lambda x.P : \omega \rightarrow \tau \text{ and } A' \vdash_V^{I''} Q : \omega)$ . (Here  $\nabla$  may not be  $\{\Rightarrow\}$ ). Then the result follows similarly using Lemma 5.4.13 except that *all* occurrences of  $Q$  are irrelevant (i.e., use Theorem 3.4.26 at each occurrence).

□

#### Theorem 5.4.15

Suppose  $A \vdash_V^I M : \tau$  and  $M$  has a normal form, then  $e_\rho^B(A) \vdash_V^I M : e_\rho^B(\tau)$ , where  $B = \bar{A} \cup \{\tau\}$ .

#### Proof

If  $\tau \equiv \omega$ , then the result follows easily since  $e_\rho^B(\omega) = \omega$  and by the use of instances of rule LEQ as appropriate after each instance of rule VAR.

Let  $N$  be the normal form of  $M$ . By Theorem 3.4.22,  $A \vdash_V^I M : \tau$  implies  $A \vdash_V^I N : \tau$ , thus, by Lemma 5.4.12,  $e_\rho^B(A) \vdash_V^I N : e_\rho^B(\tau)$ .

The result follows by iterated use of Lemma 5.4.14 on the reduction path from  $M$  to  $N$ . □

#### Definition 5.4.16

A chain,  $c$ , is a finite composition of substitutions and expansions. Write  $\langle A, V, \tau \rangle \rightarrow_c \langle A', V', \tau' \rangle$  iff  $\langle A', V', \tau' \rangle = c(\langle A, V, \tau \rangle)$ .

#### Corollary 5.4.17

Suppose  $A \vdash_V^I M : \tau$  and  $\langle A, V, \tau \rangle \rightarrow_c \langle A', V', \tau' \rangle$ , then  $A' \vdash_V^I M : \tau'$ , where  $M$  has a normal form.

#### Proof

By iterated application of Theorem 5.4.15 and Lemma 5.4.2. □

### 5.4.4 Principal Triples

The principal triple of a term in  $\Lambda^{\text{NF}}$  can be defined as follows. This particular formulation of the notion of principal triple will serve as a useful first check in showing that the algorithms in the following sections are correct.

#### Definition 5.4.18

Let  $N \in \Lambda^{\text{NF}}$ , then the *principal triple* of  $N$ ,  $\text{pt}(N)$ , is defined as follows:

1. if  $N \equiv x$ , then  $\text{pt}(N) = (\{x : \alpha\}, V_\omega[x := \Rightarrow], \alpha)$ , where  $\alpha$  is a type variable;

2. if  $N \equiv \lambda x.P$  and  $\text{pt}(P) = \langle A, V[x := b], \tau \rangle$ , then if  $x: \sigma \in A$ , for some  $\sigma$ , then  $\text{pt}(N) = \langle A_x, V[x := \rightarrow], \sigma \text{ b } \tau \rangle$  else  $\text{pt}(N) = \langle A, V[x := \rightarrow], \alpha \text{ b } \tau \rangle$ , where  $\alpha$  is a type variable which does not occur in  $A$ , and
3. if  $N \equiv xN_1 \dots N_n$  and  $\text{pt}(N_i) = \langle A_i, V_i, \tau_i \rangle$  ( $1 \leq i \leq n$ ), ( $\langle A_i, V_i, \tau_i \rangle$  is chosen so that all arrow and type variables are disjoint from  $\langle A_j, V_j, \tau_j \rangle$ ,  $i \neq j$ ,  $1 \leq i, j \leq n$ ), then  $\text{pt}(N) = \langle (\bigcap_{i=1}^n A_i) \cap \{x: \tau_1 \rightarrow_1 \dots \rightarrow_{n-1} \tau_n \rightarrow_n \alpha\}, V, \alpha \rangle$ , where the  $\rightarrow_i$  are arrow variables which do not occur anywhere in  $A_j$  and  $V_j$  ( $1 \leq i, j \leq n$ ),  $\alpha$  is a type variable which does not occur in any  $A_i$  ( $1 \leq i \leq n$ ) and  $\forall y \in X. V = \lambda y. (V_{\rightarrow}[x := \Rightarrow])(y) \vee (\bigwedge_{i=1}^n \rightarrow_i \wedge V_i)$ .

Let  $\mathcal{PT} = \{\text{pt}(N) | N \in \Lambda^{\text{NF}}\}$ .

#### Lemma 5.4.19

Let  $N \in \Lambda^{\text{NF}}$ . Suppose  $\text{pt}(N) = \langle A, V, \tau \rangle$ , then

1.  $\tau \in E_\rho^B$  implies  $B \subseteq E_\rho^B$ , where  $B = \bar{A} \cup \{\tau\}$ , and
2. if  $\langle A, V, \tau \rangle \rightarrow_c \langle A', V, \tau' \rangle$  and  $c$  is a chain of expansions, then  $\tau' \in E_\rho^{B'}$  implies  $B' \subseteq E_\rho^{B'}$ , where  $B' = \bar{A}' \cup \{\tau'\}$ .

#### Proof

By induction on  $N$  using Definition 5.4.18 and Lemma 5.4.11.  $\square$

### 5.4.5 Unification of Intersection Types

Since the set of types which may appear in the APP rule is unrestricted, and these types are from  $T_I^\nabla$ , it is necessary to define a new unification algorithm. However, the set of arrow expressions remains unchanged and so algorithm BUNIFY requires no enhancement.

The function below is specified *sequentially*, i.e., the intention is that each of the clauses in the definition will be tried in ascending numerical sequence. This allows a more compact specification of the algorithm as it is then not necessary to carefully ensure that all cases are disjoint. This practice has become common in many pattern-matching based functional programming languages (see Turner [64] or Hudak et al [52], for example).

The function  $\text{UNIFY}_I: T_I^\nabla \times T_I^\nabla \times \{X \times T_I^\nabla\} \rightarrow T_I^\nabla \rightarrow T_I^\nabla$  is defined by  $\text{UNIFY}_I(\sigma, \tau, A) = U_I(\sigma, \tau, \bar{A} \cup \{\sigma, \tau\})$ , where  $U_I$  is defined in Figure 19.

The following useful examples are based on those which appear in Ronchi della Rocca [58].

#### Example 5.4.20

Consider  $\text{UNIFY}_I(\sigma, \tau, \emptyset) = U_I(\sigma, \tau, \{\sigma, \tau\})$ , where  $\sigma = \alpha \rightarrow_i \beta$  and  $\tau = \gamma \cap \delta$ .



To calculate the value of  $U_I(\sigma, \tau, \{\sigma, \tau\})$  the unification function must perform an expansion  $e_{\sigma}^{\{\sigma, \tau\}}$ . Let  $B = \{\sigma, \tau\}$ , then  $e_{\sigma}^B(\sigma) = (\alpha' \rightarrow_i \beta') \cap (\alpha'' \rightarrow_i \beta'')$ , where  $\alpha', \alpha'', \beta'$  and  $\beta''$  are fresh type variables, and  $e_{\sigma}^B(\tau) = \tau$ .

Now substitutions  $S_1 = \text{Id}[\gamma := \alpha' \rightarrow_i \beta']$  and  $S_2 = \text{Id}[\delta := \alpha'' \rightarrow_i \beta'']$  are computed and the final result is  $S = e_{\sigma}^B; S_1; S_2$ .

Verifying,  $S(\sigma) = (\alpha' \rightarrow_i \beta') \cap (\alpha'' \rightarrow_i \beta'') = S(\tau)$ .

As will soon become clear, the following example arises in attempting to infer a type for the term  $(\lambda x.xx)(\lambda x.xx)$ .

#### Example 5.4.21

Consider  $\text{UNIFY}_I(\alpha \cap (\alpha \rightarrow_i \beta), (\gamma \cap (\gamma \rightarrow_j \delta)) \Rightarrow \delta, \emptyset)$ , then this process does not stop.

The operations performed commence by expanding  $\tau = (\gamma \cap (\gamma \rightarrow_j \delta)) \Rightarrow \delta$  to obtain  $\tau' \cap \tau'' = (\gamma' \cap (\gamma' \rightarrow_j \delta')) \Rightarrow \delta' \cap (\gamma'' \cap (\gamma'' \rightarrow_j \delta'')) \Rightarrow \delta''$ . Now the substitution  $S_1 = \text{Id}[\alpha := \tau']$  is computed and  $U_I(S_1(\alpha) \rightarrow_i \beta, \tau'', \emptyset)$  is required, which in turn requires  $U_I(S_1(\alpha) = \tau', \gamma'' \cap (\gamma'' \rightarrow_j \delta''), \emptyset)$  and so on, ad infinitum.

#### UNIFY<sub>I</sub> is Correct

##### Definition 5.4.22

Let  $\sigma' \in \sigma$  and  $\tau' \in \tau$ , then  $\sigma'$  and  $\tau'$  are *corresponding* in  $\sigma$  and  $\tau$  iff

- $\sigma' \equiv \sigma$  and  $\tau' \equiv \tau$ , or
- $\sigma \equiv \sigma_1 b_1 \sigma_2$  and  $\tau \equiv \tau_1 b_2 \tau_2$  ( $\sigma \equiv \sigma_1 \cap \sigma_2$  and  $\tau \equiv \tau_1 \cap \tau_2$ ) and  $\sigma'$  and  $\tau'$  are corresponding in  $\sigma_1$  and  $\tau_1$  or  $\sigma_2$  and  $\tau_2$ .

##### Lemma 5.4.23

If  $\text{UNIFY}_I(\sigma, \tau, A) = c$  is defined and  $c$  is a chain of substitutions, then  $c(\sigma) = c(\tau)$  and if  $\sigma' \in \sigma$  and  $\tau' \in \tau$  have corresponding occurrences in  $\sigma$  and  $\tau$ , then  $c(\sigma')$  and  $c(\tau')$  have corresponding occurrences in  $c(\sigma)$  and  $c(\tau)$ .

##### Proof

If  $\sigma$  is a type variable, then if  $\sigma = \tau$  then  $\text{UNIFY}_I(\sigma, \tau, A) = \text{Id}$  and the result follows immediately. Otherwise, it must be the case that  $\sigma \notin \tau$  since  $\text{UNIFY}(\sigma, \tau)$  is defined, then  $\text{UNIFY}_I(\sigma, \tau, A) = \text{Id}[\alpha := \tau]$  and the result holds.

Since  $c$  is a chain of substitutions, the case of either  $\sigma$  or  $\tau$  an i-type is not possible.

Suppose  $\sigma = \sigma_1 b_1 \sigma_2$  and  $\tau = \tau_1 b_2 \tau_2$ . Since  $\text{UNIFY}_I(\sigma, \tau, A)$  is defined it must be true that  $\text{BUNIFY}(b_1, b_2) = R$  is defined. By Theorem 5.1.6,

1.  $U_I(\sigma_1 \text{ b1 } \sigma_2, \tau_1 \text{ b2 } \tau_2, B)$   
 $= R; c_1; c_2$   
 where  
 $R = \text{BUNIFY}(b_1, b_2)$   
 $c_1 = U_I(R(\sigma_1), R(\tau_1), R(B))$   
 $c_2 = U_I((R; c_1)(\sigma_2), (R; c_1)(\tau_2), (R; c_1)(B))$
2.  $U_I(\sigma_1 \cap \sigma_2, \tau_1 \cap \tau_2, B)$   
 $= c_1; c_2$   
 where  
 $c_1 = U_I(\sigma_1, \tau_1, B)$   
 $c_2 = U_I(c_1(\sigma_2), c_1(\tau_2), c_1(B))$
3.  $U_I(\sigma, \tau, B)$ , where  $\sigma$ , but not  $\tau$ , is an i-type  
 $= e_\tau^B; c$   
 where  
 $c = U_I(e_\tau^B(\sigma), e_\tau^B(\tau), e_\tau^B(B))$
4.  $U_I(\alpha, \tau, B)$   
 $= \text{if } \alpha \in \tau \text{ and } \alpha \neq \tau \text{ then Id}[\alpha := \omega][\tau := \omega] \text{ else Id}[\alpha := \tau]$
5.  $U_I(\omega, \tau, B) = \text{Id}[\tau := \omega]$
6.  $U_I(\sigma, \tau, B) = U_I(\tau, \sigma, B)$

Figure 19: The Algorithm for unifying Intersection Reduction Types

$R(b_1) =_{BA} R(b_2)$ . Since  $(R; c_1)$  is a chain of substitutions, the induction hypothesis applies and so  $(R; c_1)(\sigma_1)$  and  $(R; c_1)(\tau_1)$  have corresponding instances in  $(R; c_1)(\sigma)$  and  $(R; c_1)(\tau)$ , and

$$(R; c_1)(\sigma_1) = (R; c_1)(\tau_1).$$

Similarly, both  $(R; c_1; c_2)(\sigma_2)$  and also  $(R; c_1; c_2)(\tau_2)$  have corresponding instances in  $(R; c_1; c_2)(\sigma)$  and  $(R; c_1; c_2)(\tau)$ , and

$$(R; c_1; c_2)(\sigma_2) = (R; c_1; c_2)(\tau_2).$$

Then the case follows.

The case  $\sigma = \sigma_1 \cap \sigma_2$  and  $\tau = \tau_1 \cap \tau_2$  follows by induction directly.  $\square$

**Theorem 5.4.24 (Soundness of UNIFY<sub>I</sub>)**

If  $\text{UNIFY}_I(\sigma, \tau, A) = c$ , then  $c(\sigma) = c(\tau)$ .

**Proof**

If  $c$  is a chain of substitutions, then by Lemma 5.4.23. Suppose  $\sigma$ , but not  $\tau$ , is an i-type. In this case  $c = e_\rho^B; c'$ , where  $c' = \text{U}_I(e_\rho^B(\sigma), e_\rho^B(\tau), e_\rho^B(B))$ . Then the result follows directly from the induction hypothesis. The other cases also immediately follow by induction.  $\square$

Given Theorem 5.4.15, Lemma 5.4.2 and Theorem 5.1.6, the following Completeness Theorem holds. The proof of this Theorem is very intricate, but is essentially identical to that given in the Appendix to Ronchi della Rocca [58]. Note that the optimisations described there have already been incorporated into the current unification algorithm.

**Theorem 5.4.25 (Completeness of UNIFY<sub>I</sub>)**

Let  $\langle A_i, V_i, \tau_i \rangle \in \mathcal{ST}$ ,  $i \in \{1, 2\}$ . If there exists a chain,  $c$ , such that  $c(\tau_1) = c(\tau_2)$ , then  $\text{UNIFY}_I(\tau_1, \tau_2, A_1 \mathbin{\frown} A_2) = c'$  is defined,  $c' \leq c$  and  $c'(\tau_1) = c'(\tau_2)$ .

### 5.4.6 The Type Inference Algorithm

The function  $\text{TYPE}_I: \Lambda \rightarrow \{X \times (X \rightarrow \nabla) \times T_I^\nabla\}$  is defined in Figure 20.

**Example 5.4.26**

Consider the typing of the term  $(\lambda x.xx)(xy)$ .  $\text{TYPE}_I(\lambda x.xx) = \langle \emptyset, V_\infty, (\alpha \cap (\alpha \rightarrow_1 \beta)) \Rightarrow \beta \rangle$ , and  $\text{TYPE}_I(xy) = \langle \{x: \mu \rightarrow_2 \nu, y: \mu\}, V_\infty[x := \Rightarrow], \nu \rangle$ .

Now  $\text{TYPE}_I$  will make a call of  $\text{UNIFY}_I((\alpha \cap (\alpha \rightarrow_1 \beta)) \Rightarrow \beta, \nu \rightarrow_3 \epsilon, A)$ , where  $A = \{x: \mu \rightarrow_2 \nu, y: \mu\}$ . Then,  $\text{UNIFY}_I((\alpha \cap (\alpha \rightarrow_1 \beta)) \Rightarrow \beta, \nu \rightarrow_3 \epsilon, A) = c_1; e_{c_1(\nu)}^{c_1(B)}; c_2$ , where  $B = \bar{A} \cup \{(\alpha \cap (\alpha \rightarrow_1 \beta)) \Rightarrow \beta, \nu \rightarrow_3 \epsilon\}$  and  $e_{c_1(\nu)}^{c_1(B)}$  is the expansion necessary to unify  $c_1(\nu)$  and  $c_1(\alpha \cap (\alpha \rightarrow_1 \beta))$ . Note that  $c_1(\mu \rightarrow_2 \nu) \in !c_1(B)$ .

1.  $\text{TYPE}_I(x) = \langle \{x : \alpha\}, V_{\rightarrow}[x := \Rightarrow], \alpha \rangle$   
(where  $\alpha$  is a new type variable)
2.  $\text{TYPE}_I(\lambda x.N) = \langle A_x, V[x := \rightarrow], \sigma V(x) \tau \rangle$   
where  
 $\langle A, V, \tau \rangle = \text{TYPE}_I(N)$   
and  
 $\sigma = \text{if } x \in \text{dom} A \text{ then } A(x) \text{ else } \alpha$   
(where  $\alpha$  is a new type variable)
3.  $\text{TYPE}_I(N_1 N_2) = c(\langle A, V, \alpha \rangle)$   
where  
 $\langle A_1, V_1, \sigma_1 \rangle = \text{TYPE}_I(N_1)$   
 $\langle A_2, V_2, \sigma_2 \rangle = \text{TYPE}_I(N_2)$   
 $V = \lambda x. V_1(x) \vee (\rightarrow_i \wedge V_2(x))$   
 $c = \text{UNIFY}_I(\sigma_1, \sigma_2 \rightarrow_i \alpha, A_1 \mathbin{\mathbb{M}} A_2)$   
(where  $\rightarrow_i$  and  $\alpha$  are new arrow and type variables, respectively)

Figure 20: The Type Inference Algorithm for Intersection Reduction Types

Then,  $\text{TYPE}_I((\lambda x.xx)(xy)) = \langle \{x : (\alpha_1 \rightarrow_2 \alpha) \cap (\alpha_2 \rightarrow_2 \alpha \rightarrow_1 \beta), y : \alpha_1 \cap \alpha_2\}, V_{\rightarrow}[x := \Rightarrow, y := \rightarrow_2], \beta \rangle$ .

Note that  $\text{TYPE}_I$  finds a better reduction type for  $xy(xy) \leftarrow_{\beta} (\lambda x.xx)(xy)$ . In this case  $\text{TYPE}_I$  returns  $\langle \{x : (\alpha_1 \rightarrow_3 \alpha) \cap (\alpha_2 \rightarrow_2 \alpha \rightarrow_1 \beta), y : \alpha_1 \cap \alpha_2\}, V_{\rightarrow}[x := \Rightarrow, y := \rightarrow_3 \vee (\rightarrow_2 \wedge \rightarrow_1)], \beta \rangle$ , which nicely illustrates the faithfulness of  $\text{TYPE}_I$  in emulating the (relative) weakness of the MEET rule of the intersection style type assignment system.

**$\text{TYPE}_I$  is Correct**

**Proposition 5.4.27**

Let  $N \in \Lambda^{\text{NF}}$ , then  $\text{TYPE}_I(N) = \text{pt}(N)$ .

**Proof**

By induction on the structure of  $N$ . If  $N$  is a variable then this is obvious. If  $N \equiv \lambda x.N'$ , then the result follows by the induction hypothesis.

Suppose  $N \equiv xN_1 \dots N_n$ . Let

$$\text{TYPE}_I(x) = \text{pt}(x) = \langle \{x : \alpha_0\}, V_{\rightarrow}[x := \Rightarrow], \alpha_0 \rangle.$$

By the induction hypotheses,  $\text{TYPE}_I(N_i) = \text{pt}(N_i) = \langle A_i, V_i, \tau_i \rangle$ ,  $1 \leq i \leq n$ . Since  $\text{TYPE}_I$  always introduces new type and arrow variables  $\langle A_i, V_i, \tau_i \rangle$  must

have all type and arrow variables disjoint from  $\langle A_j, V_j, \tau_j \rangle$ , for  $i \neq j$  and  $1 \leq i, j \leq n$ . Let  $\alpha_1, \dots, \alpha_n$  be new type variables and  $\rightarrow_1, \dots, \rightarrow_n$  be new arrow variables, then  $\text{TYPE}_I$  performs the following operations:

$$\begin{aligned} \text{UNIFY}_I(\alpha_{i-1}, \tau_i \rightarrow_i \alpha_i, A^i) &= [\alpha_{i-1} := \tau_i \rightarrow_i \alpha_i], \text{ and} \\ V^i(y) &= V_{\rightarrow}[x := \Rightarrow](y) \vee \left( \bigvee_{j=1}^i \rightarrow_j \wedge V_j(y) \right), \end{aligned}$$

where  $A^i = \{x : \tau_1 \rightarrow_1 \dots \tau_{i-1} \rightarrow_{i-1} \alpha_{i-1}\} \mathbin{\mathbb{M}} \left( \mathbin{\mathbb{M}}_{j=1}^i A_j \right)$  and for  $1 \leq i \leq n$ . Hence,  $\text{TYPE}_I(N) = \langle A^n, V^n, \alpha_n \rangle = \text{pt}(N)$ .  $\square$

**Theorem 5.4.28 (Soundness of  $\text{TYPE}_I$ )**

If  $\text{TYPE}_I(M) = \langle A, V, \tau \rangle$ , then  $\langle A, V, \tau \rangle$  is a suitable triple for  $M$ .

**Proof**

By induction on the structure of  $M$ .

If  $M \equiv x$ , then the result follows immediately by rule VAR.

If  $M \equiv \lambda x.N$ , then by the induction hypothesis

$$A' \vdash_{V'}^I N : \tau'.$$

The result then follows (by construction) from rule ABS.

If  $M \equiv N_1 N_2$ , then by the induction hypotheses

$$A_1 \vdash_{V_1}^I N_1 : \tau_1$$

and

$$A_2 \vdash_{V_2}^I N_2 : \tau_2.$$

By the statement of the theorem  $\text{TYPE}_I$  is defined. So the following is also defined:

$$c = \text{UNIFY}_I(\tau_1, \tau_2 \rightarrow_i \alpha, A_1 \mathbin{\mathbb{M}} A_2)$$

Now, by Theorem 5.4.24,

$$c(\tau_1) = c(\tau_2 \rightarrow_i \alpha).$$

By Lemma 5.4.2 and, if necessary, Lemma 3.4.18:

$$c(A_1 \mathbin{\mathbb{M}} A_2) \vdash_{c(V_1)}^I M_1 : c(\tau_1)$$

and

$$c(A_1 \mathbin{\mathbb{M}} A_2) \vdash_{c(V_2)}^I M_2 : c(\tau_2).$$

Now the result follows from rules APP, MEET and LEQ, and the construction of  $\langle A, V, \tau \rangle$ .  $\square$

The following Completeness Theorem for  $\text{TYPE}_I$  only holds for those terms which are strongly normalising. This is discussed further in the following section. As before, note the direct implication of the existence of principal triples for the intersection-style type system.

**Theorem 5.4.29 (Completeness of  $\text{TYPE}_I$ )**

If  $\langle A', V', \tau' \rangle$  is a suitable triple for  $M$ , and  $M$  is strongly normalising, then  $\text{TYPE}_I(M) = \langle A, V, \tau \rangle$  and there exists a chain  $c$  such that

$$\langle A, V, \tau \rangle \rightarrow_c \langle A', V', \tau' \rangle.$$

**Proof**

By induction on  $M$ . For the case where  $M$  is a variable the result is immediate. For the case  $M$  an abstraction term the proof follows directly from the induction hypothesis.

Suppose  $M \equiv N_1 N_2$ . Then, by the induction hypotheses, if  $\langle A'_i, V'_i, \tau'_i \rangle$  are suitable triples for  $N_i$  ( $i \in \{1, 2\}$ ), then  $\text{TYPE}_I(N_i) = \langle A_i, V_i, \tau_i \rangle$  and there exists chains  $c_i$  such that  $\langle A_i, V_i, \tau_i \rangle \rightarrow_{c_i} \langle A'_i, V'_i, \tau'_i \rangle$ . Since  $\langle A', V', \tau' \rangle$  is a suitable triple for  $M \equiv N_1 N_2$  there must be a chain,  $c'$ , such that  $c'(\langle A_i, V_i, \tau_i \rangle)$  is a suitable triple for  $M$ . Moreover, by Theorem 5.4.25 (since  $M$  is strongly normalising),  $c \leq c'$  and so the result follows by construction.  $\square$

### 5.4.7 Decidability

From Theorem 5.4.29 we have that  $\text{TYPE}_I$  will assign a type to all strongly normalising terms, i.e., if a term has an infinite reduction path, then  $\text{TYPE}_I$  acts like a perpetual strategy. Thus, in contrast to the previous algorithms and systems studied,  $\text{TYPE}_I$  is a *semi*-decision procedure on the strong normalisability of  $\lambda$ -terms. Note that the source of the semi-decidability is the algorithm  $U_I$  and in particular the employment of the operation of expansion which is not guaranteed to simplify the unification problem at hand.

In Appendix A a variation on  $\text{UNIFY}_I$  is introduced which is guaranteed to terminate in that the unification step is terminated after a fixed number of expansion operations. A less simple-minded way of ensuring termination might be to maintain a history of terms and try to detect the occurrence of cycles in the unification process. (This is not an effective mechanism unless the history is kept small and combined with the earlier suggestion). A much better mechanism would be to use a system in which terms which cause non-termination are restricted and treated as constants by the type assignment system (see Chapter 6 and Appendix A).

# Chapter 6

## Extensions and Future Work

The theme of this chapter is *generalisation*. Some of the generalisations consider concepts that are required to be present in a practical implementation of the analysis methodology. Others are concerned with adapting the methodology to deduce more or different information from that derived by strong head neededness analysis.

Many of the extensions presented here represent preliminary work and so the reader should not expect detailed semantic or other investigations of the various mechanisms proposed. Instead, a presentation is given of a range of possible ways in which the work of this thesis may be generalised.

In particular, the following topics are considered:

1. *term* and *type constants*,
2. special rules for *fixpoint constants*,
3. *data structures*,
4. *second-order* polymorphism,
5. *sharing analysis*,
6. *non-termination analysis*, and
7. *two stage* strong head neededness analysis.

### 6.1 Constants

This section tackles the pragmatically essential idea of *constants*. In the design of programming languages, constants are usually introduced into both the set of terms and the set of types. Typical examples of such constants are dealt with in this section, though the important case of fixpoint constants is left to the following section.

IF TRUE $MN$	$\rightarrow_\delta$	$M$
IF FALSE $MN$	$\rightarrow_\delta$	$N$
FIX $M$	$\rightarrow_\delta$	$M(\text{FIX } M)$
PLUS $\overline{m} \ \overline{n}$	$\rightarrow_\delta$	$\overline{m + n}$

Figure 21: The Delta Rules for Some Term Constants

### 6.1.1 Term Constants

Individual term constants will be introduced as required in the course of the exposition and will be immediately distinguishable by being written in sans-serif capitals.

In particular, let the set of term constants contain IF, FIX, PLUS, TRUE, FALSE and an infinite series of constants  $\overline{m}$ , one for each natural number  $m$  (thus  $\overline{0}, \overline{1}, \dots$  will be constants with which to represent the natural numbers).

Let  $\delta$  be some term constant, then  $\Lambda\delta$  is the set of  $\lambda$ -terms built up from variables and  $\delta$  by means of application and abstraction in the usual way. Similarly, define  $\Lambda\vec{\delta}$  where  $\vec{\delta}$  is a set of term constants.

Each term constant may have associated with it a particular notion of reduction. These are known as “delta rules”. Delta rules for some example constants are summarised in Figure 21. Let the set of all such delta rules be  $\Delta$ . (No confusion with the earlier use of this symbol to denote the set of arrows should arise). The delta rules are then added to the notion of reduction  $\beta$ , i.e., the notion of reduction now considered is  $\beta \cup \Delta$ .

### 6.1.2 Type Constants

Individual type constants will be distinguishable by being written using sans-serif lowercase letters.

Let  $\kappa$  be some type constant, then  $T_*^\nabla \kappa$  is the set of reduction types built up from type variables, arrows from  $\nabla$  and  $\kappa$  in the usual way for system  $*$ , where  $*$  is one of  $C, L, I$ . Similarly, define  $T_*^\nabla \tau_c$  where  $\tau_c$  is a set of type constants.

In particular, let the set of type constants,  $\tau_c$ , contain the type constants bool and int.

### 6.1.3 Type Assignment

A type assumption for a term constant,  $c$ , will be written  $c: \tau$ . A type assumption set,  $A$ , is then a set of type assumptions for variables and constants. The



following rule is added to the Curry and Intersection systems of deduction in Chapter 3:

$$\text{CONST } A \vdash_{\text{V}}^* c : \tau \quad (c : \tau \in A)$$

In the case of the LET-polymorphic system the following rule is preferred:

$$\text{CONST } A \vdash_{\text{V}}^* c : \tau \quad (c : \underline{\sigma} \in A, \tau \preceq \underline{\sigma})$$

#### 6.1.4 An Example: The Conditional

It is easy to encode the conditional statement as application:

$$\text{if } P \text{ then } M \text{ else } N \equiv PMN,$$

where it is intended that  $P$  in the above be one of the terms **K** (representing truth) or **KI** (representing falsity). So a  $\lambda$ -term to represent the conditional term is then  $\lambda abc.abc$  (see Barendregt [3], pg133). With this representation of the conditional the following type is obtained (using the Curry system for example):

$$(\alpha \rightarrow_i \alpha \rightarrow_j \alpha) \Rightarrow \alpha \rightarrow_i \alpha \rightarrow_j \alpha.$$

If concern is restricted to typed systems in which the only defined values allowed as the predicate of a conditional statement are *true* and *false*, whatever their representation, then a much more informative type can be given. Let us choose the constant **IF** to represent the conditional and the constants **TRUE** and **FALSE** to represent *true* and *false*, then the delta rules for **IF** are as in Figure 21. Now choose the following type assumptions for **IF** (in the Curry-style and Intersection type assignment systems only):

$$\text{bool} \Rightarrow \sigma \vdash \sigma (\neg b) \sigma$$

(one for each possible  $\sigma$  and  $b$ ). In the LET-Polymorphic system, choose the following type assumption (one only) for **IF**:

$$\forall \alpha. \forall \rightarrow_i. \text{bool} \Rightarrow \alpha \rightarrow_i \alpha (\neg \rightarrow_i) \alpha$$

Also choose the type assumptions

$$\text{TRUE} : \text{bool}$$

and

FALSE: bool.

This type for IF is very satisfactory as it allows the capturing of the maximum amount of strong head neededness information that could be expected to be achieved in a static context. In particular, cases such as  $\text{IF } P(fx)(fy)$ , where  $f$  is not contained in  $P$ , will have that the strong head neededness of  $f$  is  $\Rightarrow$ .

Since in practice the conditional commonly occurs in a typed context, this is sufficient justification for the inclusion of the  $\neg$  operator into the definition of Boolean arrow expressions.

## 6.2 Adding a Fixpoint Constant

It is useful to introduce a fixpoint operator as a constant. A delta rule for one such fixpoint constant, FIX, is contained in Figure 21.

The aim of this Section is simply to suggest a range of possible treatments (rules) for a fixpoint constant. No formal investigation is conducted.

One way of dealing with a *fixpoint constant* would be to introduce a rule which directly emulates the infinite typing behaviour of a *fixpoint combinator*! Adding such a rule is interesting in the weaker systems of type assignment (Curry and LET-polymorphic), since fixpoint combinators are not even typable in those systems. The rule may be formulated as follows:

$$\text{IFIX} \quad \frac{A \vdash_V^* M : \sigma_1 b_1 \tau \quad A \vdash_V^* M : \sigma_2 b_2 \sigma_1 \quad A \vdash_V^* M : \sigma_3 b_3 \sigma_2 \quad \dots}{A \vdash_V^* \text{FIX } M : \tau}$$

However, this is clearly not a pragmatically useful rule. One solution would be to introduce the following family of restricted versions of IFIX, one for each  $n \geq 2$ :

$$\text{IFIX}_n \quad \frac{A \vdash_V^* M : \sigma_1 b_1 \tau \quad A \vdash_V^* M : \sigma_2 b_2 \sigma_1 \quad \dots \quad A \vdash_V^* M : \sigma_n b_n \sigma_{n-1}}{A \vdash_V^* \text{FIX } M : \tau}$$

In the case of the intersection-based type assignment system this slight variation is probably preferable:

$$\text{IFIX}_n \quad \frac{A \vdash_V^I M : \sigma_1 \mathbf{b}_1 \tau \quad A \vdash_V^I M : \sigma_2 \mathbf{b}_2 \sigma_1 \quad \dots \quad A \vdash_V^I M : \omega \mathbf{b}_n \sigma_{n-1}}{A \vdash_V^I \text{FIX } M : \tau}$$

These systems can be implemented by inferring the type of the term, taking  $n$  copies of this type where each copy has all arrow and type variables renamed so that there are no common variables between the copies. Then the final step is to unify the copies in the manner suggested by  $\text{IFIX}_n$ .

A sensible  $\text{IFIX}_1$  rule *can* be defined—by requiring that the argument and result types agree “up to the arrows”. This notion can be formalised by introducing the relation  $\simeq$ , which is the least such relation on  $T_*^\nabla \times T_*^\nabla$  such that:

1.  $\alpha \simeq \alpha$ ,
2.  $\kappa \simeq \kappa$ , and
3.  $\frac{\sigma_1 \simeq \sigma_2 \quad \tau_1 \simeq \tau_2}{\sigma_1 \mathbf{b}_1 \tau_1 \simeq \sigma_2 \mathbf{b}_2 \tau_2}$

(And with additional clauses for types systems other than the Curry-style system, in the obvious fashion). Now the rule can be specified as:

$$\text{IFIX}_1 \quad \frac{A \vdash_V^* M : \sigma \mathbf{b} \tau}{A \vdash_V^* \text{FIX } M : \tau} \quad (\sigma \simeq \tau)$$

As a further variation, the following one step rule is derived as a special case of  $\text{IFIX}_1$ :

$$\text{FIX} \quad \frac{A \vdash_V^* M : \sigma \mathbf{b} \sigma}{A \vdash_V^* \text{FIX } M : \sigma}$$

Note that this is *not* the same as the  $\text{IFIX}_1$  rule. Moreover, it is not essential that this form of rule for  $\text{FIX}$  be specified at all, as it may be replaced by an assumption of types of the form  $(\sigma \mathbf{b} \sigma) \Rightarrow \sigma$  for the constant  $\text{FIX}$ . Milner [47] gives a corresponding ordinary type to a fixpoint constant and includes a semantic justification. In the Appendix a fixpoint constant is introduced in exactly this manner.

As an example, consider the term  $\text{FIX}(\lambda fxyz. \text{IF } x \ y \ (fxyz))$ . For simplicity, let us use the Curry-style system. To start with, suppose  $f$  has type  $\text{bool} \rightarrow;$

$\alpha \rightarrow_j \alpha \rightarrow_k \alpha$  and IF has type  $\text{bool} \Rightarrow \alpha \rightarrow_l \alpha (\neg \rightarrow_l) \alpha$ . Then, via the usual reasoning, the type of the term  $\lambda fxyz. \text{IF } x \ y \ (fxyz)$  is

$$\begin{aligned} & (\text{bool} \rightarrow_i \alpha \rightarrow_j \alpha \rightarrow_k \alpha) (\neg \rightarrow_l) \\ & \text{bool} \Rightarrow \alpha (\rightarrow_l \vee \rightarrow_k) \alpha (\neg \rightarrow_l \wedge \rightarrow_j) \alpha \end{aligned}$$

Now it is time to see the effect of the various rules for FIX described above.

Consider rule IFIX—in order to implement this rule an infinite sequence of unifications must be performed. These unifications are each between a pair of types of the forms:

$$\text{bool} \rightarrow_{i_n} \alpha \rightarrow_{j_n} \alpha \rightarrow_{k_n} \alpha$$

and

$$\text{bool} \Rightarrow \alpha (\rightarrow_{l_{n+1}} \vee \rightarrow_{k_{n+1}}) \alpha (\neg \rightarrow_{l_{n+1}} \wedge \rightarrow_{j_{n+1}}) \alpha.$$

Performing these unifications allows the application of the rule with the result that the term  $\text{FIX}(\lambda fxyz. \text{IF } x \ y \ (fxyz))$  is given the type:

$$\text{bool} \Rightarrow \alpha (\rightarrow_{l_1} \vee \neg \rightarrow_{l_2} \wedge (\rightarrow_{l_3} \vee \dots)) \alpha (\neg \rightarrow_{l_1} \wedge (\rightarrow_{l_2} \vee \neg \rightarrow_{l_3} \wedge \dots)) \alpha,$$

where  $\rightarrow_{l_1} = \rightarrow_l$ ,  $\rightarrow_{j_1} = \rightarrow_j$  and  $\rightarrow_{k_1} = \rightarrow_k$ .

In the case of IFIX<sub>n</sub>,  $n \geq 2$ , the result is similar except that only  $n$  unifications are performed and the resulting arrow expressions are finite.

For IFIX<sub>1</sub>, the result is simply:

$$\text{bool} \Rightarrow \alpha (\rightarrow_l \vee \rightarrow_k) \alpha (\neg \rightarrow_l \wedge \rightarrow_j) \alpha.$$

Now consider rule FIX. In this case there is a single unification to be performed, namely:

$$\text{bool} \rightarrow_i \alpha \rightarrow_j \alpha \rightarrow_k \alpha$$

with

$$\text{bool} \Rightarrow \alpha (\rightarrow_l \vee \rightarrow_k) \alpha (\neg \rightarrow_l \wedge \rightarrow_j) \alpha.$$

Then the result, as mechanically determined with the help of the algorithm BUNIFY of Chapter 5, is

$$\text{bool} \Rightarrow \alpha (\rightarrow_l \vee \rightarrow_k) \alpha (\neg \rightarrow_l \wedge \rightarrow_k) \alpha.$$

This is a sensible result as it expresses the fact that the second argument to

$$\text{FIX}(\lambda fxyz. \text{IF } x \ y \ (fxyz))$$

will be strongly head needed if the first branch of the conditional is taken (independently of any other condition). The third argument is required if the

PRODUCT $n$	$\rightarrow_\delta$	$\lambda x_1 \dots x_n x.x x_1 \dots x_n$
SUM $i n$	$\rightarrow_\delta$	$\lambda x_1 \dots x_n. \text{PRODUCT } \bar{2} i (\text{PRODUCT } n x_1 \dots x_n)$
SEL $i n$	$\rightarrow_\delta$	$\lambda x_1 \dots x_n. x_i \text{ (if } i \leq n)$
SELECT $i n$	$\rightarrow_\delta$	$\lambda x. x(\text{SEL } i n)$
CASE $n$	$\rightarrow_\delta$	$\lambda x. \text{SEL } (\text{SELECT } \bar{1} \bar{2} x) n$

Figure 22: The Delta Rules for Data Structures

second branch of the conditional is taken and the following invocation of the function requires its second argument, and so on. (Of course, with our “inside” knowledge of the lack of transformations performed on the predicate, we can ascertain the more precise information that either  $y$  is strongly head needed or the term is unsolvable and neither  $y$  nor  $z$  are strongly head needed in this term).

## 6.3 Data Structures

The general structured types universally present in contemporary languages (such as Miranda, Haskell, Orwell) are sum-of-product types. (ML separates product and sum types). As Peyton-Jones [55] describes, in these languages only one translation scheme is required in order to implement common types such as lists, tuples, enumerations and (disjoint) sums. The key components of the results of this translation scheme are *case*, *constructor* (for sums and products) and *selector* (or *destructor*) functions. In fact, *pattern-matching* is also completely supported by the translation scheme, as is detailed within Peyton-Jones [55].

In order to conduct an analysis of the output of the translation scheme proposed by Peyton-Jones, five main families of functions must be supported. Following the treatment in Peyton-Jones [55] these families of functions will all be constants with delta rules as summarised in Figure 22. (The form of these constants is not precisely the same as appears in Peyton-Jones, pp121–125, but it is trivial to see that they are equivalent). The families are enumerated below.

**PRODUCT  $n$**  This constant builds  $n$ -tuples. The type of **PRODUCT  $n$**  is:

$$\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} (\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} \beta) \Rightarrow \beta$$

**SUM  $i n$**  This constant builds elements of a disjoint sum. The first argument to **SUM** is an integer tag  $i \geq \bar{1}$ . The second argument,  $n \geq \bar{0}$ , specifies

the number of components in the sum (also known as the *arity* of the sum). The type of SUM  $i\ n$  is

$$\alpha_1 (\rightarrow_k \wedge \rightarrow_{i_1}) \dots \alpha_n (\rightarrow_k \wedge \rightarrow_{i_n}) \\ (\text{int} \rightarrow_j ((\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} \beta) \Rightarrow \beta) \rightarrow_k \gamma) \Rightarrow \gamma$$

SEL  $i\ n$  This constant selects the  $i$ th argument out of the  $n$  arguments to which SEL  $i\ n$  is applied. It's type is a generalisation of the type of the K combinator:

$$\alpha_1 \rightarrow \dots \alpha_i \Rightarrow \dots \alpha_n \rightarrow \alpha_i$$

SELECT  $i\ n$  This constant is intended only to be used to extract components from a product type. (Since sum type are built using products, SELECT is also applicable to them). This will be so for any correct compiler and well-typed input script, and it will be assumed here that both these criterion are met. The type of this constant is:

$$((\alpha_1 \rightarrow \dots \alpha_i \Rightarrow \dots \alpha_n \rightarrow \alpha_i) \rightarrow_j \beta) \Rightarrow \beta.$$

Since this constant is always applied only to product terms then by examining the type given for PRODUCT (SUM) above, it is easy to see that  $\rightarrow_j$  will always be unified with  $\Rightarrow$  and so the type

$$((\alpha_1 \rightarrow \dots \alpha_i \Rightarrow \dots \alpha_n \rightarrow \alpha_i) \Rightarrow \beta) \Rightarrow \beta,$$

may be chosen for SELECT instead (there is not a large advantage in so doing, since the type inference algorithm will deduce this information).

CASE  $n$  This constant always expects a sum term as argument, followed by  $n$  cases (terms) to select from. (Case on products is compiled into a let expression). The choice of which term to select is made based on the value of the tag of the sum term. Thus the type of CASE  $n$  is

$$((\alpha_1 \Rightarrow \alpha_2 \rightarrow \alpha_1) \rightarrow_i \text{int}) \Rightarrow \beta_1 \rightarrow_{i_1} \dots \beta_n \rightarrow_{i_n} \beta_k.$$

Note that for a well-typed script,  $\rightarrow_i$  will always be unified with  $\Rightarrow$ .

The translation scheme from scripts to  $\lambda$ -terms (plus constants) is detailed in Peyton-Jones [55], in particular, see Chapters 3, 4, 5 and 6 of that work.

### 6.3.1 A Detailed Example

As an example consider the type of lists, which is a structured type defined (in Orwell) by the following:

```
> list a ::= Nil | Cons a (list a)
```

In the translation scheme, `Nil` is translated to  $\text{SUM } \bar{1} \bar{0}$  and `Cons` is translated to  $\text{SUM } \bar{2} \bar{2}$ . Now consider the definition of `length` in Orwell, which computes the number of elements in a given list:

```
> length [] = 0
> length (x:xs) = 1 + length xs
```

This will be translated into the function:

$$\text{FIX } \lambda f x. \text{CASE } \bar{2} x \bar{0} (\text{PLUS } \bar{1} (f(\text{SELECT } \bar{2} \bar{2} (\text{SELECT } \bar{2} \bar{2} x))))$$

To understand the behaviour of this translation of the definition of `length`, consider the expression  $\text{SELECT } \bar{2} \bar{2} (\text{SELECT } \bar{2} \bar{2} M)$ , for some term  $M$ . This evaluates to the term  $M(\text{SEL } \bar{2} \bar{2})(\text{SEL } \bar{2} \bar{2})$  which in turn reduces to  $M(\lambda xy.y)(\lambda xy.y)$ . Now suppose that  $M$  is constructed from the `Cons` constructor of the type of lists. Then  $M \equiv \text{SUM } \bar{2} \bar{2} P Q$ , for some terms  $P$  and  $Q$ , where in a well-typed script  $Q$  will again be a sum type representing a list.  $\text{SUM } \bar{2} \bar{2} P Q$  reduces to

$$\text{PRODUCT } \bar{2} \bar{2} (\text{PRODUCT } \bar{2} P Q)$$

and so  $M(\lambda xy.y)(\lambda xy.y)$  reduces to  $Q$ , as expected.

The intersection-style system of type assignment will be used to illustrate this example.

Consider the type of  $M$  in the term  $M(\lambda xy.y)(\lambda xy.y)$ . One possible type for  $M$  is

$$(\alpha_1 \multimap \alpha_2 \Rightarrow \alpha_2) \rightarrow_i (\alpha_3 \multimap \alpha_4 \Rightarrow \alpha_4) \rightarrow_j \gamma_1.$$

Further, consider the type of the term  $M \equiv \text{SUM } \bar{2} \bar{2} P Q$ . After conducting the usual reasoning, the type obtained for this term is

$$(\text{int} \rightarrow_{k_1} ((\beta_1 \rightarrow_{k_2} \beta_2 \rightarrow_{k_3} \gamma_2) \Rightarrow \gamma_2) \rightarrow_{k_4} \gamma_3) \Rightarrow \gamma_3.$$

Since only well-typed scripts are being considered (`SELECT` is only used on a sum type), these two types for  $M$  must be unified. Using the algorithm  $U_I$  from Chapter 5 ( $U_C$  would return the same result in this case), the result is

$$(\text{int} \multimap \sigma \Rightarrow \sigma) \Rightarrow \sigma,$$

where  $\sigma$  is

$$(\beta_1 \rightarrow \beta_2 \Rightarrow \beta_2) \Rightarrow \beta_2.$$

Note that this type gives *very detailed* information about the use of this data structure: in particular, this type for the sum term  $M$  tells us that in the expression  $\text{SELECT } \bar{2} \bar{2} (\text{SELECT } \bar{2} \bar{2} (\text{SUM } \bar{2} \bar{2} P Q))$ :

1. the term  $M \equiv \text{SUM } \bar{2} \bar{2} P Q$  is strongly head needed;
2. the term  $P$  is irrelevant; and
3. the term  $Q$  is strongly head needed.

I believe that this is a strong argument for the analysis of data structures at this level. Note that absolutely no extra machinery has been introduced into the type assignment system, other than the ability to assign types to constant terms. On the other hand, it *may* be that the size of the types generated (and hence the high quality of the information obtained) is too great for a practical implementation in which many complex data structures must be analysed.

Continuing with the analysis of the translation of `length`, assume  $f: \beta_2 \rightarrow_k \text{int}$ , then since  $\text{SELECT } \bar{2} \bar{2} (\text{SELECT } \bar{2} \bar{2} x)$  has type  $\beta_2$ ,

$$f(\text{SELECT } \bar{2} \bar{2} (\text{SELECT } \bar{2} \bar{2} x)): \text{int},$$

as required by `PLUS` which has type  $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ . Note that at this stage the strong head neededness of  $x$  will be  $\rightarrow_k$ .

Now the remaining components of the body of the  $\lambda$ -abstractions may be analysed. Firstly, consider  $x$  in the expression  $\text{CASE } \bar{2} x$ . By examination of the type of  $\text{CASE } \bar{2}$  it is clear that  $x$  (being a sum type) should be assigned the type  $(\text{int} \Rightarrow \sigma \rightarrow \text{int}) \Rightarrow \text{int}$ , where  $\sigma$  is as defined above. This can be achieved by assuming  $x$  has type

$$((\text{int} \Rightarrow \sigma \rightarrow \text{int}) \Rightarrow \text{int}) \cap ((\text{int} \rightarrow \sigma \Rightarrow \sigma) \Rightarrow \sigma),$$

and then following each occurrence of an instance of the `VAR` rule for  $x$  by an appropriate instance of `LEQ`. (See Chapter 3 for details about the intersection-style type assignment system).

Suppose  $\text{CASE } \bar{2} x$  has type  $\text{int} \rightarrow_{i_1} \text{int} \rightarrow_{i_2} \gamma_3$ , then the variable strong head neededness of  $f$  in the body of the  $\lambda$ -abstractions is  $\rightarrow_{i_2}$  and for  $x$  it is  $\Rightarrow$ , since `CASE` requires its second argument. Now two instances of the `ABS` rule may be used to obtain the type

$$(\beta_2 \rightarrow_k \text{int}) \rightarrow_{i_2} \tau \Rightarrow \gamma_3,$$

where  $\tau$  is the type of  $x$ , namely

$$((\text{int} \Rightarrow \sigma \rightarrow \text{int}) \Rightarrow \text{int}) \cap ((\text{int} \rightarrow \sigma \Rightarrow \sigma) \Rightarrow \sigma).$$



The final step is the typing of the application of  $\text{FIX}$ . Note that the strong head neededness of the fixpoint variable  $f$  is  $\rightarrow_{i_2}$ . This signifies that there is a possibility that  $f$  is not required in the body of its  $\lambda$ -abstraction and this has implications for the termination behaviour of the function (see Section 6.6 in this chapter).

Using rule  $\text{FIX}$  (any  $\text{IFIX}_n$  rule with  $n \geq 1$  would do as well), the final type thus obtained for the translation of  $\text{length}$  is

$$(((\text{int} \Rightarrow \sigma \multimap \text{int}) \Rightarrow \text{int}) \cap ((\text{int} \multimap \sigma \Rightarrow \sigma) \Rightarrow \sigma)) \Rightarrow \text{int},$$

where  $\sigma$  is, as before,  $(\beta_1 \multimap \beta_2 \Rightarrow \beta_2) \Rightarrow \beta_2$ . What does this tell us about the behaviour of this function?

1. The argument to  $\text{length}$  is strongly head needed; and
2. there are two sets of occurrences of the sum term  $x$  and each set behaves as follows:
  - (a) the first set, corresponding to the type  $(\text{int} \Rightarrow \sigma \multimap \text{int}) \Rightarrow \text{int}$ , strongly head needs the tag of the sum term, but ignores the value part of the sum term, and
  - (b) the second set, corresponding to the type  $(\text{int} \multimap \sigma \Rightarrow \sigma) \Rightarrow \sigma$ , ignores the tag of the sum term, but strongly head needs the value part of the sum term. Furthermore, from the type  $\sigma$  it is clear that the value term consists of two components, the first of which is ignored and the second is strongly head needed.

## 6.4 Second-Order Polymorphism

In Chapter 3 three case studies were conducted of type assignment systems for Boolean Reduction Types. In this Section a brief sketch of a fourth case of type assignment is given.

In the case of  $\text{LET}$ -polymorphic types, all universal quantification was restricted to the outermost level of a type. Now general universal quantification of both type and arrow variables will be allowed. Thus a *second-order* system of Boolean Reduction Types will be introduced. Second-order typing itself was introduced by Girard [25] (Girard called this the *system F*) and Reynolds [57].

### Definition 6.4.1

The set of *Abstract Polymorphic Boolean Reduction Types*,  $T_2^\nabla$ , is inductively defined to be the least set satisfying:

1.  $\alpha \in \tau_v$  implies  $\alpha \in T_2^\nabla$ ,

VAR	$A_x \cup \{x : \sigma\} \vdash_{V \rightarrow [x := \Rightarrow]} x : \sigma$	
APP	$\frac{A \vdash_{V_1} N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2} N_2 : \sigma}{A \vdash_V N_1 N_2 : \tau}$	$(V = \lambda x. V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := b]} N : \tau}{A \vdash_{V[x := \Rightarrow]} \lambda x. N : \sigma \text{ b } \tau}$	
GEN	$\frac{A \vdash_V M : \tau}{A \vdash_V M : \forall \delta. \tau}$	$\delta \notin \text{FV}(A) \cup \text{FV}(V)$
INST	$\frac{A \vdash_V M : \forall \delta. \tau}{A \vdash_V M : \tau[\delta := \sigma]}$	

Figure 23: The Second-Order System for deducing Reduction Types

2.  $\sigma, \tau \in T_2^\nabla$ ,  $b \in \nabla$  implies  $\sigma \text{ b } \tau \in T_2^\nabla$ ,
3.  $\alpha \in \tau_v$ ,  $\sigma \in T_2^\nabla$  implies  $\forall \alpha. \sigma \in T_2^\nabla$ , and
4.  $\rightarrow_i \in \Delta_v$ ,  $\sigma \in T_2^\nabla$  implies  $\forall \rightarrow_i. \sigma \in T_2^\nabla$ .

It will be useful to denote by the metavariables  $\delta, \delta_1, \delta_2, \dots$  either an arrow variable or a type variable. Thus  $\forall \delta. \sigma$  will stand for either  $\forall \rightarrow_i. \sigma$  or  $\forall \alpha. \sigma$ .

A type assumption in the polymorphic system is the same as for earlier systems except that the types assumed for term variables are drawn from  $T_2^\nabla$ .

Term variable strong head neededness functions are exactly the same as described in earlier type assignment systems.

The set of free variables of an arrow expression,  $b$ , written  $\text{FV}(b)$ , is the set containing all and only the arrow variables occurring in the arrow expression,  $b$ . The set of free variables of a type  $\sigma$ , written  $\text{FV}(\sigma)$ , is defined as follows:

1. if  $\sigma \equiv \alpha$ , then  $\text{FV}(\sigma) = \{\alpha\}$ ;
2. if  $\sigma \equiv \tau_1 \text{ b } \tau_2$ , then  $\text{FV}(\sigma) = \text{FV}(\tau_1) \cup \text{FV}(b) \cup \text{FV}(\tau_2)$ ; and
3. if  $\sigma \equiv \forall \delta. \tau$ , then  $\text{FV}(\sigma) = \text{FV}(\tau) - \{\delta\}$ .

The set of free variables of an assumption set  $A$ , written  $\text{FV}(A)$ , is  $\bigcup_{x\sigma \in A} \text{FV}(\sigma)$ . Similarly, the set of free variables of a variable strong head neededness function  $V$ , written  $\text{FV}(V)$ , is  $\bigcup_{x \in X} \text{FV}(V(x))$ .

The type assignment system for polymorphic types appears in Figure 23.

## 6.5 Algebraic Reduction Types

This section investigates the claim in Wright [76] that the present system is suitable as a basis for *sharing analysis*.<sup>1</sup> The primary insight is the generalisation of the operations of the Boolean algebra to the corresponding operations of a ring, *i.e.*, replace logical disjunction by addition and logical conjunction by multiplication. This natural generalisation of the Boolean Reduction Types is called the *Algebraic Reduction Types*. In these types the exact number of uses of a subterm are kept, rather than just “none” ( $\nrightarrow$ ) or “more than zero” ( $\Rightarrow$ ). This corresponds to the *use-count* generalisation of strictness analysis (see Sestoft [60], Jensen and Mogensen [39] and Goldberg [26]).

For the sake of generality, this section considers the Intersection-style type assignment system. Similar treatments for any of the other type assignment systems presented in this thesis follow in the fashion suggested by earlier parts of this work.

### Definition 6.5.1

1. Let  $\Delta_v = \{\rightarrow_i, \rightarrow_k, \rightarrow_j, \dots\}$  be a sufficiently large set of *arrow variables* (as before), where  $i, j, k, \dots$  are (“dummy”) variables over the natural numbers, and
2. let  $\Delta_g = \{\rightarrow_0, \rightarrow_1, \dots\}$  be a set of *ground arrows*, (one for each natural number).
3. Now some operators are defined which generalise the previous operations of disjunction and conjunction over arrows. Let  $\rightarrow_i + \rightarrow_j = \rightarrow_{i+j}$ ,  $\rightarrow_i - \rightarrow_j = \rightarrow_{i-j}$  and  $\rightarrow_i \times \rightarrow_j = \rightarrow_{i \times j}$ . In the right-hand sides of these definitions  $+$  is ordinary addition,  $-$  is ordinary subtraction and  $\times$  is ordinary multiplication (all three over the natural numbers).
4. The set of *Algebraic arrow expressions* is the set of arrows

$$\Delta = T(\Delta_g \cup \Delta_v, \times, +, -).$$

As usual, various sets of types may now be constructed. For example, the set of *Intersection Algebraic Reduction Types* is the set  $T_I^\Delta$ , where  $\Delta$  is the set of Algebraic arrow expressions defined above. As usual let the meta-variables over arrow expressions be  $b, b', \dots$ . Let the meta-variables for elements of  $\Delta_g$  be  $\rightarrow_m, \rightarrow_n, \dots$ . Occasionally the multiplication symbol ( $\times$ ) will be replaced by juxtaposition.

Now irrelevance is represented by the ground arrow  $\rightarrow_0$  (previously  $\nrightarrow$ ) and various degrees of strong head neededness are represented by the arrows

<sup>1</sup>Clem Baker-Finch has independently devised a similar extension to the extension of Boolean Reduction Types described in this section.

$\rightarrow_1, \rightarrow_2, \dots$  (previously  $\Rightarrow$ ). This notion of “degrees of strong head neededness” needs clarification:

**Definition 6.5.2**

Let  $R \subseteq M$  be a redex.  $R$  is a strongly head needed redex of degree  $n$  in  $M$  if  $n$  residuals of  $R$  are reduced on the head reduction path of  $M$ .

This concept generalises to terms other than redexes by noting that residuals are a special case of descendants—exactly as was done for the concept of strong head neededness in Chapter 2. Note the (formal) correspondence between strong head neededness and  $\Rightarrow$  and the (informal) correspondence between strong head neededness of degree  $n$  and  $\rightarrow_n$ . The formalisation of the connection between these is very similar to that expressed in Chapter 4, where Boolean Reduction Types and their relation to the work of Chapter 2 was formalised.

In order to construct a type assignment system the notion of variable strong head neededness function is required. This is defined exactly as before, except that rather than being a function from term variables to Boolean arrow expressions each variable strong head neededness function is a function from term variables to *Algebraic* arrow expressions. In this context the name  $V_0$  is preferred over  $V_{\rightarrow}$  for the everywhere irrelevant variable strong head neededness function (for the obvious reason).

Finally, note that the systems of inequality and equality defined for intersection types in Chapter 3 may be defined for the present system by the substitution of Algebraic arrow expressions for Boolean arrow expressions in those definitions.

The type assignment system for Intersection Algebraic Reduction Types appears in Figure 24. (The only change is in the notation for the operators constructing the variable neededness function in the APP case and the notations for  $\Rightarrow$  and  $\rightarrow$ ).

**An Example**

Consider the term  $Twice \equiv \lambda f x. f(fx)$ . This term has the following deduction using the type assignment system of Figure 24. Let  $A = \{f: (\alpha \rightarrow_i \beta) \cap (\beta \rightarrow_j \gamma), x: \alpha\}$ , then using VAR and LEQ we obtain deductions of:

$$A \vdash_{V_0[x:=\rightarrow_1]}^I x: \alpha,$$

$$A \vdash_{V_0[f:=\rightarrow_1]}^I f: \alpha \rightarrow_i \beta$$

and

$$A \vdash_{V_0[f:=\rightarrow_1]}^I f: \beta \rightarrow_j \gamma.$$

VAR	$A_x \cup \{x : \sigma\} \vdash_{V_0[x := \rightarrow_1]}^I x : \sigma$
APP	$\frac{A \vdash_{V_1}^I N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^I N_2 : \sigma}{A \vdash_V^I N_1 N_2 : \tau}$ $(V = \underline{\lambda}x.V_1(x) + (b \times V_2(x)))$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := b]}^I N : \tau}{A \vdash_{V[x := \rightarrow_0]}^I \lambda x.N : \sigma \text{ b } \tau}$
MEET	$\frac{A \vdash_V^I N : \sigma \quad A \vdash_V^I N : \tau}{A \vdash_V^I N : \sigma \cap \tau}$
LEQ	$\frac{A \vdash_V^I N : \sigma \quad \sigma \leq \tau}{A \vdash_V^I N : \tau}$

Figure 24: The Rules for deducing Intersection Algebraic Reduction Types

Now two instances of the APP rule apply. The interesting part of this is the construction of the variable neededness functions in these two instances of APP. In the deduction of a type for  $fx$  the calculation is, for  $f$ :

$$\rightarrow_1 + (\rightarrow_i \times \rightarrow_0) = \rightarrow_{1+i \times 0} = \rightarrow_1,$$

(as expected, this is exactly analogous with the treatment of Chapter 3), and for  $x$ :

$$\rightarrow_0 + (\rightarrow_i \times \rightarrow_1) = \rightarrow_{0+i \times 1} = \rightarrow_i.$$

In the deduction of a type for  $f(fx)$ , the calculations are, for  $f$ :

$$\rightarrow_1 + (\rightarrow_j \times \rightarrow_1) = \rightarrow_{1+j \times 1} = \rightarrow_{j+1},$$

and for  $x$ :

$$\rightarrow_0 + (\rightarrow_j \times \rightarrow_i) = \rightarrow_{0+i \times j} = \rightarrow_{i \times j}.$$

Now the claim that this system of Algebraic Reduction Types is a generalisation of Boolean Reduction Types is clear. Consider the final type for *Twice*:

$$((\alpha \rightarrow_i \beta) \cap (\beta \rightarrow_j \gamma)) \rightarrow_{j+1} \alpha \rightarrow_{i \times j} \gamma.$$

Firstly, notice that the strong head neededness of  $f$  is of non-zero degree. Moreover, if the first occurrence of  $f$  in  $f(fx)$  strongly head needs  $fx$  a total of  $j$  times then  $f$  is strongly head needed a total of  $j + 1$  times—once for the first occurrence of  $f$  in  $f(fx)$  and  $j$  times for each use of the second occurrence of  $f$  in  $f(fx)$ .

Similarly, the strong head neededness given for  $x$  reflects the intuition that *both* occurrences of  $f$  must strongly head need  $x$  in order for  $x$  to be strongly head needed in the overall expression.

### 6.5.1 Constants

The treatment of constants for Algebraic Reduction Types is very similar to that for Boolean Reduction Types. In particular, the rule CONST remains unchanged for each system of type assignment. On the other hand, some innovation is required in order to get optimal information from a constant representing a conditional.

#### The Conditional

In the treatment of the conditional IF for Boolean Reduction Types the operation of logical negation was used to express the typing restriction embodied in the constant type *bool*. An analogous treatment is proposed for Algebraic Reduction Types. The negation operation is modelled in this system using *subtraction*. The conditional should be assigned a type of the form:

$$\text{bool} \rightarrow_1 \alpha \rightarrow_i \alpha \rightarrow_{1-i} \alpha.$$

Note that since the enumeration of the constant arrows  $\Delta_g$  is over the naturals as opposed to the integers, this type implies that  $i$  may only be instantiated to either 0 or 1.

As an example, in the context of the term  $f(\text{IF } p \ c \ a)$ , where  $f$  strongly head needs its argument to degree  $j$  and the conditional has type  $\text{bool} \rightarrow_1 \alpha \rightarrow_i \alpha \rightarrow_{1-i} \alpha$ , the strong head neededness of  $p$  is of degree  $j$ ,  $c$  is  $j$  or 0 and  $a$  is also  $j$  or 0. Thus  $c$  and  $a$  have upper bounds of  $j$ . In fact, since  $c$  and  $a$  are either  $j$  or 0, we can write their strong head neededness as  $j \times i$  for  $c$  and  $j - j \times i$  for  $a$ .

Note that this behaviour is exactly what the type assignment system delivers. In analysing the term  $f(\text{IF } p \ c \ a)$  the strong head neededness of  $c$  is  $0 + j \times i = j \times i$  and that of  $a$  is  $0 + j \times (1 - i) = j - j \times i$ , as required.

### 6.5.2 Fixpoints and Algebraic Reduction Types

The treatment of a fixpoint constant in a system of Algebraic Reduction Types is similar to that for a system of Boolean Reduction Types. All of the fixpoint rules for that system translate to the system for Algebraic Reduction Types, except for the rule FIX. This is because equations of the form

$$k = n + k$$

( $n \geq 1$ ) may arise which have no solution (for example, in  $\text{FIX} \lambda f x. \text{PLUS } x (fx)$ , here  $n = 1$ ). In the system of Boolean Reduction Types, any equation of the form

$$\rightarrow_k = \Rightarrow \vee \rightarrow_k$$

immediately reduces to the solvable  $\rightarrow_k = \Rightarrow$ . This is intuitively reasonable since the analysis is only concerned with whether a term is strongly head needed to degree 0 or to any degree greater than 0. In the next section is an example where an attempt to apply rule FIX results in an erroneous conclusion.

### 6.5.3 Data Structures

The treatment of data structures for Boolean Reduction Types generalises to Algebraic Reduction Types in a straightforward manner. This generalisation is detailed below.

**PRODUCT  $n$**  The type of the  $n$ -tuple builder is:

$$\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} (\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} \beta) \rightarrow_1 \beta.$$

**SUM  $i \ n$**  The type of the sum type constructor is:

$$\begin{aligned} &\alpha_1 \rightarrow_{k \times i_1} \dots \alpha_n \rightarrow_{k \times i_n} \\ &(\text{int} \rightarrow_j ((\alpha_1 \rightarrow_{i_1} \dots \alpha_n \rightarrow_{i_n} \beta) \rightarrow_1 \beta) \rightarrow_k \gamma) \rightarrow_1 \gamma. \end{aligned}$$

**SEL  $i \ n$**  Selection of the  $i$ th of  $n$  arguments results in the type:

$$\alpha_1 \rightarrow_0 \dots \alpha_i \rightarrow_1 \dots \alpha_n \rightarrow_0 \alpha_i.$$

**SELECT  $i \ n$**  Selection from a product type:

$$((\alpha_1 \rightarrow_0 \dots \alpha_i \rightarrow_1 \dots \alpha_n \rightarrow_0 \alpha_i) \rightarrow_j \beta) \rightarrow_1 \beta.$$

As before, in any well-typed context,  $j$  will be set to 1.

CASE  $n$  And lastly the switching function:

$$((\alpha_1 \rightarrow_1 \alpha_2 \rightarrow_0 \alpha_1) \rightarrow_i \text{int}) \rightarrow_1 \beta_1 \rightarrow_{i_1} \dots \beta_n \rightarrow_{i_n} \beta_k$$

If the term is well-typed then  $i$  will always be set to 1.

Consider the example from the earlier section on data structures. In the system of algebraic reduction types the type of the translation of `length` may be deduced as follows.

Firstly, the type of  $x$  in

$$\text{FIX } \lambda f x. \text{CASE } \bar{2} \ x \ \bar{0} \ (\text{PLUS } \bar{1} \ (f(\text{SELECT } \bar{2} \ \bar{2} \ (\text{SELECT } \bar{2} \ \bar{2} \ x))))$$

is

$$\tau = ((\text{int} \rightarrow_1 \sigma \rightarrow_0 \text{int}) \rightarrow_1 \text{int}) \cap ((\text{int} \rightarrow_0 \sigma \rightarrow_1 \sigma) \rightarrow_1 \sigma),$$

where  $\sigma$  is  $(\beta_1 \rightarrow_0 \beta_2 \rightarrow_1 \beta_2) \rightarrow_1 \beta_2$ .

Now the type of

$$\lambda f x. \text{CASE } \bar{2} \ x \ \bar{0} \ (\text{PLUS } \bar{1} \ (f(\text{SELECT } \bar{2} \ \bar{2} \ (\text{SELECT } \bar{2} \ \bar{2} \ x))))$$

is

$$(\beta_2 \rightarrow_k \text{int}) \rightarrow_{i_2} \tau \rightarrow_{1+ki_2} \gamma_3.$$

Let us consider some of the rules for typing the application of the fixpoint constant.

Firstly, rule IFIX. In this case the final type obtained is

$$\tau \rightarrow_{1+l_1+l_1l_2+l_1l_2l_3+\dots} \text{int}.$$

This details the correct information that (subterms of)  $x$  will be reduced each time the recursive part of the case statement is taken plus once for the first time the function is entered. In fact each  $l_i$ ,  $i \geq 1$ , is either 0 or 1 and if  $l_j = 0$ , for some  $j \geq 1$ , then  $\forall k > j. l_k = 0$ , and so a more accurate type for this term is  $\tau \rightarrow_{1+l_1+l_2+l_3+\dots} \text{int}$ . This more detailed information might be obtainable from a more informative type for the CASE constant.

Choosing rule IFIX<sub>2</sub> results in the equation

$$k = 1 + k'i'_2.$$

Substituting back into the type of the function yields the acceptable *approximation*:

$$\tau \rightarrow_{1+i_2+ki'_2} \text{int}.$$



Now, consider rule IFIX<sub>1</sub>. In this case the type obtained is

$$\tau \rightarrow_{1+ki_2} \text{int.}$$

Again, this is an acceptable approximation to the behaviour of this function.

In contrast, applying rule FIX results in the equation

$$k = 1 + ki_2.$$

This has a single solution (since variables range over the natural numbers), namely  $k = 1, i_2 = 0$ . Substituting back into the type of the function results in the troublesome type:

$$\tau \rightarrow_1 \text{int.}$$

The problem with this type is that it says that  $x$  is always used precisely once, but  $x$  may be used one *or more* times. The fine granularity of information given by Algebraic Reduction Types includes information about variable usage *for each instance* of the body of a recursive function during its reduction. Thus to equate all the usages of every instance of the body of a recursive function is erroneous in any non-trivial term. Intuitively, rule FIX works for Boolean Reduction Types since in that system the only concern is (effectively) with two arrows,  $\rightarrow_0 = \rightarrow$  and  $\rightarrow_{i>0} = \Rightarrow$ .

#### 6.5.4 Implementation

The essential component to be formalised in any implementation of Algebraic Reduction Types is the specification of a unification algorithm. However, unlike the situation for Boolean Reduction Types, I expect that such an algorithm would be undecidable. As evidence, consider Siekmann [61], in which it is recorded that the solution to Hilbert's Tenth problem is undecidable. This problem is that of finding a Diophantine (integral) solution to a polynomial equation. In this system the mere presence of rules of associativity and distributivity is enough to result in the system becoming undecidable (compare with the situation for the Boolean case). It is an open problem of Unification Theory whether distributivity alone is enough to result in the undecidability of the unification algorithm (Siekmann [61]).

(I remind the reader that familiar techniques such as Gaussian elimination are only applicable to *linear* polynomials).

However, the current system is not *exactly* the system of Hilbert's Tenth Problem. Firstly, the current system is restricted to the *positive* integers. Secondly, let  $x'$  represent the expression  $1 - x$  (implying that  $x$  is 0 or 1), then the system has the following rules:

$$\begin{array}{rcl}
x + 0 & = & x \\
x + x' & = & 1 \\
x \times 0 & = & 0 \\
x \times 1 & = & x \\
x \times (y + z) & = & x \times y + x \times z \\
x \times x' & = & 0 \\
1' & = & 0 \\
0' & = & 1 \\
(x')' & = & x
\end{array}$$

In addition, associativity and commutativity of  $+$  and  $\times$  are added as well as an infinite series of rules relating to the addition and multiplication of natural numbers (such as  $5 + 3 = 8$ , and so on). Note that terms such as  $2'$ ,  $3'$ , ... are undefined. It seems unlikely that this system will have any advantage over that of Hilbert's Tenth Problem, though this remains to be determined.

Should it be determined that exact solutions are not decidable obtainable, then work should concentrate on finding a decidable and safe approximation algorithm.

## 6.6 Non-termination

The problem of determining whether or not a term  $M$  is unsolvable is related to strong head neededness analysis: if a strongly head needed redex  $R \subseteq M$  of  $M$  is unsolvable then  $M$  is unsolvable. Thus it is natural to consider this problem once an analyser for strong head neededness information is presented. The completeness of the information obtained will of course depend on the completeness of the strong head neededness information available.

Let  $\Theta$  be a fixpoint combinator, i.e.,  $\forall M \in \Lambda. \Theta M \rightarrow_{\beta} M(\Theta M)$ . In order to determine whether a term is unsolvable in an inductive fashion from its subterms, it is convenient to tag terms which have been determined to be unsolvable.

### Definition 6.6.1

The  $\Omega$ -labelled  $\lambda$ -terms,  $\Lambda^{\Omega}$  are defined to be the least set containing  $\Lambda$  such that  $M \in \Lambda^{\Omega}$  implies  $M^{\Omega} \in \Lambda^{\Omega}$  and closed under application and abstraction in the usual manner.

The notion of reduction associated with  $\Omega$ -labelled terms is axiomatised by the following rules:

1.  $(M^{\Omega})^{\Omega} \rightarrow_{\Omega} M^{\Omega}$ ,
2.  $(M^{\Omega})N \rightarrow_{\Omega} (MN)^{\Omega}$ , and

VAR	$A_x \cup \{x : \sigma\} \vdash_{V \mapsto [x := \Rightarrow]}^I x : \sigma$
APP	$\frac{A \vdash_{V_1}^I N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^I N_2 : \sigma}{A \vdash_V^I N_1 N_2 : \tau}$ $(V = \lambda x. V_1(x) \vee (b \wedge V_2(x)))$
APP- $\Omega$	$\frac{A \vdash_{V_1}^I N_1 : \sigma \Rightarrow \tau \quad A \vdash_{V_2}^I N_2^\Omega : \sigma}{A \vdash_V^I (N_1 N_2)^\Omega : \tau}$ $(V = \lambda x. V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x : \sigma\} \vdash_{V[x := b]}^I N : \tau}{A \vdash_{V[x := \mapsto]}^I \lambda x. N : \sigma \text{ b } \tau}$
MEET	$\frac{A \vdash_V^I N : \sigma \quad A \vdash_V^I N : \tau}{A \vdash_V^I N : \sigma \cap \tau}$
LEQ	$\frac{A \vdash_V^I N : \sigma \quad \sigma \leq \tau}{A \vdash_V^I N : \tau}$
FIX- $\Omega$	$\frac{A \vdash_V^I M : \sigma \Rightarrow \sigma}{A \vdash_V^I (\Theta M)^\Omega : \sigma}$

Figure 25: The Rules for Non-termination

3.  $\lambda x. M^{\Omega \rightarrow \Omega} (\lambda x. M)^\Omega$ .

A term,  $M \in \Lambda^\Omega$ , is in  $\Omega$ -normal form if none of the above rules are applicable to  $M$  or any of its subterms.

Now a type assignment system for  $\Lambda^\Omega$  terms can be defined. For generality, choose the set of types to be  $T_V^\nabla$  (for some  $\nabla$ ), then the type assignment system appears in Figure 25. Once a type has been derived for a  $\Lambda^\Omega$  term using this type assignment system, all that remains to do to determine whether a term is unsolvable is to reduce the term to  $\Omega$ -normal form.

## 6.7 A Two Step Strong Head Neededness Algorithm

It is useful to separate the type checking phase of a compiler from its analysis phase because typically the type checking phase is cycled through several times during the development of a script. This may be conveniently done for the present methodology by splitting the type inference algorithm into two stages. The first stage performs ordinary type inference or type checking (as appropriate for the input language) and returns terms annotated with their type, while the second stage deduces the strong head neededness information for the terms. Some advantage may be taken of this scheme to simplify certain parts of the strong head neededness algorithm (as detailed below).

Firstly, it is not required that the terms output from the type checker have *every* subterm decorated with type annotations. Instead each variable should have a type annotation, but no other subterm need have a type annotation.

Secondly, type variables and constants are now not required by the strong head neededness analyser. Instead, a single dummy variable may be used. Equivalently, type variables and constants may be preserved, but the unification algorithm defined to simply ignore these (since they have already been checked in the type checking phase).

These ideas may be formalised by a deduction system which is specified below. For simplicity, a Curry-style system is considered.

### Definition 6.7.1

The set of *ordinary* types is the least set containing:

- the set of type variables, and
- all types of the form  $s \rightarrow t$ , whenever  $s$  and  $t$  are ordinary types.

The set of *annotated* variables is the set of all terms of the form  $x_s$ , where  $x$  is a term variable and  $s$  is an ordinary type.

Let  $\sigma \in T_C^\nabla$ , then write  $|\sigma|$  for that ordinary type which results from  $\sigma$  by replacing each arrow expression with the ordinary arrow  $\rightarrow$ .

Let a type assumption be a term of the form  $x_s : \sigma$ , where  $x_s$  is an annotated variable and  $\sigma \in T_C^\nabla$ . An assumption set is then a set of type assumptions, as usual.

Now the deduction system may be specified as in Figure 26. Note that if  $\nabla$  is a set of Boolean arrow expression, then principal types follow for this system in the same way as they did for the Curry-style system (see Chapters 3 and 5).

The role of the annotated variables in this deduction system is to dictate the form of the types deduced in every manner *except* for the matter of arrow expressions. Thus the type checking phase of the compiler need only maintain

VAR	$A_x \cup \{x_{ \sigma } : \sigma\} \vdash_{V \rightsquigarrow [x := \Rightarrow]}^C x : \sigma$
APP	$\frac{A \vdash_{V_1}^C N_1 : \sigma \text{ b } \tau \quad A \vdash_{V_2}^C N_2 : \sigma}{A \vdash_V^C N_1 N_2 : \tau}$ $(V = \underline{\lambda}x.V_1(x) \vee (b \wedge V_2(x)))$
ABS	$\frac{A_x \cup \{x_{ \sigma } : \sigma\} \vdash_{V[x := b]}^C N : \tau}{A \vdash_{V[x := \rightsquigarrow]}^C \lambda x.N : \sigma \text{ b } \tau}$

Figure 26: A System for Analysing Type Checked  $\lambda$ -terms

a list of types assigned to all occurrences of term variables. Then an implementation of the deduction system of Figure 26 may simply “fill in the arrows”.

# Chapter 7

## Conclusion

A new technique has been developed for analysing reduction in the  $\lambda$ -calculus. This technique gathers intensional information about  $\lambda$ -terms which may be used to justify transformations of the terms in the interests of more efficient evaluation. The reduction information collected is based on the new notion of *strong head neededness*. This property allows a finer analysis to be conducted of terms which are not head normalisable than does the alternative notion of *strictness*.

The new technique, based on a particular notion of type called the Boolean Reduction Types, has been presented as a framework for the specification of related analyses. These analyses are constructed using an appropriate choice of *type deduction system* of which three such logics were studied.

The semantics of Boolean Reduction Types has been examined. The resulting semantics is of a *denotational* form—an instrumented semantics was not required. As well, the type deduction systems that were considered as case studies were shown to be correct with respect to the semantics assigned to Boolean Reduction Types.

Implementations were conducted of all three case studies. The major innovation involved was the construction of a new unification method (two case studies were given of this method). This new unification scheme was constructed by joining together two pre-existing unification systems which had not before been used together in this fashion.

Finally, several extensions to the method were given, notably the analysis of data structures and a generalisation of Boolean Reduction Types to express sharing information.

This chapter contains a discussion of related work and a summary of the thesis and its conclusions.

## 7.1 Related Work

A large body of related literature has developed since the original work of Mycroft [50] on strictness analysis of functional programs. In this Section a review of some of the highlights of the development of this knowledge is given, with particular consideration to the relationship with the current work.

Naturally, attention is restricted to analyses which yield information about strictness, with occasional reference to some work concerned with sharing analysis. For present purposes it is convenient to divide the work into three categories:

- analyses which are based on *types*,
- analyses based on an abstract interpretation using other than types, and
- other techniques.

### 7.1.1 Analyses based on Types

In this subsection those analyses which are based on some form of *type* are considered. Each of these analyses constructs functions in a conventional manner from unconventional base sets. This is one primary difference with the current work in which functions are constructed in an unconventional way, but from conventional base sets. Another primary distinction is that the current work focuses on the identification of those functions which definitely require their argument *and* those that that definitely do not require their argument. This is in contrast with most other analyses in which the only information sought is to identify those functions which definitely do require their argument. Thus the current work seeks to find out strictly greater information than that of other analyses. Yet another difference is the issue of strong head neededness versus strictness, the former of which generalises in a natural way to the concept of sharing (Chapter 6) and has practical application to *speculative parallelism* (Partridge [53]).

Wray [71] developed first and second-order analysers which were the first to use (a primitive form of) type to describe strictness information. Types in the system of Wray have the following form:

$$mode = \begin{cases} usage \rightarrow mode \\ m_L \\ m_D \end{cases}$$

where  $usage \in \{S, L, D, A\}$ . These latter symbols stand for strict, lazy, dangerous and absent, respectively. The mode  $m_L$  is used to indicate that a function may terminate and  $m_D$  is used to indicate that the function definitely does

not terminate. Clearly such types are only suitable for first-order information about functions. Wray includes an extension to this scheme to provide second-order information about functions (*i.e.*, information is lost about arguments which themselves take arguments which are functions and so on).

Kuo and Mishra [41] also introduced a technique for analysing terms for strictness information based on a type inference scheme. Their scheme constructs types, using the function type constructor  $\rightarrow$ , from two sets:  $\emptyset$  and  $\square$ , representing the set of “looping” terms and the set of all terms, respectively. (Intuitively,  $\emptyset$  corresponds to  $S$  in Wray’s scheme and  $\square$  to  $L$ ). Kuo and Mishra call these types *strictness types*. Note that the choice made by Kuo and Mishra of the sets to analyse is the traditional one of differentiating those terms which are strict on their argument from those that *may* be strict on their argument.

Kuo and Mishra present a type deduction system based on constraints over types and an inference algorithm to automatically determine strictness types for terms. They extend their system to analyse LET-polymorphism. The technology for their algorithm is derived from Mitchell [48] and Fuh and Mishra [23, 24]. One point to note is that their algorithm requires reasoning about a set of constraints between strictness types. Unlike the system of Mitchell [48], the constraints in this system *cannot* be decomposed into inequalities between base types and variables. Indeed, for the apparently simpler case in which these constraints may be reduced, Wand and O’Keefe [70] have shown that the problem of type inference is NP-complete. (Wand and O’Keefe also show that if the inequalities are constrained such that a greatest lower bound always exists between pairs of constants, then the problem may be solved in low-order polynomial time). Wand and O’Keefe also report the incorrectness of the algorithm presented in [23] for checking the consistency of a coercion set. Kuo and Mishra [41] report that the termination of the algorithm for checking the consistency of a coercion set may be enforced by restricting the set of terms which may be analysed, for example to the set of terms typable in a LET-polymorphic type system.

This analysis scheme is extended in Leung and Mishra [42] to an analysis of which arguments are required to be evaluated to *normal form*, as well as the strictness analysis of the system of Kuo and Mishra. However, this further complicates reasoning about constraint sets (see Section 5.1 of Leung and Mishra in which the 46 rules required to be satisfied are listed).

Jensen [38] develops a relationship between the approach of Kuo and Mishra and that of the approach of Burn et al [11]. This is based on a representation of a lattice by its ideals (types naturally fitting this model). Thus Jensen is able to give the work of Kuo and Mishra a formal semantic justification. (The soundness and completeness is given via the relationship established by Jensen with the other abstract interpretation).

Coppo [15] has developed a system in essence very similar to that of Kuo



and Mishra [41]. Coppo introduces a type constant  $b$  which is interpreted in his semantics as the set  $\{\perp\}$ . (This corresponds to Kuo and Mishra's  $\emptyset$ ). The approach of Coppo is explicit about using types as the domain of an abstract interpretation, whereas other work on using types have not emphasized this aspect. Coppo demonstrates the semantic soundness and completeness of his approach by introducing two additional components to his logic, one an axiom allowing the constant symbol  $\perp$  to acquire any type and the other a rule allowing the assignment of a type to a term if all its approximants share that type.

To summarise, all these systems for strictness analysis share with this thesis the use of types to represent intensional information about functions. In contrast however, the current work is the only one based on a Boolean algebra of function type constructors (or, in the case of the generalisation to sharing analysis, an algebraic ring structure over the positive integers).

Earlier work by Wright [72, 74] employed Boolean Reduction Types in a logic based on *constraints*. Algorithms and semantics were developed for these systems. The monotonicity of function types in those logics is unusual and the semantics of this would benefit from further study.

Wright [73, 76] contains a less detailed exposition of some of the present work. In particular, the focus of these papers is on Boolean Reduction Types and the intersection-based logic. No algorithms are given, though semantics are presented.<sup>1</sup>

Recently, *linear logic* has been used by Wadler [67] to conduct an analysis of sharing using a form of type (known as *linear types*). This system determines if an argument to a function is used *exactly once*. The only other system for computing sharing information based on a logic of types that I am aware of is that based on Algebraic Reduction Types, as described in this thesis.<sup>2</sup> In comparison with the system based on linear types, a logic of Algebraic Reduction Types returns strictly greater information about the sharing behaviour of an argument to a function.

### 7.1.2 Abstract Interpretations not based on Types

Mycroft [50] first applied the methodology of abstract interpretation (Cousot and Cousot [17]) to the strictness analysis of functional programs. His analyser was restricted to first-order functions over flat domains. The extension of this approach to the higher-order case over flat domains was achieved by Burn et al [11], Hudak and Young [32] and Maurer [46]. The approach of Burn et al

<sup>1</sup>In [73] the incorrect claim that the system is complete with respect to an *ordinary*  $\lambda$ -model is made.

<sup>2</sup>Clem Baker-Finch has independently devised this same extension to Boolean Reduction Types.

restricted the analysis by admitting only terms typable in a Curry type system (extended with fixpoint and other constants)<sup>3</sup>. This ensures the termination of their method by restricting the abstract domains assigned to functions to be finite, since these domains are iterated over in order to compute the abstract fixpoints. As has been previously noted the cost of computing these fixpoints can be prohibitive. Sestoft [60] points out that the complexity of the method of Burn et al and Hudak and Young is similar for Curry typable terms, and argues that these methods are (worst-case)  $n$ -exponentially complete for this class of terms<sup>4</sup>. Despite this, considerable work has been done to alleviate the problem, see for example Hunt [36], Hankin and Hunt [28] and the references therein.

All of the above methods are confined to flat domains (unless an encoding of data structures as functions is given). Considerable work has been accomplished on removing this restriction for abstract interpretation, see for example Hughes [33], Wadler [66], Ferguson and Hughes [22] and Burn [10]. Hall and Wise [27] describe a powerful method<sup>5</sup> for analysing lists in a first-order lazy language. This method is based on an abstraction of the domain of lists which allows the structure of an arbitrary list to be fully described. An annotation which indicates strictness is attached to those components of the abstracted structure of a list which have been deduced to be required by the computation.

Sestoft presents a number of analyses in his thesis [60]. Of these analyses one is an analysis of sharing, called a *usage interval* analysis. Sestoft first defines a three point domain consisting of linearly ordered points: *Zero*, *One* and *Many*. From this he constructs the main domain used which is simply pairs of elements from the above domain. Such pairs represent an interval of uncertainty of the number of times a function will use its argument. Sestoft then presents an algorithm for deducing intervals for a lazy and higher-order language, as well as techniques for utilising the information derived in a variation of the *Krivine* machine. (The Krivine machine is a very simple call-by-name *de Bruijn*-term interpreter).

In comparison with the method given in this work for sharing analysis, it is clear that Sestoft's domain is much less precise (for example, the information that an argument is used exactly twice is not representable with Sestoft's notion of usage interval). Of course, Sestoft has traded precision for efficiency of the analysis algorithm, though it remains to be determined how much his  $O(p^3)$  algorithm gains in the "average" case and whether sufficient information is

<sup>3</sup>Alternatively a LET-polymorphic system may be chosen by the result of Abramsky [1], though this does not prevent recomputation of strictness information for polymorphic functions.

<sup>4</sup>Geoffrey Burn was the first to point this out to me (in 1989) and he attributed the observation to Werner Damm.

<sup>5</sup>This is a form of *backwards* analysis, see below for a description.

obtained for a particular application. Note that the idea of usage intervals is contained in the idea of algebraic arrow expressions:

Usage Interval	Arrow Expression
(Zero, Zero)	$\rightarrow_0$
(Zero, One)	$\rightarrow_{1-i}$
(Zero, Many)	$\rightarrow_i$
(One, One)	$\rightarrow_1$
(One, Many)	$\rightarrow_{1+i}$
(Many, Many)	$\rightarrow_{2+i}$

Also note that Sestoft's approximation fits well within the framework of Reduction Types as proposed by this thesis and formulated in an abstract manner in Wright [75]. This approximation can be modelled with the arrows:  $\rightarrow_0$ ,  $\rightarrow_1$  and  $\rightarrow_{>1}$ . These correspond to *Zero*, *One* and *Many* in Sestoft's approach. Then operations of addition and multiplication can be defined on these arrows. Thus  $\rightarrow_1 +_S \rightarrow_{>1} = \rightarrow_{>1}$ ,  $\rightarrow_0 \times_S \rightarrow_k = \rightarrow_0$  and so on in the obvious manner (the  $S$  subscript is short for *Sestoft*). Lastly, a deduction system for the Reduction Types built from these arrows may be constructed in the manner developed within this thesis.

### 7.1.3 Other Techniques

Projection Analysis (Hughes and Wadler [68], Davis and Wadler [19]) is based on the idea of transmitting demand information about the result of a function to the arguments of the functions. Thus the flow of information is "backwards"—from result to arguments. This information is described by composing *projections* with the function. A projection is an idempotent function which removes information from its argument, *i.e.*, the result of applying a projection to an argument is just the argument, but possibly with some components of the argument undefined. For example, consider the projection  $STR$ , where  $STR\perp = \perp$ <sup>6</sup> and is the identity function everywhere else. Suppose  $f$  is a strict function, then the following identity holds:

$$f; STR = STR; f; STR.$$

This indicates that in a strict context the argument to  $f$  may be evaluated before calling  $f$ , without changing the result of the expression.

The main motivation for introducing projection analysis was the study of the properties of non-flat data structures. The initial work was restricted to

<sup>6</sup>This is a slight simplification in that a lifted domain should be used for the result, see Burn [9].

first-order functions. Burn has conducted a detailed examination of the relationship between his work ([11, 8, 10]) and projection analysis. Hughes and Launchbury [35] have shown how to reverse the direction of an abstract interpretation and discuss when so doing results in a loss of information.

Dybjer [20] also proposes a backwards analysis scheme for strictness analysis, but his approach is based on computing the *inverse image* of a function. The basic idea is that a function is strict iff the inverse image of a set of total elements is a set of total elements, with respect to the function of interest. Dybjer gives laws which encapsulate a logic for computing exact inverse images of functions, as well as a suggestion of how approximations may be obtained. The method appears to be well suited to the analysis of data structures, though the treatment in [20] is limited to first-order functions.

## 7.2 Summary

In this Section the contributions and conclusions of this thesis are summarised.

### 7.2.1 Strong Head Neededness

A new property of expressions has been defined called *strong head neededness*. This property is a variation on the idea of head neededness proposed by Barendregt et al [5]. In particular, the new property provides more information about the usage of sub-expressions of a non-terminating reduction, than does the concept of head neededness as defined by Barendregt et al.

The main practical advantage of the extra information is expected to be its use in reducing unnecessary work being performed in a machine employing a *speculative* evaluation strategy (such a machine is described in detail by Partridge [53]).

The concept of strong head neededness also generalises in a very natural way to describe the repeated evaluation of a particular sub-expression during the reduction of a term. This concept forms the basis for an extension of the results of this work to sharing analysis (see Chapter 6).

In Chapter 2, an examination was conducted of strong head neededness. Amongst other results, it was determined that whether or not a particular sub-expression has this property cannot in general be decided. Hence, analyses of strong head neededness must in the general case compute a (safe) approximation to the actual information derivable.

### 7.2.2 Boolean Reduction Types and Type Deduction

With the property of strong head neededness as intuition, a new notation (Boolean Reduction Types) has been developed which relates the strong head

neededness of sub-expressions of a function to the strong head neededness of arguments of the function. This notation is a form of type which uses a Boolean algebra of function type constructors to classify groups of functions according to their strong head neededness behaviour on arguments.

Since the notation is a form of type it was natural to specify systems of logic which relate particular terms to certain types. Three case studies were presented:

- a *Curry*-style system, which is the simplest system that was presented and clearly laid out the various components of the framework to be used to infer Boolean Reduction Types;
- a *LET-polymorphic* system which is polymorphic in arrow as well as type variables. This system allows a polymorphic application to have the abstracted variable appear in multiple strong head neededness contexts in the body of the abstraction; and
- an *Intersection*-style system which allows any variable to appear in multiple strong head neededness contexts and moreover can find a type classification for every term.

These case studies demonstrated the wide applicability of the basic framework for classifying terms according to their strong head neededness behaviour, as denoted by Boolean Reduction Types.

### 7.2.3 Intensional Semantics for Reduction Types

In Chapter 4, a semantics for Boolean Reduction Types was developed which precisely captures the context-sensitive nature of the strong head neededness information described by such a type. A formal connection between the property of strong head neededness and Boolean Reduction Types was then established.

The first application of this semantics was a study of the strong head neededness behaviour of the application of a function to its argument. Each of the possible behaviours was first specified using a particular form of Boolean Reduction Types and then it was shown that under the conditions imposed that the term did indeed behave in the manner required. Finally, a general form was given which described the applicative behaviour of every application.

The second application of the semantics was a demonstration that each of the logics developed in Chapter 3 correctly associated terms with Boolean Reduction Types. The soundness of these logics was successfully shown using both an ordinary  $\lambda$ -model and Plotkin's notion of semi- $\lambda$ -model. The completeness of these logics only holds for the notion of semi- $\lambda$ -model. The logics may each be extended following the EQ rule of Hindley [29] so as to establish the completeness of these systems with respect to an ordinary  $\lambda$ -model, though

this was not done since the interest was primarily in reduction as opposed to expansion.

#### 7.2.4 Implementation of Reduction Type Inference

With a formal assurance of the correctness of the logics, the implementation of algorithms which allow the automatic derivation of types for terms could proceed. The key step here was employing an E-unification algorithm for Boolean algebras. Fortunately, such a unification algorithm had already been (re)discovered and been shown to be both correct and minimal (Martin and Nipkow [45]).

The resulting implementations were shown to be correct syntactically (*i.e.*, with respect to the deduction systems of Chapter 3). Of course, this immediately implies their *semantic* correctness as well, since in Chapter 4 the semantic correctness of the logics had already been established (as described above).

Three implementations were developed, corresponding to the three case studies of type deduction. The last of these required a careful study of the structure of deductions in order to establish the soundness of a fundamental operation employed by the algorithm, namely the operation of *expansion*.

#### 7.2.5 Extensions

With the basic methodology now developed, attention was turned to other constructs and analyses. These extensions are now briefly outlined below.

##### Constants, Fixpoints and Data Structures

Since all programming languages contain constants, fixpoints and data structures it is worthwhile paying special attention to these in any analysis. It turned out that adding constants could be done in a straightforward and practical manner to any of the example systems.

Several rules were postulated for computing information about fixpoints, including several which avoid any iteration in this computation (with a corresponding reduction in precision). The efficacy of these rules in real-life situations remains to be determined.

A particular approach to analysing arbitrary algebraic data structures was proposed. Though this approach yields very precise information its practicality is not known.

##### Algebraic Reduction Types

For a long while the author suspected that the system developed in this thesis based on Boolean Reduction Types could be generalised to perform sharing

analysis. This was established in Chapter 6 by the simple mechanism of replacing the Boolean algebra of function type constructors by an algebra of function type constructors over the positive integers. Allied to this was the generalisation of strong head neededness to *strong head neededness of degree  $n$* . Preliminary investigations show that the semantics of Chapter 4 generalise in an obvious fashion to this case. (These investigations are not reported on in the text of the thesis).

It was shown that a logic for the new set of types (Algebraic Reduction Types) could be developed following the pattern by which the types themselves were derived, namely, simply replace the Boolean Reduction Types by Algebraic Reduction Types and Boolean arrow expressions by Algebraic arrow expressions in the logics of Chapter 3. (An example of the Intersection-style system was made). Finally, some remarks were given concerning the treatment of constants, fixpoints and data structures and some aspects of what would be required of an implementation.

### Other Analyses

A sketch of how the LET-polymorphic deduction system might be extended to a second-order polymorphic system was given in Chapter 6. No discussion was given of the semantic correctness or implementation of such a system.

Two variations on the logics presented in Chapter 3 were also given. The first of these provides a static determination of terms with an infinite reduction behaviour. The detection of these terms was based on identifying the application of fixpoint combinators or constants to terms which strongly head need their argument. The strong head neededness logic was then employed to find out if a term containing such an application has an infinite reduction sequence to head normal form.

The second variation illustrated how the analysis may be split into two phases: firstly, deduce a conventional type for the term and return a set of annotations which give each occurrence of a variable in the term an ordinary type; secondly, use the logic presented to fill in the function type constructors, thus converting the type of the term from an ordinary type to a Boolean or Algebraic Reduction Type.

# Appendix A

## An Implementation in Orwell

This Appendix contains an *Orwell* ([51]) script which is an implementation of the intersection-style system for deducing Boolean Reduction Types. This implementation includes some of the extensions proposed in Chapter 6. In particular, certain term and type constants are implemented. Also, a terminating version of the unification algorithm is presented as a modification to the semi-decidable algorithm of Section 5.4.5 (the semi-decidable algorithm is also implemented in this Appendix).

### A.1 Orwell

This section of the Appendix contains a brief summary of the differences between Orwell and two similar functional languages which are perhaps more widely known in the functional language research community, namely Haskell and Miranda<sup>1</sup>.

An Orwell program is a *literate script* in which each line of the script which is to be interpreted commences with the symbol `>`. All other lines in the script are comment. This document is itself suitable as input to the *Orwell* interpreter. Like Miranda and Haskell, Orwell employs the offside rule to define the boundaries of local blocks of definitions. This rule states that any expression following an “=” symbol in a top-level equation or `where` expression must appear entirely to the right of this expression. Also like Miranda and Haskell, Orwell includes a script called the *standard prelude* which is automatically included before the interpretation of a script.

Orwell’s type system is similar to that of Miranda, both being based on the LET-polymorphic system described by Milner [47]. Both have a single numeric type and a polymorphic notion of equality which is applicable to elements of all types, including user-defined, except for functions. This is probably the main

---

<sup>1</sup>Miranda is a trademark of Research Software, Ltd.



difference from the type system of Haskell which employs a user-extendible notion of overloading of operators.

Unlike Miranda and Haskell, Orwell requires that each equation in a function definition be *disjoint*, i.e., that at most one equation is applicable in any application of the function. In patterns variables may not occur more than once. Orwell does provide a mechanism for including a “catch-all” equation to avoid writing out tedious lists of otherwise similar cases. This is implemented by including a line containing the pragma `%else`, and then following this line by the default case.

## A.2 Preliminaries

It is necessary that some preliminary definitions be made. Firstly, some general definitions not contained in the Orwell standard prelude are given, as well as the definition of *association lists*. Secondly, some functions for parsing various terms are introduced. (These parsing functions seem to have been re-invented more than once, see Fairbairn [21] and Hutton [37] for example, but are now fairly well-known though there are some extensions here). Finally, some functions for producing formatted output are described. These latter functions are essentially due to Peyton-Jones and Lester [54].

### A.2.1 Auxiliary functions

This subsection contains an excerpt from my library of auxiliary functions to complement the Orwell standard prelude. The only functions included here are those used to define the type inference system.

I prefer to write composition from left-to-right:

```
> %right 9 ;;
> (;;) :: (a -> b) -> (c -> d) -> (a -> d)
> (f ;; g) x = g (f x)
```

The following function “flattens” the application of a list valued function to each element of a list.

```
> concmap :: (a -> [b]) -> [a] -> [b]
> concmap f = map f ;; concat
```

Converting from curried forms of functions is often useful.

```
> uncurry2 f (a,b) = f a b
```

Some special cases of currying:

```
> pair x1 x2 = (x1,x2)
> triple x1 x2 x3 = (x1,x2,x3)
```

The following functions generalise the idea of currying (only a few cases are included).

```
> curry'1'of'2 f a b = f b a
> curry'1'of'3 f a b c = f c a b
> curry'2'of'3 f a = curry'1'of'2 (f a)
```

Converting a string of digits to an integer is a useful operation for parsers.

```
> stoi
>   = map (code ;; (+ (-code '0')))) ;;
>   build'num 0
>   where
>       build'num n (n':ns) = build'num (n*10+n') ns
>       build'num n [] = n
```

The following two functions are useful in “set-like” situations: the first ensures that each element of a list is unique (using Orwell’s built-in notion of equality), the second eliminates any objects which occur an even number of times in a list.

```
> uniq :: [a] -> [a]
> uniq (x:xs)
>   = uniq xs, if x $in xs
>   = x:uniq xs, otherwise
> uniq [] = []

> rem'dups (x:xs)
>   = rem'dups (xs -- [x]), if x $in xs
>   = x: rem'dups xs, otherwise
> rem'dups [] = []
```

`split` is a variation on the `filter` function: it returns a pair of lists representing all the positive and negative elements, respectively, in the input list.

```
> split :: (a -> bool) -> [a] -> ([a],[a])
> split p (x:xs)
>   = (x:ys,zs), if p x
>   = (ys,x:zs), otherwise
>   where
>       (ys,zs) = split p xs
> split p [] = ([],[])
```

`compose2` captures many common behaviours (for example, iterating a function over a binary structured data type).

```
> compose2 f1 f2 x1 x2 = f1 (f2 x1) (f2 x2)
```

The well-known finite-function builder is called `update` here.

```
> update f x t y
>   = t, if x = y
>   = f y, otherwise
```

The function `map2` is a generalisation of `map` to binary functions.

```
> map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
> %else
> map2 f xs ys = []
```

A function to print space characters is useful for formatting output

```
> spaces n
>   = ' ':spaces (n-1), if n > 0
>   = "", otherwise
```

Computing the maximum number in a list is used in this script to find the value at which renaming of variables should commence.

```
> maximum = foldr max 0
```

### A.2.2 Parsing

In this subsection functions for implementing recursive descent parsing in Orwell are presented. It is important to note that this code relies on lazy evaluation semantics in order to achieve reasonable efficiency.

A parser is a function from an input type to a list of parses, a parse being a pair of an output and the input remaining after obtaining the output.

```
> parser a b == a -> [(b,a)]
```

Atomic parses are modelled by `succeedwith` which does no further processing of the input and simply returns its first argument as result. Failure is the empty list of parses.

```
> fail inp = []
> succeedwith x inp = [(x,inp)]
> succeed = succeedwith []
```

A general mechanism for alternatives is encoded by the following function. Note the dependence on a lazy evaluation strategy and the pragma used to declare a right associative infix operator in Orwell.

```
> %right 8 ||
> (||) :: parser a b -> parser a b -> parser a b
> (p1 || p2) inp = p1 inp ++ p2 inp
```

Two parsers may be composed as follows.

```
> %right 9 >>
> (>>) :: parser a b -> parser a c -> parser a (b,c)
> (p1 >> p2) inp
>   = concmap (cont p2) (p1 inp)
>   where
>       cont p (x,rest) = map (cont' x) (p rest)
>       cont' x (y,rest) = ((x,y),rest)
```

The following variations on composition of parsers both discard the *result* of one of the parsers making up the composition. Firstly, composition with left discard.

```
> %right 9 !>
> (!>) :: parser a b -> parser a c -> parser a c
> (p1 !> p2) = p1;;concmap (snd;;p2)
```

Secondly, composition with right discard.

```
> %right 9 >!
> (>!) :: parser a b -> parser a c -> parser a b
> (p1 >! p2) inp
>   = concmap (cont p2) (p1 inp)
>   where
>       cont p (x,rest) = map (cont' x) (p rest)
>       cont' x (y,rest) = (x,rest)
```

In order to perform semantic actions (such as building up a parse tree) transformations are allowed on the results of parsing, as encapsulated here:

```
> %right 6 @
> (@) :: parser a b -> (b -> c) -> parser a c
> (p @ f) inp
>   = map (cont f) (p inp)
>   where cont f (x,rest) = (f x,rest)
```

Iteration of a parser is achieved by repeatedly applying the parser until it fails.

```
> some'of :: parser a b -> parser a [b]
> some'of p = (p >> some'of p @ uncurry2 (:)) || succeed
```

The following variation on the iterator above ensures that there are at least a minimum number of successful parses in order to obtain an overall successful parse.

```
> atleast :: num -> parser a b -> parser a [b]
> atleast 0 p = some'of p
> atleast (n+1) p = p >> atleast n p @ uncurry2 (:)
```

A further variation is useful where the first thing being parsed is not of the same type as the following things.

```
> %right 9 >*
> (>*) :: parser a b -> parser a c -> parser a (b,[c])
> p1 >* p2 = p1 >> some'of p2
```

Literals are encoded using Orwell's notion of generic equality.

```
> lit :: a -> parser [a] a
> lit x (y:xs)
>   = [(x,xs)], if x = y
>   = [], otherwise
> lit x [] = []
```

Generalising to lists of literals:

```
> accept :: [a] -> parser [a] [a]
> accept
>   = map lit ;;
>   foldr f succeed where f x1 x2 = x1 >> x2 @ uncurry2 (:)
```

Membership of a list of literals:

```
> any'of :: [a] -> parser [a] a
> any'of = map lit ;; foldr (||) fail
```

Parentheses are a common construct and deserve a parser to themselves.

```
> paren open p close = lit open !> p >! lit close
```

Similarly, lists of items with separators often occur, often in conjunction with some form of parentheses.

```
> list'body :: parser a b -> parser a c -> parser a [c]
> list'body separator item
>   = item >* body @ uncurry2 (:)
>     where body = separator !> item
```

Now a parser to make infix operators easy to implement. The arguments to `infix` are as follows:

`associator` determines associativity (usually `foldr1` or `foldl1`),

`factor` this may be a simple factor or an infix factor of lower precedence (in which case this is also constructed using `infix`),

`operator` recogniser for the syntactic form of the operator, and

`build` this function should construct a term from two factors.

```
> infix :: (d -> [b] -> e) ->
>          parser a b -> parser a c -> d -> parser a e
> infix associator factor operator build
>   = list'body operator factor @ associator build
```

Left and right associative infix operators can now be built.

```
> infixr = infix foldr1
> infixl = infix foldl1
```

Associative operators are dealt with by returning a list.

```
> infix a f o = list'body o f
```

Infix operators for which the form of the operator varies.

```
> op'infix :: ((c -> b -> b -> b) -> (b, [(c, b)]) -> b) ->
>            parser a b ->
>            parser a c ->
>            (c -> b -> b -> b) ->
>            parser a b
> op'infix associator factor operator build
>   = factor >* operator >> factor @ associator build
```

Right associative infix operators for which the form of the operator varies. (For brevity other forms are omitted).

```
> op'infixr = op'infix tangle'r
> tangle'r f (b,(c,b'):cbs) = f c b (tangle'r f (b',cbs))
> tangle'r f (b,[]) = b
```

### Some Common Parsers

Some parsers for common lexical constructs are now defined. Firstly, a categorization of characters is made:

```
> num'chars = "0123456789"
> whitespace'chars = " \n\t"
> alpha'chars
> = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
> punct'chars = "'~!@#$%^&*()_+|=|\/?>.<,:[{}]'\""
> alphanum'chars = alpha'chars ++ num'chars
```

Secondly, parsers for each category of characters:

```
> digit = any'of num'chars
> integer = atleast 1 digit
> alpha = any'of alpha'chars
> punctuation = any'of punct'chars
> white = any'of whitespace'chars
```

The following function is useful for parsers in which white space is insignificant.

```
> strip'white = filter (($in whitespace'chars);;(~))
```

### A.2.3 Printing

This section is closely based on the “pretty printing” functions of Peyton-Jones and Lester [54]. Their main motivation for introducing these functions is to avoid the repeated concatenation of strings forming the output—the result being that the cost of printing is quadratic in the length of output. Consider the expression:

```
(xs1 ++ xs2) ++ xs3
```

The time taken to compute this expression (as Peyton-Jones and Lester point out) is proportional to  $2 \times \#xs1 + \#xs2$ . Always bracketing the output expression the other way will produce a result which prints in time linear in the length of the output, but in general achieving this complicates a script considerably.

I introduce the following variation on the method of Peyton-Jones and Lester. The first step is to define an algebraic data type to represent the output. This would need enhancement for more sophisticated use, but suits the current script well.

```
> out'seq
>   ::= Empty
>       | Str [char]
>       | Append out'seq out'seq
>       | Indent out'seq
>       | Newline
```

The only unusual thing about the above definition is the case `Indent out'seq`. This is used to line up its argument `out'seq` all to the right of the current column on the printer.

To make the output easier to write, define the following infix operator for `Append`.

```
> %right 9 &&
> (&&) = Append
```

We often want to insert an output sequence between two others.

```
> between s s1 s2 = s1 && s && s2
```

A natural generalisation of `Append` is the following:

```
> join :: [out'seq] -> out'seq
> join = foldr (&&) Empty
```

Similarly, a natural generalisation of `between` can be defined:

```
> interleave :: out'seq -> [out'seq] -> out'seq
> interleave sep [] = Empty
> %else
> interleave sep os = foldr1 (between sep) os
```

A useful instance of `interleave` is a function for converting a list of output sequences into lines of output sequences.

```
> lines :: [out'seq] -> out'seq
> lines = interleave Newline
```

Print a number—uses *Orwell's* overloaded `show` function.



```
> show'num :: num -> out'seq
> show'num = show;;Str
```

Often used is the function for displaying parenthesised objects.

```
> show'par s = between s (Str "(") (Str ")")
```

### Printing Output Sequences

The following functions convert output sequences into strings ready for display or printing.

```
> print :: out'seq -> [char]
> print seq = print'it 0 [(seq,0)]

> print'it :: num -> [(out'seq, num)] -> [char]
> print'it col ((Newline, indent) : seqs)
>   = "\n" ++
>     (spaces indent) ++
>     (print'it indent seqs)
> print'it col ((Indent seq, indent) : seqs)
>   = print'it col ((seq, col) : seqs)
> print'it col ((Str s, indent) : seqs)
>   = s ++ print'it (col + #s) seqs
> print'it col ((Empty, indent) : seqs)
>   = print'it col seqs
> print'it col ((Append seq1 seq2, indent) : seqs)
>   = print'it col ((seq1, indent) : (seq2, indent) : seqs)
> print'it col [] = ""
```

### A.2.4 Association Lists

This section contains a standard treatment of association lists. Included is a parser and a printer for association lists.

```
> assoc a b == [(a,b)]

> empty'assoc = []

> update'assoc :: assoc a b -> a -> b -> assoc a b
> update'assoc xys x y = (x,y):xys

> delete :: assoc a b -> a -> assoc a b
> delete xys x = [(z,y) | (z,y) <- xys; x /= z]
```

```
> pop :: assoc a b -> a -> assoc a b
> pop ((x,y):xys) z
>   = xys, if x = z
>   = (x,y):pop xys z, otherwise
> pop [] z = []

> spop :: assoc a b -> a -> b -> (assoc a b, b)
> spop ((x,y):xys) z b
>   = (xys,y), if x = z
>   = ((x,y):xys',b'), otherwise
>     where (xys',b') = spop xys z b
> spop [] z b = ([],b)

> in'dom :: a -> assoc a b -> bool
> x $in'dom xys = x $in dom xys

> val :: assoc a b -> a -> b
> val ((x,y):xs) z = y, if x = z
>                   = val xs z, otherwise

> sval :: assoc a b -> a -> b -> b
> sval ((x,y):xs) z a
>   = y, if x = z
>   = sval xs z a, otherwise
> sval [] z a = a

> range :: assoc a b -> [b]
> range a = [y | (x,y) <- a]

> dom :: assoc a b -> [a]
> dom a = [x | (x,y) <- a]

> show'assoc showx showy ((x,y):xys)
>   = showxy, if xys = []
>   = showxy &&
>     Str ", " &&
>     show'assoc showx showy xys, otherwise
>   where
>     showxy = showx x && Str ":" && showy y
> show'assoc showx showy [] = Empty
```

```

> read'assoc readx ready
>   = strip'white ;; parse'assoc readx ready ;; hd ;; fst

> parse'assoc readx ready
>   = paren
>     '{'
>       (list'body (lit ',') (readx >> lit ':' !> ready))
>     '}'
>   || (lit '{' >> lit '}' @ const empty'assoc)

```

### A.3 An Implementation of Boolean Algebra

The representation used of a Boolean algebra (as a ring structure) is also an algebraic data type. Another possibility would be a list based representation, but this may be a little less clear for didactic purposes.

```

> ba
>   ::= Zero
>       | One
>       | BAVar num
>       | Sum [ba]
>       | Prod [ba]

```

Some associated recognisers which are used in guards:

```

> is'Sum (Sum bs) = True
> %else
> is'Sum b = False

> is'BAVar (BAVar n) = True
> %else
> is'BAVar b = False

```

A function for finding the lexically greatest arrow variable:

```

> max'bavar = arrow'vars;; maximum

```

Computation of a list of all the arrow variables in a ring expression is concisely expressed using the auxiliary function `conccmap`.

```

> arrow'vars (BAVar n) = [n]
> arrow'vars (Sum bs) = conccmap arrow'vars bs
> arrow'vars (Prod bs) = conccmap arrow'vars bs

```

```
> %else
> arrow'vars b = []
```

### A.3.1 Parsing and Printing

Now I introduce a parser and pretty printer for Boolean algebras.

The parser converts the input string directly into a ring term in sum-of-products normal form using the `operator`. Lexical analysis is very crude—simply remove all “white” space (tab, newline and space characters). The hard work is done by the call to `parse'ba`. No error checking is performed on the result!

```
> read'ba = strip'white;;parse'ba;;hd;;fst
```

By convention, addition binds less tightly than does multiplication and so this determines the structure of the parser. The parsing itself is very simple—`infix` associative operators are defined using `infixa` and atoms are defined as literals using `lit` or as a parenthesised expression using `paren`. The hard work of transforming to sum-of-products normal form is done by the functions `xor` and `conj` which are discussed below. The simplicity of the parser is a testimony to appropriate abstraction and the naturalness of the expression of this abstraction by higher-order functions.

```
> parse'ba = infixa ba'conj (lit '+') @ xor
> ba'conj = infixa ba'atom (lit '.') @ conj
> ba'atom = simple'atom || paren '(' parse'ba ')'

> simple'atom
>   = (lit '1' @ const One)
>     || (lit '0' @ const Zero)
>     || (lit 'v' !> integer @ stoi;;BAVar)
```

Pretty printing a ring expression is defined by mimicking their inductive definition. The functions `map` and `interleave` nicely capture the iteration required.

```
> show'ba Zero = Str "0"
> show'ba One = Str "1"
> show'ba (BAVar n) = Str "v" && show'num n
> show'ba (Prod bs) = interleave (Str ".") (map show'ba bs)
> show'ba (Sum bs) = interleave (Str "+") (map show'ba bs)
```

### A.3.2 Sum-of-Products Normal Form

check'single is used to avoid building unnecessary levels of Sum's and Prod's.

```
> check'single :: ([ba] -> ba) -> ba -> [ba] -> ba
> check'single f e [] = e
> check'single f e [x] = x
> %else
> check'single f e bs = f bs
```

Simplification of a sum term is done by xor'simplify which is called from xor.

```
> xor :: [ba] -> ba
> xor [x] = x
> %else
> xor bs = (xor'simplify;;check'single Sum Zero) bs
```

Apart from implementing the simplifications expressed by the rewrite system:

$$\begin{aligned}
 0 + x &\rightarrow x \\
 x + x &\rightarrow 0 \\
 0 * x &\rightarrow 0 \\
 1 * x &\rightarrow x \\
 x * x &\rightarrow x \\
 x * (y + z) &\rightarrow x * y + x * z,
 \end{aligned}$$

xor'simplify also orders all Boolean variables into ascending lexical order using xor'insert which simplifies the unification process described below.

```
> xor'simplify (Zero:bs) = xor'simplify bs
> xor'simplify (Sum bs:rest)
>   = xor'simplify (bs ++ rest)
> xor'simplify (BAVar n:rest)
>   = xor'insert (BAVar n) (xor'simplify rest)
> xor'simplify [] = []
> %else
> xor'simplify (x:rest)
>   = xs--[x], if x $in xs
>   = x:xs, otherwise
>   where xs = xor'simplify rest
> xor'insert (BAVar n) (BAVar m:rest)
```

```

> = BAVar n : BAVar m : rest, if n < m
> = rest, if n = m
> = BAVar m : conj'insert (BAVar n) rest, otherwise
> xor'insert b [] = [b]
> %else
> xor'insert b (b':bs) = b' : xor'insert b bs

```

Similarly, constructing a product term involves both simplification and sorting of variables.

```

> conj :: [ba] -> ba
> conj [x] = x
> %else
> conj bs
>   = (split (is'Sum));;
>       uncurry2 (++);;
>       conj'simplify;;
>       check'single Prod One) bs

> conj'simplify (Zero:bs) = [Zero]
> conj'simplify (One:bs) = conj'simplify bs
> conj'simplify (Prod bs:rest) = conj'simplify (bs ++ rest)
> conj'simplify (Sum bs:rest) = [xor [conj (b:rest) | b <- bs]]
> conj'simplify (BAVar n:rest)
>   = conj'insert (BAVar n) (conj'simplify rest)
> conj'simplify [] = []

> conj'insert (BAVar n) (BAVar m:rest)
>   = BAVar n : BAVar m : rest, if n < m
>   = BAVar n : rest, if n = m
>   = BAVar m : conj'insert (BAVar n) rest, otherwise
> conj'insert b [] = [b]
> %else
> conj'insert b (b':bs)
>   = [Zero], if b' = Zero
>   = b' : conj'insert b bs, otherwise

```

### A.3.3 Unification of Boolean Rings

The following algorithm is based on Boole's variable elimination procedure of 1847. The first step is to add (using xor) the terms to be unified (they are then implicitly equated to Zero). After this the equation solver is called.

```
> bunify :: ba -> ba -> ba -> ba
> bunify b1 b2 = bsolve (xor [b1,b2])
```

The equation solver stops as soon as it has an equation of the form `Zero = Zero`. (If there is no solution then the Orwell system will always halt with an error message, this being “equivalent” to the unsolvable value *bottom*). Otherwise `bsolve` eliminates a variable, recursively calls itself and, if the recursive call returns a defined result, builds up a substitution for the eliminated variable (which is called the “pivot” variable here).

```
> bsolve :: ba -> ba -> ba
> bsolve Zero = id
> %else
> bsolve b
>   = apply'to'ba
>       (update s piv
>         (xor [ piv,
>               fb,
>               conj [tb,piv],
>               conj [fb,piv],
>               conj [fb,tb,piv] ]))
>       )
>   where
>       (ts,fs,piv) = pivot b
>       (tb,fb) = (ts b,fs b)
>       s = bsolve (conj [tb,fb])
```

`pivot` computes the actual variable elimination. It returns functions to convert the variable being eliminated to both true and false (1 and 0) and also the variable itself. This latter is so that the pivot variable can be used to build the result substitution.

```
> pivot (BAVar n)
>   = (apply'to'ba (update id (BAVar n) One),
>      apply'to'ba (update id (BAVar n) Zero),
>      BAVar n)
> pivot (Prod bs) = pivot (hd bs)
> pivot (Sum bs) = pivot (hd (filter (~=One) bs))
```

The following function homomorphically extends substitution on variable arrows to substitution on ring expressions.

```
> apply'to'ba f (BAVar n) = f (BAVar n)
```

```

> apply'to'ba f (Prod bs) = conj (map (apply'to'ba f) bs)
> apply'to'ba f (Sum bs) = xor (map (apply'to'ba f) bs)
> %else
> apply'to'ba f b = b

```

## A.4 Lambda Terms

Now it is time to define  $\lambda$ -terms. Since these terms will not be reduced they are represented by an algebraic data type in the ordinary (naive) way:

```

> lam ::= V num | APP lam lam | ABS num lam

```

The following recognition functions are useful below.

```

> app'term (APP e1 e2) = True
> %else
> app'term e = False
> abs'term (ABS n e) = True
> %else
> abs'term e = False
> var'term (V n) = True
> %else
> var'term e = False

```

The free variables of a term are returned in a list, being careful to ensure that each variable occurs at most once in the list.

```

> fv (V n) = [n]
> fv (APP e1 e2) = uniq (compose2 (++) fv e1 e2)
> fv (ABS n e) = fv e -- [n]

```

### A.4.1 Parsing $\lambda$ -terms

A parser for  $\lambda$ -terms is easy to construct. Lexical analysis is again simplistic and error checking omitted.

```

> read'lam = strip'white ;; parse'lam ;; hd ;; fst

```

The parser works by assigning the empty parser to application (since it is represented by concatenation of symbols) and then building an infix left associative parser which combines the resulting elements using APP (thus building a parse tree).

Subterms of an application term are either variables, parenthesised expressions or abstractions which are straightforward to define.



```

> parse'lam = infixl simple'term succeed APP
> simple'term = (var @ V) || paren '(' parse'lam ')' || abs
> abs = lit '\\ ' !> var >> lit '.' !> parse'lam @ uncurry2 ABS
> var = lit 'x' !> integer @ stoi

```

### A.4.2 Printing $\lambda$ -terms

The function to print a  $\lambda$ -term is defined below. The main feature of this is the APP case in which attention is paid to including the minimum number of parentheses so as to enhance the readability of the output.

```

> show'lam (V n) = Str "x" && show'num n
> show'lam (ABS n e2)
>   = Str "\\x" && show'num n && Str "." && show'lam e2
> show'lam (APP e1 e2)
>   = par'lam e1 && Str " " && par'lam e2,
>     if abs'term e1 & ~var'term e2
>   = par'lam e1 && Str " " && show'lam e2, if abs'term e1
>   = show'lam e1 && Str " " && par'lam e2, if ~var'term e2
>   = show'lam e1 && Str " " && show'lam e2, otherwise

> par'lam = show'lam ;; show'par

```

## A.5 Intersection Reduction Types

The next step is the definition of the reduction types—in this case intersection Boolean reduction types. The definition incorporates an additional constant Bool, as discussed in Chapter 6.

```

> ty
>   ::=
>     TV num
>     | Omega
>     | I ty ty
>     | F ba ty ty
>     | Bool

```

As usual, recognisers are helpful to supplement pattern matching.

```

> tv'type (TV n) = True
> %else
> tv'type t = False
> ftype (F b t1 t2) = True

```

```

> %else
> ftype t = False
> itype (I t1 t2) = True
> %else
> itype t = False
> omega t = t = Omega
> type'constant t = t = Bool

```

The arrow expression is extracted from a reduction type as follows:

```

> arrow (F b t1 t2) = b

```

Type variables are found by a walk over the structure of a type, using `compose2` to direct the iteration.

```

> tvars (TV n) = [n]
> tvars Omega = []
> tvars (I t1 t2) = compose2 (++) tvars t1 t2
> tvars (F b t1 t2) = compose2 (++) tvars t1 t2
> %else
> tvars t = []

```

The arrow variables in a type are also found by a walk over the structure of a type, again using `compose2` to direct the iteration.

```

> avars (I t1 t2) = compose2 (++) avars t1 t2
> avars (F b t1 t2) = arrow'vars b ++ compose2 (++) avars t1 t2
> %else
> avars t = []

> max'avar = avars;; maximum
> max'tvar = tvars;; maximum

```

### A.5.1 Parsing Reduction Types

Parsing reduction types follows the now familiar pattern. One difference is the need to record the infix arrow expression used to construct the type. This is ably accomplished by `op'infixr`. As in the case for Boolean ring expressions, the resulting parse tree is reduced to a canonical form using functions `reduce'F` and `reduce'I` which are defined below.

```

> read'ty = strip'white ;; parse'ty ;; hd ;; fst

```

```

> parse'ty
>   = op'infixr
>       inter'ty
>       (accept "->[" !> parse'ba >! lit '']')
>       reduce'F
> inter'ty = infixr simple'ty (lit '&') reduce'I

> simple'ty
>   = tvar ||
>       (lit 'w' @ const Omega) ||
>       (lit 'B' @ const Bool) ||
>       paren '(' parse'ty ')'

> tvar = lit 'a' !> integer @ stoi ;; TV

```

### A.5.2 Simplification of Reduction Types

The two functions below correspond to a rewrite system for intersection reduction types:

$$\begin{aligned}
 \sigma \cap \sigma &\rightarrow \sigma \\
 \sigma \cap \omega &\rightarrow \sigma \\
 \omega \cap \sigma &\rightarrow \sigma \\
 \rho b(\sigma \cap \tau) &\rightarrow (\rho b \sigma) \cap (\rho b \tau) \\
 \rho b \omega &\rightarrow \omega.
 \end{aligned}$$

An important advantage for using simplified types is the ability to use syntactic identity to determine equality amongst them.

```

> reduce'I :: ty -> ty -> ty
> reduce'I t1 t2
>   = t1, if t1 = t2 \ / omega t2
>   = t2, if omega t1
>   = t1, if type'constant t1 & tv'type t2
>   = t2, if type'constant t2 & tv'type t1
>   = Omega, if type'constant t1 & type'constant t2
>   = I t1 t2, otherwise

> reduce'F :: ba -> ty -> ty -> ty
> reduce'F b t1 (I t2 t3)
>   = I (reduce'F b t1 t2) (reduce'F b t1 t3)
> reduce'F b t1 Omega = Omega
> %else

```

```
> reduce'F b t1 t2 = F b t1 t2
```

### A.5.3 Printing Reduction Types

In printing intersection reduction types, attention is given to minimising the number of parentheses.

```
> show'ty (TV n) = Str "a" && show'num n
> show'ty Omega = Str "w"
> show'ty Bool = Str "B"
> show'ty (I t1 t2) = show'i t1 t2
> show'ty (F b1 (F b2 t1 t2) t3)
>   = Str "(" &&
>     show'ty (F b2 t1 t2) &&
>     Str ") ->[" &&
>     show'ba b1 &&
>     Str "]" " &&
>     show'ty t3
> %else
> show'ty (F b t1 t2)
>   = show'ty t1 &&
>     Str " ->[" &&
>     show'ba b &&
>     Str "]" " &&
>     show'ty t2

> show'i t1 t2
>   = par'ty t1 &&
>     Str " & " &&
>     par'ty t2, if ftype t1 & ftype t2
>   = par'ty t1 &&
>     Str " & " &&
>     show'ty t2, if ftype t1
>   = show'ty t1 &&
>     Str " & " &&
>     par'ty t2, if ftype t2
>   = show'ty t1 &&
>     Str " & " &&
>     show'ty t2, otherwise

> par'ty = show'ty ;; show'par
```

### A.5.4 Substitution for types

*Chains*, as defined in Chapter 5, are represented by functions from types to types.

```
> chain == ty -> ty
```

The first kind of chain element is the substitution. The following functions are useful for constructing substitutions of various kinds.

```
> subst :: ty -> num -> chain
> subst t n = subst'all' t [n]

> subst'all :: ty -> [num] -> chain
> subst'all t (n:ns) t' = subst'all' t (n:ns) t'
> subst'all t [] t' = t'

> subst'all' :: ty -> [num] -> chain
> subst'all' t ns (TV m)
>   = t, if m $in ns
>   = TV m, otherwise
> subst'all' t ns (F b t1 t2)
>   = compose2 (reduce'F b) (subst'all' t ns) t1 t2
> subst'all' t ns (I t1 t2)
>   = compose2 reduce'I (subst'all' t ns) t1 t2
> %else
> subst'all' t ns t' = t'
```

Applying an arrow substitutions to a type is also a required operation.

```
> extend'ba'to'ty :: (ba -> ba) -> ty -> ty
> extend'ba'to'ty f (F b t1 t2)
>   = compose2 (F (f b)) (extend'ba'to'ty f) t1 t2
> extend'ba'to'ty f (I t1 t2)
>   = compose2 reduce'I (extend'ba'to'ty f) t1 t2
> %else
> extend'ba'to'ty f t = t
```

The second kind of chain element is expansion whose definition is given in a later section.

### A.5.5 State-based functions

Before describing expansions some functions to deal with state dependent concepts are introduced. A *state-based function* is any function of the following

form:

```
> statefun a b c d == a -> b -> (c, d)
```

In the definition of `statefun`, `a` is the state, `b` is the input, `c` is the output and `d` is the output state.

Any ordinary function can be converted into a state-based one as follows:

```
> make'state :: (b -> c) -> statefun a b c a
> make'state f n x = (f x, n)
```

The constant state function is defined by ignoring its input.

```
> const'state :: b -> statefun a b b a
> const'state t n t' = (t, n)
```

The following function is a useful iterator over state-based functions.

```
> map'state :: statefun a b c d -> statefun a [b] [c] d
> map'state f s (x:xs)
>   = (y:ys,s'')
>   where
>       (ys,s'') = map'state f s' xs
>       (y,s') = f s x
> map'state f s [] = ([],s)
```

Composing state-based functions is captured by `compose'state`.

```
> compose'state :: statefun a b c d ->
>                 statefun d c e f ->
>                 statefun a b e f
> compose'state c1 c2 n t
>   = c2 n1 t1
>   where
>       (t1,n1) = c1 n t
```

`compose2'state` is like the useful auxiliary function `compose2` except for the need to thread the states `n`, `n1` and `n2`. Note that in `compose2'state b->f` is often instantiated to a state-based function.

```
> compose2'state :: (a -> a -> b -> f) ->
>                 statefun b e a b ->
>                 (e -> e -> b -> f)
> compose2'state u c t1 t2 n
```

```

> = u t1' t2' n2
>   where
>       (t1',n1) = c n t1
>       (t2',n2) = c n1 t2

```

The following is useful in conjunction with `compose2's` state.

```

> coalesce :: (a -> b -> c) -> a -> statefun b d c d
> coalesce f t1 t2 = pair (f t1 t2)

```

### A.5.6 Expansion

Expansion is a somewhat intricate operation whose definition is made much simpler using the state-based function abstractions of the previous subsection. Defining this operation is a two step process: first define the expansion context and then define the expanding substitution (see Chapter 5).

Some auxiliary concepts are useful. Firstly, we need to generate all non-i-subtypes of a type.

```

> nonisub :: ty -> [ty]
> nonisub (TV n) = [TV n]
> nonisub (F b t1 t2)
>   = F b t1 t2 : compose2 (++) nonisub t1 t2
> nonisub (I t1 t2)
>   = compose2 (++) nonisub t1 t2
> %else
> nonisub t = [t]

```

Now a function to check if a type variable is the rightmost type variable in a type. (This expects the type to be in reduced form).

```

> right'tvar :: [num] -> ty -> bool
> right'tvar ms (TV n) = n $in ms
> right'tvar ms (F b t1 t2)
>   = right'tvar ms t2, otherwise
> %else
> right'tvar ms t = False

```

#### The Expansion Context

As defined in Chapter 5 the expansion context is built in an iterative fashion. All the non-i-subtypes of the expansion base type must be included, the actual iteration then being performed by `ec'`.

```

> ec :: [ty] -> ty -> [ty]
> ec b t
>   = nonisub t ++
>     concmap nonisub (ec' (uniq (concmap nonisub b)) (tvars t))

> ec' b newts
>   = [], if ts' = []
>   = ts' ++ ec' b' (concmap tvars ts'), otherwise
>   where
>       (in,out)
>         = split
>           (uncurry2 right'tvar)
>           [(newts,t) | t <- b]
>       ts' = range in
>       b'  = range out

```

### The Expanding Substitution

Using the expansion context the expanding substitution can be defined. This is accomplished by first building a pair of renaming substitutions. Finally, an i-type is constructed which consists of two renamed duplicates of the input type. Note the used of the state-based function abstractors.

```

> e :: [ty] -> statefun num ty chain num
> e b n t
>   = (expansion,n2)
>   where
>       expansion
>         = exp' context
>           (foldr (;;) id subs1)
>           (foldr (;;) id subs2)
>       (subs1,n1)
>         = map'state rename n context
>       (subs2,n2)
>         = map'state rename n1 context
>       context = uniq (ec b t)

> rename :: statefun num ty chain num
> rename m (TV n) = (subst (TV m) n, m+1)
> rename m (F b t1 t2)
>   = compose2'state (coalesce (;;)) rename t1 t2 m
> rename m (I t1 t2)
>   = compose2'state (coalesce (;;)) rename t1 t2 m

```



```

> %else
> rename m t = (id, m)

> exp' :: [ty] -> chain -> chain -> chain
> exp' c sub1 sub2 s
>   = I (sub1 s) (sub2 s), if s $in c
>   = exp'' c sub1 sub2 s, otherwise

> exp'' :: [ty] -> chain -> chain -> chain
> exp'' c sub1 sub2 (TV m) = TV m
> exp'' c sub1 sub2 (I t1 t2)
>   = compose2 I (exp' c sub1 sub2) t1 t2
> exp'' c sub1 sub2 (F b t1 t2)
>   = compose2 (reduce'F b) (exp' c sub1 sub2) t1 t2
> %else
> exp'' c sub1 sub2 t = t

```

## A.6 Unification

Unification of reduction types is complicated by the need to thread a state through the process. This state is an integer representing the supply of new names for type variables. However, appropriate use of higher-order functions makes this process much clearer and isolates the behaviour into well-defined places.

The more abstract version of this algorithm in Chapter 5 serves as a useful guide as to the various parts of the following algorithm. A point to note is that the expansions occur in two places rather than the single occurrence in  $U_I$ . (See the treatment of type variables and the catch-all “%else” clause).

```

> unify :: [ty] -> ty -> ty -> num -> (chain,num)
> unify b (TV m) t
>   = pair (subst'all (TV m) tv), if (itype t) & occurs
>   = pair (subst'all Omega (m:tv)), if (~tv'type t) & occurs
>   = u'compose em unify b (TV m) t, if itype t
>   = pair (subst t m), otherwise
>   where
>       occurs = m $in tv
>       tv = tvars t
>       em = curry'2'of'3 e b (TV m)
> unify b (F b1 t1 t2) (F b2 t3 t4)
>   = u'compose
>     (combine bu (unify b' (bu t1) (bu t3))) unify b' t2 t4

```

```

>   where
>       bu = extend'ba'to'ty (bunify b1 b2)
>       b' = map bu b
> unify b (I t1 t2) (I t3 t4)
>   = u'compose (unify b t1 t3) unify b t2 t4
> unify b Omega t = pair (subst'all Omega (tvars t))
> unify b Bool Bool = pair id
> %else
> unify b t1 t2
>   = unify b t2 t1, if tv'type t2 \ / omega t2
>   = u'compose e2 unify b t1 t2, if itype t1
>   = u'compose e1 unify b t1 t2, if itype t2
>   where
>       e1 = curry'2'of'3 e b t1
>       e2 = curry'2'of'3 e b t2

> combine f1 f2 n = (f1;;f3,m) where (f3,m) = f2 n

> u'compose f1 f2 b t1 t2 n
>   = (c1;;c2,n2)
>   where
>       (c1,n1) = f1 n
>       (c2,n2) = compose2 (f2 (map c1 b)) c1 t1 t2 n1

```

### A.6.1 Guaranteeing Termination

The following version of `unify` guarantees termination by only permitting a finite number of expansion steps to be taken during the unification process.

```

> finite'unify :: [ty] -> ty -> ty -> num -> (chain,num)
> finite'unify = unify' 5

> unify' :: num -> [ty] -> ty -> ty -> num -> (chain,num)
> unify' 0 b t1 t2
>   = pair (subst'all Omega (tvars t1 ++ tvars t2))
> unify' (m+1) b t1 t2 = unify'' m b t1 t2

> unify'' :: num -> [ty] -> ty -> ty -> num -> (chain,num)
> unify'' n b (TV m) t
>   = pair (subst'all Omega tvt),
>   if (~tv'type t) & (m $in tvt)
>   = u'compose em (unify' n) b (TV m) t, if itype t
>   = pair (subst t m), otherwise

```

```

>   where
>       tvt = tvars t
>       em = curry'2'of'3 e b (TV m)
> unify'' n b (F b1 t1 t2) (F b2 t3 t4)
>   = u'compose
>       (combine bu (unify'' n b' (bu t1) (bu t3)))
>       (unify'' n) b t2 t4
>   where
>       bu = extend'ba'to'ty (bunify b1 b2)
>       b' = map bu b
> unify'' n b (I t1 t2) (I t3 t4)
>   = u'compose (unify'' n b t1 t3) (unify'' n) b t2 t4
> unify'' n b Omega t = pair (subst'all Omega (tvars t))
> unify'' n b Bool Bool = pair id
> %else
> unify'' n b t1 t2
>   = unify'' n b t2 t1, if tv'type t2 \ / omega t2
>   = u'compose e2 (unify' n) b t1 t2, if itype t1
>   = u'compose e1 (unify' n) b t1 t2, if itype t2
>   where
>       e1 = curry'2'of'3 e b t1
>       e2 = curry'2'of'3 e b t2

```

## A.7 Constants

Some example constants from Chapter 6 are defined here. Each time a constant is typed the type assigned to it is given unique type and arrow variables.

```

> lookup'const :: statefun (num,num) string ty (num,num)
> lookup'const (an, vn) s = freshen (an, vn) (get'type s)

> get'type :: string -> ty
> get'type "Y" = read'ty "(a1->[v1]a1)->[1]a1"
> get'type "IF" = read'ty "B->[1]a1->[v1]a1->[v1+1]a1"
> get'type "ZERO" = read'ty "Int ->[1] B"
> get'type "PLUS" = read'ty "Int ->[1] Int ->[1] Int"
> get'type "PRED" = read'ty "Int ->[1] Int"

```

## A.8 Type Inference

Now the algorithm for performing type inference can be defined, closely based on that appearing in Chapter 5 for type inference of intersection types.

```
> type :: statefun (assoc num ty)
>           lam
>           (ty,num -> ba)
>           (assoc num ty)
> type a e
>   = ((t,v),a')
>   where
>       ((t,v,c),(a',n,vn))
>       = pp (a,
>             find'max max'tvar,
>             find'max max'avar)
>             e
>       find'max f = (foldr max (-1) (map f (range a)))+1

> pp'state == (assoc num ty, num, num)

> pp :: statefun pp'state lam (ty,num -> ba,chain) pp'state
> pp (a,mn,vn) (V n)
>   = (
>       (TV mn, update (const Zero) n One, id),
>       (update'assoc a' n s, mn+1, vn)
>   )
>   where
>       (a',s') = spop a n Omega
>       s = reduce'I s' (TV mn)
> pp (a,mn,vn) (N n)
>   = ((Int, const Zero, id), (a,mn,vn))
> pp (a,mn,vn) (C c)
>   = ((c'type, const Zero, id), (a,mn',vn'))
>   where
>       (c'type,(mn',vn')) = lookup'const (mn,vn) c
> pp (a,mn,vn) (ABS n e)
>   = ((reduce'F (v n) s t, update v n Zero, c), (a',mn',vn'))
>   where
>       (a1,s1) = spop a n Omega
>       ((t,v,c),(a2,mn',vn')) = pp (a1,mn,vn) e
>       (a3,s) = spop a2 n Omega
>       a' = update'assoc a3 n s1, if s1 /= Omega
```

```

>           = a3, otherwise
> pp (a,mn,vn) (APP e1 e2)
>   = (
>     (c3 t,
>       build'app'v
>         (v1;;ty'to'ba (c2;;c3))
>         (ty'to'ba c3 (BAVar new'avar))
>         (v2;;ty'to'ba c3),
>       c1;;c2;;c3
>     ),
>     (
>       [(x,c3 y) | (x,y) <- a2],
>       m4,
>       new'avar+1
>     )
>   )
>   where
>     ((t1,v1,c1),(t2,v2,c2),(a2,m2,new'avar))
>       = compose2'state triple pp e1 e2 (a,mn,vn)
>     t = TV m2
>     t1' = c2 t1
>     t2' = F (BAVar new'avar) t2 t
>     m3 = m2+1
>     b = t1' : t2' : range a2
>     (c3,m4) = unify b t1' t2' m3

> build'app'v v1 b v2 x
>   = xor [b1, b2, conj [b1,b2]]
>   where
>     b1 = v1 x
>     b2 = conj [b, v2 x]

> ty'to'ba f b = arrow (f (F b Bool Bool))

```

## A.9 The Top-level of the Type Inference System

The top-level of the implementation is defined by the following pair of functions. The first simply calls the second with the representation of the empty assumption set, the second arranges the parsing, type inference and printing phases of the implementation.

```

> ti :: string -> string
> ti = tia "{}"

> tia :: string -> string -> string
> tia as s
>   = print
>     (show'type'assign (m,type (read'assoc var parse'ty as) m))
>     where m = read'lam s

```

To print out the results of type inference the following function is useful. Note that by using `show'lam m` instead of just `s` the term is always printed in the canonical format.

```

> show'type'assign (m,((t,v),a))
>   = Str "{"                                     &&
>     show'assoc (show'num;;(Str "x"&&)) show'ty a &&
>     Str "}" |- V-/->["                           &&
>     interleave (Str ",")
>     (map2 (between (Str ":="))
>     [Str "x" && show'num x | x <- active]
>     [(v;;show'ba) x | x <- active])             &&
>     Str "]" "                                     &&
>     show'lam m                                   &&
>     Str ":"                                     &&
>     show'ty t
>     where active = fv m

```

## A.10 Examples

In this section some example executions from this script are shown. These were run on an Apple Macintosh LC using the Frontline Orwell system [51] of the University of Western Australia.<sup>2</sup>

A simple term which requires the use of *expansion* (see Chapter 5) is the following:

```

{x2:a7 & a11, x1:(a7 ->[v1] a1 ->[v0] a6) & (a11 ->[v1] a1)}
|- V-/->[x1:=1,x2:=v1] (\x1.x1 x1) (x1 x2):a6

```

Reductions = 4990, Cells = 20903, Name space used = 6380

---

<sup>2</sup>Macintosh is a trademark of Apple Computer, Inc.

A more complex term which requires the use of several expansions<sup>3</sup> is the following:

```
{ } |- V-/->[]
\x1.(\x2.x1 (x2 (\x3.\x4.x3 x4)) (x2 (\x3.\x4.x4 x3)))
(\x5.x5 x5 x5):
(((a55 ->[1] a56) ->[1] a55 ->[1] a56) ->[v2.v5.v8+v5+v8]
  ((a112 ->[1] (a112 ->[1] a114) ->[1] a114) ->[1] a92)
    ->[1] a92)
->[v5.v8] a18)
->[1] a18
```

Reductions = 52609, Cells = 297509, Name space used = 6380

Some examples involving constants:

```
{ } |- V-/->[]
Y (\x1.\x2.\x3.\x4.IF x2 x3 (x1 x2 x4 x3)):
B ->[1] a11 ->[1+v7] a11 ->[v7] a11
```

Reductions = 19305, Cells = 73827, Name space used = 5029

```
{ } |- V-/->[]
Y (\x1.\x2.\x3.\x4.IF x2 x3 (x1 x2 x3 x4)):
B ->[1] a9 ->[1+v5*v7+v7] a11 ->[v6*v7] a9
```

Reductions = 13048, Cells = 42159, Name space used = 5029

```
{ } |- V-/->[]
Y (\x1.\x2.\x3.\x4.
  IF (ZERO x4) (PLUS x2 x3) (x1 x3 x2 (PRED x4))):
Int ->[1+v11] Int ->[1+v8*v11+v8+v11] Int ->[1] Int
```

Reductions = 22448, Cells = 75039, Name space used = 5029

---

<sup>3</sup>Even on a relatively slow machine, interpreting SK-combinator graphs, this type was found for this term very quickly. However, the use of many expansions has caused a ten-fold increase in reductions required and space used.

# Bibliography

- [1] S. Abramsky. Strictness analysis and polymorphic invariance. In *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, Berlin, 1986.
- [2] C.A. Baker-Finch. Relevant implication and hereditary strictness. Draft manuscript, 1991.
- [3] H.P. Barendregt. *The Lambda-Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.
- [4] H.P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, December 1983.
- [5] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed Reduction and Spine Strategies for the lambda-calculus. Technical report, Centre for Mathematics and Computer Science, Amsterdam, May 1986.
- [6] G. Boole. *The Mathematical Analysis of Logic*. Macmillan, 1847. Reprinted 1948, B. Blackwell.
- [7] K. Bruce and A. Meyer. A Completeness Theorem for Second-Order Polymorphic Lambda-Calculus. In D.B. MacQueen G. Kahn and G.D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.
- [8] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, March 1987.
- [9] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *Principles of Programming Languages*, pages 151–156, 1990.
- [10] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1991.



- [11] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.
- [12] F. Cardone and M. Coppo. Two Extensions of Curry's Type Inference System. Technical Report RT90-6, Universita di Torino, August 1990.
- [13] F. Cardone and M. Coppo. Type Inference with Recursive Types: Syntax and Semantics. Technical Report RT90-5, Universita di Torino, August 1990.
- [14] M. Coppo. An Extended Polymorphic Type System for Applicative Languages. In *Mathematical Foundations of Computer Science*, number 88 in Lecture Notes in Computer Science. Springer-Verlag, September 1980.
- [15] M. Coppo. Type Inference, Abstract Interpretation and Strictness Analysis. Draft manuscript, 1992.
- [16] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal Type Schemes and  $\lambda$ -calculus Semantics. In 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.
- [17] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
- [18] H.B. Curry and R. Feys. *Combinatory Logic, Volume 1*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1958.
- [19] K. Davis and P. Wadler. Backwards Strictness Analysis: Proved and Improved. In *Functional Programming*. Springer-Verlag and the British Computer Society, 1989.
- [20] P. Dybjer. Inverse Image Analysis. In *ICALP'14*, volume 267 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.
- [21] J. Fairbairn. Making Form follow Function: an exercise in functional programming style. Technical Report 89, University of Cambridge, June 1986.
- [22] A.B. Ferguson and R.J.M. Hughes. An Iterative Powerdomain Construction. In *Functional Programming*. Springer-Verlag and the British Computer Society, 1989.

- [23] Y-C. Fuh and P. Mishra. Type Inference with Subtypes. In *ESOP'88*, pages 94–114, Nancy, France, March 1988. LNCS 300.
- [24] Y-C. Fuh and P. Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. In *TAPSOFT'89*, pages 167–183, Barcelona, Spain, March 1989. LNCS 352.
- [25] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, University of Paris VII, 1972.
- [26] B. Goldberg. Detecting Sharing of Partial Applications in Functional Programs. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.
- [27] C.V. Hall and D.S. Wise. Compiling Strictness into Streams. In *Principles of Programming Languages*. ACM Press, 1987.
- [28] C. Hankin and S. Hunt. Approximate Fixed Points in Abstract Interpretation. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
- [29] J.R. Hindley. The Completeness Theorem for Typing  $\lambda$ -terms. *Theoretical Computer Science*, 22:1–17, 1983.
- [30] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.
- [31] J. Hsiang. Refutational Theorem Proving Using Term-Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
- [32] P. Hudak and R. Young. Higher-order strictness analysis in untyped lambda calculus. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–109. ACM, 1986.
- [33] R.J.M. Hughes. Strictness detection in Non-flat Domains. In *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 1985.
- [34] R.J.M. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
- [35] R.J.M. Hughes and J. Launchbury. Reversing Abstract Interpretations. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

- [36] S. Hunt. Frontiers and Open Sets in Abstract Interpretation. In *Functional Programming Languages and Computer Architecture*. ACM Press, 1989.
- [37] G. Hutton. Parsing Using Combinators. In *Proceedings of Glasgow Workshop on Functional Programming*, Glasgow, 1989. Springer-Verlag.
- [38] T.P. Jensen. *Strictness Analysis in Logical Form*, 1991.
- [39] T.P. Jensen and T.A. Mogensen. A Backwards Analysis for Compile-time Garbage Collection. In N.D. Jones, editor, *European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [40] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, State University of Utrecht, 1980.
- [41] T-M. Kuo and P. Mishra. Strictness Analysis: A New Perspective based on Type Inference. In *FPCA '89*, pages 260–272, London, United Kingdom, September 1989.
- [42] A. Leung and P. Mishra. Reasoning about Simple and Exhaustive Demand in Higher-Order Lazy Languages. Technical Report 91-01, State University of New York (Stony Brook), 1991.
- [43] L. Löwenheim. Über das Auflösungsproblem im logischen Klassenkalkül. *Sitzungsber. Berl. Math. Gesell.*, 7:89–94, 1908.
- [44] D.B. MacQueen, G.D. Plotkin, and R. Sethi. An Ideal Model for Polymorphic Types. In *ACM Symposium on Principles of Programming Languages*, volume 11, pages 165–174, January 1984.
- [45] U. Martin and T. Nipkow. Boolean Ring Unification—the story so far. *Journal of Symbolic Computation*, 7:275–293, 1989.
- [46] D. Maurer. Strictness Computation using Special  $\lambda$ -expressions. In *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 1985.
- [47] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [48] J.C. Mitchell. Coercion and Type Inference (Summary). In *Principles of Programming Languages*, pages 175–185, 1984.
- [49] J.C. Mitchell. Type Inference and Type Containment. In D.B. MacQueen G. Kahn and G.D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

- [50] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag, 1980.
- [51] A. Nielsen and M. Jager. *Frontline Orwell—User’s Guide*. University of Western Australia, October 1991.
- [52] S. Peyton-Jones P. Hudak and P. Wadler. *Report on the Programming Language Haskell*, August 1991.
- [53] A.S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*. PhD thesis, University of Tasmania, 1991.
- [54] S. Peyton-Jones and D. Lester. *Implementing Functional Languages: a tutorial*. Prentice-Hall, 1992.
- [55] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [56] G.D. Plotkin. A Semantics for Type Checking. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 1–17, Sendai, Japan, 1991. Springer-Verlag.
- [57] J.C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.
- [58] S. Ronchi Della Rocca. Principal Type Schemes and Unification for Intersection Type Discipline. *Theoretical Computer Science*, 59:181–209, 1988.
- [59] S. Ronchi Della Rocca and B. Venneri. Principal Type Schemes for an Extended Type Theory. *Theoretical Computer Science*, 28:151–169, 1984.
- [60] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, 1991.
- [61] J.H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7:207–274, 1989.
- [62] M. Stone. The Theory of Representations for Boolean Algebra. *Transactions of the American Mathematical Society*, 40:37–111, 1936.
- [63] M.E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

- [64] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In G. Goos and J. Hartmanis, editors, *Functional Programming and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.
- [65] D.A. Turner. The Semantic Elegance of Applicative Languages. In *Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, 1981.
- [66] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C.L. Hankin, editors, *Abstract interpretation of declarative languages*. Ellis Horwood Limited, 1987.
- [67] P. Wadler. Is there a use for Linear Logic? In *Partial Evaluation and Program Manipulation*. ACM Press, 1991.
- [68] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, 1987.
- [69] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.
- [70] M. Wand and P. O’Keefe. On the complexity of type inference with coercion. In *FPCA ’89*, pages 293–297, London, United Kingdom, September 1989.
- [71] Stuart C. Wray. *Implementation and Programming Techniques for Functional Languages*. PhD thesis, University of Cambridge, 1986.
- [72] D.A. Wright. Strictness Analysis Via (Type) Inference. Technical Report TR89-3, University of Tasmania, September 1989.
- [73] D.A. Wright. An Intensional Type Discipline. Technical Report R91-3, University of Tasmania, July 1991.
- [74] D.A. Wright. A New Technique for Strictness Analysis. In *Theory and Practice of Software Development*, number 494 in *Lecture Notes in Computer Science*, Brighton, United Kingdom, April 1991. Springer-Verlag.
- [75] D.A. Wright. Abstract reduction types: a framework for static analysis. Draft manuscript in preparation, 1992.
- [76] D.A. Wright. An Intensional Type Discipline. In *Australian Computer Science Communications*, volume 14, January 1992.