University College London

Department of Computer Science

The Analysis of Resource Use in the λ-Calculus by Type Inference

Simon A. Courtenage

A thesis submitted for the degree of

Doctor of Philosophy

in the University of London

September, 1995

Abstract

This thesis is concerned primarily with the definition and semantics of resource use in the λ-calculus and the implicational fragment of intuitionistic propositional logic. A secondary aim is the subsequent derivation of a type system which can infer the expected reduction behaviour of functions upon their arguments.

The term *resource use* refers to the view of arguments as the resources a function requires to produce a result. In this thesis, resource use will be taken to mean a property of the formal parameter of a function that describes the use that the function will make of arguments substituted for the parameter. Knowledge of the resource use of a function parameter can lead to many practical benefits in the efficient compilation of functional programs.

Recent research has investigated the derivation of resource use information using type inference. Types inferred for functions contain resource use information about the way that arguments will be evaluated when applied to those functions. However, the justification of the correctness of these type systems relies on the given interpretation of type expressions as sets of terms possessing those types.

The main contribution of this thesis is the definition of resource use in both the λ-calculus and in the implicational fragment of intuitionistic propositional logic that corresponds to the typed λ-calculus under the Curry-Howard isomorphism. We are able to demonstrate the correspondence of resource use between (typed) λ-terms and the proofs in intuitionsitic logic, though we find that this correspondence is not up to equivalence.

Subsequently, we derive a type system for inferring resource use in λ-terms and also discuss the implementation of the type system. We find that we are unable to apply previously-used methods of unification over types containing resource use information when the range of resource use information is expanded.

Acknowledgements

I would like to thank Chris Clack for his supervision during the course of this work. I am also very grateful for the support and encouragement he gave during the difficult periods I encountered.

I would also like to thank colleagues and friends who have helped me in one way or another during the time I have studied for this PhD. Among them I would like to mention Clem Baker-Finch of the University of Canberra, and David Wright of the University of Tasmania, for their helpful comments and willingness to discuss their work; Gavin Bierman at Cambridge for his comments and advice on logic and type systems; Dave Parrott, Stuart Clayman, and Dave Lewis from University College for their many practical contributions; Chris Hankin, Simon Hughes, and Ian Mackie from Imperial College for helpful discussions; Mike Luck of Warwick University for reading an early draft of this thesis; and Alex Poulovassilis from King's College London. My thanks also to friends at the Department of Computer Science at University College London for their pleasant company and hospitality.

My thanks also go to my family for their encouragement during all these years of study.

Most of all, I would like to thank Maria for her love and support. This thesis is dedicated to her.

This work was supported by a research studentship from the Science and Engineering Research Council.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Despite the many advances made in the implementation of functional programming languages in recent years, static program analysis continues to be an important tool in compiler development as researchers and developers seek to match the conceptual advantages of functional programming languages with adequate run-time performance.

Functional programming languages have been designed to enable programmers to write clear, concise programs at a level of abstraction that does not involve them in architectural issues such as memory allocation, leaving these to be dealt with automatically by the compiler and the run-time system. To manage memory allocation automatically requires a garbage collection process to be run alongside the program, checking at intervals for redundant memory cells that can be collected and returned to the free pool of memory. This process, however, imposes its own cost on run-time performance, by consuming CPU cycles, and is not as efficient as the programmer, through the use of program statements, in managing the program's memory.

The goal of static analysis therefore is to provide information about expected program behaviour which can be used to guide the compiler in optimising the translation of program code to minimise a program's reliance on run-time support. Traditionally, the framework for developing static analyses has been Abstract Interpretation [2]. Recently, however, much interest has been shown in the potential of type inference as a means of performing static analysis as well as ensuring program correctness.

One such proposal for a system of type inference, which is able to deduce computational information about a program, has been made by Wright [93] and Baker-Finch [3] [4]. In these type systems, information is inferred that indicates the manner in which arguments to functions will be evaluated. In this thesis, we refer to such type systems as being *resource-aware*, and to the use that a function makes of its parameters, based on the expected evaluation of arguments substituted for those parameters, as *resource use*.

For example, in Wright's type system, two possible functional types are $\sigma \not\rightarrow \tau$ or $\sigma \rightarrow \tau$. The first type indicates that any argument to a function of this type will *not* be evaluated during evaluation of the application, while the second indicates that it *will* be evaluated. Baker-Finch has described a similar system of type inference, derived from a system of logic in which hypotheses are *tagged* to indicate whether they are necessary or not to a deduction of a proposition.

Both Wright and Baker-Finch have supplied interpretations of types containing information about reduction behaviour using the concept of *needed redexes* from Barendregt *et al* [7], and, on the basis of these interpretations, have proved the soundness and completeness of their type inference systems with respect to the type interpretation. One criticism of their work, however, is that they have not shown the existence of a relationship between the reduction behaviour of λ-terms and the information inferred about reduction behaviour by a resource-aware type system. Any relationship that may be presumed to exist does so only on the basis of the supplied interpretation of types.

In this thesis, we investigate the relationship between resource use in the λ-calculus and in intuitionistic logic. Our aim is to demonstrate the existence of this relationship, and subsequently, to use it as the basis for deriving a resource-aware type system in which it can be shown that the resource use inferred by the type system for a λ-term corresponds to the reduction behaviour of the term in question.

1.1 Goals and contributions of the thesis

Our primary goal in this thesis is to provide a semantics of resource use in the λ-calculus and in intuitionistic logic, and to demonstrate the relationship between these two semantics.

Subsequently, as a secondary goal, we intend to define a resource-aware type inference system on the basis of our understanding of this relationship, and to give an algorithm that implements the inference system.

The main contributions of this thesis can be stated as follows:

- In Chapter 3, we define resource use in λ-terms, following Wright and Baker-Finch in describing reduction behaviour in terms of needed and head-needed redexes (as defined by Barendregt *et al* in [7]), and give a semantics in the form of a system of inference rules. We define three resource use domains: the first is for strictness, absence, and non-strictness; the second extends the first with linearity; the third is Bierman's lattice of type annotations. In order to define a linear resource use, we extend the definitions of neededness and head-neededness to measure the degree to which a redex may be needed or head-needed.
- In Chapter 4, we give analogous definitions of needed and head-needed redexes in terms of proofs in intuitionistic logic, and subsequently define a system of intuitionistic logic which infers the resource use made of hypotheses in a proof. A variation on this system of intuitionistic logic is defined, in which only first-order resource use information is inferred. We show that neededness and head-neededness in intuitionistic logic are equivalent to the same concepts in the typed λ-calculus. For resource use, however, we are only able to show that a relationship holds, but not up to equivalence.
- We show in Chapter 5, that the type system derived from the system of intuitionistic logic with resource use, given in Chapter 4, does not have the Subject Reduction property, i.e., that the type of a term is not preserved over the reduction of the term. However, it is the case that the types of a term and its reduct are related by means

of an ordering on types induced by an ordering over the resource use domains.

- In Chapter 6, we show that the resource use domains and the operators defined over them do not form a boolean algebra, and therefore, that the method of boolean unification used by Wright in [93] is not suitable, except for very simple domains. We give a type inference algorithm for the type inference system in Chapter 5.

1.2 Static analyses of functional programming languages

In this section, we briefly review static analysis for functional programs in order to provide a context for this thesis.

As Sestoft mentions in [83], the expressive power and flexibility of functional languages require that they be implemented in a very general way. However, for any particular function, a more restrictive and consequently more efficient implementation may be possible. The aim of static analysis is to find out instances of a functional program where a more restrictive implementation is possible by analysis of program properties.

The properties a program useful to producing more efficient implementations include, for example, strictness, storage use, and sharing. Strictness, in functional programming languages, concerns the termination or non-termination of a program. Strictness is defined as follows: a function f is strict if $f\bot = \bot$, where $\bot$ is the undefined value. In lazy functional programming languages, knowledge that a function is strict in its argument can be used to evaluate function arguments in advance of, or in parallel with, evaluation of the function application, consequently reducing the amount of space needed to construct closures for unevaluated arguments and, on parallel machines, recovering implicit parallelism lost to lazy evaluation. Store use properties, such as single-threading of data structures, or discarding of heap cells in the local context of the need to allocate heap cells, can also result in useful optimisations, such as destructive updating of heap cells, without compromising referential transparency. The benefits of these optimisations are in minimising the costs of heap administration and avoiding garbage collection. Sharing is related to storage use, and describes the number of program references pointing to a sub-expression of the program. Implementations such as the Spineless, Tagless G-Machine [77] can make beneficial use of knowledge about sharing to avoid updating a graph node after evaluation of a term when it is known that the term is not shared.

Traditionally, the basis for static analysis of functional languages has been *abstract interpretation* [2] [53] [16], which provides a formal framework in which program analyses can be developed and proved correct. The underlying idea of abstract interpretation is to evaluate a program in a non-standard semantics, in which values denote properties, such as non-termination, abstracted from the standard semantics. Using abstract interpretation, both *forward* analyses, those that propagate values from the leaves of a program's syntax tree to the root, i.e., to derive a value for the program as a whole, and backwards analyses, those that propagate values from the root to the leaves of the syntax tree (see, for example, Hughes's abstract interpretation of continuations in [48]), can be formulated. The correctness of the analysis is given by showing that the property denoted by the value of a program in the non-standard semantics is satisfied by the standard semantic interpretation of the program. The computability of the analysis follows from defining properties in the non-standard semantics as approximations to values in the standard semantics, and constructing finite lattices of properties.

Abstract interpretation was originally developed for imperative languages using operational semantics by the Cousots [23] and applied to first-order functional languages using denotational semantics with domains rather than sets of values by Mycroft in [74]. Burn, Hankin, and Abramsky extended Mycroft's work to higher-order functional languages in [14]. More recent work by Hunt has shown that abstract interpretation can be expressed using partial equivalence relations (PERs) in order to capture certain properties, such as *head-strictness* of lists, which escape Mycroft/Burn, Hankin, and Abramksy style abstract interpretation based on domains. Prior to Hunt's work, *projections* were discovered as a means of formulating backwards static program analyses (for example, Hughes and Wadler [90]). Backwards analyses using projections as abstract values are also able to define properties elusive to abstract interpretation based on the Mycroft/Burn, Hankin, and Abramsky approach. Launchbury [59] showed that projections can be associated with equivalence relations. Using this idea, Hunt's work on abstract interpretation, demonstrated that the properties expressible by projections can captured using PERs. Work has also been carried out on the relationship between abstract interpretation-based forwards analyses and projection-based backwards analyses, in order to understand their comparative power. See, for example, Hughes and Launchbury [51], Burn [15], and Neuberger and Mishra [75].

The best-known example of the use of abstract interpretation in static analysis of functional languages is *strictness analysis*, first proposed by Mycroft in [74], and further explored by, among others, Clack and Peyton-Jones [19], Burn, Hankin and Abramsky [14], Wray [92], and Wadler [88]. In strictness analysis the aim is to discover if a function is strict in its argument. For example, first-order forwards strictness analysis involves abstracting the value domain D of the standard interpretation to a two-point domain $\mathbf{2} = \{0, 1\}$ where 0 corresponds to $\{\perp\}$, where $\perp$ is the undefined value, and and 1 corresponds to D. If the abstract interpretation of a function produces 0, then the function is strict. For backwards strictness analysis using projections, see Hughes [49] [50], Hughes and Wadler [90], Davis and Wadler [29], and Davis [28].

Program analyses that capture properties about store-use are also possible using abstract interpretation: for example, Hudak's reference count analysis [47], Bloss's update analysis [11], the store-use analyses of Hughes [52], and Goldberg's sharing analysis [38]. Typically, in these analyses, the standard and non-standard semantics are extended in some way to capture operational information, for example, the abstract stores in Hudak's analysis, Bloss's update paths semantics, or Hughes' store semantics.

One of the disadvantages of abstract interpretation as a framework for static analysis is its complexity. Much work is needed, for example, to perform fix-pointing over domains. Even if these domains are finite, the cost can be prohibitive even for relatively small domains. Hunt has shown in [53] that finding fix-points, especially in the higher-order case, by exploring the lattice using an efficient method based on *frontiers* [18] can be intractable. Furthermore, Hunt suggests that higher-order functions often produce the worst-case scenario for this method. To solve this problem, he shows how lattices can be reduced in size to improve the search for fix-points using the frontiers method without having to change the abstract interpretation.

An alternative framework for static analysis, which has been recently proposed, is *non-standard type inference.* The perceived advantage of type inference over abstract interpretation is that it is able to make use of fast implementations of Hindley-Milner style type inference algorithms.

In non-standard type inference, types model the properties under analysis and are inferred using a program logic based on that of type inference. The work by Kuo and Mishra in [58] seems to have been the first application of type inference to static analysis, and was followed by Jensen [54] [55] and Benton [9]. Burn describes the general framework of using non-standard type inference as a program logic in [12] [13]. In this approach, however, it has been found that to achieve the deductive power of abstract interpretation, it is necessary to include conjunctive types, also known as *intersection* types, into the type discipline. The problem of the undecidability of type inference for intersection types is avoided by analysing only those programs that are known to be type-correct. However, the introduction of conjunctive types, as Hankin and Le Métayer point out, removes the link with efficient implementations. Their work, in [40], [42] and [41], has concentrated on how efficient algorithms can be recovered using lazy evaluation of type expressions.

A different approach to using type inference as the basis for static analysis is described by Wright in [93] and Baker-Finch in [3] [4]. In Wright's and Baker-Finch's work, static analysis is incorporated into the conventional type inference phase of program compilation, rather than performed as a separate pass after type inference (Kuo and Mishra note in [58] that they assume programs to be type-correct). The work of Wright and Baker forms the foundation of the work presented in this thesis, and is described in more detail in Section 2.6 of Chapter 2.

1.3 Overview of thesis

The plan of this thesis is as follows: in Chapter 2, we review background material on the λ-calculus, type inference, and intuitionistic logic necessary to an understanding of the thesis; in Chapter 3, we define resource use in the λ-calculus and provide its semantics; in Chapter 4, we define resource use in the implicational fragment of intuitionistic logic, and demonstrate the relationship between resource use in intuitionistic logic and resource use in typed λ-terms; in Chapter 5, we derive a type system for λ-terms, and discuss the representation of principal types and let-polymorphism; we discuss the use of boolean ring unification for unification over resource use expressions in Chapter 6, and give an implementation of the type system of Chapter 5; finally, in Chapter 7, we summarise the contributions of the thesis, discuss directions for future work, and conclude.

Chapter 2

λ-Calculus, types and intuitionistic logic

In this chapter, we review the basic concepts of λ-calculus, type inference and intuitionistic logic necessary to an understanding of this thesis. We also discuss an equivalence between the typed λ-calculus and intuitionistic logic known as the Curry-Howard isomorphism. An overview of the resource-aware type systems of Wright and Baker-Finch is also provided.

2.1 The λ-calculus

In this section, we provide a brief introduction to the λ-calculus. (For further reading see Barendregt's comprehensive text on the λ-calculus [6], Hindley and Seldin's book [43], or the more recent book by Hankin [39].)

A λ-term is a word formed from the alphabet consisting of variables $x \in \mathit{Var}$, where Var is a countably infinite set of variable symbols, the symbol '.', brackets ( ) and the lambda sign λ. The set of λ-terms Λ is defined inductively as the least set satisfying

$$\begin{aligned} x \in \mathit{Var} &\Rightarrow x \in \Lambda \\ M \in \Lambda, x \in \mathit{Var} &\Rightarrow (\lambda x.M) \in \Lambda \\ M, N \in \Lambda &\Rightarrow (MN) \in \Lambda \end{aligned}$$

This is also known as the λK-calculus. (Church's original definition, now known as the λI-calculus, contained the additional stipulation that, in forming an abstraction $\lambda x.M$, the variable x had to occur within M.)

Functions, or operators, are represented by λ-abstractions, i.e., by terms of the form $\lambda x.M$. In such a term, the variable x is said to be *bound* and M is the scope of the binding. An occurrence of a variable within a term that is not bound is said to be *free*. The set of free variables in a term M is denoted by $FV(M)$.

Applications of functions to arguments by the juxtaposition of two terms, e.g., $(\lambda x.M)N$. By convention, function application associates to the left, so that, for example, the term MNP is equivalent to $(MN)P$. The value of a function, represented by $\lambda x.M$, at N is calculated by substituting N for free occurrences of x throughout the body of the abstraction, represented by $M[N/x]$. Substitution is defined by

$$\begin{array}{lll}
(i) & x[N/x] & = N \\
(ii) & y[N/x] & = y, y \neq x \\
(iii) & (\lambda x.M)[N/x] & = \lambda x.M \\
(iv) & (\lambda y.M)[N/x] & = \lambda y.(M[N/x]),\ if\ y \neq x,\ and\ y \notin FV(N)\ or\ x \notin FV(M) \\
(v) & (\lambda y.M)[N/x] & = \lambda z.(M[z/y][N/x]),\ if\ y \neq x\ and\ y \in FV(N)\ and\ x \in FV(M) \\
 & & \quad and\ z \notin FV(M) \cup FV(N) \\
(vi) & (MM')[N/x] & = (M[N/x])(M'[N/x])
\end{array}$$

In the process of substitution it may be necessary to rename bound variables to prevent free occurrences of a variable in a term N to be substituted in a term $\lambda x.M$ from being unintentionally bound. For example, contracting the following term

$$(\lambda x.(\lambda y.yx))y$$

will cause the argument y to be bound by the inner λ-abstraction $\lambda y.yx$ when this was not intended, producing a function applies a value to itself, rather than a function which applies a value to a constant. To prevent this situation, bound variables are renamed during substitution of an argument where necessary. In the above example, substitution of the argument y with renaming of bound variables will result in

$$(\lambda z.((yx)[z/y]))[y/x]$$

i.e., y is renamed to be a new variable z throughout the body of the abstraction before y is substituted for x. This act of renaming bound variables is called *α-conversion* or *congruence.* Congruent terms, i.e., those terms that are syntactically equivalent up to renaming of bound variables, have the same interpretation. In discussing λ-terms we will use $\equiv$ to denote syntactic equivalence. For example, $\lambda x.x \equiv \lambda y.y$. In the rest of this thesis, for those λ-terms that occur in definitions, proofs, examples etc., we will adopt Barendregt's variable convention (see Chapter 1, §2.1, of [6]), i.e., that bound variables are chosen to be different from the free variables.

In the λ-calculus, a *combinator* is a *closed* term. In other words, it is a term with no free variables. Some common combinators have been given special names; these are I, K, S, and Y, defined as follows.

$$\begin{aligned} I &\equiv \lambda x.x \\ K &\equiv \lambda x.\lambda y.x \\ S &\equiv \lambda x.\lambda y.\lambda z.xz(yz) \\ Y &\equiv \lambda x.(\lambda y.x(yy))(\lambda y.x(yy)) \end{aligned}$$

A λ-term of the form $(\lambda x.M)N$ is called a β-redex and the corresponding term $M[N/x]$ is its *contractum.* If a term P contains a redex R, denoted by $R \in P$, then P *β-contracts* to P', i.e., $P \rightarrow_\beta P'$, where P' is P with the redex R replaced by its contractum. (We will sometimes write $P \xrightarrow{R}_\beta P'$ to denote the fact that the redex R is contracted.) A term P *β-reduces* to P', i.e.,$P \twoheadrightarrow_\beta P'$ iff P' is reached by zero or more contractions from P. The reduction relation $\twoheadrightarrow_\beta$ is the reflexive, transitive closure of $\rightarrow_\beta$. A *reduction sequence* is a series of contractions. The notation $\mathcal{R} : M \twoheadrightarrow_\beta M'$ states that $\mathcal{R}$ is a reduction sequence reducing M to M'.

The reduction relation $\twoheadrightarrow_\beta$ induces a notion of conversion of λ-terms, expressed by the equivalence relation $=_\beta$.

Definition 2.1 *Given λ-terms M and N, then M β-converts to N (and vice versa), written*

$$M =_\beta N$$

iff N is obtained from M by a finite (possibly empty) series of β-contractions, reversed β-contractions, and α-conversions.

A term is in *normal form* (nf) if it contains no redexes. A term N is a normal form of a term M if N contains no redexes and $M \twoheadrightarrow_\beta N$. A λ-term that does not reduce to a normal form is one for which all reduction paths are infinitely long, i.e., do not terminate. For those terms with normal forms, the Church-Rosser Theorem for β-reduction implies that the normal form of a λ-term is unique.

Theorem 2.1 (Church-Rosser theorem) *Given a λ-term M, if $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta N'$, then there exists a R such that $N \twoheadrightarrow_\beta R$ and $N' \twoheadrightarrow_\beta R$.*

A redex is *maximal* (or *outermost*) if it is not contained in any other redex, and is the *leftmost* maximal redex if it is the leftmost of the maximal redexes of a term. In the term

$$\lambda x.((\lambda y.(Ky))(Ix)((SKK)x))$$

the redex $(\lambda y.(Ky))(Ix)$ is leftmost maximal.

If a term M is of the form

$$\lambda x_1 \ldots \lambda x_n.(\lambda y.N)N_1 \ldots N_m$$

for $n \geq 0, m \geq 0$, then $(\lambda y.N)N_1$ is the *head redex* of M. The head redex is also the leftmost redex (but not conversely). For example, given the terms

$$\begin{aligned}&\lambda x.((Ix)(SKx))\\&\lambda x.(x(Kyx))\end{aligned}$$

the redex *(Ix)* in the first term is a head redex, while, in the second term, although *(Kyx)* is the leftmost redex, it is not a head redex. A term is in head normal form (hnf) iff it does not contain a head redex. The following are examples of terms in head normal form:

$$\begin{aligned}&x\\&x(Iy)\\&\lambda x.(x(K(Iy)x))\end{aligned}$$

A reduction sequence in which only leftmost maximal redexes are contracted is a *normal* or *leftmost* reduction sequence, denoted as $\mathcal{L}$. As a notational convention, we will use $\Downarrow_{\mathcal{L}} M$ to represent the reduction of a term M by leftmost reduction. *Head reduction* describes a reduction sequence in which only head redexes are contracted, denoted by $\mathcal{H}$. The notation $\Downarrow_{\mathcal{H}} M$ denotes the reduction of a term M by head reduction. Head reduction is closely linked with the solvability of a λ-term. If M' is the closure of an arbitrary λ-term M (i.e., M' is $\lambda x_1 \ldots x_n.M$ where $\{x_1, \ldots, x_n\} = FV(M)$), then M is solvable if

$$\exists n, \exists N_1 \ldots N_n \in \Lambda . M' N_1 \ldots N_n =_\beta I$$

Wadsworth has shown (Theorem 8.3.14 in Barendregt [6]) that, in the λK-calculus, if a term is solvable then it has a head-normal form. Finite head reduction sequences are therefore associated with solvable terms, while infinite head reduction sequences, i.e., those that do not produce head normal forms, are associated with unsolvable terms.

2.1.1 Labelled λ-calculi

In Klop [56], a variant of the λ-calculus is described in which labels are attached to λ-terms in order to be able to *trace* terms across reduction steps. The definition of the set of labelled λ-terms Λ_L follows the definition of the set of untyped λ-terms Λ.

Definition 2.2 *Let L be a set of labels. Then Λ_L is defined inductively as the least set satisfying*

$$\begin{aligned} x \in Var, l \in L &\Rightarrow x^l \in \Lambda_L \\ x \in Var, M \in \Lambda_L, l \in L &\Rightarrow (\lambda x.M)^l \in \Lambda_L \\ M, N \in \Lambda_L, l \in L &\Rightarrow (MN)^l \in \Lambda_L \end{aligned}$$

Example 2.1 *The following is a term in Λ_L (where $L \equiv \mathbb{N}$).*

$$M \equiv (((\lambda x.(\lambda y.y^1 x^2)^3)^4 z^5)^6 a^7)^8$$

A term in Λ_L can be written as M^φ where M is the term obtained by erasing the labels, and φ is a mapping from the sub-terms of M to labels. Therefore, another way to view labelled λ-terms is as ordinary (i.e., unlabelled) λ-terms with an associated mapping φ. For our purposes, in Chapter 3, we take a mapping φ for a term M to be one whose domain is the set of all sub-terms of M and whose range is the *multi-set* of labels in M, such that distinct occurrences of a label a in M result in distinct occurrences of a in the range of φ.

Reduction in Klop's labelled λ-calculus is defined as in the unlabelled case, with provision for labels:

$$((\lambda x.M)^i N)^j \to_\beta M[N/x]$$

Note that the label of the application, and of the λ-abstraction, are erased in the contraction of the redex.

An earlier version of a labelled λ-calculus is described by Lévy in [61]. In Lévy's labelled λ-calculus, β-reduction is defined as

$$((\lambda x.M)^i N)^j \to_\beta j\bar{i}.M[\underline{i}.N/x]$$

where the scope of $j\bar{i}.$ is $M[\underline{i}.N/x]$ and

$$\begin{array}{lcl} i.x^j & = & x^{ij} \\ i.(\lambda x.M)^j & = & (\lambda x.M)^{ij} \\ i.(MN)^j & = & (MN)^{ij} \end{array}$$

Substitution of a term N for x in M, denoted $M[N/x]$, is as for the unlabelled λ-calculus, except for the case where M is a variable. Then substitution is defined by

$$\begin{array}{lcl} x^i[N/x] & = & i.N \\ y^i[N/x] & = & y^i \end{array}$$

The important aspect of Lévy's labelled λ-calculus from our point of view is that labels of redexes are not lost when they are contracted, as is the case in Klop's labelled λ-calculus. Instead they are concatenated to other labels as reduction proceeds. In fact, labels are only erased in the case of contraction where the bound variable does not appear among the free variables of the body of the abstraction, i.e.,

$$(\lambda x.M)N, \quad x \notin FV(M)$$

In this instance, the labels attached to N and its sub-terms are erased on contraction. Lévy has also shown that his calculus has the Church-Rosser property.

As with Klop's version of a labelled λ-calculus, labels in Lévy's calculus can be represented by a mapping φ from sub-terms to labels.

2.1.2 The typed λ-calculus

The traditional mathematical view of a function is as a set of ordered pairs with a domain and range, which are themselves sets of values. We can apply the operator-process concept of the λ-calculus to the study of the set-theoretic idea of functions from an algorithmic perspective by specifying domains and ranges of λ-terms using *types*. The resulting calculus is known as the typed λ-calculus (see Chapter 13 of Hindley and Seldin [43], or Appendix A of Barendregt [6]), and forms the basis for the study of many conventional programming languages, for example, in denotational semantics (see Stoy [85]).

The language of type expressions is built up from type constants, e.g., o, which represent fixed sets of values such as the set of integers or the set of boolean values, and the connective $\rightarrow$.

An intuitive interpretation of a type $\sigma \rightarrow \tau$, where σ and τ denote arbitrary type expressions, is that it represents a set of functions mapping from a domain represented by σ to a range which is a subset of the set represented by τ. Note that $\lambda x^{o}.x^{o\rightarrow o}$ is distinct from $\lambda x^{\sigma}.x^{\sigma\rightarrow\sigma}$ for any type $\sigma \neq$ o, even though they both perform the same operation of identity.

The language of typed λ-terms is similar to the untyped λ-calculus, with variables, abstraction and application as the three syntactic forms. In the typed λ-calculus, however, terms and bound variables of abstractions are annotated with types to denote domains and ranges. The language of typed λ-terms $\Lambda^{\rightarrow}$ is defined to be the least set satisfying

$$\begin{aligned} x^{\sigma} \in \mathit{TypedVar}_{\sigma} &\Rightarrow x^{\sigma} \in \Lambda^{\rightarrow} \\ M^{\tau}, x^{\sigma} \in \Lambda^{\rightarrow} &\Rightarrow (\lambda x^{\sigma}.M^{\tau})^{\sigma\rightarrow\tau} \in \Lambda^{\rightarrow} \\ M^{\sigma\rightarrow\tau}, N^{\sigma} \in \Lambda^{\rightarrow} &\Rightarrow (M^{\sigma\rightarrow\tau}N^{\sigma})^{\tau} \in \Lambda^{\rightarrow} \end{aligned}$$

(where σ and τ denote arbitrary types, and $\mathit{TypedVar}_{\sigma}$ is an infinite set of variables x^{σ} of type σ for each σ).

The typed λ-calculus has substitution, reduction, convertibility, normal-form etc. as defined for the untyped λ-calculus, with the minor difference of type annotations to terms. It has one important property, however, that the untyped λ-calculus does not: *Strong Normalisation.*

Theorem 2.2 *Strong Normalisation Theorem* (Theorem 13.12 of Hindley and Seldin [43]) *In the typed λ-calculus, there are no infinite β-reductions.*

In other words, as a consequence of Strong Normalisation, every term in the typed λ-calculus has a normal form. Hence, not all terms in the untyped λ-calculus have an analogous typed λ-term. The untyped version of the identity function $\lambda x.x$ has infinitely many typed analogues $\lambda x^{\sigma}.x^{\sigma \to \sigma}$ for every type σ. However, terms such as $\lambda x.xx$ have no typed equivalent (unless we allow intersection types [21]). The Y combinator used to implement recursion in the untyped λ-calculus also has no equivalent typed λ-term. Consequently, a fix-point constant *fix* must be introduced into the typed λ-calculus if recursion is needed.

2.2 Models of the λ-calculus

In this section, we review models of the λ-calculus, which provide the semantics of λ-terms.

The following definitions of λ-models are standard and more detailed presentations of the topic may be found in Hindley and Seldin [43], Barendregt [6], and Meyer [71].

Definition 2.3 *(Definition 11.3 of [43].) A* λ-model *is a triple*

$$\mathcal{D} = < D, \cdot, [\![\ ]\!] >$$

where $< D, \cdot >$ is an applicative structure in which D is a non-empty set and $\cdot$ is a binary operator of type $D \times D \to D$, and $[\![\ ]\!]$ is a mapping from λ-terms to D in the context of *an environment $\rho :: Var \rightarrow D$, such that*

(i) $[\![x]\!]\rho = \rho(x)$

(ii) $[\![MN]\!]\rho = [\![M]\!]\rho \cdot [\![N]\!]\rho$

(iii) $[\![\lambda x.M]\!]\rho \cdot a = [\![M]\!]\rho[x \mapsto a], \forall a \in \mathrm{D}$

(iv) $[\![M]\!]\rho = [\![M]\!]\sigma, \text{if } \rho(x) = \sigma(x), \forall x \in \mathrm{FV}(M)$

(v) $[\![\lambda x.M]\!]\rho = [\![\lambda y.(M[y/x])]\!]\rho, \text{ if } y \notin \mathrm{FV}(M)$

(vi) $\forall d \in D.[\![M]\!]\rho[\mathrm{x} \mapsto d] = [\![N]\!]\rho[x \mapsto d], \text{ then } [\![\lambda x.M]\!]\rho = [\![\lambda x.N]\!]\rho$

Note that clause (v) defines the semantic equivalence of congruent λ-terms. Clause (vi) defines the equivalence of two functions if they map the same arguments to the same results for all possible applications.

The simplest λ-model to construct is a *term model*, in which the set D contains convertibility classes of λ-terms.

Definition 2.4 *(Definition 11.16 of [43].) Define convertibility classes for λ-terms, such that*

$$[M] = \{N | M =_\beta N\}$$

Then a term-model *of the λ-calculus (either $\lambda\beta$ or $\lambda\beta\eta$) is a model $< D, \cdot, [\![\;\;]\!] >$ where*

$$\begin{array}{lll} \text{(i)} & D & = & \{[M] | M \in \Lambda\} \\ \text{(ii)} & [M]\cdot[N] & = & [MN] \\ \text{(iii)} & [\![\mathrm{M}]\!]\rho & = & [M[N_1/x_1, \ldots, N_n/x_n]] \\ & & & \text{where} \\ & & & \{x_1, \ldots, x_n\} = \mathrm{FV}(M) \\ & & & \rho(x_i) = [N_i] \end{array}$$

Note that, for (iii), we define $[M[N_1/x_1, \ldots, N_n/x_n]]$ to be the act of simultaneously substituting N_1 for x_1, $\ldots N_n$ for x_n in M.

The class of models to which the term model above belongs are models of *conversion*, i.e., two λ-terms M and N equate to the same semantic element if $M{=_\beta}N$. (In the case of the term model, this follows directly from the fact that the elements of the semantic domain are simply the convertibility classes of terms.) An alternative model of *reduction* has been proposed by Plotkin [78], in which the underlying term relation is the reduction relation $\twoheadrightarrow_\beta$. (Plotkin's reduction model has been used as the basis for the term and type semantics of the resource-aware type systems of Wright [93] and Baker-Finch [3].)

2.3 Intuitionistic logic

In this thesis, we are concerned with intuitionistic implicative propositional logic, where the only logical connective is *implication*, denoted by $\rightarrow$. Figure 2.1 presents implicative propositional intuitionistic logic in Natural Deduction style using asymmetrical sequents. In the rest of this thesis, we will refer to implicative propositional intuitionistic logic simply as intuitionistic logic or as IL. (For further reading on mathematical logic, see Mendelson [70], Gallier [34], or Van Dalen [87].)

A sequent in the system of intuitionistic logic presented in Figure 2.1 is of the form $\Gamma \vdash \sigma$ (where Γ is a multiset of propositions and σ is a single proposition) and is interpreted as meaning that, given the conjunction of the propositions in Γ, we can infer σ. Hence, the propositions in Γ act as hypotheses or assumptions for the inference of σ.

IL is divided into three types of rules: the Axiom rule, Structural Rules and Logical Rules. The Axiom rule is the start point for any proof. The Structural rules define the manipulations that can be performed over the structure of sequents. Since we are using sequents in which the right-hand side is restricted to a single proposition, the Structural rules are defined only over the multiset of propositions on the left-hand side of $\vdash$. (Some presentations of IL include a structural rule Exchange which permutes the propositions on the left-hand side of the sequent. However, since we consider multisets of propositions, rather than sequences, we have omitted this rule.) The Logical rules are those used for manipulating derived propositions by introducing or eliminating propositional connectives, in this case $\rightarrow$.

Proofs (or deductions) of propositions in intuitionistic logic can be envisaged as trees of deductions, in which nodes of the tree are premises or conclusions, and applications of

Axiom

$$\dfrac{}{\sigma \vdash \sigma}\ \text{Axiom}$$

Structural Rules

$$\dfrac{\Gamma, \sigma, \sigma \vdash \tau}{\Gamma, \sigma \vdash \tau}\ \text{Contraction} \qquad \dfrac{\Gamma \vdash \tau}{\Gamma, \sigma \vdash \tau}\ \text{Weakening}$$

Logical Rules

$$\dfrac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \rightarrow \tau}\ \rightarrow \text{Intro}$$

$$\dfrac{\Gamma \vdash \sigma \rightarrow \tau \qquad \Lambda \vdash \sigma}{\Gamma, \Lambda \vdash \tau}\ \rightarrow \text{Elim}$$

Figure 2.1: Intuitionistic implicative propositional logic

inference rules act as edges. For example, a proof of $\sigma \vdash \tau \rightarrow \sigma$ can be represented by the tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \vdash \sigma}\ \text{Axiom}}{\sigma, \tau \vdash \sigma}\ \text{Weakening}}{\sigma \vdash \tau \rightarrow \sigma}\ \rightarrow \text{Intro}}{\vdash \sigma \rightarrow \tau \rightarrow \sigma}\ \rightarrow \text{Intro} \qquad \dfrac{}{\sigma \vdash \sigma}\ \text{Axiom}}{\sigma \vdash \tau \rightarrow \sigma}\ \rightarrow \text{Elim}$$

As a notational convenience, we will sometimes abbreviate proof trees as follows:

$$\dfrac{\begin{matrix}[P] \\ \vdots\end{matrix}}{\Gamma \vdash \tau}\ \mathit{Rule}$$

or sometimes, just

$$\begin{array}{c}[P]\\ \vdots\\ \Gamma \vdash \tau\end{array}$$

where $[P]$ stands for the proof tree above the conclusion of $\Gamma \vdash \tau$ by rule *Rule*. We will use the letters P, Q, R to range over proof trees in this fashion. Also, we will sometimes write $P : \sigma$ to mean that a proposition σ is the conclusion of a proof tree P.

A hypothesis occurring on the left of a sequent is in fact a multi-set of occurrences of the same hypothesis, called a *parcel* by Girard [37] (or *assumption class* by Troelstra in [86]). A parcel of a hypothesis contains different occurrences of the hypothesis introduced by the Axiom rule at different positions in the proof tree. Parcels are merged to form larger parcels by Contraction. Empty or *dummy* parcels are introduced by Weakening.

Example 2.2 *In the following proof, the rule* $\to$ Intro *discharges a parcel containing two occurrences of the hypothesis* σ*.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \to \sigma \to \tau \vdash \sigma \to \sigma \to \tau}\,\text{Axiom} \quad \dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma \to \sigma \to \tau, \sigma \vdash \sigma \to \tau}\,\to \text{Elim} \quad \dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma \to \sigma \to \tau, \sigma, \sigma \vdash \tau}\,\to \text{Elim}}{\sigma \to \sigma \to \tau, \sigma \vdash \tau}\,\text{Contraction}}{\sigma \to \sigma \to \tau \vdash \sigma \to \tau}\,\to \text{Intro}$$

Proof trees can be simplified by eliminating redundant pairs of Introduction and Elimination rules (see [37] or [86]). In implicative propositional intuitionistic logic, there is one rule governing *proof normalisation* for pairs of $\to$ Intro and $\to$ Elim rules, described below.

$$\dfrac{\dfrac{\begin{array}{c}[Q]\\ \vdots\\ \hline \Gamma, \sigma \vdash \tau\end{array}}{\Gamma \vdash \sigma \to \tau}\,\to \text{Intro} \qquad \begin{array}{c}[P]\\ \vdots\\ \Lambda \vdash \sigma\end{array}}{\Gamma, \Lambda \vdash \tau}\,\to \text{Elim} \qquad \Rightarrow \qquad \begin{array}{c}[Q']\\ \vdots\\ \Gamma, \Lambda \vdash \tau\end{array}$$

where Q' is obtained from Q by substitution of

$$\begin{array}{c}[P]\\ \vdots\\ \Lambda \vdash \sigma\end{array}$$

for the hypothesis σ, in other words by replacing in Q the proof tree P for every occurrence of

$$\frac{}{\sigma \vdash \sigma}\text{ Axiom}$$

where the σ introduced is included in the hypothesis parcel for σ in the sequent concluding Q. As a notational shorthand for substitution, we will write

$$(Q : \tau)[P/\sigma]$$

to denote the substitution, as described above, of the proof $P : \sigma$ for occurrences of the Axiom rule introducing the undischarged hypothesis σ in $Q : \tau$.

A proof is said to be in normal form if there are no redundant proof steps to be eliminated, i.e., there are no contractions of the above form that can be applied to the proof ([86], Section 4.1.4). Strong normalisation is also true for intuitionistic logic, i.e., proofs possess unique normal forms. As a result, there are no infinite normalisation sequences of proofs.

2.4 Intuitionistic logic and the typed λ-calculus

There exists a correspondence between intuitionistic logic and the typed λ-calculus, which shows that, in an essential way, the two systems are equivalent. This correspondence is known as the Curry-Howard isomorphism (see Howard [46], Girard [37], or Troelstra [86]).

Under the Curry-Howard isomorphism, types and terms in the typed λ-calculus correspond to propositions and proofs in intuitionistic logic. In particular, terms are encodings of the proofs of the propositions corresponding to the types of the terms, i.e., variables, abstractions and applications of terms correspond to the use of the Axiom, $\rightarrow$ Intro and $\rightarrow$ Elim rules in proofs.

Example 2.3 *The typed λ-term $(\lambda x^{\sigma}.\lambda y^{\tau}.x)^{\sigma\rightarrow\tau\rightarrow\sigma}$ corresponds to the proof*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \vdash \sigma}\text{Axiom}}{\sigma, \tau \vdash \sigma}\text{Weakening}}{\sigma \vdash \tau \rightarrow \sigma}\rightarrow\text{Intro}}{\vdash \sigma \rightarrow \tau \rightarrow \sigma}\rightarrow\text{Intro}$$

Not only do the two systems share the same structure, but the processes of normalisation on each side are equivalent. The proof normalisation rule which eliminates redundant proof steps involving the connective $\rightarrow$ (see Section 2.3 above) corresponds to β-reduction of redexes in the typed λ-calculus.

The following translation Ψ of IL proofs into λ-terms is derived from the translation described by Troelstra [86] (Section 4.1.6) for a Natural Deduction presentation of IL.

1. To a proof consisting only of an Axiom rule, we assign a new typed variable x, i.e.,

$$\Psi\left(\dfrac{}{\sigma \vdash \sigma}\text{Axiom}\right) = x_{\sigma}$$

2. Contracted hypotheses result in renaming to a single variable:

$$\Psi\left(\dfrac{\overset{\vdots}{\Gamma, \sigma, \sigma \vdash \tau}}{\Gamma, \sigma \vdash \tau}\text{Contraction}\right) = \Psi\left(\overset{\vdots}{\Gamma, \sigma, \sigma \vdash \tau}\right)[z_{\sigma}/x_{\sigma}, z_{\sigma}/y_{\sigma}]$$

where x_{σ} and y_{σ} are the typed variables assigned to the two distinct hypotheses σ, σ in the premise to the Contraction rule.

3. The translation of a deduction ending in an application of Weakening is simply the translation of the premise:

$$\Psi\left(\cfrac{\overset{\vdots}{\Gamma \vdash \tau}}{\Gamma, \sigma \vdash \tau}\ \text{Weakening}\right) = \Psi\left(\overset{\vdots}{\Gamma \vdash \tau}\right)$$

4. An abstraction is translated inductively:

$$\Psi\left(\cfrac{\overset{\vdots}{\Gamma, \sigma \vdash \tau}}{\Gamma \vdash \sigma \to \tau} \to \text{Intro}\right) = \lambda x^{\sigma}.\Psi\left(\overset{\vdots}{\Gamma, \sigma \vdash \tau}\right)$$

where the variable x^{σ} is the variable associated with the hypothesis σ in the translation of the premise $\Gamma, \sigma \vdash \tau$ to the $\to$ Intro rule.

5. Applications are also translated inductively:

$$\Psi\left(\cfrac{\overset{\vdots}{\Gamma \vdash \sigma \to \tau} \quad \overset{\vdots}{\Delta \vdash \sigma}}{\Gamma, \Delta \vdash \tau} \to \text{Elim}\right) = \Psi\left(\overset{\vdots}{\Gamma \vdash \sigma \to \tau}\right)\ \Psi\left(\overset{\vdots}{\Delta \vdash \sigma}\right)$$

In the rest of this thesis, we will use $\equiv_{CHI}$ to stand for equivalence between objects in intuitionistic logic and the typed λ-calculus, for example, between a proof and a typed λ-term.

2.5 The Curry type system

In the typed λ-calculus (see Section 2.1.2 above), a consequence of the type annotations on λ-terms is that there are distinct copies of what is essentially the same term for different domains and ranges. For example, for each type σ and τ in the set of types, we have a different version of $(\lambda f^{\sigma \to \tau}.\lambda x^{\sigma}.fx)^{(\sigma \to \tau) \to \sigma \to \tau}$ each of which is a special case (for the specified domain and range) of the term in the untyped λ-calculus $\lambda f.\lambda x.fx$.

An alternative approach is to take the analogue in the untyped λ-calculus of these terms and to treat it as the general case of which the typed terms are specialisations. We can *reconstruct* any of the typed terms from the general untyped term using a system of type inference to deduce a type for the term. A term in the untyped λ-calculus may be assigned an infinite number of types, but we can treat these as being special cases of a small number of general type expressions called *type schemes*. Type schemes have variables in the type expressions for which type expressions can be substituted to create any of the valid types for the term.

Definition 2.5 *Assume a number of type constants and an infinite number of type variables. Then type schemes are defined as follows:*

- *all type constants and type variables are type schemes,*
- *if σ, τ are type schemes, so is $\sigma \to \tau$.*

Throughout this thesis, we will denote type variables by α, β, γ and arbitrary type schemes by σ, τ, ϕ, ψ.

The result of simultaneously substituting type schemes $\sigma_1, \ldots, \sigma_n$ for type variables $\alpha_1, \ldots, \alpha_n$ in a type scheme τ is denoted by

$$\tau[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$$

Mappings such as $[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ are called *substitutions* (in this thesis, the letters S and T will range over substitutions).

The rules for inferring the types of λ-terms were first developed by Curry in [24] [25]. Curry's system for inferring type schemes of λ-terms (also known as Curry's system of F-deducibility) is presented in Figure 2.2. In this system, rules are of the form

$$\frac{P_1 \ldots P_n}{C}\ \textit{Rule}$$

$$\frac{}{A, x:\sigma \vdash x:\sigma}\ \text{Axiom}$$

$$\frac{A, x:\sigma \vdash M:\tau}{A \vdash \lambda x.M : \sigma \to \tau}\ \to \text{Intro}$$

$$\frac{A_1 \vdash M : \sigma \to \tau \qquad A_2 \vdash N : \sigma}{A_1 \cup A_2 \vdash MN : \tau}\ \to \text{Elim}$$

Figure 2.2: Curry type inference system

where $P_1 \dots P_n$ are premises to the rule and C is the conclusion following from the premises by application of rule *Rule*. Premises and conclusions are statements or *type judgements* of the form

$$A \vdash M : \sigma$$

where A is a *base* or set of mappings from variables to types used as a type environment from which the right-hand side of the sequent, $M : \sigma$, can be inferred.

The type system is *syntax-directed*, i.e., the type of a term is derived from the types of its immediate sub-terms. The Axiom rule types occurrences of variables with their assumed type. The $\to$ Intro rule *introduces* types for functions, deriving them from the type assumed for the variable to be bound in the λ-abstraction and the type derived for the term forming the body of the abstraction. The $\to$ Elim rule *eliminates* function types to derive the result of the application of a function-typed term to an argument term whose type matches that expected by the function.

Example 2.4 *The type $\tau \to \sigma$ is derived for the λ-term $(\lambda x.\lambda y.x)z$ as follows:*

$$\frac{\dfrac{\dfrac{\dfrac{}{x:\sigma, y:\tau \vdash x:\sigma}\ \text{Axiom}}{x:\sigma \vdash \lambda y.x : \tau \to \sigma}\ \to \text{Intro}}{\vdash \lambda x.\lambda y.x : \sigma \to \tau \to \sigma}\ \to \text{Intro} \qquad \dfrac{}{z:\sigma \vdash z:\sigma}\ \text{Axiom}}{z:\sigma \vdash (\lambda x.\lambda y.x)z : \tau \to \sigma}\ \to \text{Elim}$$

As might be expected, if a type can be inferred for a λ-term, then it has a normal form (since it is the untyped analogue of a typed λ-term, which, by the Strong Normalisation theorem, will possess a normal form).

Curry's system has the advantage of being decidable, but the disadvantage of not being complete with respect to a semantics in which convertible terms denote the same semantic element. If M has type σ and $M =_\beta N$, then we would expect M and N to have the same type. To achieve this kind of type equivalence, we need to add an extra rule

$$\frac{\Gamma \vdash M : \sigma \qquad M =_\beta N}{\Gamma \vdash N : \sigma}\,\text{Eq}$$

However, adding the rule Eq to Curry's type system makes type inference an undecidable problem, since the set of typable terms is not closed under conversion.

In the following theorems of the soundness and completeness of the Curry type system, we assume the inclusion of the Eq rule and a term model based on convertibility classes of λ-terms.

2.5.1 Soundness and completeness of the Curry type system

The Curry type system has been shown to be *sound* and *complete* with respect to a variety of semantic interpretations of λ-terms by Hindley in [44] [45]. Soundness indicates that the type system infers types for terms that are valid with respect to semantic interpretations of both terms and types, i.e., that if a term M is associated with type σ in the type system, then the semantic meaning of M is a member of the set of values which is the semantic meaning of σ. Completeness means that any relationship between terms and types that is possible in the semantics can be inferred by the type system.

The *simple semantics* of the λ-calculus is given using the semantic interpretation function $[\![\ ]\!]_\lambda :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow D$ as follows:

$$\begin{array}{rcl}
[\![x]\!]_\lambda \eta & = & \eta(x) \\
[\![\lambda x.M]\!]_\lambda \eta & = & \lambda z.[\![M]\!]_\lambda \eta[x \mapsto z] \\
[\![MN]\!]_\lambda \eta & = & [\![M]\!]_\lambda \eta([\![N]\!]_\lambda \eta)
\end{array}$$

The semantics of type expressions is defined using an environment $\nu :: TypeVar \rightarrow 2^D$ which maps type variables to subsets of 2^D (the powerset of D). Not every subset of D is denoted by a type, however: as noted by Milner [72], only those subsets which are downwards-closed and directed complete form the semantics of type expressions [1] (when D is viewed as a domain). Hence, for any type variable α, $\nu(\alpha) \in 2^D$.

Using the type variable environment ν, we define a semantic interpretation $[\![\ ]\!]_t :: TypeExpr \rightarrow TypeEnv \rightarrow 2^D$ of type expressions as follows

$$\begin{array}{rcl}
[\![\alpha]\!]_t \nu & = & \nu(\alpha) \\
[\![\sigma \rightarrow \tau]\!]_t \nu & = & \{f \in D | \forall e.e \in [\![\sigma]\!]_t \nu \Rightarrow (fe) \in [\![\tau]\!]_t \nu\}
\end{array}$$

Note that the semantic meaning of a type $\tau_1 \rightarrow \tau_2$ is given by application, i.e., that $\tau_1 \rightarrow \tau_2$ is the set of all functions f such that, for all values e in $[\![\tau_1]\!]\nu$, the result of applying f to e is in $[\![\tau_2]\!]\nu$.

Satisfaction in the Curry type system relates the assignment to a term M of a type σ with the membership of $[\![M]\!]_\lambda \eta$ in $[\![\sigma]\!]_t \nu$ for a domain D and environments η, ν. We say that a statement $x : \tau$ is satisfied by D and environments η and ν iff

$$[\![x]\!]_\lambda \eta \in [\![\tau]\!]_t \nu$$

A base A is satisfied iff all its members are satisfied. We write $A \models M : \sigma \Leftrightarrow \forall D, \eta, \nu$ satisfying A, $M : \sigma$ is also satisfied, i.e., that $[\![M]\!]_\lambda \eta \in [\![\sigma]\!]_t \nu$.

The soundness of the Curry type system is expressed in Theorem 2.3.

Theorem 2.3 *If $A \vdash M : \sigma$, then $\forall D, \eta, \nu$ which satisfy the base A in the simple semantics $A \models M : \sigma$, i.e., $[\![M]\!]_\lambda \eta \in [\![\sigma]\!]_t \nu$.*

[1]Such subsets are called *ideals* by MacQueen, Plotkin and Sethi in [66].

Theorem 2.4 states that the Curry type system (with Eq typing rule) is complete with respect to the semantics of λ-terms and types. See [44] for the proof of this theorem.

Theorem 2.4 *If, $\forall D, \eta, \nu$ in the simple semantics, $A \models M : \sigma$, then $A \vdash M : \sigma$.*

2.6 Resource-aware type inference systems

In this section, we give a brief overview of the type systems of Wright and Baker-Finch, which infer information about the reduction behaviour of functions as part of their types. Throughout this thesis, we will refer to these type systems as *resource-aware* type systems, since it is our belief that there exists a connection between the system of intuitionistic logic underlying the type systems of Wright and Baker-Finch and those systems of logic called *resource-conscious*, such as *linear logic* [36] and *relevant logic* [31] (in fact, as mentioned below, Baker-Finch has developed his type system from a variant of relevant logic).

2.6.1 Wright's reduction types

In his thesis [93], Wright introduced a new type discipline for the λ-calculus which he termed *reduction types*. Reduction types capture information, in the form of types, about the expected evaluation of arguments when they are supplied to functions. As part of this type discipline, Wright introduced two function arrows $\twoheadrightarrow$ and $\Rightarrow$, interpreted as follows: a function of type $\sigma \twoheadrightarrow \tau$ is a function from arguments of type σ to results of type τ such that, during reduction of the application of the function to an argument, the arguments are not evaluated, while if a function has type $\sigma \Rightarrow \tau$, then the arguments are evaluated, i.e., the function is *strict*.

In Figure 2.3, we reproduce Wright's rules for deducing reduction types using Curry-style type inference. In this type system, a *variable neededness function* V is attached to the deduction symbol $\vdash$, which maps term variables to the function arrows to be used to create a function type when those variables are discharged to form λ-abstractions. For example, given the term x, the variable neededness function is $V[x := \Rightarrow]$. Consequently,

$$\frac{}{A, x : \sigma \vdash_{[x:=\Rightarrow]} x : \sigma}\,\text{Var}$$

$$\frac{A, x : \sigma \vdash_{V[x:=b]} M : \tau}{A \vdash_{V[x:=\nrightarrow]} \lambda x.M : \sigma b \tau}\,\text{Abs}$$

$$\frac{A \vdash_{V_1} M : \sigma b \tau \qquad A \vdash_{V_2} N : \sigma}{A \vdash_{V} MN : \tau}\,\text{App}$$

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

Figure 2.3: Wright's Curry-style rules for deducing reduction types

the type of $\lambda x.x$ is $\sigma \Rightarrow \sigma$, indicating that any argument to this function will be definitely evaluated in head reduction to head normal form.

A more complicated example is the term $\lambda f.\lambda g.\lambda x.f(gx)$, which is assigned type

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

The type of this term involves the use of variable arrows $\rightarrow_i$ and $\rightarrow_j$ and the boolean operator $\wedge$. Variable arrows are instantiated over the set of *ground* arrows, $\{\Rightarrow, \nrightarrow\}$. The set of variable and ground arrows together with the boolean operators $\wedge$ and $\vee$ are shown to be a boolean algebra. Consequently, as variable arrows are instantiated, the simplification rules of boolean algebra can be used to simplify the arrow expressions in the type.

Wright has also given an extension to this system for let-polymorphism, which involves quantification over variable arrows. However, neither the Curry-style nor let-polymorphic type systems are able to type all typable λ-terms, particularly in those cases where higher-order variables are passed arguments of different intensionality. A further extension to intersection types is made in order to type all typable λ-terms. (This point is further discussed in Section 5.1.1 of Chapter 5.)

2.6.2 Baker-Finch's strictness types

In [3] and [4], Baker-Finch describes a type system similar to Wright's, but developed using relevant implication from *relevance logic* [31]. It is well-known that relevance logic proofs correspond to terms in the λI-calculus, in which all terms are strict in their arguments, and hence relevant implication corresponds to the function type constructor of strict functions. Baker-Finch extends relevance logic by incorporating $\twoheadrightarrow$ for constant implication (i.e., an implication that does not depend on its argument) and $\supset$ for ordinary intuitionistic implication (which essentially denotes non-strictness), and by tagging hypotheses according to their method of introduction (e.g., by Axiom or by Weakening rules) so that the appropriate implication introduction rule can be applied when a hypothesis is discharged. A type system is derived from this system of logic by the usual approach of assigning terms to logical rules that represent their encoding under the Curry-Howard isomorphism. As in Baker-Finch's extension to relevance logic, tags are attached to hypotheses in order to determine which form of function arrow should be used when a variable is discharged. In Baker-Finch's system, the tag # denotes strictness in a hypothesis, the empty string ϵ denotes absence, and ? denotes non-strictness.

Baker-Finch's type system $R_\supset$ is reproduced in Figure 2.4. (In [3], several type systems are described, but since they represent a series of incremental extensions, and we wish to illustrate the main idea behind Baker-Finch's work, we choose to describe only $R_\supset$ here.) In the system $R_\supset$, the need to extend the type system to incorporate intersection types in order to type all Curry-typable λ-terms is avoided by the introduction of conventional intuitionistic implication and by adding a side-condition to the App rules that allows types with more specific reduction information than is expected to be applied as arguments to implications. This side-condition is given in terms of an ordering $\trianglelefteq$ over types, defined as follows:

$$\begin{aligned}
&\alpha \trianglelefteq \alpha \\
&\sigma \rightarrow \tau \trianglelefteq \sigma' \rightarrow \tau', \text{ if } \sigma' \trianglelefteq \sigma \text{ and } \tau \trianglelefteq \tau' \\
&\sigma \twoheadrightarrow \tau \trianglelefteq \sigma' \twoheadrightarrow \tau', \text{ if } \sigma' \trianglelefteq \sigma \text{ and } \tau \trianglelefteq \tau' \\
&\sigma \supset \tau \trianglelefteq \sigma' \supset \tau', \text{ if } \sigma' \trianglelefteq \sigma \text{ and } \tau \trianglelefteq \tau' \\
&\sigma \rightarrow \tau \trianglelefteq \sigma' \supset \tau', \text{ if } \sigma' \trianglelefteq \sigma \text{ and } \tau \trianglelefteq \tau' \\
&\sigma \twoheadrightarrow \tau \trianglelefteq \sigma' \supset \tau', \text{ if } \sigma' \trianglelefteq \sigma \text{ and } \tau \trianglelefteq \tau'
\end{aligned}$$

$$\frac{}{A_{irr}, x\#\sigma \vdash x : \sigma}\ \text{Var}$$

$$\frac{A, x\#\sigma \vdash M : \tau}{A \vdash \lambda x.M : \sigma \rightarrow \tau} \rightarrow \text{Abs} \qquad \frac{A, x : \sigma \vdash M : \tau}{A \vdash \lambda x.M : \sigma \twoheadrightarrow \tau} \twoheadrightarrow \text{Abs}$$

$$\frac{A, x?\sigma \vdash M : \tau}{A \vdash \lambda x.M : \sigma \supset \tau} \supset \text{Abs}$$

$$\frac{A \vdash M : \sigma \rightarrow \tau \quad A' \vdash N : \sigma'}{A \cup A' \vdash MN : \tau} \rightarrow \text{App}\ \sigma' \trianglelefteq \sigma$$

$$\frac{A \vdash M : \sigma \twoheadrightarrow \tau \quad A' \vdash N : \sigma'}{A \vdash MN : \tau} \twoheadrightarrow \text{App}\ \sigma' \trianglelefteq \sigma$$

$$\frac{A \vdash M : \sigma \supset \tau \quad A' \vdash N : \sigma'}{A \cup A'_{int} \vdash MN : \tau} \supset \text{App}\ \sigma' \trianglelefteq \sigma$$

Figure 2.4: Baker-Finch's type assignment system $R_\supset$

As a result, typings such as

$$\frac{\begin{array}{c}\vdots \\ \vdash \lambda f.\lambda x.fx : (\sigma \supset \tau) \rightarrow \sigma \supset \tau\end{array} \quad \begin{array}{c}\vdots \\ \vdash M : \sigma \rightarrow \tau\end{array} \quad \sigma \rightarrow \tau \trianglelefteq \sigma \supset \tau}{\vdash (\lambda f.\lambda x.fx)M : \sigma \supset \tau} \supset \text{App}$$

are possible. In this case, an argument with strict function type $\sigma \rightarrow \tau$ is supplied as argument to a function expecting an non-strict functional argument of type $\sigma \supset \tau$.

Note that, in the Var rule, A_{irr} denotes a base of typing hypotheses in which all tags are the empty string ϵ. Also, in rule $\supset$ App, A_{int} denotes a base in which the tags of typing hypotheses are ?.

Chapter 3

Usage and neededness in the λ-calculus

In this chapter, we define the use made of a formal parameter by a function. In other words, for functional λ-terms of the form $\lambda x.M$, we provide a semantics of the use made of x in M. Broadly speaking, the semantics of the use made of x in M is given by whether or not the value of a redex R substituted for x in M is *required* during evaluation, for all possible values of R. The meaning of *required* in this context is derived from neededness and head-neededness as defined by Barendregt, Kennaway, Klop and Sleep [7]. Our approach is based on work by Wright [93] and Baker-Finch [3] [4] who use neededness to provide the semantics of resource-aware type expressions (see Section 2.6 of Chapter 2 for an introduction to the resource-aware type systems of Wright and Baker-Finch). However, our aim in this chapter will be to use neededness (and head-neededness) to define properties about functions in their parameters that describe how those parameters are used, rather than to use the neededness of an argument in an application to interpret reduction information annotating a type expression,

The rest of this chapter is as follows: in Section 3.1, we review the work of Barendregt *et al* on neededness and head-neededness of redexes; in Section 3.2, we show how a simple resource use domain describing first-order strictness and absence can be defined using head-neededness; Section 3.3 extends the work of Barendregt *et al* to consider how much a redex may be needed or head-needed; based on this extended view of neededness and head-neededness, Section 3.4 shows how more complex resource use domains can be defined; related work is discussed in Section 3.6; Section 3.7 concludes.

A preliminary version of the work in this chapter (and Chapter 4) has appeared in [22]. However, as some differences in the treatment of resource use have been made since the publication of that work, it should be considered as being superseded by the work given in this chapter.

3.1 Neededness in the λ-calculus

Neededness describes the property possessed by some redexes in λ-terms that sooner or later they are contracted to normal form whatever reduction sequence is followed. For example, any reduction sequence taking the term

$$(\lambda x.x)R$$

(where R is a redex) to normal form will contract R. In addition, we can define a sharper notion of neededness called *head-neededness* which is based on reduction to head-normal form. Redexes known to be needed or head-needed can be reduced in parallel, thus reducing the length of a reduction sequence without affecting its termination characteristics. (A similar benefit is conferred by strictness analysis, which recovers the parallelism of a functional program lost to the sequentialism of lazy evaluation.)

Neededness and head-neededness were first defined by Barendregt, Klop, Kennaway and Sleep in [7] for the untyped λ-calculus, together with approximation algorithms to find subsets of the set of needed redexes. (Approximation is necessary since the problem of finding all needed redexes can be reduced to the Halting Problem.) In this section, we review the definitions of neededness etc. from that work necessary to define the simple resource use domain in Section 3.2.

Stating that a redex is needed if it is contracted on all reduction paths, however, obscures the fact that the redex actually contracted may be a copy of the original redex.

For example, given the term

$$(\lambda x.(\lambda y.\lambda z.z)xx)R$$

where R is some redex, one possible reduction sequence is

$$\begin{array}{ll} & (\lambda x.(\lambda y.\lambda z.z)xx)R \\ \rightarrow_\beta & (\lambda y.\lambda z.z)RR \\ \rightarrow_\beta & (\lambda z.z)R \\ \rightarrow_\beta & R \\ \rightarrow_\beta & \ldots \end{array}$$

in which it is the second copy of R, created by substitution in the first step of the reduction, which is contracted. To enable us to identify copies of a redex created during reduction with their original, we require the notion of *descendant.* (The following definition of descendant uses Klop's labelled λ-calculus, for a discussion of which see Section 2.1.1.)

Definition 3.1 *Let M be an unlabelled λ-term and let ϕ be an initial mapping of sub-terms of M to labels in a label set L. Furthermore, let $M \twoheadrightarrow_\beta M'$, and φ be the mapping of sub-terms of M' to L. Then a sub-term N' in M' is a descendant of a sub-term N in M if $\phi(N) = \varphi(N')$.*

In [56] and [6], a descendant of a redex is also called a *residual.* Note that residuals of redexes are themselves redexes. The following examples of descendants are taken from Barendregt [6] (Chapter 11, Section 2), adapted to use Klop's labelled λ-calculus.

Example 3.1 *Let R be the redex SKK in the following one-step reductions:*

$$\begin{array}{llll} \text{(i)} & ((\lambda x.x^1x^2)^3M^4)^5R^6 & \rightarrow & (M^4M^4)^5R^6 \\ \text{(ii)} & (\lambda x.x^1x^2)^3R^4 & \rightarrow & R^4R^4 \\ \text{(iii)} & (\lambda x.y^1)^2R^3 & \rightarrow & y^1 \end{array}$$

In (i), *R has one descendant or residual, while in* (ii), *it has two. In* (iii), *there are no descendants of R.*

Using descendants, we can define neededness and head-neededness as follows:

Definition 3.2 (Definition 3.1 of Barendregt *et al [7]) If R is a redex in M, then*

- *R is* needed *in M if every reduction sequence of M to normal form contracts a descendant of R.*
- *R is* head-needed *in M if every reduction sequence of M to head normal form contracts a descendant of R.*

Note that the leftmost reduction path to normal form contracts all needed redexes and only needed redexes (Theorem 3.6 of Barendregt *et al* [7]). This arises as a consequence of the fact that the position of leftmost redexes means that their contracted forms cannot be substituted into λ-abstractions and possibly erased. Similarly, the head reduction path to head normal form contracts all and only head needed redexes.

Example 3.2 (Example 3.2 of Barendregt *et al* [7]) *In the term*

$$\lambda x.\lambda y.(Ix(Ky(Iy)))$$

the redex Ix is needed and head-needed, the redex $(Ky(Iy))$ is needed but not head-needed, and the redex (Iy) is neither needed nor head-needed.

If a redex R is not needed in a term M, then, using the definition of neededness, there exists a reduction sequence $\mathcal{S}$ in which R is not contracted, i.e., R is *erased* in $\mathcal{S}$ or R is *erasable* in $\mathcal{S}$ (Definition 3.3 of Barendregt *et al* [7]). In Example 3.2, the redex *(Iy)* is erasable.

A corollary of the definitions of neededness and head-neededness is that every head-needed redex is also needed (but not vice versa), since every reduction to normal form contains a reduction to head-normal form [7][1].

Recursion in the λ-calculus is usually implemented using the Y combinator (see Section 2.1 of Chapter 2 for a definition of Y) or by a fix-point constant *fix*. The presence of

[1]Every head redex is a leftmost redex, but not conversely (See Chapter 8, §4 of Barendregt [6]).

the Y combinator in a λ-term, however, may result in an infinite series of reductions. For example, β-reduction of the term

$$Y(\lambda f.\lambda x.fx)$$

is non-terminating. One possible reduction for this term is

$$\begin{array}{ll} & Y(\lambda f.\lambda x.fx) \\ \rightarrow_\beta & (\lambda f.\lambda x.fx)(Y(\lambda f.\lambda x.fx)) \\ \rightarrow_\beta & \lambda x.(Y(\lambda f.\lambda x.fx))x \\ \rightarrow_\beta & \lambda x.((\lambda f.\lambda x.fx)(Y(\lambda f.\lambda x.fx)))x \\ \rightarrow_\beta & \lambda x.(\lambda x.(Y(\lambda f.\lambda x.fx))x)x \\ \rightarrow_\beta & \cdots \end{array}$$

Where reduction does not terminate in a normal form (head normal form), all redexes are considered to be needed (head-needed) [7]. For example, in the λ-term M defined as

$$(\lambda x.xx)(\lambda x.xx)(KzR)$$

where R is any redex, R is erasable. However, R is also said to be head-needed since M does not possess a head normal form. Hence, for example, any redex R in the term

$$(Y(\lambda f.\lambda x.fx))R$$

will be head-needed.

The following proposition shows that non-neededness persists over reduction.

Theorem 3.1 (Theorem 3.5 of Barendregt *et al* [7].) *Let $M \twoheadrightarrow_\beta M'$. Let S be a redex in M and S' be a descendant of S in M'. Then*

(i)	*S is not needed*	$\Rightarrow$	*S' is not needed*
(ii)	*S is not head-needed*	$\Rightarrow$	*S' is not head-needed*

Equivalently,

(i)	*S' is needed*	$\Rightarrow$	*S is needed*
(ii)	*S' is head-needed*	$\Rightarrow$	*S is head-needed*

Consequently, the reduction relation $\twoheadrightarrow_\beta$ preserves neededness and head-neededness, as described in the following proposition.

Proposition 3.1 (Proposition 3.7 of Barendregt *et al* [7]) *Assume M has a normal form (head normal form) and that R is a needed (head needed) redex of M. If $M \twoheadrightarrow_\beta N$ is a reduction sequence that does not reduce any descendant of R then R has a needed (head-needed) descendant in N.*

Following from Theorem 3.1 and Proposition 3.1, Lemma 3.1 shows that neededness is preserved under conversion.

Lemma 3.1 (Lemma 3.11 of Barendregt *et al* [7]) *If $F =_\beta F'$, then R needed (head-needed) in $FR \Rightarrow R$ needed (head-needed) in $F'R$.*

Therefore, if a redex R is needed (head-needed) in XR, then

$$\forall Y \in [X] \Rightarrow R \text{ is needed (head needed) in } YR$$

where $[X]$ denotes the convertibility class of X. In other words, if R is needed (head needed) in XR, then R is needed (head needed) when supplied to all the terms in the convertibility class of X.

3.2 A simple definition of resource use

In this section, we show how a domain of simple resource use values which represent first-order strictness and absence can be defined on the basis of neededness and head-neededness. In other words, we can describe a term $\lambda x.M$ as being first-order strict, absent, or non-strict in its parameter x. In addition, we present a set of inference rules for deriving the resource use of variables in λ-terms and show these to be sound with respect to our definition of resource use.

As Barendregt *et al* [7] observe

$$R \text{ is head needed in } C[R] \Leftrightarrow C[\;] \text{ is strict in its argument } [\;].$$

In other words, head neededness corresponds to first-order strictness[2]. This observation provides the basis for our definition of a *strict* resource use. Intuitively, if a redex R is head needed in $(\lambda x.M)R$, then $\lambda x.M$ is strict in its bound variable x Furthermore, if there exists an R such that R is head-needed in $(\lambda x.M)R$, it is easy to show that for all redexes R', R' is head-needed in $(\lambda x.M)R'$. Hence, we define a function as being strict in its parameter if all redexes substituted for the parameter are head-needed. Note that if $(\lambda x.M)R$ is non-terminating for all possible values of R, then the function is considered to be strict in its parameter x. The reason being that all redexes are head-needed on non-terminating reduction sequences (as mentioned in Section 3.1 above).

To define a resource use of absence, head neededness is insufficient. Absence attached to a parameter x of a term $\lambda x.M$ indicates not only that a value substituted for x will not be evaluated as a first-order value, but that it will not be evaluated in the context of values substituted for other parameters. For example, given the term

$$\lambda x.\lambda y.yx$$

any value R substituted for x will not be head needed in the evaluation of $(\lambda x.\lambda y.yx)R$ to head normal form. However, it may become head-needed as a result of values substituted for y. For example, R is head needed in

$$(\lambda x.\lambda y.yx)R(\lambda z.z)$$

but not in

$$(\lambda x.\lambda y.yx)R(Kz)$$

To be certain that a function parameter will not be used (is absent), we must be sure that values substituted for the parameter will not be head-needed in the context of arguments substituted for further parameters (e.g., *(Kz)* for y in the example above). Note that not even neededness is sufficient to define absence in this (higher-order) case.

[2]The restriction to first-order strictness is due to the fact that no term occurs to the left of a head redex, other than λ-bindings, and consequently, the head redex does not appear as an argument to sub-terms to its left.

However, we can define a simpler first-order version of absence. A parameter x of a term $\lambda x.M$ is said to be (first-order) absent if a redex R, substituted for x in M, is *not needed*, for all possible values of R. (Note that, similar to strictness discussed above, if there exists an R such that R is not needed in $(\lambda x.M)R$, then for all R', R' is not needed in $(\lambda x.M)R'$.) For example, x is absent in

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

since any redex R substituted for x will not be needed (or head needed) and hence will not be required by the values of arguments substituted for y or z.

We also define a third resource use of *non-strictness* to represent the use of variables such as z in the example above. For such variables, we cannot determine in general whether they will be used strictly or will be absent or neither. Typically, this is because they occur in the position of arguments to other, higher-order variables (such as y in the above example).

Definition **3.3** formally defines simple resource use (initially for variables that occur free in λ-terms). Note that $\Downarrow_L$ ($\Downarrow_H$) denote the leftmost (head) evaluation to normal (head normal) form.

Definition 3.3 *For $x \in Var$, $M \in \Lambda$, say that $x : u \in M$ (x has use u in M) where u is defined as follows:*

(i) $u \equiv \mathbf{S} \Rightarrow \forall R.R$ *is head needed in* $\Downarrow_H M[R/x]$,

(ii) $u \equiv \mathbf{A} \Rightarrow \forall R.R$ *is not needed in* $\Downarrow_L M[R/x]$,

(iii) $u \equiv \mathbf{N} \Rightarrow \forall R.R$ *is or is not needed in* $\Downarrow_L M[R/x]$.

It is easy to show that the resource use of a free variable x in a λ-term M is preserved by β-conversion.

Lemma 3.2 *Let $x : u \in M$ according to Definition 3.3. Let $M =_\beta M'$. Then $x : u \in M'$.*

Proof Follows from Lemma 3.1 on the preservation of neededness and head-neededness under β-conversion.

$$\begin{tikzcd} & \mathrm{N} & \\ \mathrm{S} \arrow[ur] & & \mathrm{A} \arrow[ul] \end{tikzcd}$$

Figure 3.1: Resource use domain S

□

We also define a relation between a function applied to an argument and a resource use value that says that the evaluation of the argument is *congruent* with the definition of the resource use value.

Definition 3.4 *Let M be a λ-term. Then $M \cong u$ implies that, given any redex R*

(i) $u \equiv \mathrm{S} \Rightarrow$ *R is head needed in $\Downarrow_H MR$,*

(ii) $u \equiv \mathrm{A} \Rightarrow$ *R is not needed in $\Downarrow_L MR$,*

(iii) $u \equiv \mathrm{N} \Rightarrow$ *R is or is not needed in $\Downarrow_L MR$.*

We also define a reflexive, transitive, and anti-symmetric relation or partial order $\sqsubseteq_{\mathrm{S}}$ over the simple resource values S, A and N as follows:

$$\begin{array}{lll} \mathrm{S} & \sqsubseteq_{\mathrm{S}} & \mathrm{N} \\ \mathrm{A} & \sqsubseteq_{\mathrm{S}} & \mathrm{N} \\ u & \sqsubseteq_{\mathrm{S}} & u, \quad u \in \{\mathrm{S}, \mathrm{A}, \mathrm{N}\} \end{array}$$

For example, any $\lambda x.M$ with use S in x also has use N in x, since any redex which is definitely head-needed is also needed, and hence possibly needed. Figure 3.1 shows the domain[3] S derived from the ordering of the simple resource use values by $\sqsubseteq_{\mathrm{S}}$.

[3]A domain is a pair $(D, \sqsubseteq)$ where D is a set with distinguished element $\perp_D$ and $\sqsubseteq$ is a partial order over D such that $\forall d \in D. \perp_D \sqsubseteq d$. In our presentation of S we have omitted $\perp_{\mathrm{S}}$ since it plays no role in the semantics of resource use (similarly for the other resource use domains presented in this chapter).

$$\frac{}{x \rhd x : \mathbf{S}}\,\mathrm{Var}_1 \qquad \frac{}{y \rhd x : \mathbf{A}}\,\mathrm{Var}_2$$

$$\frac{M \rhd x : u}{\lambda y.M \rhd x : u}\,\mathrm{Abs}_1 \qquad \frac{M \rhd x : u}{\lambda x.M \rhd x : \mathbf{A}}\,\mathrm{Abs}_2$$

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}\lambda y.M' \quad M \rhd x : u_1 \quad N \rhd x : u_2 \quad M' \rhd y : u_3}{MN \rhd x : u_1 + (u_2 \times u_3)}\,\mathrm{App}_1$$

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}yM_1 \ldots M_n \quad M \rhd x : u_1 \quad N \rhd x : u_2}{MN \rhd x : u_1 + (u_2 \sqcup \mathbf{A})}\,\mathrm{App}_2$$

$$\frac{}{MN \rhd x : u}\,\mathrm{App}_3 \text{ where } u \text{ is N if } x \in FV(MN) \text{ and A otherwise}$$

Figure 3.2: $\rhd_{\mathsf{S}}$: inference rules for simple resource use

In Figure 3.2, we present a system $\rhd_{\mathsf{S}}$ of inference rules for the derivation of resource use of free variables in λ-terms. In $\rhd_{\mathsf{S}}$, a statement of the form $M \rhd x : u$ is a deduction of resource use u made of a free variable x in a term M. An inference rule is of the form

$$\frac{S_1 \ldots S_n}{S}\,\mathrm{Rule}$$

where premises $S_1 \ldots S_n$ are statements about the deduction of resource use of variables in sub-terms, leading to the inference of resource use of a variable in a term composed of those sub-terms in S.

Note that in the case of the App$_n$ rules ($1 \leq n \leq 3$), head reduction of the application appears as a premise to the rules. The question of whether the application has a head normal form is important, since the unsolvability of a term implies non-strictness (following from the fact that all redexes in an unsolvable term are considered to be needed but not head-needed). If the head reduction path is finite, i.e., the application has a head normal form, then either of App$_1$ or App$_2$ may be applied, depending on the head normal form of the functional sub-term of the application. If the head-normal form is an abstraction, then

$\times$	S	A	N
S	S	A	N
A	A	A	A
N	N	A	N

$+$	S	A	N
S	S	S	S
A	S	A	N
N	S	N	N

Table 3.1: Operators $\times$ and $+$ over domain $\mathcal{S}$

the resource use of the bound variable in the body of the abstraction participates in the resource use of the variable free over the application. Otherwise, if the functional term is of the form $yM_1 \ldots M_n$ $(n \geq 0)$, then, since we lack information about what values y might take on, we assume either non-strictness or absence for the resource use of occurrences of the variable in the argument sub-term of the application. If, however, the head reduction path is infinite, and the variable in question is in the set of free variables of the application, we must assume that any redex substituted for the variable will be needed. Therefore, the resulting resource use of the variable is N. However, if the variable is not in the set of free variables, then we can assume absence for its resource use.

The App$_1$ and App$_2$ rules in Figure 3.2 make use of binary operators $\times$ and $+$ to derive resource use values for occurrences of a free variable occurring in the functional and argument sub-components of an application term. The operators $\times$ and $+$ are defined in Table 3.1 below. Note that both $\times$ and $+$ are commutative and associative, and that S is the identity for $\times$ and $\mathrm{A} \times u = \mathrm{A}$, while A is the identity for $+$ and $\mathrm{S} + u = \mathrm{S}$.

Example 3.3 *Given a term* $(\lambda x.((\lambda y.y)x))z$*, the deduction of the resource use made of the free variable* z *is represented as follows (rule labels have been omitted for the purpose of presentation).*

Let $[P]$ *represent*

$$\dfrac{\dfrac{(\lambda y.y)x\Downarrow_{\mathcal{H}}x \neq \bot \quad \lambda y.y\Downarrow_{\mathcal{H}}\lambda y.y \quad \dfrac{\dfrac{}{y \triangleright z : \mathbf{A}}}{\lambda y.y \triangleright z : \mathbf{A}} \quad \dfrac{}{x \triangleright z : \mathbf{A}} \quad \dfrac{}{y \triangleright y : \mathbf{S}}}{(\lambda y.y)x \triangleright z : \mathbf{A} + (\mathbf{S} \times \mathbf{A})}}{\lambda x.((\lambda y.y)x) \triangleright z : \mathbf{A} + (\mathbf{S} \times \mathbf{A})}$$

and let $[Q]$ *represent*

$$\dfrac{(\lambda y.y)x\Downarrow_{\mathcal{H}}x \neq \bot \quad \lambda y.y\Downarrow_{\mathcal{H}}\lambda y.y \quad \dfrac{\dfrac{}{y \triangleright x : \mathbf{A}}}{\lambda y.y \triangleright x : \mathbf{A}} \quad \dfrac{}{x \triangleright x : \mathbf{S}} \quad \dfrac{}{y \triangleright y : \mathbf{S}}}{(\lambda y.y)x \triangleright x : \mathbf{A} + (\mathbf{S} \times \mathbf{S})}$$

in the proof

$$\dfrac{(\lambda x.(\lambda y.y)x)z\Downarrow_{\mathcal{H}}z \neq \bot \quad \lambda x.(\lambda y.y)x\Downarrow_{\mathcal{H}}\lambda x.(\lambda y.y)x \quad [P] \quad [Q] \quad \dfrac{}{z \triangleright z : \mathbf{S}}}{(\lambda x.((\lambda y.y)x))z \triangleright z : (\mathbf{A} + (\mathbf{S} \times \mathbf{A})) + ((\mathbf{A} + (\mathbf{S} \times \mathbf{S})) \times \mathbf{S})}$$

The inferred resource use of z simplifies to **S** *according to the definitions of* $+$ *and* $\times$ *in Table 3.1 below.*

Example 3.4 *Let* $[P]$ *represent*

$$\dfrac{(\lambda x.x)y\Downarrow_{\mathcal{H}}y \neq \bot \quad \lambda x.x\Downarrow_{\mathcal{H}}\lambda x.x \quad \dfrac{\dfrac{}{x \triangleright z : \mathbf{A}}\,\mathrm{Var}_2}{\lambda x.x \triangleright z : \mathbf{A}} \quad \dfrac{}{y \triangleright z : \mathbf{A}}\,\mathrm{Var}_2 \quad \dfrac{}{x \triangleright x : \mathbf{S}}\,\mathrm{Var}_1}{(\lambda x.x)y \triangleright z : \mathbf{A} + (\mathbf{S} \times \mathbf{A}) = \mathbf{A}}\,\mathrm{App}_1$$

in the inference of the resource use of z in the λ-term $((\lambda x.x)y)z$

$$\dfrac{((\lambda x.x)y)z \Downarrow_{\mathcal{H}} yz \neq \bot \quad (\lambda x.x)y \Downarrow_{\mathcal{H}} y \quad [P] \quad \dfrac{}{z \rhd z : \mathbf{S}}\,\text{Var}_1}{((\lambda x.x)y)z \rhd x : \mathbf{A} + (\mathbf{S} \sqcup \mathbf{A})}\,\text{App}_2$$

Hence z has resource use $\mathbf{A} + (\mathbf{S} \sqcup \mathbf{A}) = \mathbf{N}$ in $((\lambda x.x)y)z$.

The following lemmas are concerned with the derivation of the resource use of a free variable in a compound term, given its resource use in the sub-expressions of the term.

Lemma 3.3 *Let $x \in Var$ and $x : u \in N$. Given $\lambda y.M$ such that $x \notin FV(M)$. If $y : \mathbf{A} \in M$, then $x : \mathbf{A} \in (\lambda y.M)N$.*

Proof By definition of use $\mathbf{A}$ for y in M, N is erased in evaluating $(\lambda y.M)N$ to normal form. Therefore, any redex R substituted for x in $(\lambda y.M)N$ will also be erased, and so $x : \mathbf{A} \in (\lambda y.M)N$ by Definition 3.3.

□

Lemma 3.4 *Let $x \in Var$ and $x : \mathbf{A} \in N$. Then for any $\lambda y.M$ such that $x \notin FV(M)$, $x : \mathbf{A} \in (\lambda y.M)N$.*

Proof Let $N' \equiv N[R/x]$ for any redex R. Then R is erased in the evaluation of any descendant of N' contracted during the evaluation of $(\lambda y.M)N'$. Hence $x : \mathbf{A} \in (\lambda y.M)N$ by Definition 3.3.

□

Lemma 3.5 *Let $x \in Var$. Let N be a λ-term such that $x : u \in N$, and let $\lambda y.M$ be a λ-term such that $x \notin FV(M)$. If $y : \mathbf{N} \in M$, then $u \equiv \mathbf{A}$ implies $x : \mathbf{A} \in (\lambda y.M)N$, and $u \equiv \mathbf{S}$ or $u \equiv \mathbf{N}$ implies $x : \mathbf{N} \in (\lambda y.M)N$. Otherwise, if $\lambda y.M$ has use $\mathbf{S}$ in y, then $x : u \in (\lambda y.M)N$.*

Proof For the first case, where y has use N: if $u \equiv$ **A**, then by Lemma 3.4, $x : \mathbf{A} \in (\lambda y.M)N$; if $u \equiv$ S or $u \equiv$ N, then by Definition 3.3, it is not known if N is needed (head needed) in the evaluation of $(\lambda y.M)N$ to normal (head normal) form. Hence it is not known if any redex R substituted for x in N will be needed (head-needed), and therefore $x : \mathrm{N} \in (\lambda y.M)N$. If y has use S, then N will be head needed in the evaluation to head normal form, and therefore the use of x in the application reduces to the use of x in N.

□

The following lemma demonstrates the soundness of the definition of $\times$.

Lemma 3.6 *Assume MN, such that $M \Downarrow_{\mathcal{H}} \lambda y.M'$, and $y : u_1 \in M$. Let $x \in Var$, such that $x \notin FV(M)$, and assume $x : u_2 \in N$. Then $x : u_1 \times u_2 \in MN$.*

Proof By case analysis over the possible values of u_1, and u_2:

$$\begin{array}{lll}
u_1 \equiv \mathrm{S} & \Rightarrow & u_1 \times u_2 = u_2, \text{ by Lemma 3.5} \\
u_1 \equiv \mathbf{A} & \Rightarrow & u_1 \times u_2 = u_1, \text{ by Lemma 3.3} \\
u_1 \equiv \mathrm{N} & \Rightarrow & u_1 \times u_2 = u_1, \text{ if } u_2 \equiv \mathrm{S}, \text{ by Lemma 3.5} \\
 & & u_1 \times u_2 = u_2, \text{ if } u_2 \equiv \mathbf{A}, \text{ by Lemma 3.4} \\
 & & u_1 \times u_2 = u_1, \text{ if } u_2 \equiv \mathrm{N}, \text{ by Lemma 3.3}
\end{array}$$

An examination of the definition of $\times$ in Table 3.1 shows that $\times$ satisfies these conditions.

□

In rule App$_2$ of $\rhd_{\mathrm{S}}$, when the functional sub-term of an application turns out not to be a λ-abstraction, we assume either non-strictness or absence for the resource use of those occurrences of a variable in the argument sub-term, reasoning that we know nothing about the way the argument to a higher-order variable (e.g., the variable y in the second premise to the rule) will be evaluated. The following lemma shows the correctness of the deduction of this resource use in the conclusion to App$_2$ as being $u \sqcup \mathbf{A}$ (where u is the resource use of the variable in the argument).

Lemma 3.7 *Assume $x \in Var$. Given M such that $x \notin FV(M)$, and $M \Downarrow_{\mathcal{H}} yM_1 \ldots M_n$. For all possible uses u of x in N, x has use $u \sqcup \mathbf{A}$ in $(yM_1 \ldots M_n)N$.*

Proof Clearly, N is be needed but not head-needed in $(yM_1 \ldots M_n)N$ (since it will not be substituted into a λ-abstraction, it cannot be copied or erased). Therefore, if $x : \mathrm{S} \in N$, any redex R substituted for x in N will be head-needed in N and therefore needed in $(yM_1 \ldots M_n)N$, i.e., $x : \mathrm{N} \in (yM_1 \ldots M_n)N$. A similar argument follows to show that if $x : \mathrm{N} \in N$, then $x : \mathrm{N} \in (yM_1 \ldots M_n)N$, and also that if $x : \mathbf{A} \in N$, then $x : \mathbf{A} \in (yM_1 \ldots M_n)N$. An examination of $u \sqcup \mathbf{A}$ for all values $u \in \mathrm{S}$ shows that it meets these requirements.

□

So far we have dealt with the derivation of the resource use of a free variable present in the argument sub-term of an application, but not present in the functional sub-term. To derive the resource use of a free variable present in both component terms of an application requires an operator to compose the resource use values of separate occurrences of a variable in distinct sub-terms. For example, $x : \mathrm{S} \in \lambda y.x$, while $x : \mathbf{A} \in (Kzx)$. Since $y : \mathbf{A} \in x$, then by Lemma 3.3, the occurrence of x in (Kzx) has resource use $\mathbf{A}$ in $(\lambda y.x)(Kzx)$. Therefore, the resource use of x in $(\lambda y.x)(Kzx)$ is derived from the resource uses S and $\mathbf{A}$ for the two distinct occurrences of the variable. In this case, $x : \mathrm{S} \in (\lambda y.x)(Kzx)$, since any redex R substituted for x will be head-needed. To perform this composition, we require the binary operator $+$. Lemma 3.8 below demonstrates the soundness of the definition of $+$ given in Table 3.1.

Lemma 3.8 *Let $x_1 \ldots x_2$ be the distinct occurrences of the same variable x in term M. If $u_1 \ldots u_n$ are the resource use values of $x_1 \ldots x_n$ respectively, i.e., $x_i : u_i \in M$, then $x : u_1 + \ldots + u_n \in M$ (where $+$ is defined in Table 3.1).*

Proof The proof proceeds by case analysis. If any u_i of $u_1 \ldots u_n$ is S, then by Definition 3.3, any redex R substituted for x will be head-needed, i.e., $x : \mathrm{S} \in M$. If no

u_i of $u_1 \ldots u_n$ is S, then if any u_j of $u_1 \ldots u_n$ is N, then also by Definition 3.3, any redex R substituted for x may be needed in M. Hence $x : \mathbf{N} \in M$. Furthermore, only if $u_1 = \mathbf{A} \ldots u_n = \mathbf{A}$ do we have $x : \mathbf{A} \in M$. The definition of $+$ satisfies these properties, since for all values $u \in \mathrm{S}$, $\mathrm{S} + u = \mathrm{S}$, while for all $u \neq \mathrm{S}$, $\mathrm{N} + u = \mathrm{N}$. Lastly, since $\mathbf{A}$ is the identity for $+$, $\mathbf{A} + u = \mathbf{A} \Rightarrow u \equiv \mathbf{A}$.

□

We now show that the inference system $\rhd_{\mathrm{S}}$ described in Figure 3.2 is sound with respect to Definition 3.3 of resource use. In the following theorem of the soundness of $\rhd_{\mathrm{S}}$, we write $M \models x : u$ to mean that $x : u \in M$ according to Definition 3.3.

Theorem 3.2 *(Soundness of inference of resource use.)*
$M \rhd x : u \Rightarrow M \models x : u$.

Proof The proof is by structural induction over the height of a proof using the inference rules. We show that for each case, the resource use assigned to a free variable by the inference rules is the resource use of the variable according to Definition 3.3.

Case 1:

$$\frac{}{x \rhd x : \mathrm{S}}\ \mathrm{Var}_1$$

For all redexes R, $x[R/x] \equiv R$. Hence, R is a head redex and x has resource use S by Definition 3.3.

Case 2:

$$\frac{}{y \rhd x : \mathbf{A}}\ \mathrm{Var}_2$$

For all redexes R, $y[R/x] \equiv y$. Hence R is erased, and the resource use value of x is $\mathbf{A}$.

Case 3:

$$\frac{M \rhd x : u}{\lambda y.M \rhd x : u}\ \mathrm{Abs}_1$$

By the induction hypothesis, we have that $M \models x : u$. By the rules of substitution for the λ-calculus,

$$(\lambda y.M)[R/x] \equiv \lambda y.M[R/x]$$

(assuming changes of bound variables to avoid name clash problems). If R is a head-needed redex in M, i.e., the head reduction path contains a reduction of M to $RN_1 \ldots N_k$, then R is also head-needed in $\lambda y.M$, since R is also a head-redex in $\lambda y.RN_1 \ldots N_k$. Hence, $x : u \in \lambda y.M$.

Case 4:

$$\frac{M \rhd x : u}{\lambda x.M \rhd x : \mathbf{A}} \text{Abs}_2$$

Since x is not free in $\lambda x.M$, R is erased in $(\lambda x.M)[R/x]$ and consequently is not contracted in the head reduction of $(\lambda x.M)[R/x]$. Therefore, by Definition 3.3, $x : \mathbf{A} \in M$.

Case 5:

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}\lambda y.M' \quad M \rhd x : u_1 \quad N \rhd x : u_2 \quad M' \rhd y : u_3}{MN \rhd x : u_1 + (u_2 \times u_3)} \text{App}_1$$

By induction, we have that $M \models x : u_1$, $N \models x : u_2$, and $M' \models y : u_3$. Also, we have that $MN{\Downarrow_{\mathcal{H}}}\lambda y.M'$ and that, also by induction, $M' \models y : u_3$. Rename the occurrences of x in N with a fresh variable w. By Lemma 3.6, we have

$$MN \models w : u_2 \times u_3$$

and subsequently, by Lemma 3.8, we have

$$MN \models x : u_1 + (u_2 \times u_3)$$

Case 6:

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}yM_1 \ldots M_n \quad M \rhd x : u_1 \quad N \rhd x : u_2}{MN \rhd x : u_1 + (u_2 \sqcup \mathbf{A})} \text{App}_2$$

By induction, $M \models x : u_1$ and $N \models N : u_2$. Also, we have that $MN{\Downarrow_{\mathcal{H}}}yM_1 \ldots M_n$. Let w rename x in N. Then $w : u_2 \sqcup \mathbf{A} \in MN$ by Lemma 3.7. Finally, by Lemma 3.8, we have $MN \models x : u_1 + (u_2 \sqcup \mathbf{A})$.

Case 7:

$$\frac{}{MN \rhd x : u}\text{App}_3 \text{ where } u \text{ is N if } x \in FV(MN) \text{ and A otherwise}$$

If $x \in FV(MN)$, then any redex R substituted for x in MN is may or may not be needed, and therefore, we have $x : \mathrm{N} \in MN$. If $x \notin FV(MN)$, then any redex R substituted for x in MN does not appear in $(MN)[R/x]$, and hence is not needed (despite the non-termination of the application), i.e., $x : \mathrm{A} \in MN$.

□

It would also be desirable to demonstrate the completeness of $\rhd_S$, which would show that whenever $x : u \in M$, then $M \rhd x : u$. A proof of such a theorem would follow a structural induction over λ-terms. However, the rule App$_3$ means that the system is not sufficiently strong to be complete in this sense, since it allows us to infer a resource use of N for a free variable in a term MN without reference to the resource use of the variable in the sub-terms M and N.

The inference system $\rhd_S$ presented in Figure 3.2 is unsuitable to be used as a static analysis of strictness and absence in λ-terms because of the use of head reduction and the test for head normal form in the premises of the App rules. Its usefulness, however, will become apparent in the following chapter on resource use in intuitionistic logic, where we will describe the relationship between resource use in λ-terms and intuitionistic logic by structural induction over both intuitionistic proofs and typed λ-terms.

3.3 Neededness in detail

In this section, we extend the work of Barendregt, Kennaway, Klop and Sleep, as reviewed in Section 3.1 above, to measure in more detail the neededness and head-neededness of redexes during reduction. This will allow us to develop a framework for defining more complex resource use domains based on neededness and head-neededness in Section 3.4 below. The essence of our approach will be to define the *degree* to which a redex may be needed (head-needed) in terms of the number of descendants of the redex that are contracted on the leftmost (head) reduction path. This idea is similar to that outlined by Wright in [93] (and described in greater detail in a joint work with Baker-Finch [94]) as the basis for a semantics of a type language incorporating complete sharing information (see Section 3.6 below).

As an example of what we mean by degrees of neededness, consider the λ-term

$$(\lambda f.\lambda x.f(fx))(SKK)(Iy)$$

and its reduction to normal form using leftmost reduction, i.e.,

$$\begin{array}{ll} & (\lambda f.\lambda x.f(fx))(SKK)(Iy) \\ \rightarrow_\beta & (\lambda x.(SKK)((SKK)x))(Iy) \\ \rightarrow_\beta & (SKK)((SKK)(Iy)) \\ \twoheadrightarrow_\beta & I((SKK)(Iy)) \\ \rightarrow_\beta & ((SKK)(Iy)) \\ \twoheadrightarrow_\beta & I(Iy) \\ \rightarrow_\beta & (Iy) \\ \rightarrow_\beta & y \end{array}$$

in which two descendants of the redex (SKK) are contracted, while only one descendant of the redex *(Iy)* is contracted. Consequently, by our intuitive definition, the *degree* to which the redex *(SKK)* is needed is greater than that of *(Iy)*.

In other words, we measure the degree to which a redex may be needed not just by whether a descendant is contracted on the leftmost reduction path to normal form, but by *how many* descendants are contracted. Similarly, we measure the degree of head-neededness by how many descendants are contracted in head reduction to head normal form.

For any term M and sub-redex R of M, the number of descendants of R contracted will vary according to the reduction path followed. The leftmost-innermost reduction path $\mathcal{LI}$ (as used in strict functional languages, such as Hope [32] for evaluating function application) for the example above would be as follows:

$$
\begin{aligned}
& (\lambda f.\lambda x.f(fx))(SKK)(Iy) \\
\rightarrow_\beta\ & (\lambda f.\lambda x.f(fx))I(Iy) \\
\rightarrow_\beta\ & (\lambda x.I(Ix))(Iy) \\
\rightarrow_\beta\ & (\lambda x.(Ix))(Iy) \\
\rightarrow_\beta\ & (\lambda x.x)(Iy) \\
\rightarrow_\beta\ & (\lambda x.x)y \\
\rightarrow_\beta\ & y
\end{aligned}
$$

in which the redex *(SKK)* is contracted only once.

Other reduction sequences may contract a redex R more times than on the leftmost sequence. For example, for the term

$$(\lambda z.(\lambda x.\lambda y.x)zz)(Ia)$$

the leftmost reduction sequence contracts only one descendant of the redex *(Ia)*.

$$
\begin{aligned}
& (\lambda z.(\lambda x.\lambda y.x)zz)(Ia) \\
\rightarrow_\beta\ & (\lambda x.\lambda y.x)(Ia)(Ia) \\
\rightarrow_\beta\ & (\lambda y.(Ia))(Ia) \\
\rightarrow_\beta\ & (Ia) \\
\rightarrow_\beta\ & a
\end{aligned}
$$

Another reduction sequence for this term is

$$
\begin{aligned}
& (\lambda z.(\lambda x.\lambda y.x)zz)(Ia) \\
\rightarrow_\beta\ & (\lambda x.\lambda y.x)(Ia)(Ia) \\
\rightarrow_\beta\ & (\lambda x.\lambda y.x)a(Ia) \\
\rightarrow_\beta\ & (\lambda x.\lambda y.x)aa \\
\rightarrow_\beta\ & (\lambda y.a)a \\
\rightarrow_\beta\ & a
\end{aligned}
$$

in which two descendants of *(Ia)* are contracted, one of which is later erased.

In general, it would seem that the only correlation between the leftmost reduction sequence and other reduction sequences is the one provided by the definition of neededness, i.e., that if a redex is needed, then at least one descendant is contracted on all reduction sequences to normal form. However, the following results establish the important fact that the number of descendants of a redex contracted on the leftmost reduction path to normal form corresponds to the number of descendants of the same redex contracted on other reduction paths whose contracted forms are *not eventually erased.* The formulation and proof of these results uses Lévy's labelled λ-calculus [61] (see Section 2.1.1 of Chapter 2). In Lévy's version of a labelled λ-calculus, labels of redexes are not deleted when the redexes are contracted, as is the case in Klop's version [56]. Note that Baker-Finch in [3] uses a similar method to define neededness by adapting the labelled λ-calculus of Klop such that labels are not erased unless the term in which they occur is erased. However, it would seem that for this adapted calculus to be Church-Rosser it would have to impose an order on the collection of labels on λ-terms during reduction, in much the same way as Lévy does by concatenating label sequences. Hence, Baker-Finch's description of a labelled λ-calculus is likely to resemble Lévy's.

For the purposes of the following discussion, we define membership of the range of a mapping φ also means being a sub-string of a string in $range(\varphi)$, assuming a unique initial labelling.

Lemma 3.9 *Let R be a sub-redex of term M. Let φ be an initial labelling of M, assigning label a to R. Let ϑ be the labelling of M_{nf}, the normal form of M. Then*

$$R \textit{ erasable in } M \Rightarrow a \notin range(\vartheta)$$

Proof If R is erasable, then it is not contracted on the leftmost reduction path. In Lévy's labelled λ-calculus, $x^i[R/y] = x^i$. Hence, the labels in R do not appear in M_{nf} and so $a \notin range(\vartheta)$.

□

Corollary 3.3 *Let R be a sub-redex of term M. Let φ be an initial labelling of M, assigning label a to R, and let ϑ be the labelling of M_{nf}. If $a \in range(\vartheta)$, then R is needed in M.*

Proof By Lemma 3.9. If $a \in range(\vartheta)$, then R is not erasable, i.e., R is needed.

□

Corollary 3.4 *Let R be the leftmost redex in M. Let φ be an initial labelling of M assigning label a to R. Let ϑ be the labelling of M_{nf}. Then $a \in range(\vartheta)$, and furthermore, there exists only one occurrence of a in $range(\vartheta)$.*

Theorem 3.5 below states that the number of times a descendant of R is contracted in the leftmost reduction of M to normal form is the same as the number of labels of R left in the normal form of M. The importance of this theorem lies in the fact that it establishes that measuring the degree of neededness of a redex using the number of residuals of the redex contracted on the leftmost reduction path is sound in the sense that it corresponds to the way in which the value of the redex is required on other reduction paths. For example, if a reduction path contracts fewer residuals of R than the leftmost reduction path, then as a result of the following theorem, we can be sure that contracted forms of R are duplicated and are involved in the contraction of other redexes. Moreover, we can be sure that the extent to which residuals of R are contracted and its contracted form required in the contraction of other redexes (which are not themselves erased) corresponds to the number of residuals of R contracted on the leftmost reduction path.

We introduce the following notation to denote descendants of redexes contracted on a particular reduction path.

Notation 3.1 *Let R be a redex in M and let*

$$\mathcal{S} : M \twoheadrightarrow_{\beta} M'$$

denote a reduction sequence $\mathcal{S}$ from M to M'. Then the descendants of R contracted in $\mathcal{S}$ are denoted by $\{R\}_{\mathcal{S}}$. Write $|\{R\}_{\mathcal{S}}|$ for the number of descendants contracted.

Hence for a term M with sub-redex R, the descendants of R contracted on the leftmost (head) reduction path are denoted $\{R\}_{\mathcal{L}}$ ($\{R\}_{\mathcal{H}}$).

Theorem 3.5 *Let R be a sub-redex of M. Let φ be an initial labelling of M, assigning label a to R. Let ϑ be the labelling of M_{nf}. Then*

$$|\{R\}_{\mathcal{L}}| = \textit{no. of distinct occurrences of } a \in range(\vartheta)$$

Proof Assume $|\{R\}_{\mathcal{L}}| = m$, i.e., that m descendants of R occur as contracted redexes on the leftmost reduction path to normal form. By Corollary 3.4, there are at least m distinct occurrences of the label a in $range(\vartheta)$. Furthermore, since leftmost redexes cannot be substituted as arguments after their contraction, the labels of contracted residuals of R cannot be duplicated. Therefore, there are exactly m occurrences of a in $range(\vartheta)$.

□

The implication of Corollary 3.6 is that if the leftmost reduction path contracts just one residual of a redex, then on all other reduction paths to normal form, all but one of the contracted forms of the redex is erased.

Corollary 3.6 *Let R be a sub-redex of term M. Let φ be an initial labelling of M assigning label a to R, and let ϑ be the labelling of M_{nf}. If $|\{R\}_{\mathcal{L}}| = 1$, then there is exactly one occurrence of a in $range(\vartheta)$.*

Lemma 3.10 *Let M be a λ-term with sub-redex R. If $|\{R\}_{\mathcal{L}}| \geq 1$, then for all other reduction sequences $\mathcal{S}$ to normal form, $|\{R\}_{\mathcal{S}}| \geq 1$.*

Proof By Definition 3.2, if a redex is needed, then at least one descendant of the redex is contracted on all reduction paths to normal form.

□

In the case where head reduction to head normal form contracts a single descendant of a redex, and leftmost reduction to normal form of the same term also contracts a single descendant of the same redex, the following theorem shows that the redex contracted in the leftmost reduction occupies the head position in the term, i.e., it is also the contracted head redex.

Theorem 3.7 *Let R be a sub-redex of term M. If $|\{R\}_{\mathcal{H}}| = 1$ and $|\{R\}_{\mathcal{L}}| = 1$, then a residual of R is contracted in the head position in $\mathcal{L}$.*

Proof Head redexes are leftmost redexes. Therefore, if one descendant of R is contracted on the head reduction path to head normal form, then one or more descendants of R must be contracted on the leftmost reduction path to normal form. Furthermore, if $|\{R\}_{\mathcal{L}}| = 1$, the descendant of R contracted must occupy the head position for $|\{R\}_{\mathcal{H}}| = 1$ to be satisfied.

□

We now show that degrees of neededness and head-neededness are preserved under conversion of λ-terms, that is to say that if n residuals of a redex R are contracted in the leftmost reduction of MR and $M =_\beta M'$, then n residuals of R are contracted in the leftmost reduction of $M'R$. The proof of this theorem is greatly simplified by the use of labels.

Theorem 3.8 *If $M =_\beta M'$, then if n descendants of R are contracted on the leftmost (head) reduction of MR to normal (head normal) form, then n descendants of R are contracted in the leftmost (head) reduction of $M'R$ to normal (head normal) form.*

Proof Follows from the confluence of Lévy's labelled λ-calculus and Theorem 3.5.

□

The effect of non-termination with respect to neededness is to make all redexes needed (similarly for head-neededness). In the context of this section, non-termination has the effect of making the degree of neededness and head-neededness non-zero but infinite, i.e., equivalent to contracting an infinite number of descendants of a redex on the leftmost and head reduction paths.

3.4 Resource use in detail

In this section, we extend the domain of resource use values defined in Section 3.2. The work of the previous section now makes it possible to define resource use values in a manner similar to that used in Section 3.2, but which capture more information about the *degree* to which a function parameter is used, for example, whether a variable is used once or more than once within the body of the function that binds it.

As in Section 3.2, our aim is to take a property pertaining to arguments to a function as the basis for defining a property of the bound variable of the function. In this section, we take the extended view of degrees of neededness and head-neededness from Section 3.3 to be the properties of arguments with which we will define resource use properties of function variables. In the rest of this section, we first show how the simple resource use domain shown in Figure 3.1 can be extended to include a value describing *linearity* of use. Then we give a semantics in similar terms to a lattice of argument usage annotations described by Bierman in [10].

3.4.1 Adding a linear resource use

A *linear* resource use describes the case where the argument value substituted for a bound variable in a function is guaranteed to be evaluated exactly once. Worthwhile optimisations such as in-place or destructive updating of data structures can be made once it is known that the argument to a function will be used only once (see, for example, [11] [47]). Also, by distinguishing between a linear resource use and a strict but (possibly) non-linear value, we describe elementary sharing, which is also of use in optimising the performance of abstract machines such as the Spineless Tagless G-machine [77].

We define linearity to be first-order strictness without sharing. In other words, a bound variable x of a function $\lambda x.M$ is used linearly if any argument substituted for x will be evaluated once and exactly once.

Below, we define a linear resource use as well as strictness, absence and non-strictness

$$\begin{tikzcd}
& \mathbf{N} & \\
\mathbf{S} \arrow[ur] & & \mathbf{A} \arrow[ul] \\
\mathbf{L} \arrow[u] & &
\end{tikzcd}$$

Figure 3.3: Resource use domain L

in terms of the number of descendants of argument redexes occurring on the head and leftmost reduction paths.

Definition 3.5 *For $x \in Var$, $M \in \Lambda$, say that $x : u \in M$ (x has use u in M) where u is defined as follows:*

(i) $u \equiv \mathbf{L} \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| = 1$ *in* $\Downarrow_H M[R/x]$ *and* $|\{R\}_{\mathcal{L}}| = 1$ *in* $\Downarrow_L M[R/x]$

(ii) $u \equiv \mathbf{S} \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| \geq 1$ *in* $\Downarrow_H M[R/x]$

(iii) $u \equiv \mathbf{A} \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| = 0$ *in* $\Downarrow_H M[R/x]$ *and* $|\{R\}_{\mathcal{L}}| = 0$ *in* $\Downarrow_L M[R/x]$

(iv) $u \equiv \mathbf{N} \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| \geq 0$ *in* $\Downarrow_H M[R/x]$

The definition of linearity in Definition 3.5 above states that a variable x has use **L** (linear) in a term M if any redex R substituted for x in M has exactly one descendant contracted on the head reduction path to head normal form and exactly one descendant contracted in the leftmost reduction to normal form. By Theorem 3.7, the descendant contracted on the leftmost path is contracted as a head redex. The restriction in this case of the number of descendants contracted on the leftmost path is necessary to ensure true linearity. For example, the term $\lambda f.fx$ is linear in its bound variable f. However, this is not the case for the term $\lambda f.\lambda g.\lambda x.fx((gf)x)$, since for some redex R substituted for f, $|\{R\}_{\mathcal{L}}|$ will be greater than 1. (For example, $R \equiv ((\lambda a.\lambda b.\lambda c.acb)K)$.) Further arguments for the bound variables g and x may also require the value substituted for f.

Note that the form of linearity defined here is not the same as linearity in the sense used in Linear Logic-based term calculi (see, for example, Abramsky's Linear Term Calculus [1], Mackie's linear functional language Lilac [64], or the linear logic term calculus of Benton *et al* [8]). For example, the λ-term $\lambda x.Kxx$ is linear in x according to Definition 3.5 (since $|\{R\}_{\mathcal{H}}| = 1$ and $|\{R\}_{\mathcal{L}}| = 1$ for any redex R). However, x would not be considered linear in any equivalent term in the Linear Term Calculus, since x occurs twice syntactically.

The following lemma demonstrates that resource use with linearity is preserved by β-conversion.

Lemma 3.11 *Let $x : u \in M$ according to Definition 3.5. Let $M =_\beta M'$. Then $x : u \in M'$.*

Proof Follows from Theorem 3.8 on the preservation of degrees of neededness and head-neededness under β-conversion.

$\square$

Lemma 3.11 is equivalent to Lemma 3.2 for domain S given in Section 3.2 above. In that section, we also defined in Definition 3.4 a relation $\cong$ on functions and their arguments with respect to resource use values. The relation $\cong$ can also be defined for functions and arguments over L in the same manner.

We define a reflexive, transitive, and anti-symmetric ordering $\sqsubseteq_{\mathsf{L}}$ over the resource use values in Definition 3.5 as follows:

$$\begin{array}{ccc} \mathbf{L} & \sqsubseteq_{\mathsf{L}} & \mathbf{S} \\ \mathbf{S} & \sqsubseteq_{\mathsf{L}} & \mathbf{N} \\ \mathbf{A} & \sqsubseteq_{\mathsf{L}} & \mathbf{N} \\ u & \sqsubseteq_{\mathsf{L}} & u \end{array}$$

The semantics of the ordering relation is defined by reference to the meaning of the resource use values given in Definition 3.5. We interpret a statement such as

$$|\{R\}_{\mathcal{H}}| \geq 1 \ \textit{in} \ \Downarrow_H M[R/x]$$

using the set of values over which $|\{R\}_{\mathcal{H}}|$ ranges that satisfy the inequality in the statement (in this case, the set of positive integers). Conjunctions of such statements are interpreted as of pairs of such sets. For example, the conjunctive statement

$$|\{R\}_{\mathcal{H}}| \geq 1 \wedge |\{R\}_{\mathcal{L}}| \geq 1$$

is interpreted by the pair

$$(\{1\ldots\}, \{1\ldots\})$$

(where $\{1\ldots\}$ is the set of positive integers). The semantics of the ordering relation over resource use values is then given in terms of a subset relation between components of pairs of sets of positive integers. This is formalised in the following definition.

Definition 3.6 *Let u and u' be two resource use values as defined in Definition 3.5. Let the interpretation (s,s') represent u and (t,t') represent u'. Then $u \sqsubseteq u'$ iff*

$$s \subseteq t \wedge s' \subseteq t'$$

Subsequently, we are able to define the domain L of resource values defined in Definition 3.5 as shown in Figure 3.3.

We define a system $\triangleright_{\mathsf{L}}$ of inference rules for resource use with linearity in Figure 3.4, in a similar fashion to $\triangleright_{\mathsf{S}}$ defined in Section 3.2 for strictness and absence. The inference rules for resource use with linearity are almost identical to those for simple resource use, with the exception that the resource use inferred in the rule Var_1 is the linear value **L**, rather than **S**. Note also that the binary operators $\times$ and $+$ are re-defined (and overloaded) for L (see Table 3.2.)

We now show that $\triangleright_{\mathsf{L}}$ is sound with respect to Definition 3.5. In the following theorem, we write $M \models x : u$ to mean that $x : u \in M$ according to Definition 3.5.

Theorem 3.9 Soundness of $\triangleright_{\mathsf{L}}$.

$$M \triangleright_{\mathsf{L}} x : u \Rightarrow M \models x : u$$

Proof The proof follows the proof of soundness in the case of simple resource use (see Section 3.2 above), except for rule Var_1 for which the proof is as follows:

$$\frac{}{x \rhd_{\mathsf{L}} x : \mathbf{L}}\,\text{Var}_1 \qquad \frac{}{y \rhd_{\mathsf{L}} x : \mathbf{A}}\,\text{Var}_2$$

$$\frac{M \rhd_{\mathsf{L}} x : u}{\lambda y.M \rhd_{\mathsf{L}} x : u}\,\text{Abs}_1 \qquad \frac{M \rhd_{\mathsf{L}} x : u}{\lambda x.M \rhd_{\mathsf{L}} x : \mathbf{A}}\,\text{Abs}_2$$

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}\lambda y.M' \quad M \rhd_{\mathsf{L}} x : u_1 \quad N \rhd_{\mathsf{L}} x : u_2 \quad M' \rhd_{\mathsf{L}} y : u_3}{MN \rhd_{\mathsf{L}} x : u_1 + (u_2 \times u_3)}\,\text{App}_1$$

$$\frac{MN{\Downarrow_{\mathcal{H}}}P \neq \bot \quad M{\Downarrow_{\mathcal{H}}}yM_1 \ldots M_n \quad M \rhd_{\mathsf{L}} x : u_1 \quad N \rhd_{\mathsf{L}} x : u_2}{MN \rhd_{\mathsf{L}} x : u_1 + (u_2 \sqcup \mathbf{A})}\,\text{App}_2$$

$$\frac{}{MN \rhd_{\mathsf{L}} x : u}\,\text{App}_3 \text{ where } u \text{ is N if } x \in FV(MN) \text{ and } \mathbf{A} \text{ otherwise}$$

Figure 3.4: $\rhd_{\mathsf{L}}$: inference rules for resource use with linearity

$\times$	L	A	S	N
L	L	A	S	N
A	A	A	A	A
S	S	A	S	N
N	N	N	N	N

$+$	L	A	S	N
L	L	L	S	S
A	L	A	S	N
S	S	S	S	S
N	S	N	S	N

Table 3.2: Operators of $\times$ and $+$ over domain L

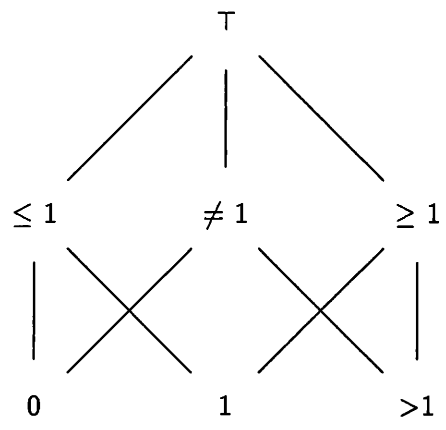

Figure 3.5: Bierman's domain

Case: $x \rhd_{\mathsf{L}} x : \mathbf{L}$.

For all redexes R, $x[R/x] = R$. By definition, R is a head redex, and furthermore, is evaluated once. In other words,

$$\forall R. |\{R\}_{\mathcal{H}}| = 1 \textit{ in } \Downarrow_{\mathcal{H}} x[R/x] \textit{ and } |\{R\}_{\mathcal{L}}| = 1 \textit{ in } \Downarrow_{\mathcal{L}} x[R/x]$$

which is the definition of **L** in Definition 3.5.

□

3.4.2 Bierman's domain

In [10], Bierman describes a complete lattice of values intended as resource use annotations to function types, reproduced in Figure 3.5. No semantics of the values in the lattice is given by Bierman, but it is easy to see that the resource use domain described in Section 3.4.1 above is a subdomain of Bierman's, with the following equivalences: $\top \equiv$ N; $\geq 1 \equiv$ S; $1 \equiv$ L; and $0 \equiv$ A. Therefore, the domain points 1, ≥ 1, 0, and $\top$ are defined as **L**, **S**, **A** and **N** respectively in Definition 3.5. We are also able to define the remaining points in similar fashion in Definition 3.7 below. (Note that, in keeping with our presentation of resource use domains in this chapter, we have omitted the least element $\perp$, since it plays no role in the semantics of resource use.)

Definition 3.7 *For $x \in Var$, $M \in \Lambda$, say that x has use u in M where u is defined as follows:*

(i) $u \equiv\ > 1 \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| > 1$ *in* $\Downarrow_H M[R/x]$ *and* $|\{R\}_{\mathcal{L}}| > 1$ *in* $\Downarrow_L M[R/x]$

(ii) $u \equiv\ \leq 1 \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| \leq 1$ *in* $\Downarrow_H M[R/x]$ *and* $|\{R\}_{\mathcal{L}}| \leq 1$ *in* $\Downarrow_L M[R/x]$

(iii) $u \equiv\ \neq 1 \Rightarrow \forall R.|\{R\}_{\mathcal{H}}| \neq 1$ *in* $\Downarrow_H M[R/x]$ *and* $|\{R\}_{\mathcal{L}}| \geq 0$ *in* $\Downarrow_L M[R/x]$

The semantics of the ordering relation $\sqsubseteq_{\mathsf{B}}$ over values in Bierman's use domain are as described for the ordering over the domain in Section 3.4.1 above.

3.5 Resource use and the typed λ-calculus

The work in this chapter on resource use in the (untyped) λ-calculus also applies to the typed λ-calculus, with one exception. Since the typed λ-calculus is strongly-normalising, we do not need rule App$_3$ in $\rhd_{\mathsf{S}}$ and $\rhd_{\mathsf{L}}$, nor the premises concerning reduction to head normal form in rules App$_1$ and App$_2$ in these systems.

However, we may wish to introduce a family of fix-point constants $fix_{\sigma\to\sigma}$ for each type σ into the typed λ-calculus to implement recursion. In which case, given a term $fix\ M$, we must find the resource use of a free variable in M over the calculation of the fix-point of M.

3.6 Related work

Wright in [93] uses *strong* head-neededness to define the semantics of resource-aware type expressions. Strong head-neededness differs from head-neededness in that it is defined with respect only to residuals contracted on the head reduction path, rather than all reduction paths to head normal form. For example, given the type expression

$$\sigma \Rightarrow \tau_1 b_1 \ldots \tau_n b_n \alpha$$

(i.e., a function of this type is strict in its first argument), then the semantics is (partly) described by

$$\forall d \in \sigma. \forall e_i \in \tau_i (1 \leq i \leq n). d \text{ strongly head-needed in } f \cdot d \cdot e_1 \ldots e_n$$

In Baker-Finch's work [3], persistent labels are used to determine whether a redex is needed. In Klop's concept of labelled reduction [56] the label of a redex is dispensed with once the redex has been contracted. However, in Baker-Finch's semantics, the label of the redex persists, and is erased only if the term it is attached to is erased during reduction. Therefore if the label of a redex appears in the normal form, then the redex with that label is said to be needed. Strictness and absence are defined by Baker-Finch as follows:

Definition 3.8 (Definition 3.2.1 of [3]). *Given a term M and a sequence of terms $P_1 \ldots P_n$ $(n \geq 0)$,*

(i) M *is* strict *in the context of* $P_1 \ldots P_n$ *if no term* N *is erased in* $MNP_1 \ldots P_n$,

(ii) M *is* constant *in the context of* $P_1 \ldots P_n$ *if any term* N *is erasable in* $MNP_1 \ldots P_n$.

An extension to this idea is outlined by Wright in [93] and described in more detail by Wright and Baker-Finch in [94], which describes complete sharing information in terms of the number of descendants of a redex that occur on the head reduction path, again given all possible values that may be substituted for further parameters. For example, functions linear in their first parameter are said (by their type) to be those functional λ-terms M such that only one descendant of the redex N is contracted in the head reduction of

$$MNP_1 \ldots P_n \qquad (n \geq 0)$$

for all possible values of N and $P_1, \ldots, P_n$. Generally, functions that use their parameter to degree m are those functions M that, for all possible values of $N, P_1, \ldots, P_n$, m descendants of N are contracted during head reduction of

$$MNP_1 \ldots P_2$$

Also of relevance, as mentioned in [93] and [3], is Sestoft's definition of usage intervals in [83]. Usage intervals are defined using the values *Zero*, *One* and *Many*, and give the lower and upper bounds within which the actual use of a function argument may fall; for example, the interval [*Zero*, *One*] means that a function argument may not be used, but if it is then it will be used once at most. Moreover, the inference system $\rhd_{\mathsf{L}}$ appears very similar to Sestoft's usage analysis function $\mathcal{U}_e$ (see Section 5.1.2 of [83]). Like our system $\rhd_{\mathsf{S}}$, $\mathcal{U}_e$ is essentially first-order but is extended to analyse higher-order functions by means of a closure analysis. The semantics and soundness of $\mathcal{U}_e$ are not considered in [83], unlike $\rhd_{\mathsf{S}}$ in this chapter.

Lastly, the inference system $\rhd_{\mathsf{S}}$ presented in Section 3.2 also bears some resemblance to the *context analysis* of Wadler and Hughes in [90]. Context analysis is based upon the idea of determining the need for the value of a free variable given the need for the value of its surrounding expression or *context.* As Davis and Wadler describe in [30], the context analysis in [90] was first-order and low-fidelity, by which is meant that the analysis is carried out on each free variable separately, which excludes the possibility of discovering *joint* strictness in two or more variables. It is easy to see that $\rhd_{\mathsf{S}}$ is similarly low-fidelity in that each free variable is examined separately from other free variables.

3.7 Summary and conclusions

In this chapter, we have presented a semantics of resource use in the λ-calculus, based on the neededness and head-neededness of arguments to functions. We defined a resource use domain describing first-order strictness and absence, and presented a system of inference rules for deriving the resource use of free variables, similar to the context analysis of Wadler and Hughes. A proof of the soundness of the inference system was given.

We also showed how neededness and head-neededness can be extended to *degrees* of neededness and head-neededness based on the number of residuals of a redex contracted on the leftmost and head reduction paths. We also established a correspondence between this measure of the degree of neededness (head-neededness) of a redex and the number of times a redex's value was required on other reduction paths using Lévy's labelled λ-calculus. Subsequently, we defined resource use beyond strictness and absence, for example, to include linearity and to provide a semantics for the annotations in Bierman's domain.

Chapter 4

Intuitionistic logic and resource use

In this chapter, we investigate the definition and semantics of resource use in intuitionistic logic, and its correspondence with the semantics of resource use in the typed λ-calculus (as described in Chapter 3) under the Curry-Howard isomorphism. Our aim is to demonstrate that resource use inferred for hypotheses in intuitionistic proofs is equivalent to the resource use of free variables in typed λ-terms, thus providing a justification of the inference of resource use for λ-terms by the type system in Chapter 5. As we will see, however, we are unable to demonstrate an equivalence between our definitions of resource use in proofs and λ-terms, although we are able to show a correspondence based on approximation.

The plan of this chapter is as follows: we define neededness and head-neededness for intuitionistic logic, and show their equivalence with neededness and head-neededness in the typed λ-calculus. Subsequently, we define a resource use domain of strictness and absence, and extend intuitionistic logic to incorporate resource use in proofs and propositions. We also show that the correspondence with resource use defined for the typed λ-calculus is an approximation rather than an equivalence. We also outline how the more complex resource use domains of Chapter 3 can be defined in intuitionistic logic.

Note that in the rest of this thesis, the term *intuitionistic logic* refers to intuitionistic

Axiom

$$\frac{}{\sigma \vdash \sigma}\ \text{Axiom}$$

Structural Rules

$$\frac{\Gamma, \sigma, \sigma \vdash \tau}{\Gamma, \sigma \vdash \tau}\ \text{Contraction} \qquad \frac{\Gamma \vdash \tau}{\Gamma, \sigma \vdash \tau}\ \text{Weakening}$$

Logical Rules

$$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \rightarrow \tau} \rightarrow \text{Intro}$$

$$\frac{\Gamma \vdash \sigma \rightarrow \tau \quad \Lambda \vdash \sigma}{\Gamma, \Lambda \vdash \tau} \rightarrow \text{Elim}$$

Figure 4.1: Intuitionistic implicational propositional logic

implicational propositional logic (see Figure 4.1 above). Also, reference to the λ-calculus is to the typed λ-calculus unless otherwise stated.

4.1 Neededness and head-neededness in intuitionistic logic

In this section, we define neededness and head-neededness for propositions and their proofs in intuitionistic logic.

4.1.1 Proofs in redex form

In the discussion of neededness and head-neededness in intuitionistic logic proofs, we will require the following definitions. The first defines the notion of a proof in *redex* form.

Definition 4.1 *Let P be a proof of proposition τ, i.e., $P : \tau$. P is a proof* redex *or proof in redex form if it is of the form*

$$\dfrac{\dfrac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau} \to \text{Intro} \qquad \begin{matrix} \vdots \\ \Lambda \vdash \sigma \end{matrix}}{\Gamma, \Lambda \vdash \tau} \to \text{Elim}$$

In other words, a redex proof is one in which the last rule used is $\to$ Elim, *which directly follows a use of* $\to$ Intro.

In addition to the notion of a proof redex, we will need the definition of leftmost and head proof redex. Both definitions rely on the concept of a *labelling* of rules in a proof.

A labelling has been defined by Troelstra [86] (called a *coding*), and similarly by Gallier [34], which assigns a finite string of natural numbers to each node in a proof tree. Here we define the labelling in the form of a function Φ_P over a proof P. In the following definitions, $\mathbb{N}$ is the set of natural numbers and $\mathbb{N}^*$ is the set of strings, including the empty string ϵ, generated over $\mathbb{N}$.

Definition 4.2 *For a proof tree $P : \sigma$, define by induction a function Φ_P mapping nodes (sequents) in $P : \sigma$ to values in $\mathbb{N}^*$ as follows:*

- *to the conclusion of P, assign the empty string ϵ*
- *if a deduction in P, assigned n by Φ_P, has m premises above it, then the premises are labelled left to right as $n0, \ldots, n(m-1)$ (where juxtaposition represents concatenation).*

The application of Φ_P to any sub-proof $R : \phi \in P : \sigma$, written $\Phi_P(R : \phi)$, returns the label assigned to the conclusion of R.

We also define a lexicographic ordering $\leq$ over strings (taken from Gallier [34], Section 2.2.2 of Chapter 2), which induces a depth-first, left-right ordering on nodes in a proof tree when they are labelled according to the scheme given in Definition 4.2.

Definition 4.3 *Let $\mathbb{N}_+^*$ denote the set of strings over natural numbers. Let $\leq$ be an ordering over strings $u, v \in \mathbb{N}_+^*$ such that $u \leq v$ if either u is a prefix of v, or there exist strings $x, y, z \in \mathbb{N}_+^*$, and natural numbers $i, j \in \mathbb{N}_+$ such that $i < j$, such that $u = xiy$ and $v = xjz$.*

Example 4.1 *The following are orderings over strings in $\mathbb{N}_+^*$:*

$$\begin{aligned} \epsilon &\leq 112 \\ 112 &\leq 12 \\ 12 &\leq 1211 \\ 100 &\leq 11 \end{aligned}$$

Example 4.2 *The rules of the following proof tree $P : \phi$ are annotated with their labels under a depth-first left-right labelling Φ_P.*

$$\dfrac{\dfrac{\begin{matrix}[Q_1]\\ \vdots\\ \Gamma \vdash \sigma \to (\tau \to \phi)^{00}\end{matrix} \quad \begin{matrix}[Q_2]\\ \vdots\\ \Gamma' \vdash \sigma^{01}\end{matrix}}{\Gamma, \Gamma' \vdash \tau \to \phi^{0}} \to \text{Elim} \qquad \dfrac{\begin{matrix}[Q_3]\\ \vdots\\ \Lambda \vdash \sigma \to \tau^{10}\end{matrix} \quad \begin{matrix}[Q_4]\\ \vdots\\ \Lambda' \vdash \sigma^{11}\end{matrix}}{\Lambda, \Lambda' \vdash \tau^{1}} \to \text{Elim}}{\Gamma, \Gamma', \Lambda, \Lambda' \vdash \phi^{\epsilon}} \to \text{Elim}$$

Hence,

$$\Phi_P \left(\begin{matrix}[Q_3]\\ \vdots\\ \Lambda \vdash \sigma \to \tau\end{matrix} \right) = 10$$

The definitions of leftmost and head redex that follow are motivated by the definitions of leftmost and head redexes for the λ-calculus (see Section 2.1 of Chapter 2). In Section 4.1.3 below, we show that these definitions of leftmost and head redex are equivalent to leftmost and head redex in the λ-calculus under the Curry-Howard isomorphism.

Definition 4.4 *A proof redex $P : \sigma \in Q : \tau$ is leftmost in $Q : \tau$ if for all proof redex $R : \phi \in Q : \tau$, $\Phi_Q(P) \leq \Phi_Q(R)$.*

The following example illustrates the definition of leftmost proof redexes.

Example 4.3 *Let $P : \sigma$ denote the proof redex*

$$\dfrac{\dfrac{\dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\vdash \sigma \to \sigma}\,\to\text{Intro} \qquad \dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma \vdash \sigma}\,\to\text{Elim}$$

and let $Q : \tau$ denote the proof redex

$$\dfrac{\dfrac{\dfrac{}{\tau \vdash \tau}\,\text{Axiom}}{\vdash \tau \to \tau}\,\to\text{Intro} \qquad \dfrac{}{\tau \vdash \tau}\,\text{Axiom}}{\tau \vdash \tau}\,\to\text{Elim}$$

in the proof $R : (\sigma \to \tau \to \sigma) \to \sigma$ (in which nodes are labelled) as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \to \tau \to \sigma \vdash \sigma \to \tau \to \sigma^{000}}\,\text{Axiom} \qquad \begin{matrix}[P]\\ \vdots \\ \sigma \vdash \sigma^{001}\end{matrix}}{\sigma \to \tau \to \sigma, \sigma \vdash \tau \to \sigma^{00}}\,\to\text{Elim} \qquad \begin{matrix}[Q]\\ \vdots \\ \tau \vdash \tau^{01}\end{matrix}}{\sigma \to \tau \to \sigma, \sigma, \tau \vdash \sigma^{0}}\,\to\text{Elim}}{\sigma, \tau \vdash (\sigma \to \tau \to \sigma) \to \sigma^{\epsilon}}\,\to\text{Intro}$$

According to the labelling function Φ_R,

$$\begin{aligned} \Phi_R(P : \sigma) &= 001 \\ \Phi_R(Q : \tau) &= 01 \end{aligned}$$

Therefore, $\Phi_R(P : \sigma) \leq \Phi_R(Q : \tau)$, i.e., $P : \sigma$ is the leftmost proof redex.

We define head proof redexes as special cases of leftmost redexes, such that they occur on what Troelstra refers to as the *spine* of a proof tree (Chapter 4,§2 of [86]).

Definition 4.5 *A proof redex $P : \sigma \in Q : \tau$ is a head proof redex if $P : \sigma$ is leftmost in $Q : \tau$ and there exists no occurrence of an axiom $R : \phi$ such that $\Phi_Q(R : \phi) \leq \Phi_Q(P : \sigma)$.*

Example 4.4 *In the proof tree*

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma,\phi\vdash\sigma\to\tau^{000}\end{array}}{\Gamma\vdash\phi\to(\sigma\to\tau)^{00}}\to\text{Intro}\qquad\dfrac{}{\phi\vdash\phi^{01}}\text{Axiom}}{\Gamma,\phi\vdash\sigma\to\tau^{0}}\to\text{Elim}\qquad\dfrac{}{\sigma\vdash\sigma^{1}}\text{Axiom}}{\Gamma,\phi,\sigma\vdash(\sigma\to\tau)\to\tau^{\epsilon}}\to\text{Intro}$$

the proof redex

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma,\phi\vdash\sigma\to\tau^{000}\end{array}}{\Gamma\vdash\phi\to(\sigma\to\tau)^{00}}\to\text{Intro}\qquad\dfrac{}{\phi\vdash\phi^{01}}\text{Axiom}}{\Gamma,\phi\vdash\sigma\to\tau^{0}}\to\text{Elim}$$

is a head proof redex according to Definition 4.5.

Note that, in Example 4.3, the leftmost proof redex is *not* the head proof redex because of the presence of an axiom in the proof with a label 000 that precedes that of the leftmost redex according to the ordering $\leq$ over strings.

Normal form for proofs in intuitionistic logic is defined in Section 2.3 of Chapter 2. A proof is said to be in normal form if it contains no proof redexes. We also define a proof as being in *head normal form* if it contains no head proof redexes.

Example 4.5 *The following proof is in head normal form (but not normal form).*

$$\dfrac{\dfrac{\dfrac{}{\sigma\to\tau\vdash\sigma\to\tau}\text{Axiom}\qquad\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma,\phi\vdash\sigma\end{array}}{\Gamma\vdash\phi\to\sigma}\to\text{Intro}\qquad\dfrac{}{\phi\vdash\phi}\text{Axiom}}{\Gamma,\phi\vdash\sigma}\to\text{Elim}}{\Gamma,\sigma\to\tau,\phi\vdash\tau}\to\text{Elim}}{\Gamma,\sigma\to\tau\vdash\phi\to\tau}\to\text{Intro}$$

From the definitions of leftmost and head redex we derive leftmost and head normalisation of proofs. Leftmost proof normalisation consists of the successive normalisation of leftmost redexes in a proof to normal form, while head proof normalisation refers to the successive normalisation of head redexes until head normal form is reached.

As a matter of notation, we will use $\mathcal{L}$ and $\mathcal{H}$ to refer to the leftmost and head proof normalisation paths, respectively. Also, we write $\Downarrow_{\mathcal{L}} Q : \tau$ to denote the normalisation of a proof $Q : \tau$ by the leftmost normalisation path (and similarly, $\Downarrow_{\mathcal{H}} Q : \tau$ to denote head normalisation).

4.1.2 Neededness and head-neededness

For proofs in intuitionistic logic, we define a needed proof redex to be one that will eventually be contracted, whatever normalisation path is followed to reduce the proof in which it occurs to normal form. As with redexes in λ-terms, a notion of *descendants* is required in order to identify a proof with its original across normalisation steps.

Definition 4.6 *The descendants of a sub-proof $P : \sigma \in Q : \tau$, after the normalisation of $Q : \tau$ to $Q' : \tau$, are all those sub-proofs of $Q' : \tau$ that can be traced back to $P : \sigma$, identified at the level of the rule used to deduce the conclusion of $P : \sigma$.*

Example 4.6 *In the normalisation of the proof*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \to \sigma \to \tau \vdash \sigma \to \sigma \to \tau}\,\text{Axiom} \quad \dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma \to \sigma \to \tau, \sigma \vdash \sigma \to \tau}\,\to\text{Elim} \quad \dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma \to \sigma \to \tau, \sigma, \sigma \vdash \tau}\,\to\text{Elim}}{\sigma \to \sigma \to \tau, \sigma \vdash \tau}\,\text{Contraction}}{\sigma \vdash (\sigma \to \sigma \to \tau) \to \tau}\,\to\text{Intro}}{\vdash \sigma \to (\sigma \to \sigma \to \tau) \to \tau}\,\to\text{Intro} \quad \dfrac{[P]}{\Gamma \vdash \sigma}}{\Gamma \vdash (\sigma \to \sigma \to \tau) \to \tau}\,\to\text{Elim}$$

to the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma\to\sigma\to\tau\vdash\sigma\to\sigma\to\tau}\,\text{Axiom}\quad\dfrac{[P]}{\Gamma\vdash\sigma}}{\Gamma,\sigma\to\sigma\to\tau\vdash\sigma\to\tau}\,\to\text{Elim}\quad\dfrac{[P]}{\Gamma\vdash\sigma}}{\Gamma,\Gamma,\sigma\to\sigma\to\tau\vdash\tau}\,\to\text{Elim}}{\Gamma,\sigma\to\sigma\to\tau\vdash\tau}\,\text{Contraction}}{\Gamma\vdash(\sigma\to\sigma\to\tau)\to\tau}\,\to\text{Intro}$$

two descendants of the sub-proof $P : \sigma$ are created. (Note that possibly multiple applications of Contraction are required in order to merge the different occurrences of the base Γ.)

As a small digression, we note that when a proof is substituted for a hypothesis in a proof normalisation step, the number of descendants of the proof resulting from the substitution depends upon the size of the hypothesis's parcel (parcels of hypotheses are discussed in Section 2.3 of Chapter 2). The correspondence between the number of elements in a parcel of a hypothesis and the number of free occurrences of a variable in a typed λ-term is implied by the Curry-Howard isomorphism.

As mentioned in Section 2.3 of Chapter 2, the Axiom rule introduces a hypothesis parcel of size 1, while Contraction merges parcels into larger ones by multi-set union and Weakening introduces empty parcels. Sub-proofs are erased or discarded when substituted for a hypothesis whose parcel is empty. On the other hand, the result of Contraction is to increase the number of descendants of a sub-proof that may be created during proof normalisation.

We define neededness for intuitionistic logic using descendants as follows:

Definition 4.7 *Let $P : \sigma$ be a proof redex in proof $Q : \tau$. Then $P : \sigma$ is needed iff every normalisation of $Q : \tau$ to normal-form normalises a descendant of $P : \sigma$.*

Head-neededness for proofs is similarly defined as follows.

Definition 4.8 *Let $P : \sigma \in Q : \tau$ be a proof redex. Then $P : \sigma$ is head-needed iff a descendant of $P : \sigma$ is normalised in every normalisation sequence taking $Q : \tau$ to head normal form.*

As is the case with redexes in the λ-calculus, proof redexes contracted on the leftmost normalisation path are needed redexes, and proof redexes contracted on the head normalisation path are head-needed. In the λ-calculus, this arises because the position of leftmost and head redexes means that their contracted forms cannot be erased later in the reduction path. The same observation applies to leftmost and head proof redexes in intuitionistic logic.

We also define a notion of *erasure* for proof redexes.

Definition 4.9 *Let $P : \sigma$ be a proof redex in $Q : \tau$, and $\mathcal{S}$ a normalisation path of $Q : \tau$ to $Q' : \tau$, such that no descendant of $P : \sigma$ is contracted on $\mathcal{S}$ and $P : \sigma$ does not occur in $Q' : \tau$. Then $P : \sigma$ is* erased *in $\mathcal{S}$.*

If there exists a normalisation path $\mathcal{S}$ such that $P : \sigma$ in $Q : \tau$ is erased, then $P : \sigma$ is *erasable.* It follows from Definition 4.7 that no needed proof redex is erasable, and no erasable proof redex is needed.

4.1.3 Relationship with neededness in the λ-calculus

The definitions of neededness and head-neededness for intuitionistic logic have been motivated by neededness and head-neededness for the (typed) λ-calculus, and here we show that they are in fact equivalent under the Curry-Howard isomorphism.

Troelstra (Section 4.1.6 of [86]) has shown that the equivalence between a proof redex and a λ-term holds across the contraction of the proof redex and the λ-term, shown as follows (using the translation Ψ discussed in Section 2.4 of Chapter 2):

$$\dfrac{\dfrac{\begin{matrix}[Q]\\ \vdots\\ \Gamma,\sigma\vdash\tau\end{matrix}}{\Gamma\vdash\sigma\to\tau}\to\text{Intro}\qquad\begin{matrix}[P]\\ \vdots\\ \Lambda\vdash\sigma\end{matrix}}{\Gamma,\Lambda\vdash\tau}\to\text{Elim}$$

translates to

$$(\lambda x:\sigma.\Psi\left(\dfrac{\begin{matrix}[Q]\\ \vdots\\ \Gamma,\sigma\vdash\tau\end{matrix}}{\Gamma\vdash\sigma\to\tau}\to\text{Intro}\right))\Psi\left(\begin{matrix}[P]\\ \vdots\\ \Lambda\vdash\sigma\end{matrix}\right)$$

and

$$\begin{matrix}[Q']\\ \vdots\\ \Gamma,\Lambda\vdash\tau\end{matrix}$$

(where $Q':\tau\equiv Q:\tau[P/\sigma]$ as defined in Section 2.3 of Chapter 2) is translated to

$$\Psi\left(\dfrac{\begin{matrix}[Q]\\ \vdots\end{matrix}}{\Gamma\vdash\sigma\to\tau}\to\text{Intro}\right))[\Psi\left(\begin{matrix}[P]\\ \vdots\\ \Lambda\vdash\sigma\end{matrix}\right)/x]$$

The persistence of the equivalence between a proof redex and its corresponding λ-term across contraction also holds between a proof and a term in which these redexes occur.

We define equivalence of reduction sequences of proofs and typed λ-terms as follows.

Definition 4.10 *For a proof $Q:\tau$ and its corresponding λ-term $\Psi(Q:\tau)$, a proof normalisation path $\mathcal{R}$ taking $Q:\tau$ to normal form and a reduction path $\mathcal{S}$ taking $\Psi(Q:\tau)$ to normal form are equivalent, denoted $\mathcal{R}\equiv_{CHI}\mathcal{S}$, iff at each step they contract equivalent sub-parts of $Q:\tau$ and $\Psi(Q:\tau)$.*

To help prove the equivalence between neededness and head-neededness in intuitionistic logic and the λ-calculus, we re-define the translation Ψ from intuitionistic logic proofs to typed λ-terms defined by Troelstra (see Section 2.4 of Chapter 2). The new translation function Ψ_l makes use of the labelling function mentioned in Section 4.1.1 above, in order to translate a proof P into a *labelled* typed λ-term. (Details of the translations of the Contraction and Weakening rules have been omitted, since they are the same as for Ψ.)

Definition 4.11 *Ψ_l is defined for a proof $Q : \tau$ as follows:*

1. *To a proof consisting only of an Axiom rule, we assign a new typed variable, labelled by the labelling function Φ_Q for the particular occurrence of the Axiom rule, i.e.,*

$$\Psi_l\left(\frac{}{\sigma \vdash \sigma}\text{Axiom}\right) = x_\sigma^{\Phi_Q(\sigma\vdash\sigma)}$$

2. *An abstraction is translated inductively:*

$$\Psi_l\left(\cfrac{\begin{matrix}\vdots\\ \Gamma,\sigma\vdash\tau\end{matrix}}{\Gamma\vdash\sigma\to\tau}\to\text{Intro}\right) = \left(\lambda x_\sigma.\Psi_l\left(\begin{matrix}\vdots\\ \Gamma,\sigma\vdash\tau\end{matrix}\right)\right)^{\Phi_Q(\Gamma\vdash\sigma\to\tau)}$$

3. *Applications are also translated inductively:*

$$\Psi_l\left(\cfrac{\begin{matrix}\vdots\\ \Gamma\vdash\sigma\to\tau\end{matrix}\quad\begin{matrix}\vdots\\ \Delta\vdash\sigma\end{matrix}}{\Gamma,\Delta\vdash\tau}\to\text{Elim}\right) = \left(\Psi_l\left(\begin{matrix}\vdots\\ \Gamma\vdash\sigma\to\tau\end{matrix}\right)\ \Psi_l\left(\begin{matrix}\vdots\\ \Delta\vdash\sigma\end{matrix}\right)\right)^{\Phi_Q(\Gamma,\Delta\vdash\tau)}$$

Example 4.7 *The following are examples of the translation of proofs into labelled typed λ-terms using Ψ_l:*

1.

$$\Psi_l\left(\cfrac{\cfrac{}{\sigma\vdash\sigma}\text{Axiom}}{\vdash\sigma\to\sigma}\to\text{Intro}\right) = (\lambda x_\sigma.x^0)^\epsilon_{\sigma\to\sigma}$$

2.

$$\Psi_l\left(\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma\vdash\sigma}\text{Axiom}}{\sigma,\tau\vdash\sigma}\text{Weakening}}{\sigma\vdash\tau\to\sigma}\to\text{Intro}}{\vdash\sigma\to\tau\to\sigma}\to\text{Intro}\qquad\dfrac{\dfrac{}{\sigma\to\sigma\vdash\sigma\to\sigma}\text{Axiom}\qquad\dfrac{}{\sigma\vdash\sigma}\text{Axiom}}{\sigma\to\sigma\vdash\sigma}\to\text{Elim}}{\sigma\to\sigma,\sigma\vdash\tau\to\sigma}\to\text{Elim}\right)$$

$$= ((\lambda x_\sigma.(\lambda y_\tau.x_\sigma^{0000})^{00}_{\tau\to\sigma})^{0}_{\sigma\to\tau\to\sigma}(I^{10}_{\sigma\to\sigma}y^{11}_\sigma)^{1}_\sigma)^{\epsilon}_{\tau\to\sigma}$$

We also define a function over the translated λ-terms that returns the label attached to them.

Definition 4.12 *For any proof $P:\sigma$ in proof $Q:\tau$, the label associated with the translation of $P:\sigma$ is given by*

$$label(\Psi_l(P:\sigma)) = \Phi_Q(P:\sigma)$$

We also require the following lemma that relates the relative position of λ-redexes within a λ-term to the ordering over their associated labels.

Lemma 4.1 *Let N and P be redexes in a labelled typed λ-term M, where labels are assigned to sub-terms according to the labelling function Φ for an equivalent proof. If N is to the left of P in M, then $label(N) \leq label(P)$.*

Proof Follows directly from the fact that pre-order traversal is used for both the labelling of proofs and for the definition of N to the left of P in M.

□

Lemma 4.2 *$P:\sigma$ is a leftmost proof redex in a proof $Q:\tau \Leftrightarrow \Psi_l(P:\sigma)$ is a leftmost redex in $\Psi_l(Q:\tau)$.*

Proof ($\Rightarrow$) Let M be any redex in $\Psi_l(Q : \tau)$, i.e., $M \equiv \Psi_l(R : \phi)$ where $R : \phi$ is a proof redex in $Q : \tau$. Assume that M is to the left of $\Psi_l(P : \sigma)$. By Lemma 4.1,

$$label(M) \leq label(\Psi_l(P : \sigma))$$

which, by definition of *label*, implies that

$$\Phi_Q(R : \phi) \leq \Phi_Q(P : \sigma)$$

which contradicts Definition 4.4 of leftmost proof redex. Therefore, $\Psi_l(P : \sigma)$ is the leftmost redex in $\Psi_l(Q : \tau)$.

($\Leftarrow$) If $\Psi_l(P : \sigma)$ is leftmost in $\Psi_l(Q : \tau)$, then there exists no redex M in $\Psi_l(Q : \tau)$ such that M is to the left of $\Psi_l(P : \sigma)$. Therefore, by Lemma 4.1, there exists no M such that

$$label(M) \leq label(\Psi_l(P : \sigma))$$

Let $R : \phi$ be the proof translated to M by Ψ_l. Then by definition of the function *label*, there is no $R : \phi$ such that

$$\Phi_Q(R : \phi) \leq \Phi_Q(P : \sigma)$$

By Definition 4.4, therefore, $P : \sigma$ is the leftmost proof redex.

□

Lemma 4.3 *$P : \sigma$ is a head proof redex in a proof $Q : \tau \Leftrightarrow \Psi_l(P : \sigma)$ is a head redex in $\Psi_l(Q : \tau)$.*

Proof The proof follows along the same lines as the proof of Lemma 4.2.

□

Lemma 4.4 *Assume a proof $Q : \tau$. Let $\mathcal{L}_{IL}$ denote the leftmost normalisation path taking $Q : \tau$ to normal form, and let $\mathcal{L}_\lambda$ denote the leftmost reduction path taking $\Psi_l(Q : \tau)$ to normal form. Then*

$$\mathcal{L}_{IL} \equiv_{CHI} \mathcal{L}_\lambda$$

Similarly for head proof normalisation $\mathcal{H}_{IL}$ and the head reduction path $\mathcal{H}_\lambda$.

Proof The proof follows from Lemma 4.2, which states that leftmost redexes in a proof and a λ-term are equivalent (under the translation Ψ_l), and from the persistence of the equivalence between a proof and a λ-term across contraction steps. (Substitute Lemma 4.3 for Lemma 4.2 to prove the same for head proof normalisation and head reduction paths.)

$\square$

The main result of this section is given in Theorem 4.1, and states that our definitions of neededness and head-neededness, in intuitionistic logic and the typed λ-calculus, are equivalent under the Curry-Howard isomorphism.

Theorem 4.1 *Let $P : \sigma$ be a proof redex in $Q : \tau$. Then $P : \sigma$ is needed (head-needed) in $Q : \tau \Leftrightarrow \Psi_l(P : \sigma)$ is needed (head-needed) in $\Psi_l(Q : \tau)$.*

Proof Directly from Lemma 4.4 and from the definitions of neededness (head-neededness) in intuitionistic logic and λ-calculus.

$\square$

4.2 Simple resource use in intuitionistic logic

In this section, we show how simple resource use, covering strictness, absence, and non-strictness, can be defined in intuitionistic logic.

In intuitionistic logic, resource use is taken to be concerned with the use made of the resources of deductions, in other words about hypotheses. For example, given a proof tree

$$\begin{array}{c}[Q]\\ \vdots \\ \Gamma, \sigma \vdash \tau\end{array}$$

the hypothesis σ is a resource, and a description of its resource use is concerned with the dependency of the proof upon the hypothesis. In this section, we define this dependency in terms of the neededness and head-neededness of the proofs of σ substituted for it in the proof tree $Q : \tau$.

4.2.1 Strictness, absence, and non-strictness in intuitionistic logic

We define the resource use domain S containing strictness, absence, and non-strictness for intuitionistic logic following the approach taken in Section 3.2 of Chapter 3. This domain is as previously defined in Figure 3.1, with the same ordering $\sqsubseteq_{\mathsf{S}}$ over resource use domain S and with operators $+$ and $\times$ as defined in Table 3.1 of that section.

Our intuition of what strictness means for intuitionistic logic is that any hypothesis σ used in the deduction of a proposition τ should be *necessary* in some sense to the deduction.

To be more precise about what necessary means, we first consider the circumstances under which a hypothesis might be *unnecessary.* A hypothesis can be thought of as unnecessary if the deduction can be made without it; in particular, if a proof of the hypothesis is substituted for it in the deduction, which is not then contracted in a normalisation of the deduction, then the hypothesis is obviously unnecessary. The converse of this idea is that a necessary, or *strict*, hypothesis is one such that a proof substituted for the hypothesis in the deduction will be contracted during normalisation.

We determine strictness by the head-neededness of any proof substituted for a hypothesis in a deduction, i.e., if all proofs of a hypothesis σ substituted for it in a deduction are head-needed in normalisation, then the deduction is strict in the hypothesis. Neededness is insufficient to define strictness, as the following example illustrates.

Example 4.8 *Let $Q : \sigma$ be the proof*

$$\dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\text{Axiom} \qquad \dfrac{}{\sigma \vdash \sigma}\text{Axiom}}{\sigma \to \sigma, \sigma \vdash \sigma}\to\text{Elim}$$

The result of substituting a proof redex $P : \sigma$ for the hypothesis σ in $Q : \sigma$ is the proof

$Q' : \sigma$

$$\dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\,\text{Axiom} \qquad \begin{array}{c}[P]\\ \vdots\\ \Lambda \vdash \sigma\end{array}}{\sigma \to \sigma, \Lambda \vdash \sigma}\,\to \text{Elim}$$

By Definition 4.7, $P : \sigma$ is needed in $Q' : \sigma$, but not head-needed according to Definition 4.8. However, $P : \sigma$ may be subsequently erased depending on the proof substituted for hypothesis $\sigma \to \sigma$. In particular, $P : \sigma$ will be erased if the following proof

$$\dfrac{\dfrac{\dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma, \sigma \vdash \sigma}\,\text{Weakening}}{\sigma \vdash \sigma \to \sigma}\,\to \text{Intro}$$

of $\sigma \to \sigma$ (where the discharged hypothesis is the one introduced by Weakening) is substituted for the hypothesis in $Q : \tau$.

As the above example demonstrates, although a proof substituted for a hypothesis may be needed, the hypothesis may yet prove to be unnecessary to the deduction as a result of proofs substituted for other hypotheses. This problem is avoided if head-neededness is adopted as the basis for the definition of strictness. By Definition 4.5, no axiom lies to the left of a head proof redex in the deduction, and therefore, no later substitution of proofs for hypotheses can result in its erasure. Therefore, we define a deduction to be strict in a particular hypothesis if all proofs substituted for that hypothesis are head-needed in the deduction.

A hypothesis in a deduction has a resource use of *absence* if it is unnecessary to the deduction, as explained above. In this case, the basis for the definition of absence is that any proof substituted for the hypothesis is *not needed* in the deduction. That a proof substituted for the hypothesis is not head-needed is insufficient to establish absence, by a line of argument similar to that demonstrating that head-neededness is required to define strictness. If the proof is not head-needed but needed, then later substitutions of proofs may result in the proof being head-needed after all.

Finally, the resource use describing *non-strictness* is simply the resource use of uncertainty. If a deduction is non-strict in a hypothesis, then we have no information about how it will be used.

The following definition formalises this discussion about the semantics of resource use.

Definition 4.13 *Given the proof*

$$\begin{array}{c}[Q]\\ \vdots\\ \Gamma \vdash \tau\end{array}$$

then for any $\sigma \in \Gamma$, write $\sigma^u \in Q : \tau$ where u is defined as follows:

(i)	$u \equiv \mathrm{S} \Rightarrow$	$\forall P : \sigma \,.\, P : \sigma$ *head-needed in* $Q : \tau[P/\sigma]$
(ii)	$u \equiv \mathrm{A} \Rightarrow$	$\forall P : \sigma \,.\, P : \sigma$ *is not needed in* $Q : \tau[P/\sigma]$
(iii)	$u \equiv \mathrm{N} \Rightarrow$	$\forall P : \sigma \,.\, P : \sigma$ *is or is not needed in* $Q : \tau[P/\sigma]$

Sequents are now written with resource use annotations on hypotheses to indicate their resource use with respect to the proof in which they appear. For example,

$$\begin{array}{c}[Q]\\ \vdots\\ \Gamma, \sigma^{\mathrm{S}} \vdash \tau\end{array}$$

indicates that the hypothesis σ is used strictly in the deduction of τ, i.e., that any proof $P : \sigma$ substituted for occurrences of the hypothesis σ in Q will be head-needed in the normalisation of the proof $Q : \tau[P/\sigma]$ resulting from the substitution.

We also allow resource use annotations to be discharged along with the hypothesis parcel they annotate on application of the $\rightarrow$ Intro rule. For example,

$$\begin{array}{c}[Q]\\ \vdots\\ \dfrac{\Gamma, \sigma^{\mathrm{S}} \vdash \tau}{\Gamma \vdash \sigma \xrightarrow{\mathrm{S}} \tau} \rightarrow \text{Intro}\end{array}$$

introduces an implicative proposition which is strict in its first argument. Hence, in

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \sigma \xrightarrow{\mathsf{S}} \tau}\to \text{Intro} \qquad \begin{array}{c}[P]\\ \vdots \\ \Lambda \vdash \sigma\end{array}}{\Gamma, \Lambda^{\times \mathsf{S}} \vdash \tau}\to \text{Elim}$$

the sub-proof $P : \sigma$ will be head-needed in the normalisation of the proof. Note that the strict use of the sub-proof $P : \sigma$ also affects the open hypotheses in the base Λ. In the conclusion of the $\to$ Elim Rule, we write $\Lambda^{\times \mathsf{S}}$ to denote the fact that the resource use of the hypotheses in Λ must now take account of the strict use of the proof in which they appear. (This notation and the $\times$ operator are defined and explained in more detail in Section 4.2.2 below.)

4.2.2 Incorporating resource use into intuitionistic logic

We would like to incorporate resource use into intuitionistic logic such that the resource use made of hypotheses, and discharged with hypotheses to annotate implications, can be inferred within the existing rules. In this section, we present an extended system of intuitionistic logic in which resource use is represented and inferred for hypotheses. Also, we show that this system is sound, in the sense that if a hypothesis is annotated with resource use u in the conclusion of a proof, then any proof of the hypothesis substituted for it will be head-needed and needed according to the definition of u in Definition 4.13.

The extended system of intuitionistic logic with simple resource use, IL_{S}, is presented in Figure 4.2. To denote deduction in a sequent in IL_{S}, we use the symbol $\vdash_{\mathsf{S}}$, e.g.,

$$\Gamma \vdash_{\mathsf{S}} \sigma$$

Where necessary, to establish the context as being IL_{S}, we will also annotate bases and propositions, for example, $\Gamma_{\mathsf{S}}, \tau_{\mathsf{S}}$.

Definition 4.14 *The language of propositions in* IL_{S} *is given by the following grammar:*

$$\begin{array}{rcl} \sigma & ::= & \alpha \mid \sigma_1 \xrightarrow{u} \sigma_2 \\ u & ::= & \mathsf{S} \mid \mathsf{A} \mid \mathsf{N} \end{array}$$

Axiom

$$\frac{}{\sigma^{\mathrm{S}} \vdash_{\mathrm{S}} \sigma}\ \text{Axiom}$$

Structural Rules

$$\frac{\Gamma, \sigma^{i}, \sigma^{j} \vdash_{\mathrm{S}} \tau}{\Gamma, \sigma^{i+j} \vdash_{\mathrm{S}} \tau}\ \text{Contraction} \qquad \frac{\Gamma \vdash_{\mathrm{S}} \tau}{\Gamma, \sigma^{\mathrm{A}} \vdash_{\mathrm{S}} \tau}\ \text{Weakening}$$

Logical Rules

$$\frac{\Gamma, \sigma^{i} \vdash_{\mathrm{S}} \tau}{\Gamma \vdash_{\mathrm{S}} \sigma \xrightarrow{i} \tau} \rightarrow \text{Intro} \qquad \frac{\Gamma \vdash_{\mathrm{S}} \sigma \xrightarrow{i} \tau \quad \Lambda \vdash_{\mathrm{S}} \sigma' \quad \sigma' \subseteq_{\mathrm{S}} \sigma}{\Gamma, \Lambda^{\times i} \vdash_{\mathrm{S}} \tau} \rightarrow \text{Elim}$$

Figure 4.2: IL_{S}: intuitionistic logic and resource use

The language of propositions should also include resource use expressions, such as A + S, as annotations to the implication, but since these expressions evaluate to atomic values (such as S for this example), we do not include them.

The rules of IL_{S} are essentially the same as those for intuitionistic logic, except that values of S annotate hypotheses and the implicative connective $\rightarrow$. Also, we add an extra condition to the $\rightarrow$ Elim rule, to reflect, for example, the fact that a proof of the strict implication $\sigma \xrightarrow{\mathrm{S}} \sigma$ can be substituted for the non-strict hypothesis $\sigma \xrightarrow{\mathrm{N}} \sigma$. The condition to $\rightarrow$ Elim makes use of an ordering $\subseteq_{\mathrm{S}}$ over propositions which is induced by the ordering $\sqsubseteq_{\mathrm{S}}$ over values in S.

Definition 4.15 *A partial ordering* $\subseteq_{\mathrm{S}}$ *on propositions in* IL_{S} *is defined as follows:*

$$\alpha \subseteq_{\mathrm{S}} \alpha$$
$$\sigma \xrightarrow{i} \tau \subseteq_{\mathrm{S}} \sigma' \xrightarrow{j} \tau' \quad \Leftrightarrow \quad \sigma' \subseteq_{\mathrm{S}} \sigma, \tau \subseteq_{\mathrm{S}} \tau', i \sqsubseteq_{\mathrm{S}} j$$

(where α denotes an atomic proposition).

Example 4.9 *The following are examples of the ordering between propositions induced by*

$\subseteq_S$:

$$\begin{array}{lcl} \sigma\xrightarrow{\mathbf{S}}\tau & \subseteq_S & \sigma\xrightarrow{\mathbf{N}}\tau \\ (\sigma\xrightarrow{\mathbf{N}}\tau)\xrightarrow{\mathbf{N}}\phi & \subseteq_S & (\sigma\xrightarrow{\mathbf{S}}\tau)\xrightarrow{\mathbf{N}}\phi \end{array}$$

One important feature of the ordering $\subseteq_S$ is its anti-monotonicity in the first argument to the implicative connective. For example, see the second instance given in Example 4.9 above).

Without this *coercion*, we would be unable to produce, for example, a proof tree with a proof of $\sigma\xrightarrow{\mathbf{S}}\tau$ as the minor premise and a proof of $(\sigma\xrightarrow{\mathbf{N}}\tau)\xrightarrow{\mathbf{A}}\phi$ as the major premise in an application of $\rightarrow$ Elim. But with the current $\rightarrow$ Elim rule, we can have

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma\vdash_S(\sigma\xrightarrow{\mathbf{N}}\tau)\xrightarrow{\mathbf{A}}\phi\end{array} \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma,\sigma^{\mathbf{S}}\vdash_S\tau\end{array}}{\Gamma\vdash_S\sigma\xrightarrow{\mathbf{S}}\tau}\rightarrow\text{Intro} \quad \sigma\xrightarrow{\mathbf{S}}\tau\subseteq_S\sigma\xrightarrow{\mathbf{N}}\tau}{\Gamma,\Lambda^{\times\mathbf{A}}\vdash_S\phi}\rightarrow\text{Elim}$$

Note that the definitions of Contraction and $\rightarrow$ Elim in IL$_S$ require operators $+$ and $\times$ over resource use values in their conclusions. These operators over resource use domain S are as defined in Table 3.1 of Chapter 3. The notation $\Lambda^{\times i}$ used in the conclusion of the $\rightarrow$ Elim rule is defined as

$$\{\sigma^{j\times i}|\sigma^j\in\Lambda\}$$

Example 4.10 *The following proof tree is an example of a deduction in* IL$_S$ *(for reasons of presentation, the side-condition to the* $\rightarrow$ Elim *has been omitted in this example):*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma^{\mathbf{S}}\vdash_S\sigma}\text{Axiom}}{\sigma^{\mathbf{S}},\tau^{\mathbf{A}}\vdash_S\sigma}\text{Weakening}}{\sigma^{\mathbf{S}}\vdash_S\tau\xrightarrow{\mathbf{A}}\sigma}\rightarrow\text{Intro}}{\vdash_S\sigma\xrightarrow{\mathbf{S}}\tau\xrightarrow{\mathbf{A}}\sigma}\rightarrow\text{Intro} \quad \dfrac{\dfrac{}{(\sigma\xrightarrow{\mathbf{S}}\sigma)^{\mathbf{S}}\vdash_S\sigma\xrightarrow{\mathbf{S}}\sigma}\text{Axiom} \quad \dfrac{}{\sigma^{\mathbf{S}}\vdash_S\sigma}\text{Axiom}}{(\sigma\xrightarrow{\mathbf{S}}\sigma)^{\mathbf{S}},\sigma^{\mathbf{S}\times\mathbf{S}}\vdash_S\sigma}\rightarrow\text{Elim}}{(\sigma\xrightarrow{\mathbf{S}}\sigma)^{\mathbf{S}\times\mathbf{S}},\sigma^{\mathbf{S}\times\mathbf{S}\times\mathbf{S}}\vdash_S\tau\xrightarrow{\mathbf{A}}\sigma}\rightarrow\text{Elim} \quad \dfrac{}{\tau^{\mathbf{S}}\vdash_S\tau}\text{Axiom}}{(\sigma\xrightarrow{\mathbf{S}}\sigma)^{\mathbf{S}\times\mathbf{S}},\sigma^{\mathbf{S}\times\mathbf{S}\times\mathbf{S}},\tau^{\mathbf{S}\times\mathbf{A}}\vdash_S\sigma}\rightarrow\text{Elim}$$

Note that the conclusion of the proof simplifies to

$$(\sigma \xrightarrow{\mathrm{S}} \sigma)^{\mathrm{S}}, \sigma^{\mathrm{S}}, \tau^{\mathrm{A}} \vdash_{\mathrm{S}} \sigma$$

by the definitions of $+$ *and* $\times$.

4.2.3 Soundness of resource use inference

We would like to show that if an open hypothesis in a proof $Q : \tau$ has resource use u according to Definition 4.13, then u is the resource use annotation attached to the hypothesis by the rules of IL_{S}. However, this is not the case, as the following example shows. Let $Q : \tau$ be

$$\dfrac{\dfrac{}{(\sigma \xrightarrow{\mathrm{S}} \tau)^{\mathrm{S}} \vdash \sigma \xrightarrow{\mathrm{S}} \tau}\,\text{Axiom} \qquad \dfrac{}{\sigma^{\mathrm{S}} \vdash \sigma}\,\text{Axiom} \qquad \sigma \subseteq \sigma}{(\sigma \xrightarrow{\mathrm{S}} \tau)^{\mathrm{S}}, \sigma^{\mathrm{S}\times\mathrm{S}} \vdash \tau}\to\text{Elim}$$

The open hypothesis σ in the conclusion of this proof has resource use $\mathrm{S} \times \mathrm{S} = \mathrm{S}$, yet it is obvious that no proof redex replacing the axiom introducing σ in this proof will be head-needed. The problem is clearly that the axiom $\sigma \xrightarrow{\mathrm{S}} \tau$ contains more precise information about the resource use to be made of any argument to the implication than is assumed by the definition of resource use made in Definition 4.13.

To resolve this mis-match, we introduce a variation on IL_{S} in which the resource use contained in implicative propositions introduced as axioms is non-specific, i.e., is N. This system is called $\mathrm{IL}_{\mathrm{S}f}$ (where f stands for *first-order*) and is given in Figure 4.3.

In $\mathrm{IL}_{\mathrm{S}f}$, the propositions introduced by Axiom are treated by the operator $\downarrow$, defined over propositions as follows:

$$\begin{array}{lcl} \alpha\downarrow & = & \alpha \\ (\sigma \xrightarrow{j} \tau)\downarrow & = & \sigma\downarrow \xrightarrow{\mathrm{N}} \tau\downarrow \end{array}$$

The effect is to ensure that all implications introduced as axioms now contain only the resource use value N.

Axiom

$$\frac{}{\sigma\downarrow^{\mathsf{S}}\vdash_{\mathsf{S}}\sigma\downarrow}\ \text{Axiom}$$

Structural Rules

$$\frac{\Gamma,\sigma^{i},\sigma^{j}\vdash_{\mathsf{S}}\tau}{\Gamma,\sigma^{i+j}\vdash_{\mathsf{S}}\tau}\ \text{Contraction}\qquad\frac{\Gamma\vdash_{\mathsf{S}}\tau}{\Gamma,\sigma^{\mathsf{A}}\vdash_{\mathsf{S}}\tau}\ \text{Weakening}$$

Logical Rules

$$\frac{\Gamma,\sigma^{i}\vdash_{\mathsf{S}}\tau}{\Gamma\vdash_{\mathsf{S}}\sigma\xrightarrow{i}\tau}\to\text{Intro}\qquad\frac{\Gamma\vdash_{\mathsf{S}}\sigma\xrightarrow{i}\tau\quad\Lambda\vdash_{\mathsf{S}}\sigma'\quad\sigma'\subseteq\sigma}{\Gamma,\Lambda^{\times i}\vdash_{\mathsf{S}}\tau}\to\text{Elim}$$

Figure 4.3: System $\mathrm{IL}_{\mathsf{S}_f}$ for first-order resource use in intuitionistic logic

A definition of soundness of inference of resource use might be as follows: let τ be a provable proposition in $\mathrm{IL}_{\mathsf{S}_f}$ by proof Q and set of open hypotheses Γ, i.e.,

$$\begin{array}{c}[Q]\\ \vdots\\ \Gamma\vdash\tau\end{array}$$

Then for any σ, such that $\sigma^{v}\in\Gamma$: let u be the resource use that σ receives according to Definition 4.13, based on the head-neededness and neededness of any proof redex substituted for the axiom introducing σ (if any) in $Q:\tau$. Then soundness would state that the resource use inferred for σ by the rules of $\mathrm{IL}_{\mathsf{S}_f}$, i.e,, v, is equivalent to the resource use u for σ according to Definition 4.13.

Unfortunately, it is relatively simple to construct an example to show that this is not the case in $\mathrm{IL}_{\mathsf{S}_f}$. In fact, instead of an equivalence we have an approximation. We take the following proof tree as an example. (Note that for presentation purposes, rule labels have been omitted, as well as application of $\downarrow$ in the Axiom rules, although the effect of this operator is shown):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\sigma \xrightarrow{N} \sigma^{S} \vdash \sigma \xrightarrow{N} \sigma} \quad \overline{\sigma^{S} \vdash \sigma} \quad \sigma \subseteq_{S} \sigma}{\sigma \xrightarrow{N} \sigma^{S}, \sigma^{N} \vdash \sigma}}{\sigma \xrightarrow{N} \sigma^{S} \vdash \sigma \xrightarrow{N} \sigma}}{\vdash (\sigma \xrightarrow{N} \sigma) \xrightarrow{S} \sigma \xrightarrow{N} \sigma} \qquad \dfrac{\overline{\sigma^{S} \vdash \sigma}}{\vdash \sigma \xrightarrow{S} \sigma} \quad \sigma \xrightarrow{S} \sigma \subseteq_{S} \sigma \xrightarrow{N} \sigma}{\vdash \sigma \xrightarrow{N} \sigma} \qquad \overline{\sigma^{S} \vdash \sigma} \quad \sigma \subseteq_{S} \sigma}{\sigma^{N} \vdash \sigma}$$

which normalises by contraction of the head proof redex to

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\sigma^{S} \vdash \sigma}}{\vdash \sigma \xrightarrow{S} \sigma} \quad \overline{\sigma^{S} \vdash \sigma} \quad \sigma \subseteq_{S} \sigma}{\sigma^{S} \vdash \sigma}}{\vdash \sigma \xrightarrow{S} \sigma} \qquad \overline{\sigma^{S} \vdash \sigma} \quad \sigma \subseteq_{S} \sigma}{\sigma^{S} \vdash \sigma}$$

which, in turn, normalises by contraction of the head proof redex to

$$\dfrac{\dfrac{\overline{\sigma^{S} \vdash \sigma}}{\vdash \sigma \xrightarrow{S} \sigma} \quad \overline{\sigma^{S} \vdash \sigma} \quad \sigma \subseteq_{S} \sigma}{\sigma^{S} \vdash \sigma}$$

which reduces to the head normal form (and normal form)

$$\overline{\sigma^{S} \vdash \sigma}$$

Any proof redex substituted for the open hypothesis σ in the conclusion of the original proof will therefore become the head proof redex and therefore head-needed in the normalisation to head normal form. But IL_{S_f} is unable to deduce this in the original proof tree, deriving instead the less informative resource use N. The reason for this is the (lack of) interaction between the coercion premise between propositions for $\rightarrow$ Elim and the derivation of resource use in the conclusion of the rule. The coercion premise allows proofs of propositions that are more specific in their resource use information to be used as the minor premise to $\rightarrow$ Elim than is specified by the argument component of the implication which is the major premise. However, it is the resource use behaviour of the argument component of the implication that prevails in the conclusion, not that of the minor premise. As a result, given a proof

$$\begin{array}{c}[Q]\\ \vdots \\ \Gamma \vdash \tau\end{array}$$

and its normalised form

$$\begin{array}{c}[Q']\\ \vdots \\ \Gamma' \vdash \tau\end{array}$$

then, for all $\sigma^u \in \Gamma'$, $\sigma^v \in \Gamma$, we have $u \subseteq_S v$.

Consequently, we define soundness as an approximation, rather than an equivalence, for IL_{S_f}. Theorem 4.2 below demonstrates the soundness of the inference of the resource use of hypotheses in IL_{S_f}. We write $Q : \tau \models \sigma^u$ to mean that $\sigma^u \in Q : \tau$ according to Definition 4.13 above.

Theorem 4.2 *Resource use soundness of* IL_{S_f}. *If* $\Gamma \vdash_S \tau$ *is provable in* IL_{S_f} *by proof* Q, *then*

$$\forall \sigma^v \in \Gamma . Q : \tau \models \sigma^u \Rightarrow u \sqsubseteq_S v$$

Proof The proof is by structural induction over the height of proofs in IL_{S_f}.

Case 1: Let $Q : \tau$ be

$$\frac{}{\sigma \downarrow^S \vdash_S \sigma \downarrow}\text{Axiom}$$

For any proof redex $P : \sigma$, $P : \sigma$ occurs as the head redex in $Q : \sigma[P/\sigma]$, and hence is head-needed. Therefore, $Q : \sigma \models \sigma^{\mathbf{S}}$. By the reflexive property of $\sqsubseteq$, we have $\mathbf{S} \sqsubseteq_{\mathbf{S}} \mathbf{S}$.

Case 2: Let $Q : \tau$ be

$$\frac{\begin{array}{c}[Q'] \\ \vdots \\ \Gamma \vdash_{\mathbf{S}} \tau\end{array}}{\Gamma, \sigma^{\mathbf{A}} \vdash_{\mathbf{S}} \tau}\ \text{Weakening}$$

By induction,

$$\forall \phi^i \in \Gamma . Q' : \tau \models \phi^j \Rightarrow j \sqsubseteq_{\mathbf{S}} i$$

For any proof redex $P : \sigma$, by substitution $Q : \tau[P/\sigma] \equiv Q' : \tau[P/\sigma]$. Since σ has been introduced by Weakening, it is not an open hypothesis in Γ (other occurrences of $\sigma \in \Gamma$ are distinct). Therefore,

$$Q' : \tau[P/\sigma] \equiv Q' : \tau$$

i.e., $P : \sigma$ is erased in $Q : \tau$. So we have $Q : \tau \models \sigma^{\mathbf{A}}$. Again, by the reflexive nature of $\sqsubseteq$, we have $\mathbf{A} \sqsubseteq_{\mathbf{S}} \mathbf{A}$. Also, any proof redex $R : \phi$ to be substituted in $Q : \tau$ for an open hypothesis $\phi^i \in \Gamma$, will be substituted in $Q' : \tau$, and hence we have that

$$\forall \phi^i \in \Gamma . Q : \tau \models \phi^j \Rightarrow j \sqsubseteq_{\mathbf{S}} i$$

Case 3: Let $Q : \tau$ be

$$\frac{\begin{array}{c}[Q'] \\ \vdots \\ \Gamma, \sigma^i, \sigma^j \vdash_{\mathbf{S}} \tau\end{array}}{\Gamma, \sigma^{i+j} \vdash_{\mathbf{S}} \tau}\ \text{Contraction}$$

By induction, $Q' : \tau \models \sigma^{i'} \Rightarrow i' \sqsubseteq_{\mathbf{S}} i$ and $Q' : \tau \models \sigma^{j'} \Rightarrow j' \sqsubseteq_{\mathbf{S}} j$ for the two distinct parcels of σ in the premise. If $i = \mathbf{S}$, then, for all values of i' such that $Q' : \tau \models \sigma^{i'}$, any proof redex substituted for σ^{i+j} in $Q : \tau$ will be head-needed as a result of being substituted for σ^i in the premise, i.e., the resource use of the contracted hypothesis σ in the conclusion is

S. Similarly, if $j = \mathrm{S}$, then σ in the conclusion also has resource use S. Similar arguments hold for the other possible combinations of values in S for i and j that show that if $i = \mathrm{N}$ and $j \neq \mathrm{S}$ (and vice versa), then the resource use of the contracted hypothesis is N, and that it is A only if both $i = \mathrm{A}$ and $j = \mathrm{A}$. An examination of the definition of $+$ shows that in each case $i + j$ produces the result required for the resource use of σ in the conclusion given all possible values for i and j in the premise. Hence $Q : \tau \models \sigma^{i+j}$, and by the monotonic property of $+$, we have that $i' + j' \sqsubseteq_{\mathrm{S}} i + j$.

Case 4: Let $Q : \tau$ be

$$\dfrac{\begin{matrix}[P] \\ \vdots \\ \Gamma, \sigma^{i} \vdash_{\mathrm{S}} \tau\end{matrix}}{\Gamma \vdash_{\mathrm{S}} \sigma \xrightarrow{i} \tau} \rightarrow \text{Intro}$$

By induction,

$$P : \tau \models \sigma^{i'} \Rightarrow i' \sqsubseteq_{\mathrm{S}} i \text{ and } \forall \phi^{j} \in \Gamma . P : \tau \models \phi^{j'} \Rightarrow j' \sqsubseteq_{\mathrm{S}} j$$

The discharge of the hypothesis σ^{i} does not affect the resource use of other hypotheses in Γ (any substitution of a proof redex $R : \phi$ for a hypothesis $\phi^{j} \in \Gamma$ takes place above the conclusion to $Q : \tau$, i.e., in $P : \tau$). Therefore, we have

$$\forall \phi^{j} \in \Gamma . Q : \tau \models \phi^{j'} \Rightarrow j' \sqsubseteq_{\mathrm{S}} j$$

Case 5: Let $Q : \tau$ be

$$\dfrac{\begin{matrix}[P_1] \\ \vdots \\ \Gamma \vdash_{\mathrm{S}} \sigma \xrightarrow{i} \tau\end{matrix} \quad \begin{matrix}[P_2] \\ \vdots \\ \Lambda \vdash_{\mathrm{S}} \sigma'\end{matrix} \quad \sigma' \subseteq_{\mathrm{S}} \sigma}{\Gamma, \Lambda^{\times i} \vdash_{\mathrm{S}} \tau} \rightarrow \text{Elim}$$

By induction,

$$\forall \phi^{j} \in \Gamma . P_1 : \sigma \xrightarrow{i} \tau \models \phi^{j'} \Rightarrow j' \sqsubseteq_{\mathrm{S}} j$$
$$\forall \psi^{k} \in \Lambda . P_2 : \sigma \models \psi^{k'} \sqsubseteq_{\mathrm{S}} k$$

The resource use annotations of those hypotheses in Γ are preserved in the conclusion, since the head-neededness or neededness of any proof redex substituted for such a hypothesis in $P_1 : \sigma \xrightarrow{i} \tau$ is preserved in $Q : \tau$. For those hypotheses ψ^k in Λ, we need to show that $Q : \tau \models \psi^{k' \times i'}$ where $k' \sqsubseteq_S k$ and $i' \sqsubseteq_S i$. This follows by induction on the rule used to produce the implication $\sigma \xrightarrow{i} \tau$, either an application of $\rightarrow$ Elim or Axiom, and the monotonicity of $\times$.

□

4.3 Relationship with resource use in the λ-calculus

We would like to show that the resource use inferred for propositions in IL_{S_f} is equivalent to the resource use inferred for λ-terms by $\rhd_S$. For example, given a proof

$$\begin{array}{c}[P]\\ \vdots \\ \Gamma, \sigma^i \vdash_S \tau\end{array}$$

then in the corresponding λ-term $\Psi(P : \tau)$, x_σ has resource use i, i.e., $\Psi(P : \tau) \rhd_S x_\sigma : i$.

However, as the following theorem and its proof demonstrate, this is not entirely the case. We are unable to demonstrate the existence of an equivalence. Instead, we show that the resource use inferred for propositions in IL_{S_f} is related to that inferred for variables in λ-terms by $\rhd_S$ by the ordering $\sqsubseteq_S$ over S. Taking the proof $P : \tau$ in the example above, with open hypothesis σ^i, we have

$$\Psi(P : \tau) \rhd_S x_\sigma : i'$$

such that $i' \sqsubseteq_S i$.

Theorem 4.3 *Given a proof*

$$\begin{array}{c}[P]\\ \vdots \\ \Gamma \vdash_S \tau\end{array}$$

then

$$\forall \sigma^u \in \Gamma . \Psi(P : \tau) \rhd_{\mathsf{S}} x_\sigma : u' \Rightarrow u' \sqsubseteq_{\mathsf{S}} u$$

Proof The proof can be given directly by induction over the height of the proof $P : \tau$ and the structure of the corresponding λ-term $\Psi(P : \tau)$, but also follows from Theorem 4.2 on the resource use soundness of of $\mathrm{IL}_{\mathsf{S}_f}$ and Theorem 4.1 on the equivalence under the Curry-Howard isomorphism of needed and head-needed redexes in proofs and λ-terms.

□

4.4 Resource use in intuitionistic logic in detail

In this section, we discuss how to provide more detailed resource use information other than simple strictness and absence. We briefly consider how neededness and head-neededness in intuitionistic logic can be extended in the manner of Section 3.3 in Chapter 3 for λ-terms, and then define more complex resource use domains for intuitionistic logic. In particular, we define IL_{L}, in which the resource use domain of strictness and absence is augmented with a linear resource use value.

4.4.1 Extending neededness and head-neededness

In order to define more detailed resource use domains for intuitionistic logic, we must define degrees of neededness and head-neededness, just as was done for λ-terms in Section 3.3 of Chapter 3. Rather than the exhaustive process followed in Section 4.1 above, we describe briefly how to measure the degree to which proof redexes may be needed or head-needed. The important concepts, such as neededness and head-neededness, have already been defined for intuitionistic logic, and the relationship with degrees of neededness and head-neededness in the λ-calculus follow as a result.

We introduce the following notation to denote the number of descendants of a proof redex normalised on a particular proof normalisation sequence:

Notation 4.1 *Let $P : \sigma$ be a proof redex in a proof $Q : \tau$. Let $\mathcal{S}$ be a proof normalisation path taking $Q : \tau$ to $Q' : \tau$. Then the descendants of $P : \sigma$ normalised on $\mathcal{S}$ is denoted by $\{P : \sigma\}_{\mathcal{S}}$, and the number of descendants normalised by $|\{P : \sigma\}_{\mathcal{S}}|$.*

In particular, $\{P : \sigma\}_{\mathcal{L}}$ and $\{P : \sigma\}_{\mathcal{H}}$ denote the descendants of $P : \sigma$ normalised on the leftmost and head normalisation paths to leftmost and head normal form, respectively.

It can be shown that given a proof redex $P : \sigma$ that is needed in a proof $Q : \tau$, then if $P : \sigma$ is replaced by any other proof redex $P' : \sigma$ then $P' : \sigma$ was also needed. As the following theorem demonstrates, in general, it is not the case that if n descendants of $P : \sigma$ are contracted in the leftmost normalisation of $Q : \tau$ then n descendants of any other proof redex $P' : \sigma$ replacing it in $Q : \tau$ will be contracted.

Theorem 4.4 *Let $P : \sigma$ be a proof redex in $Q : \tau$. Let $\mathcal{S}$ be a normalisation path of $Q : \tau$. Let $P' : \sigma$ be any other proof redex of σ replacing $P : \sigma$ in $Q : \tau$. Then*

$$|\{P : \sigma\}_{\mathcal{S}}| = n \not\Rightarrow |\{P' : \sigma\}_{\mathcal{S}}| = n$$

Proof We assume that

$$|\{P : \sigma\}_{\mathcal{S}}| = n \Rightarrow |\{P' : \sigma\}_{\mathcal{S}}| = n$$

is true, and then provide a counter-example to invalidate the assumption. Let

$$Q : (\sigma \to \sigma) \to \sigma \to \sigma$$

be the IL proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\text{Axiom} \quad \dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\text{Axiom} \quad \dfrac{}{\sigma \vdash \sigma}\text{Axiom}}{\sigma \to \sigma, \sigma \vdash \sigma}\to\text{Elim}}{\sigma \to \sigma, \sigma \to \sigma, \sigma \vdash \sigma}\to\text{Elim}}{\sigma \to \sigma, \sigma \vdash \sigma}\text{Contraction}}{\sigma \to \sigma \vdash \sigma \to \sigma}\to\text{Intro}}{\vdash (\sigma \to \sigma) \to \sigma \to \sigma}\to\text{Intro}$$

and let $P : \sigma \to \sigma$ be the proof

$$\dfrac{\dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\,\text{Axiom}}{\vdash (\sigma \to \sigma) \to (\sigma \to \sigma)}\,\to\text{Intro} \qquad \dfrac{\dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\vdash \sigma \to \sigma}\,\to\text{Intro}}{\vdash \sigma \to \sigma}\,\to\text{Elim}$$

in the proof $R : \sigma \to \sigma$ as follows:

$$\dfrac{\begin{array}{c}[Q]\\ \vdots\\ \vdash (\sigma \to \sigma) \to \sigma \to \sigma\end{array} \quad \begin{array}{c}[P]\\ \vdots\\ \vdash \sigma \to \sigma\end{array}}{\vdash \sigma \to \sigma}\,\to\text{Elim}$$

In the leftmost normalisation of proof $R : \sigma \to \sigma$ two descendants of $P : \sigma \to \sigma$ are normalised (the details of the normalisation are omitted). Let $P' : \sigma \to \sigma$ be the proof

$$\dfrac{\dfrac{\dfrac{}{\sigma \to \sigma \vdash \sigma \to \sigma}\,\text{Axiom}}{\vdash (\sigma \to \sigma) \to (\sigma \to \sigma)}\,\to\text{Intro} \qquad \dfrac{\dfrac{\dfrac{}{\sigma \vdash \sigma}\,\text{Axiom}}{\sigma, \sigma \vdash \sigma}\,\text{Weakening}}{\sigma \vdash \sigma \to \sigma}\,\to\text{Intro}}{\sigma \vdash \sigma \to \sigma}\,\to\text{Elim}$$

in which the occurrence of σ discharged in the right-most branch of the proof tree is the one introduced by Weakening.

Let $R' : \sigma \to \sigma$ be $R : \sigma \to \sigma$ with $P' : \sigma \to \sigma$ replacing $P : \sigma \to \sigma$, i.e.,

$$\dfrac{\begin{array}{c}[Q]\\ \vdots\\ \vdash (\sigma \to \sigma) \to \sigma \to\end{array} \quad \begin{array}{c}[P']\\ \vdots\\ \vdash \sigma \to \sigma\end{array}}{\sigma \vdash \sigma \to \sigma}\,\to\text{Elim}$$

In the leftmost normalisation of this proof, just one descendant of $P' : \sigma \to \sigma$ is normalised (again the actual details of the normalisation are omitted), thus contradicting the initial assumption.

□

4.4.2 Linear resource use in intuitionistic logic

In this section, we define a system of resource-aware intuitionistic logic similar to $\text{IL}_{\mathsf{S}f}$, but whose domain of resource use is L (see Section 3.4.1 of Chapter 3).

Axiom

$$\frac{}{(\sigma\downarrow)^{\mathsf{L}} \vdash_{\mathsf{S}} \sigma\downarrow}\ \text{Axiom}$$

Structural Rules

$$\frac{\Gamma, \sigma^{i}, \sigma^{j} \vdash_{\mathsf{S}} \tau}{\Gamma, \sigma^{i+j} \vdash_{\mathsf{S}} \tau}\ \text{Contraction} \qquad \frac{\Gamma \vdash_{\mathsf{S}} \tau}{\Gamma, (\sigma\downarrow)^{\mathsf{A}} \vdash_{\mathsf{S}} \tau}\ \text{Weakening}$$

Logical Rules

$$\frac{\Gamma, \sigma^{i} \vdash_{\mathsf{S}} \tau}{\Gamma \vdash_{\mathsf{S}} \sigma \xrightarrow{i} \tau} \to \text{Intro} \qquad \frac{\Gamma \vdash_{\mathsf{S}} \sigma \xrightarrow{i} \tau \quad \Lambda \vdash_{\mathsf{S}} \sigma' \quad \sigma' \subseteq \sigma}{\Gamma, \Lambda^{\times i} \vdash_{\mathsf{S}} \tau} \to \text{Elim}$$

Figure 4.4: $\mathrm{IL}_{\mathsf{L}_f}$: intuitionistic logic with linear resource use

Figure 4.4 presents the system $\mathrm{IL}_{\mathsf{L}_f}$ of intuitionistic logic with resource use annotations from domain L. This system is essentially the same as $\mathrm{IL}_{\mathsf{S}_f}$, except that the resource use of a hypothesis introduced in the Axiom is L, since any proof redex substituted for the axiom will obviously be normalised just once on the head and leftmost normalisation paths, and because multiple uses only arise from application of the Contraction rule. An ordering $\subseteq_{\mathsf{L}}$ over propositions is also defined, as follows.

Definition 4.16 *A partial ordering $\subseteq_{\mathsf{L}}$ on propositions in IL_{L_f} is defined as follows:*

$$\begin{array}{l} \alpha \subseteq_{\mathsf{L}} \alpha \\ \sigma \xrightarrow{i} \tau \subseteq_{\mathsf{L}} \sigma' \xrightarrow{j} \tau' \quad \Leftrightarrow \quad \sigma' \subseteq_{\mathsf{L}} \sigma, \tau \subseteq_{\mathsf{L}} \tau', i \sqsubseteq_{\mathsf{L}} j \end{array}$$

(where α denotes an atomic proposition).

The definitions of resource use values in domain L are given in Definition 4.17 below.

Definition 4.17 *Given the proof*

$$\begin{array}{c} [Q] \\ \vdots \\ \Gamma \vdash \tau \end{array}$$

then for any $\sigma \in \Gamma$, write $\sigma^u \in Q : \tau$ where u is defined as follows:

$$\begin{array}{llll}
\textit{(i)} & u \equiv \mathbf{L} & \Rightarrow & \forall P : \sigma \,.\, |\{P : \sigma\}_{\mathcal{H}}| = 1 \textit{ in } \Downarrow_{\mathcal{H}} Q : \tau[P/\sigma] \textit{ and} \\
 & & & |\{P : \sigma\}_{\mathcal{L}}| = 1 \textit{ in } \Downarrow_{\mathcal{L}} Q : \tau[P/\sigma] \\
\textit{(ii)} & u \equiv \mathrm{S} & \Rightarrow & \forall P : \sigma \,.|\{P : \sigma\}_{\mathcal{H}}| \geq 1 \textit{ in } \Downarrow_{\mathcal{H}} Q : \tau[P/\sigma] \textit{ and} \\
 & & & |\{P : \sigma\}_{\mathcal{L}}| \geq 1 \textit{ in } \Downarrow_{\mathcal{L}} Q : \tau[P/\sigma] \\
\textit{(iii)} & u \equiv \mathrm{A} & \Rightarrow & \forall P : \sigma \,.|\{P : \sigma\}_{\mathcal{H}}| = 0 \textit{ in } \Downarrow_{\mathcal{H}} Q : \tau[P/\sigma] \textit{ and} \\
 & & & |\{P : \sigma\}_{\mathcal{L}}| = 0 \textit{ in } \Downarrow_{\mathcal{L}} Q : \tau[P/\sigma] \\
\textit{(iv)} & u \equiv \mathrm{N} & \Rightarrow & \forall P : \sigma \,.|\{P : \sigma\}_{\mathcal{H}}| \geq 0 \textit{ in } \Downarrow_{\mathcal{H}} Q : \tau[P/\sigma] \textit{ and} \\
 & & & |\{P : \sigma\}_{\mathcal{L}}| \geq 0 \textit{ in } \Downarrow_{\mathcal{L}} Q : \tau[P/\sigma]
\end{array}$$

4.5 Related work

In [3], Baker-Finch describes a system of intuitionistic logic $R_{\supset}$, derived from *Relevant Logic* [31], which has three versions of the implication operator: $\rightarrow$ for strict or *relevant* implication, $\twoheadrightarrow$ for absent or *constant* implication, and $\supset$ for ordinary implication. In this system, tags are attached to hypotheses at the point they are introduced in order to determine which implicative operator should be used to discharge them. For example, if a hypothesis is introduced by Axiom, then it is given a relevant tag #, and if introduced by Weakening, the tag assigned to the hypothesis is the empty string ϵ. An additional tag of ? is used to indicate non-strictness. Hypotheses with tag ϵ are discharged using $\twoheadrightarrow$, and those with tag # are discharged using $\rightarrow$.

The relationship with our system $\mathrm{IL}_{\mathrm{S}_f}$ is immediately apparent, since the tags #, ϵ, and ? used by Baker-Finch intuitively correspond to the resource use annotations S, A, and N, respectively. Hence, $\xrightarrow{\mathrm{S}}$, $\xrightarrow{\mathrm{A}}$, and $\xrightarrow{\mathrm{N}}$ are respectively equivalent to $\rightarrow$, $\not\rightarrow$, and $\supset$ in Baker-Finch's system $R_{\supset}$. Moreover, intuitionistic implication in $R_{\supset}$ only occurs as a result of implicational axioms. The ? tag is not attached to a hypothesis by the Axiom rule, by Weakening, or by Contraction, but occurs only as a result of taking part in the minor premise to $\supset$ Elim, the elimination rule for $\supset$. Therefore, $\supset$ is either introduced in an implicational axiom or as a result of the discharge of a hypothesis which acquired the ? tag by taking part in a minor premise to $\supset$ Elim for an axiomatic occurrence of

$\supset$. This has obvious similarities with our restriction in $\mathrm{IL}_{\mathsf{S}_f}$ that implicational axioms must only contain a resource use of N. An important difference, however, is that in $R_\supset$, implicational axioms can be made using implicational arrows other than $\supset$. Also, another difference is that Contraction in $R_\supset$ only takes place over hypotheses assigned the same tag. By contrast, in IL_{S}, Contraction is defined over hypotheses with different resource use annotations using the operator $+$.

In IL_{S}, the semantics of the resource use annotations for hypotheses and implications are defined with respect to neededness and head-neededness for intuitionistic logic proofs. In addition, neededness and head-neededness, and subsequently, resource use, in intuitionistic logic are related to their corresponding concepts for the typed λ-calculus through the Curry-Howard isomorphism. Baker-Finch, however, does not define the semantics of the tags in $R_\supset$, other than # which is the tag of relevant implication. Nor are the tags related to the reduction behaviour of λ-terms.

Baker-Finch also describes in [3], and with Wright in [94], an extension to $R_\supset$ which uses the set of natural numbers $\mathbb{N}$ as the domain of annotations for hypotheses (thus creating an infinite family of implications for each value $n \in \mathbb{N}$, for example, $\xrightarrow{0}$, $\xrightarrow{1}$, $\xrightarrow{2}$ and so on). We have not extended our resource-aware system of intuitionistic logic to cover this domain of annotations, primarily for practical reasons. Wright has already discussed a resource-aware type system similar to such a logic in [93], and has mentioned that unification in an implementation would be undecidable.

We should also mention Girard's linear logic [36] (see also Wadler [89] and Abramsky [1] for good introductions to linear logic). It would seem that $\mathrm{IL}_{\mathsf{L}_f}$ and linear logic are related, since they both contain a notion of linearity. However, it is easy to produce simple proofs in $\mathrm{IL}_{\mathsf{L}_f}$ with linear hypotheses (i.e., hypotheses annotated with L) that are not linear in an equivalent linear logic proof. The reason is that in $\mathrm{IL}_{\mathsf{L}_f}$ applications of the Contraction rule may preserve linearity. For example,

$$\frac{\begin{array}{c}\vdots\\ \Gamma, \sigma^{\mathbf{L}}, \sigma^{\mathbf{A}} \vdash \tau\end{array}}{\Gamma, \sigma^{\mathbf{L}} \vdash \tau}\text{ Contraction}$$

In linear logic, however, Contraction is only permitted over non-linear hypotheses, i.e., those to which the ! operator is applied (as is also the case for Weakening). Another way of looking at this difference is that in linear logic, linear hypotheses are those whose parcel is limited to exactly one occurrence of the hypothesis by forbidding Contraction over it (if we ignore the similar restriction on Weakening). In $\mathrm{IL}_{\mathsf{L}_f}$, however, it is not the parcel size that determines linearity, but how many of the descendants of proof redexes substituted for the occurrences of a hypothesis in a parcel that will be head-needed and needed.

4.6 Summary

In this chapter, we have defined neededness and head-neededness for intuitionistic logic. We have also shown the equivalence of these definitions with neededness and head-neededness in the typed λ-calculus by means of the Curry-Howard isomorphism. Subsequently, we defined resource use for intuitionistic logic for strictness, absence, and non-strictness in IL_{S}. However, it was shown that in order to prove the soundness of resource use inference, it was necessary to restrict the resource use of implicative axioms, leading to the definition of the logical system $\mathrm{IL}_{\mathsf{S}_f}$. The relationship of resource use in $\mathrm{IL}_{\mathsf{S}_f}$ and the typed λ-calculus was shown to be an approximation based on the domain ordering $\sqsubseteq_{\mathsf{S}}$, rather than an equivalence, because implicative propositions in the system $\mathrm{IL}_{\mathsf{S}_f}$ were less expressive in terms of resource use annotations than types in the typed λ-calculus.

We also outlined how the definitions of neededness and head-neededness in intuitionistic logic could be extended in the same way as neededness and head-neededness were extended for the λ-calculus. Subsequently, we showed how resource use information could be extended to include, for example, a linear resource use value.

Chapter 5

A resource-aware type inference system

In this chapter, we describe a resource-aware type inference system for the typed λ-calculus, based on the resource-aware systems of intuitionistic logic discussed in Chapter 4. In this system, types inferred for λ-terms contain information about the resource use behaviour of those terms with respect to their parameters, where resource use is as defined in Chapters 3 and 4. For example, the type $\sigma \xrightarrow{\mathbf{S}} \tau \xrightarrow{\mathbf{A}} \sigma$ is the type of functions with a strict resource use in their first parameter and a resource use of absence in their second parameter, for example, a λ-term such as $\lambda x.\lambda y.x$.

We present the type inference system as a generic framework that can be parameterised over any of the resource use domains we have seen so far, simply by redefining the resource use constant labelling axioms and the $\times$ and $+$ operators for the appropriate domain. Furthermore, the basis of the type system we describe here are the *first-order* systems of resource-aware intuitionistic logic, such as $\mathrm{IL}_{\mathbb{S}f}$, presented in Chapter 4. There, for $\mathrm{IL}_{\mathbb{S}f}$, we showed that the resource use inferred for hypotheses in $\mathrm{IL}_{\mathbb{S}f}$ proofs corresponded to the resource use of free variables in equivalent λ-terms only up to approximation. As a result, in this chapter, the resource use inferred for λ-terms by type inference *approximates* the actual resource use of those λ-terms, in the sense that, for some term $\lambda x.M$, the resource use u inferred by the type system for x in M and the actual resource use v of x in M

$$\frac{}{\Gamma^{\mathbf{A}}\downarrow, x : (\sigma\downarrow)^c \vdash x : \sigma}\ \text{Axiom}$$

$$\frac{\Gamma, x : \sigma^i \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \xrightarrow{i} \tau}\ \rightarrow \text{Intro}$$

$$\frac{\Gamma \vdash M : \sigma \xrightarrow{i} \tau \qquad \Lambda \vdash N : \sigma' \qquad \sigma' \subseteq \sigma}{\Gamma + \Lambda^{\times i} \vdash MN : \tau}\ \rightarrow \text{Elim}$$

Figure 5.1: F_r: a resource-aware type inference system

(given, for example, by $\rhd_{\mathsf{S}}$) are such that $v \sqsubseteq u$ (where $\sqsubseteq \equiv \sqsubseteq_{\mathsf{S}}$ if the resource use domain is S).

In the rest of this chapter, we describe the resource-aware type inference system, and discuss its properties, for example, Subject Reduction in the light of approximate information. We show that the type system is sound with respect to the simple semantics (see Chapter 2, Section 2.5.1) and to the semantics of resource use for λ-terms(see Chapter 3). Finally, we discuss an extension to the type system to accommodate Milner-style let-polymorphism.

5.1 Resource-aware type inference

In this section, we describe a system of resource-aware type inference F_r, which follows from the application of resource-aware intuitionistic logic (discussed in Chapter 4) to λ-terms as type inference rules in the same way that conventional intuitionistic logic is applied as a type inference system for λ-terms in the form of Curry's system of F-deducibility under the Curry-Howard isomorphism.

Figure 5.1 presents the resource-aware system of type inference F_r based on the first-order systems of resource-aware intuitionistic logic, such as IL_{S_f}, in Chapter 4. We derive the type system from the logical system as follows: at each stage in a proof, the λ-term to be typed represents the encoding of the proof so far according to the Curry-Howard isomorphism. The inference system is presented as a general framework that can be parameterised over any of the resource use domains considered in Chapters 3 and 4, so that, for example, F_r^{S} is the inference system that infers resource use values in domain S, and F_r^{L} infers resource use values in domain L. In each case, the definition of the $+$ and $\times$ operators is the definition of those operators associated with the relevant resource use domain.

The language of *resource types*, i.e., the type expressions incorporating resource use values, is given by the following grammar:

$$\sigma ::= \alpha \mid \sigma_1 \xrightarrow{u} \sigma_2$$

where σ ranges over types, α over atomic types (type constants and type variables), and u over values in the relevant resource use domain, for example, S.

The variable typing rule in the generic system F_r is defined as

$$\frac{}{\Gamma^{\mathbf{A}} \downarrow, x : (\sigma \downarrow)^c \vdash x : \sigma} \text{Axiom}$$

where the annotation c to the hypothesis or type assumption $x : (\sigma \downarrow)$ is, for example, S if the type inference system is parameterised over S and L if it is parameterised over L. Also, the resource use annotations of functionally-typed variables introduced by the Axiom rule are set to N using the $\downarrow$ operator, as defined for $\mathrm{IL}_{\mathsf{S}_f}$ (see Section 4.2.3, Chapter 4). The Axiom rule also combines both the introduction of a variable and its type with the Weakening of other hypotheses in the assumption set, which we denote by $\Gamma^{\mathbf{A}} \downarrow$, where $\Gamma^{\mathbf{A}}$ is a set of type assumptions $x : \phi^i$ such that $i \equiv \mathbf{A}$. The meaning of $\Gamma^{\mathbf{A}} \downarrow$ is as follows:

$$\Gamma^{\mathbf{A}} \downarrow = \{x : (\phi \downarrow)^{\mathbf{A}} | x : \phi^{\mathbf{A}} \in \Gamma^{\mathbf{A}}\}$$

In the rest of this chapter, we will adopt the convention of omitting $\downarrow$ from presentations of prooftrees in examples, showing instead, where necessary, its effect on functionally-typed hypotheses.

In the $\rightarrow$ Elim rule, we perform Contraction on the common hypotheses between the bases of the two sequents which are premises to the rule, identifying those hypotheses by variable names. Before we can contract two hypotheses that appear in the two bases for the same variable, however, we must take account of the effect that application of the function to its argument has on the resource use of those hypotheses in the base Λ of the premise typing the argument. We denote the changes in the resource use of the hypotheses by $\Lambda^{\times i}$, defined as

$$\{x : \phi^{j \times i} | x : \phi^j \in \Lambda\}$$

Contraction takes place over common variables in the bases Γ and $\Lambda^{\times i}$ using the $+$ operator. The contraction of bases $\Gamma + \Lambda^{\times i}$ in the conclusion of the typing rule $\rightarrow$ Elim is defined as follows:

$$\{x : \sigma^{j+k} \mid x : \sigma^j \in \Gamma, x : \sigma^k \in \Lambda^{\times i}\} \cup (\Gamma \bigtriangleup \Lambda^{\times i})$$

(where $\Gamma \bigtriangleup \Lambda^{\times i}$ is the symmetric set difference of the bases). Both $+$ and $\times$ are as defined for the relevant resource use domain in Chapter 3.

Also in $\rightarrow$ Elim, the ordering relation $\subseteq$ defined over propositions in Section 4.2.2 of Chapter 4 appears in the third premise to the rule as a *coercion relation* over resource types. This follows the definition of the logical rule $\rightarrow$ Elim in $\mathrm{IL}_{\mathsf{S}_f}$; the definition of $\subseteq$ over resource types is

$$\begin{aligned} &\alpha \subseteq \alpha \\ &\sigma \xrightarrow{i} \tau \subseteq \sigma' \xrightarrow{j} \tau' \quad \Leftrightarrow \quad \sigma' \subseteq \sigma, \tau \subseteq_{\mathsf{S}} \tau', i \sqsubseteq j \end{aligned}$$

(where α denotes an atomic proposition and $\sqsubseteq$ is the ordering over the relevant resource use domain). As a matter of terminology, if a type σ and a type τ are such that $\sigma \subseteq \tau$, then σ is said to be a *subtype* of the *supertype* τ. In Section 5.2.1 below, we give a semantics of the ordering relation and discuss in more detail the anti-monotonic property of the relation over function types.

5.1.1 Typing multiple occurrences of variables

Both Wright, in [93], and Baker-Finch, in [3], discuss the problem of typing multiple occurrences of variables. For example, consider the term $\lambda f.g(fI)(f(Kz))$. The first occurrence of the variable f could be assigned type $(\sigma \xrightarrow{\mathbf{S}} \sigma) \xrightarrow{\mathbf{S}} \tau$, while the second occurrence could be given type $(\sigma \xrightarrow{\mathbf{A}} \sigma) \xrightarrow{\mathbf{S}} \tau$ for its argument. The problem, therefore, is to derive a type for a variable that fits the context of each occurrence of the variable, where the context is provided by the arguments (if any) to the variable. In other words, we must provide a type for a variable such that, if the variable is used as a function, the argument components of the function type for the variable match the types of the arguments supplied to the variable. As the above example shows, different occurrences of such a variable may be given extensionally equivalent but intensionally different arguments, and so, the problem further refines itself to the resolution of the different resource uses of arguments to a functional variable. So far, two different solutions to this problem have been proposed, by Wright and Baker-Finch.

Wright's solution requires the extension of the type system to *intersection types*, which, in the above example, allows the variable f to possess *both* types, joined by conjunction. However, this is at the expense of the decidability of type inference. Type inference with intersection type is known to be undecidable [81], although a semi-decidable algorithm exists[1].

A simpler solution has been proposed by Baker-Finch, which is to include intuitionistic implication in the language of function types, along with coercion of types as a side-condition to the application typing rule (see Figure 2.4, Section 2.6 of Chapter 2). This allows, for example, a strict function to be used where a non-strict function is expected. This is similar to our approach, based on the first-order resource-aware intuitionistic systems described in Chapter 4, except that Baker-Finch's system still allows for the possibility of introducing functional types as axioms with resource use information that is more specific than non-strictness.

[1]Hankin and Le Métayer [40] have described a non-standard type inference system for strictness analysis that uses lazily evaluated types to recover efficiency in the presence of conjunctive types. It is not known whether this approach can be applied to Wright's type system to improve its performance.

5.1.2 Example typings

The following are examples of typings produced by the type inference system F_{τ}^{S}.

Example 5.1 *The type assigned to $\lambda x.x$ is*

$$\dfrac{\dfrac{}{x : \sigma^{\mathrm{S}} \vdash x : \sigma}\ \text{Axiom}}{\vdash \lambda x.x : \sigma \xrightarrow{\mathrm{S}} \sigma}\ \to \text{Intro}$$

Example 5.2 *The type inferred for the λ-term $\lambda x.\lambda y.x$ is*

$$\dfrac{\dfrac{\dfrac{}{x : \sigma^{\mathrm{S}}, y : \tau^{\mathrm{A}} \vdash x : \sigma}\ \text{Axiom}}{x : \sigma^{\mathrm{S}} \vdash \tau \xrightarrow{\mathrm{A}} \sigma}\ \to \text{Intro}}{\vdash \lambda x.\lambda y.x : \sigma \xrightarrow{\mathrm{S}} \tau \xrightarrow{\mathrm{A}} \sigma}\ \to \text{Intro}$$

Example 5.3 *Given the λ-term $\lambda f.\lambda g.\lambda x.fx(gx)$, the inferred type is as follows. Let*

$$\begin{array}{c}[P]\\ \vdots \\ f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}}, x : \sigma^{\mathrm{N}} \vdash fx : \tau \xrightarrow{\mathrm{N}} \phi\end{array}$$

denote the typing

$$\dfrac{\dfrac{}{f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}} \vdash f : \sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi}\ \text{Axiom} \qquad \dfrac{}{x : \sigma^{\mathrm{S}} \vdash x : \sigma}\ \text{Axiom} \qquad \sigma \subseteq_{\mathrm{S}} \sigma}{f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}}, x : \sigma^{\mathrm{S} \times \mathrm{N} = \mathrm{N}} \vdash fx : \tau \xrightarrow{\mathrm{N}} \phi}\ \to \text{Elim}$$

and let

$$\begin{array}{c}[Q]\\ \vdots \\ g : (\sigma \xrightarrow{\mathrm{N}} \tau)^{\mathrm{S}}, x : \sigma^{\mathrm{N}} \vdash gx : \tau\end{array}$$

denote the typing

$$\dfrac{\dfrac{}{g : \sigma \xrightarrow{\mathrm{N}} \tau^{\mathrm{S}} \vdash g : \sigma \xrightarrow{\mathrm{N}} \tau}\ \text{Axiom} \qquad \dfrac{}{x : \sigma^{\mathrm{S}} \vdash x : \sigma}\ \text{Axiom} \qquad \sigma \subseteq_{\mathrm{S}} \sigma}{g : (\sigma \xrightarrow{\mathrm{N}} \tau)^{\mathrm{S}}, x : \sigma^{\mathrm{S}\times\mathrm{N}=\mathrm{N}} \vdash gx : \tau}\ \rightarrow \text{Elim}$$

in the following derivation of the type for $\lambda f.\lambda g.\lambda x.fx(gx)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[P]\\ \vdots\\ f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}}, x : \sigma^{\mathrm{N}} \vdash fx : \tau \xrightarrow{\mathrm{N}} \phi\end{array} \qquad \begin{array}{c}[Q]\\ \vdots\\ g : (\sigma \xrightarrow{\mathrm{N}} \tau)^{\mathrm{S}}, x : \sigma^{\mathrm{N}} \vdash gx : \tau\end{array} \quad \tau \subseteq_{\mathrm{S}} \tau}{f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}}, g : (\sigma \xrightarrow{\mathrm{N}} \tau)^{\mathrm{S}\times\mathrm{N}=\mathrm{N}}, x : \sigma^{\mathrm{N}+(\mathrm{N}\times\mathrm{N})=\mathrm{N}} \vdash fx(gx) : \phi}\ \rightarrow \text{Elim}}{f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}}, g : (\sigma \xrightarrow{\mathrm{N}} \tau)^{\mathrm{N}} \vdash \lambda x.fx(gx) : \sigma \xrightarrow{\mathrm{N}} \phi}\ \rightarrow \text{Intro}}{f : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi)^{\mathrm{S}} \vdash \lambda g.\lambda x.fx(gx) : (\sigma \xrightarrow{\mathrm{N}} \tau) \xrightarrow{\mathrm{N}} \sigma \xrightarrow{\mathrm{N}} \phi}\ \rightarrow \text{Intro}}{\vdash \lambda f.\lambda g.\lambda x.fx(gx) : (\sigma \xrightarrow{\mathrm{N}} \tau \xrightarrow{\mathrm{N}} \phi) \xrightarrow{\mathrm{S}} (\sigma \xrightarrow{\mathrm{N}} \tau) \xrightarrow{\mathrm{N}} \sigma \xrightarrow{\mathrm{N}} \phi}\ \rightarrow \text{Intro}$$

5.1.3 Properties of the type inference system

In this section, we discuss properties of F_r. We show that F_r is as expressive as Curry type inference, i.e., that any term typable by F_r is Curry-typable, and vice versa; and that the inference of resource use by the type system corresponds to the resource use of terms. As we would expect, this correspondence is based on approximation, and follows from the work on relating resource use in $\mathrm{IL}_{\mathrm{S}f}$ and the typed λ-calculus, covered in Chapter 4. Also, as a consequence, the Subject Reduction theorem does not hold in the form used for the Curry type system, i.e., types of terms are not preserved across reduction.

We show that every λ-term which possesses a Curry type is also typable in the resource-aware type system F_r and vice versa. We first introduce a function Υ that translates resource types into Curry types by simply stripping resource use annotations from function types, defined as follows:

$$\begin{array}{lcl} \Upsilon(\alpha) & = & \alpha \\ \Upsilon(\sigma \xrightarrow{i} \tau) & = & \Upsilon(\sigma) \rightarrow \Upsilon(\tau) \end{array}$$

In the following theorems, $\vdash_C$ refers to typing in the Curry type system and $\vdash_r$ refers to typing in the resource-aware type system.

Theorem 5.1 *Let $\Gamma \vdash_C M : \sigma$ be a valid Curry typing. Then there exists a Γ_r such that*

$$\forall x : \phi \in \Gamma . x : \psi^i \in \Gamma_r \text{ and } \Upsilon(\psi) \equiv \phi$$

and there exists σ_r such that $\Upsilon(\sigma_r) \equiv \sigma$, such that $\Gamma_r \vdash_r M : \sigma_r$ is a valid typing in F_r.

Proof The proof is by induction over the proof of $\Gamma \vdash_C M : \sigma$. The Axiom and $\rightarrow$ Intro cases are straightforward. So is the $\rightarrow$ Elim case, except that we must deal with the additional premise concerning coercion between types. By induction, we know that if $\Gamma \vdash_C M : \sigma \rightarrow \tau$ and $\Lambda \vdash_C N : \sigma$ are provable, then so are $\Gamma_r \vdash_r M : \sigma_r \xrightarrow{i} \tau_r$ and $\Lambda_r \vdash_r N : \sigma_r'$. However, we must also show that $\sigma_r' \subseteq \sigma_r$. This follows from a sub-induction over the structure of σ as follows. If $\sigma \equiv \alpha$, then $\sigma_r' \subseteq \sigma_r$ since $\subseteq$ is reflexive over atomic types. If $\sigma \equiv \phi \rightarrow \psi$, then, assuming the equivalences

$$\sigma_r \equiv \sigma_{r_1} \xrightarrow{i} \sigma_{r_2} \text{ and } \sigma_r' \equiv \sigma_{r_1}' \xrightarrow{j} \sigma_{r_2}'$$

we require that

$$\sigma_{r_1} \subseteq \sigma_{r_1}', \ \sigma_{r_2}' \subseteq \sigma_{r_2}, \text{ and } j \sqsubseteq i$$

Since σ was discharged as a result of $\rightarrow$ Intro or $\sigma \rightarrow \tau$ was introduced by Axiom, then

$$\sigma_{r_1} \xrightarrow{i} \sigma_{r_2} \equiv \sigma_{r_1} \xrightarrow{\mathrm{N}} \sigma_{r_2}$$

Hence we have that $j \sqsubseteq \mathrm{N}$ for any value of j. For the same reasons, all resource use annotations in σ_{r_1} must be N. Similarly, the resource use annotations in σ_{r_1}' must be N, and hence

$$\sigma_{r_1} \subseteq \sigma_{r_1}'$$

by reflexivity of $\subseteq$. Then $\sigma_{r_2}' \subseteq \sigma_{r_2}$ can be shown to hold by induction.

□

The following theorem shows that any term typable by F_r is Curry typable.

Theorem 5.2 *Let $\Gamma_r \vdash_r M : \sigma_r$ be a valid typing in F_r. Let there be a Γ such that*

$$\forall x : \sigma^i \in \Gamma_r . x : \Upsilon(\sigma) \in \Gamma$$

Then $\Gamma \vdash_C M : \Upsilon(\sigma)$ is a valid Curry typing

Proof By a straightforward induction over the proof of $\Gamma_r \vdash_r M : \sigma_r$. For the $\rightarrow$ Elim case, and the premise $\sigma_r^{'} \subseteq \sigma_r$, note that erasure of resource use annotations from types turns $\subseteq$ into $\equiv$, i.e., equivalence.

$\square$

Theorem 5.3 states that the resource use of typing hypotheses inferred by the type system is an approximation of the resource use of the free variables of those hypotheses in the term being typed. This theorem is essentially a restatement of Theorem 4.3 given in Section 4.3 of Chapter 4 as it relates to $\mathrm{IL}_{\mathsf{S}_f}$ (the system of resource-aware intuitionistic logic in which implicative axioms are introduced with resource use annotation N), and hence we have not given a separate proof.

Theorem 5.3 *If* $\Gamma_r \vdash_r M : \sigma_r$ *is a valid typing in* F_r *then*

$$\forall x : \phi^i \in \Gamma_r . M \ \triangleright x : i^{'} \Rightarrow i^{'} \sqsubseteq i$$

Proof Follows directly from the proof of Theorem 4.3.

$\square$

It follows from this theorem that we are unable to show that types of terms are preserved over reduction steps. The reason why inferred resource use only approximates actual resource use in λ-terms is that in applications of $\rightarrow$ Elim information is lost in the derivation of resource use of hypotheses in the conclusion to the rule when the argument has a more specific type than that expected by the function. Therefore we expect, for example, that the type of a term not in normal form and its type when it is in normal form will not be the same, although this difference can be explained solely in terms of the resource use annotations.

5.2 Soundness of type inference

In this section, we prove the soundness and completeness properties for the resource-aware type system F_r.

5.2.1 The semantics of resource types

The semantics of resource type variables is defined by an environment $\nu :: TypeVar \rightarrow 2^D$ which maps type variables to subsets of D (where 2^D is the powerset of D). We define a semantic interpretation of resource type expressions $[\![\ ]\!]_t$ of type expressions as follows

$$\begin{aligned} [\![\alpha]\!]_t \nu &= \nu(\alpha) \\ [\![\sigma \xrightarrow{i} \tau]\!]_t \nu &= \{f \in D | \forall e.e \in [\![\sigma]\!]_t \nu \Rightarrow (fe) \in [\![\tau]\!]_t \nu \wedge f \cong i\} \end{aligned}$$

For Curry types, membership of $\beta \rightarrow \sigma$ depends only on the applicative behaviour of a functional element *d*, i.e., *d* is a member of $\beta \rightarrow \sigma$ iff for all values *e* in the type β, the application of *d* to *e* produces a value in the range type σ. For resource types, we similarly define membership according to applicative behaviour of functional elements but qualify membership according to the operational behaviour of that element, so that a value *f* belongs to $\sigma \xrightarrow{i} \tau$ iff its applicative behaviour is as for Curry types *and* its operational behaviour as regards its arguments is congruent with the resource use *i* annotating the function type, as defined by the $\cong$ operator (see Definition 3.4, Section 3.2 of Chapter 3).

5.2.2 Satisfaction for the resource aware type system

Satisfaction establishes a correspondence between assignments of types to terms and the inclusion relation between the term and type semantics (see Section 2.5.1 of Chapter 2 for satisfaction in Curry type inference). For resource types, we use the same notion of satisfaction, but extended in order to show that the resource use of hypotheses in a type judgement is also satisfied with respect to the λ-term being typed.

A type assignment $x : \sigma^i$ is satisfied by a model $\mathcal{D} =< D, \cdot, [\![\ ]\!] >$ and environments $\eta :: TermVar \rightarrow D$, $\nu :: TypeVar \rightarrow 2^D$ if $[\![x]\!]_\lambda \eta \in [\![\sigma]\!]_t \nu$. A base Γ of such type assignments is satisfied iff all its members are satisfied. A typing $\Gamma \vdash M : \sigma$ is satisfied if, for all environments η, ν satisfying Γ, then

$$[\![M]\!]_\lambda \eta \in [\![\sigma]\!]_t \nu \text{ and } \forall x : \sigma^j \in \Gamma . [\![M]\!]_\lambda \eta \rhd [\![x]\!]_\lambda \eta : i \Rightarrow i \sqsubseteq j$$

This definition of satisfaction also takes into account the satisfaction of resource use annotations of hypotheses with respect to the semantics of resource use given in Chapter 3. Note that satisfaction of resource use annotations to hypotheses is defined using the ordering relation $\sqsubseteq$ because we have only approximation, not equivalence, between resource use in the type system and resource use in λ-terms. If a typing $\Gamma \vdash M : \sigma$ is satisfied according to this definition, then we write $\Gamma \models M : \sigma$.

5.2.3 Soundness

Theorem 5.4 (Soundness of resource-aware type inference)

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma \models M : \sigma$$

Proof The proof of soundness for resource-aware type inference is by induction over the structure of λ-terms.

Case 1 : $M \equiv x$

By the Axiom rule,

$$\frac{}{\Gamma^{\mathbf{A}} \downarrow, x : (\sigma \downarrow)^c \vdash x : \sigma} \text{Axiom}$$

where $\Gamma^{\mathbf{A}} = y_1 : \phi_1^{\mathbf{A}} \ldots y_n : \phi_n^{\mathbf{A}}$, any η, ν satisfying $\Gamma^{\mathbf{A}}, x : \sigma^c$ also satisfies $x : \sigma$. Also, by rule Var$_1$ of $\rhd$,

$$x \rhd x : c$$

and, by rule Var$_2$ of $\rhd$

$$\forall y \in dom(\Gamma^{\mathbf{A}}) . x \rhd y : \mathbf{A}$$

Therefore,

$$\Gamma^{\mathbf{A}}, x : \sigma^c \models x : \sigma$$

Case 2 : $M \equiv \lambda x.M$.

By induction,

$$\Gamma, x : \sigma^i \vdash M : \tau \Rightarrow \Gamma, x : \sigma^i \models M : \tau$$

which, by definition, implies

$$[\![M]\!]_\lambda \eta \in [\![\tau]\!]_t \nu, [\![M]\!]_\lambda \eta \rhd x : i' \Rightarrow i' \sqsubseteq i \text{ and } \forall y : \phi^j \in \Gamma.[\![M]\!]_\lambda \eta \rhd y : j' \Rightarrow j' \sqsubseteq j$$

By definition of the term model

$$[\![M]\!]_\lambda \eta \in [\![\tau]\!]_t \nu \Rightarrow \forall a \in [\![\sigma]\!]_t \nu.([\![M]\!]_\lambda \eta'[x \mapsto a] \in [\![\tau]\!]_t \nu$$

which can be rewritten as

$$\forall a \in [\![\sigma]\!]_t \nu.([\![\lambda x.M]\!]_\lambda \eta' \cdot a) \in [\![\tau]\!]_t \nu$$

and consequently, we have

$$[\![\lambda x.M]\!]_\lambda \eta' \in [\![\sigma \xrightarrow{i} \tau]\!]_t \nu$$

Case 3 : $M \equiv M_1 M_2$

By induction we have

$$\begin{array}{lll} \Gamma \vdash M_1 : \sigma \xrightarrow{i} \tau & \Rightarrow & \Gamma \models M_1 : \sigma \xrightarrow{i} \tau \\ \Lambda \vdash M_2 : \sigma & \Rightarrow & \Lambda \models M_2 : \sigma \end{array}$$

i.e., that

$$\begin{array}{lll} [\![M_1]\!]_\lambda \eta \in [\![\sigma \xrightarrow{i} \tau]\!]_t \nu & \text{and} & \forall x : \phi^j \in \Gamma.[\![M_1]\!]_\lambda \eta \rhd x : j' \Rightarrow j' \sqsubseteq j \\ [\![M_2]\!]_\lambda \eta \in [\![\sigma]\!]_t \nu & \text{and} & \forall y : \psi^k \in \Gamma.[\![M_2]\!]_\lambda \eta \rhd y : k' \Rightarrow k' \sqsubseteq k \end{array}$$

By definition of the type semantics, we have

$$\forall a \in [\![\sigma]\!]_t \nu.[\![M_1]\!]_\lambda \eta \cdot a \in [\![\tau]\!]_t \nu$$

which implies that $[\![M_1]\!]_\lambda\eta \cdot [\![M_2]\!]_\lambda\eta \in [\![\tau]\!]_t\nu$ and hence that

$$[\![M_1M_2]\!]_\lambda\eta \in [\![\tau]\!]_t\nu \qquad (5.1)$$

For the typing hypotheses in Λ, rename the term variables $y_1, \ldots, y_n \in dom(\Lambda)$ to fresh variables $w_1, \ldots, w_n$. If $\lambda z.M_1' \in [\![M_1]\!]_\lambda\eta$, then rule App$_1$ of $\triangleright$ applies for each $w : \psi^k$ with conclusion

$$[\![M_1M_2]\!]_\lambda\eta \triangleright w : \mathbf{A} + (k' \times i')$$

where $\mathbf{A}$ follows from the fact that $w \notin FV(M_1)$, and $i' \sqsubseteq i$. If $zM_1' \ldots M_n' \in [\![M_1]\!]_\lambda\eta$, then rule App$_2$ applies with conclusion

$$[\![M_1M_2]\!]_\lambda\eta \triangleright w : k' \sqcup \mathbf{A}$$

Furthermore, for any $y \in dom(\Gamma)$, we have, by Lemma 3.8, that

$$[\![M_1M_2]\!]_\lambda\eta \triangleright y : j' + l$$

where $y : \psi^j \in \Gamma$ and $j' \sqsubseteq j$, and if $y \in dom(\Lambda)$, then l is $k' \times i'$ or $k' \times \mathrm{N}$ from above. If $y \notin dom(\Lambda)$, then l is $\mathbf{A}$ where $\mathbf{A}$ is the identity for $+$. By definition of $+$ and $\times$ over bases, therefore, we have

$$\Gamma + \Lambda^{\times i} \models M_1M_2 : \tau$$

□

5.2.4 Principal types in F_r

In Section 2.5 of Chapter 2, principal types for Curry type inference were defined as type schemes from which all valid typings for λ-terms could be obtained by substitution of type expressions for the variables of the type scheme. In this section, we show how principal resource types can be defined.

The use of the ordering relation $\subseteq$ in the $\rightarrow$ Elim rule in F_r implies that a λ-term may have many possible typings that differ only in their resource use annotations. This can be seen more easily if we replace the premise $\sigma' \subseteq \sigma$ to the $\rightarrow$ Elim rule with a Coerce rule as follows:

$$\frac{\Gamma \vdash M : \sigma \qquad \sigma \subseteq \tau}{\Gamma \vdash M : \tau}\text{Coerce}$$

With the addition of this rule to the type system, then (in F_{τ}^{L}) the λ-term $\lambda f.\lambda x.fx$ has the types

$$(\sigma \xrightarrow{\mathbf{N}} \tau) \xrightarrow{\mathbf{L}} \sigma \xrightarrow{\mathbf{N}} \tau$$
$$(\sigma \xrightarrow{\mathbf{N}} \tau) \xrightarrow{\mathbf{S}} \sigma \xrightarrow{\mathbf{N}} \tau$$
$$(\sigma \xrightarrow{\mathbf{N}} \tau) \xrightarrow{\mathbf{N}} \sigma \xrightarrow{\mathbf{N}} \tau$$

which differ only in the resource use annotation for the first argument of the function type. We see, therefore, that the set of valid typings for a λ-term are given by instantiations of type variables and coercions of resource use annotations in function types.

To represent principal types in the presence of coercion, induced by the ordering over resource use annotations, we follow the approach taken by Mitchell in [73] (further developed by Fuh and Mishra in [33]) on type inference with subtyping. In Mitchell's system, subtyping is used to take account of the possible coercions that may be present between types. For example, type *Integer* may be coerced to, or be a subtype of, type *Real*. Subtype relations, such as $Integer \subseteq Real$, are contained in a *coercion set* C, which is present as an additional component in typing sequents. For example, the $\rightarrow$ Elim rule in Mitchell's type inference system is

$$\frac{C;\Gamma \vdash M : \sigma \rightarrow \tau \qquad C;\Lambda \vdash N : \sigma}{C;\Gamma,\Lambda \vdash MN : \tau}\rightarrow \text{Elim}$$

The Coerce rule uses derivation over the coercion set as follows

$$\frac{C;\Gamma \vdash M : \sigma \qquad C \vdash \sigma \subseteq \tau}{C;\Gamma \vdash M : \tau}\text{Coerce}$$

where the rules for deriving coercions are as follows:

$$\frac{}{\sigma \subseteq \sigma} \quad \text{Reflexivity}$$

$$\frac{\sigma \subseteq \tau \quad \tau \subseteq \phi}{\sigma \subseteq \phi} \quad \text{Transitivity}$$

$$\frac{\sigma_1 \subseteq \sigma \quad \tau \subseteq \tau_1}{\sigma \to \tau \subseteq \sigma_1 \to \tau_1} \quad \text{Function}$$

A coercion $\sigma \subseteq \tau$ is provable from a coercion set C, written $C \vdash \sigma \subseteq \tau$, if it can be derived from the coercions in C using the above rules. By overloading the notation, $C \vdash C'$ is taken to mean that

$$\forall \sigma \subseteq \tau \in C'.C \vdash \sigma \subseteq \tau$$

The type for a term M in this system of type inference with coercions is represented as a pair (C, σ), where C is a set of coercions between types and σ is a type expression for M. A type (C', σ') is defined as an instance of (C, σ) if there exists a substitution S of types for type variables such that

$$C' \vdash SC \text{ and } \sigma' = S\sigma$$

A type (C, σ) is therefore the principal type for M if all valid typings of M arise as instances of (C, σ).

In our approach to principal types, we define coercion sets as containing relations between resource use values that represent the ordering of the relevant resource use domain. For example, if the resource use domain is S, then the associated coercion set R over resource use values in S is

$$\{S \sqsubseteq_{\mathsf{S}} N, A \sqsubseteq_{\mathsf{S}} N\}$$

$$\frac{}{R; \Gamma^{\mathbf{A}}, x : \sigma^{c} \vdash x : \sigma} \text{ Axiom}$$

$$\frac{R; \Gamma, x : \sigma^{i} \vdash M : \tau}{R; \Gamma \vdash \lambda x.M : \sigma \xrightarrow{i} \tau} \rightarrow \text{Intro}$$

$$\frac{R; \Gamma \vdash M : \sigma \xrightarrow{i} \tau \quad R; \Lambda \vdash N : \sigma' \quad R \vdash \sigma \subseteq \sigma'}{R; \Gamma + \Lambda^{\times i} \vdash MN : \tau} \rightarrow \text{Elim}$$

Figure 5.2: F_{rc}: resource-aware type inference with coercions

However, we define the rules of derivation of coercions as operating over types as follows:

$$\frac{}{\sigma \subseteq \sigma} \quad \text{Reflexivity}$$

$$\frac{\sigma \subseteq \tau \quad \tau \subseteq \phi}{\sigma \subseteq \phi} \quad \text{Transitivity}$$

$$\frac{\sigma_1 \subseteq \sigma \quad \tau \subseteq \tau_1 \quad i \sqsubseteq_S j}{\sigma \xrightarrow{i} \tau \subseteq \sigma_1 \xrightarrow{j} \tau_1} \quad \text{Function}$$

Therefore, a coercion $\sigma \subseteq \tau$ is derivable from a coercion set R over resource use values, i.e.,

$$R \vdash \sigma \subseteq \tau$$

if it can be derived from the relations between resource use values in R and the rules above.

In Figure 5.2, we present the revised system of resource-aware type inference, F_{rc}, in which the coercion set over resource use values is made explicit. Types for λ-terms inferred by this system are represented as a pair (R, σ), where R represents the relevant resource use domain and σ is an inferred type expression.

We define type instance as follows: a type (R, σ) has as an instance (R, τ) if there exists a substitution S of types for type variables such that

$$R \vdash \tau \subseteq S\sigma$$

A type (R, σ) for a λ-term M is therefore the principal type for M if all valid typings (R, τ) for M arise as instances of (R, σ).

The presence of the resource coercion set R in the definition of a type for a term would appear to be superfluous, since, for example, we could define type instance as $\sigma \subseteq S\tau$ in which the resource use domain is implicit in the parameterisation of the type inference system. In the next section, however, we make use of the resource coercion set in introducing let-polymorphism into the type system, and so we take the opportunity to introduce it now.

5.3 Introducing let-polymorphism

Milner has shown in [72], and with Damas in [26], how a restricted form of *parametric polymorphism*[2], called *let-polymorphism*, allows different occurrences of a variable in the body of an expression to be given different types.

The essence of let-polymorphism is that a variable may have different types at different occurrences if the argument to be substituted for the variable is already known. For example, in the λ-term $(\lambda f.g(fI)(fK))(\lambda z.z)$, the variable f is typed as $\sigma \to \sigma$ at its first occurrence and as $(\phi \to \psi \to \phi) \to (\phi \to \psi \to \phi)$ at its second occurrence. Milner introduced a special syntactic construct to represent such situations, e.g.,

$$let\ f = \lambda z.z\ in\ g(fI)(fK)$$

as well as type-schemes in which type variables are bound in much the same way as term variables are bound in λ-terms. Type expressions can then be substituted for such bound type variables to create instances of a type-scheme for a variable such as f in the example

[2]Parametric polymorphism arises from the second-order λ-calculus, and was discovered independently by Girard [35] and Reynolds [79].

above. This permits different instances of the type of f to be introduced for each separate occurrence of the variable. Furthermore, the type system is extended to include the two typing rules Gen and Inst, given as

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma}\text{Gen}(\alpha \text{ not free in } \Gamma) \qquad \frac{\Gamma \vdash M : \sigma \quad \sigma > \tau}{\Gamma \vdash M : \tau}\text{Inst}$$

In Damas and Milner [26], a type $\sigma \equiv \forall\alpha.\tau$ is said to have as *generic instance* a type $\sigma' \equiv \forall\beta.\tau'$ if $\tau' \equiv \tau[\phi/\alpha]$ for some type ϕ and β is not free in σ. Then instantiation is denoted by $\sigma > \sigma'$.

Wright has already shown how his resource-aware type system can be extended to include let-polymorphism. In Wright's let-polymorphic type system, binding takes place over arrow variables as well as type variables, producing types such as

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

for the λ-term $\lambda x.\lambda y.xy$. Specialisation of this type takes place over arrow and type variables to produce types such as $(\sigma \Rightarrow \tau) \Rightarrow \sigma \Rightarrow \tau$ and $(\sigma \twoheadrightarrow \tau) \Rightarrow \sigma \twoheadrightarrow \tau$.

The introduction of Wright's approach into F_{rc}, however, has the result that types inferred for λ-terms under the let-polymorphic system do not match types inferred for equivalent terms under F_{rc}. To see this, take as an example the term

$$let\ f = \lambda x.\lambda y.xy\ in\ \lambda x.K(fIx)(f(Kz)x)$$

The type of this term if we adopt Wright's approach would be

$$\sigma \xrightarrow{\text{S}} \sigma$$

The equivalent term, without the let-construct, is $\lambda x.K((\lambda x.\lambda y.xy)Ix)((\lambda x.\lambda y.xy)(Kz)x)$. Under F_{rc}, this term has type

$$\sigma \xrightarrow{\text{N}} \sigma$$

The difference is due to the restriction imposed on the resource use annotations of function types introduced by the Axiom rule to be N (see Section 5.1 above).

One solution to this difficulty is to not quantify over resource use. Under this proposal, the quantified type for $\lambda x.\lambda y.xy$ would be

$$\forall\alpha.\forall\beta.(\alpha\xrightarrow{\mathrm{N}}\beta)\xrightarrow{\mathrm{N}}\alpha\xrightarrow{\mathrm{N}}\beta$$

Although the λ-term is clearly strict in its first (functional) parameter, the resource use is N in order to comply with the restriction that function types introduced as axioms are annotated only with N. Hence, the λ-term

$$let\ f = \lambda x.x\ in\ \lambda z.fz$$

is typed as follows: let

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha \vdash f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha}}{f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha \vdash f:\sigma\xrightarrow{\mathrm{N}}\sigma}\,\text{Inst} \qquad \dfrac{}{x:\sigma^{\mathrm{S}}\vdash x:\sigma}}{f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha, z:\sigma^{\mathrm{N}}\vdash fz:\sigma}}{f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha\vdash \lambda z.fz:\sigma\xrightarrow{\mathrm{N}}\sigma}$$

be denoted by

$$\begin{array}{c}[P]\\ \vdots \\ f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha\vdash \lambda z.fz:\sigma\xrightarrow{\mathrm{N}}\sigma\end{array}$$

in

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\alpha^{\mathrm{S}}\vdash x:\alpha}}{\vdash\lambda x.x:\alpha\xrightarrow{\mathrm{S}}\alpha}}{\vdash\lambda x.x:\forall\alpha.\alpha\xrightarrow{\mathrm{S}}\alpha}\,\text{Gen} \qquad \begin{array}{c}[P]\\ \vdots \\ f:\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha\vdash \lambda z.fz:\sigma\xrightarrow{\mathrm{N}}\sigma\end{array} \qquad \forall\alpha.\alpha\xrightarrow{\mathrm{S}}\alpha\subseteq\forall\alpha.\alpha\xrightarrow{\mathrm{N}}\alpha}{\vdash let\ f=\lambda x.x\ in\ \lambda z.fz:\sigma\xrightarrow{\mathrm{N}}\sigma}\,\text{Let}$$

In this example, important strictness information concerning the term associated with the let-bound variable *f* has been lost, and as a consequence, we are unable to deduce that the function is strict in the parameter *z*.

To circumvent this problem, we allow a limited form of quantification for resource use variables. In addition, the quantification is *bounded*, using an approach based on bounded quantification of type variables (see Cardelli and Wegner [17] for a discussion of universal and bounded quantification). The type system F_r-*let* which incorporates let-polymorphism

$$\dfrac{}{\{\mathrm{S} \sqsubseteq i\} \cup \mathrm{A} \sqsubseteq I; (\Gamma\downarrow)^I, x : (\sigma\downarrow)^i \vdash x : \sigma}\ \text{Axiom}$$

$$\dfrac{R; \Gamma, x : \sigma^i \vdash M : \tau}{R; \Gamma \vdash \lambda x.M : \sigma \xrightarrow{i} \tau}\ \to \text{Intro}$$

$$\dfrac{R; \Gamma \vdash M : \sigma \xrightarrow{i} \tau \qquad R; \Lambda \vdash N : \sigma' \qquad R \vdash \sigma' \subseteq \sigma}{R; \Gamma + \Lambda^{\times i} \vdash MN : \tau}\ \to \text{Elim}$$

$$\dfrac{R; \Gamma \vdash M : \sigma}{R; \Gamma \vdash M : \forall\alpha.\sigma}\ \text{Gen}_\text{t}\ (\alpha \text{ not free in } \Gamma) \qquad \dfrac{R; \Gamma \vdash M : \sigma \qquad \sigma > \sigma'}{R; \Gamma \vdash M : \sigma'}\ \text{Inst}_\text{t}$$

$$\dfrac{R, u \sqsubseteq i; \Gamma \vdash M : \sigma}{R; \Gamma \vdash M : \forall u \sqsubseteq i.\sigma}\ \text{Gen}_\text{u} \qquad \dfrac{R; \Gamma \vdash M : \forall u \sqsubseteq i.\sigma \qquad R \vdash u \sqsubseteq v}{R; \Gamma \vdash M : \sigma[v/i]}\ \text{Inst}_\text{u}$$

$$\dfrac{R; \Gamma, x : \sigma^i \vdash M : \tau \qquad R; \Lambda \vdash N : \sigma' \qquad R \vdash \sigma' \subseteq \sigma}{R; \Gamma + \Lambda^{\times i} \vdash \mathit{let}\ x\ =\ N\ \mathit{in}\ M : \tau}\ \text{Let}$$

Figure 5.3: F_r-*let*: polymorphic resource-aware type inference

with bounded quantification over use variables is presented in Figure 5.3. The first new feature of this system is that variables are introduced in the Axiom rule with variables in place of constants for their resource use annotations. In the axiom rule, the base of typing hypotheses is now $\Gamma^I, x : \sigma^i$, where I indicates the variables annotating the typing hypotheses in Γ. (Note that the resource use annotations in functional types introduced using the Axiom rule are still N.) The second new feature is that typing sequents are now of the form

$$R; \Gamma \vdash M : \sigma$$

where R is a *resource coercion set*, containing orderings $u \sqsubseteq u'$ between resource use values and/or variables. In the Axiom rule, we introduce a resource coercion set as $\{\mathrm{S} \sqsubseteq i\} \cup \mathrm{A} \sqsubseteq I$, where I is the set of variables annotating the typing hypotheses in a base and

$$A \sqsubseteq I = \{\mathrm{A} \sqsubseteq j | j \in I\}$$

In other words, the variable annotating the typing hypothesis for which the Axiom is a tautology is constrained to be a value of S or above in the resource use domain, i.e., a constant u such that $\mathrm{S} \sqsubseteq u$, and for the variables annotating the hypotheses in Γ, i.e., those variables in the set I, they are constrained to be A or above according to the resource use domain. Under this type system, F_r-*let*, the λ-term *let* $f = \lambda x.x$ *in* $\lambda z.fz$ receives the type $\sigma \xrightarrow{\mathrm{S}} \sigma$. However, $F_r -$ *let* still loses information about resource use in certain cases, as shown in the following example.

Example 5.4 *The term*

$$let\ f = \lambda x.\lambda y.xy\ in\ \lambda x.f(\lambda a.a)x$$

is typed as $\sigma \xrightarrow{\mathrm{N}} \sigma$ *as follows. Let*

$$\tau \equiv (\alpha \xrightarrow{\mathrm{N}} \beta) \xrightarrow{i} \alpha \xrightarrow{\mathrm{N}} \beta$$

and

$$\tau' \equiv (\alpha \xrightarrow{\mathrm{N}} \beta) \xrightarrow{\mathrm{S}} \alpha \xrightarrow{\mathrm{N}} \beta$$

Then let the deduction of the sub-term $\lambda x.\lambda y.xy$*,*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{S} \sqsubseteq i\}; x : \alpha \xrightarrow{\mathrm{N}} \beta^i \vdash x : \alpha \xrightarrow{\mathrm{N}} \beta}\,\text{Axiom} \quad \dfrac{}{\emptyset; y : \alpha^{\mathrm{S}} \vdash y : \alpha}\,\text{Axiom}}{\{\mathrm{S} \sqsubseteq i\}; x : \alpha \xrightarrow{\mathrm{N}} \beta^i, y : \alpha^{\mathrm{N}} \vdash xy : \beta}\,\to \text{Elim}}{\{\mathrm{S} \sqsubseteq i\}; x : \alpha \xrightarrow{\mathrm{N}} \beta^i \vdash \lambda y.xy : \alpha \xrightarrow{\mathrm{N}} \beta}\,\to \text{Intro}}{\{\mathrm{S} \sqsubseteq i\}; \vdash \lambda x.\lambda y.xy : (\alpha \xrightarrow{\mathrm{N}} \beta) \xrightarrow{i} \alpha \xrightarrow{\mathrm{N}} \beta}\,\to \text{Intro}}{\{\mathrm{S} \sqsubseteq i\}; \vdash \lambda x.\lambda y.xy : \forall\alpha.\forall\beta.\tau}\,\text{Gen}_{\mathrm{t}}}{\emptyset; \vdash \lambda x.\lambda y.xy : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau}\,\text{Gen}_{\mathrm{u}}$$

be denoted by

$$\begin{array}{c} [P] \\ \vdots \\ \emptyset; \vdash \lambda x.\lambda y.xy : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \end{array}$$

and let the deduction of the sub-term $\lambda z.f(\lambda a.a)z$ *(where some rule labels have been omitted for purposes of presentation),*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash f : \forall\alpha.\forall\beta.\tau}}{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash f : \forall\alpha.\forall\beta.\tau[\mathrm{S}/i]}\,\mathrm{Inst_u}}{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash f : \tau'}\,\mathrm{Inst_t} \quad \dfrac{\dfrac{}{a : \sigma^{\mathrm{S}} \vdash a : \sigma}}{\vdash \lambda a.a : \sigma \xrightarrow{\mathrm{S}} \sigma} \quad \sigma \xrightarrow{\mathrm{S}} \sigma \subseteq \sigma \xrightarrow{\mathrm{N}} \sigma}{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash f(\lambda a.a) : \sigma \xrightarrow{\mathrm{N}} \sigma} \qquad \dfrac{}{z : \sigma^{\mathrm{S}} \vdash z : \sigma}}{\dfrac{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau, z : \sigma^{\mathrm{N}} \vdash f(\lambda a.a)z : \sigma}{f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash \lambda z.f(\lambda a.a)z : \sigma \xrightarrow{\mathrm{N}} \sigma}\,\rightarrow \mathrm{Intro}}$$

be denoted by

$$\begin{array}{c} [Q] \\ \vdots \\ f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash \lambda z.f(\lambda a.a)z : \sigma \xrightarrow{\mathrm{N}} \sigma \end{array}$$

in the typing

$$\dfrac{\begin{array}{c} [P] \\ \vdots \\ \emptyset; \vdash \lambda x.\lambda y.xy : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \end{array} \quad \begin{array}{c} [Q] \\ \vdots \\ f : \forall \mathrm{S} \sqsubseteq i.\forall\alpha.\forall\beta.\tau \vdash \lambda z.f(\lambda a.a)z : \sigma \xrightarrow{\mathrm{N}} \sigma \end{array}}{\emptyset; \vdash \mathit{let}\ f = \lambda x.\lambda y.xy\ \mathit{in}\ \lambda z.f(\lambda a.a)z : \sigma \xrightarrow{\mathrm{N}} \sigma}\,\mathrm{Let}$$

5.4 Related work

The type systems presented in this chapter are closely related to those of Wright [93] and Baker-Finch [3] [4] (see Section 2.6, Chapter 2, for an overview of these type systems).

In [93], Wright considers three type systems, for Curry-style type inference, an extension to let-polymorphism, and for intersection types. Only the last of these is able to type all Curry-typable λ-terms for reasons discussed in Section 5.1.1 above. Baker-Finch's system, on the other hand, is able to type all Curry-typable λ-terms. However, this is at the expense of expressiveness, for the same reason that the correspondence between resource use inferred in F_r and resource use in λ-terms is only an approximation, not an equivalence.

The correctness of resource use inference in both Wright's and Baker-Finch's type systems is due to the proofs of the soundness and completeness of the type system, in which the semantics of types are extended with neededness to provide a semantics of resource use (in the form of different function arrows). We argue, however, that soundness and completeness do not constitute a sufficient proof of correctness of resource use inference, since while together they show that the type system is *consistent* with respect to the interpretation, they do not show that the information inferred by the type system corresponds to the reduction behaviour of λ-terms.

In [94], Wright and Baker-Finch discuss an extension to Wright's intersection-style type system to capture precise sharing information. In this system, the domain of resource use annotations is $\mathbb{N}$, the domain of natural numbers. However, as Wright comments in [93], unification in an implementation of such a type system is likely to be undecidable because of its reduction to Hilbert's 10th problem.

A similar type system is outlined by Bierman in [10] which uses the domain of annotations reproduced in Section 3.4.2 of Chapter 3. A sketch of how principal types in Bierman's work can be represented using sets of relations between resource use values is given. However, no further details about this type system are available. Baker-Finch has shown in [3] how Bierman's lattice can be incorporated into his method of resource-use typing by redefining the operators $+$ and $\times$ over the elements of the lattice.

Also related to the work presented in this chapter are non-standard program logics for strictness analysis , for example, Fuh and Mishra [58] and Jensen [54]. As mentioned in Section 1.2 of Chapter 1, non-standard program logics abstract type semantics to represent properties of interest, and infer information about those properties for programs using type inference rules. The key difference, therefore, between this approach and that of Wright and Baker-Finch derives from that between denotational and operational semantics. Non-standard program logics define properties from an abstraction of a denotational model of types. In Wright and Baker-Finch's work (and in the work presented in this chapter), however, the semantics of types are augmented with an operational semantics based on reduction, and it is within this augmented semantics that we are able to represent information about reduction behaviour.

5.5 Summary

In this chapter, we defined a type system capable to deriving resource use information about λ-terms. The type system was derived from the first-order resource-aware systems of intuitionistic logic discussed in Chapter 4. Properties of the type inference system were described, particularly Theorem 5.3, which stated the correspondence between the resource use for λ-terms inferred by the type system and the actual resource use of those terms. Since the basis for this relationship is the relationship between resource use in the logical system $\mathrm{IL}_{\mathbb{S}_f}$ and the typed λ-calculus, a similar problem occurs, i.e., that the correspondence is one of approximation rather than equivalence.

The soundness of the type system was also given. We also introduced a notion of resource coercion sets, based on the work of Mitchell [73] on type inference with subtyping, that allowed a succinct representation of principal types and of let-polymorphism.

Chapter 6

An algorithm for resource aware type inference

In this chapter, we give an algorithm to implement the type inference system F_{rc}, with resource coercion sets (described in Section 5.2.4 of Chapter 5). The framework for our implementation is the typing algorithm given by Leivant in [60], which is well suited to typing with subtypes and also forms the basis for Mitchell's type inference algorithms with subtyping in [73]. However, we must also perform unification over resource use expressions annotating function types and typing hypotheses within the unification of type expressions.

Wright has already shown that Boolean unification as described by Martin and Nipkow [68] [69] can be used to unify expressions involving his strict and constant function arrows, producing single most general unifiers of his arrow expressions. Since our resource use expressions are similar to Wright's arrow expressions, it seems appropriate to follow his approach in using boolean unification. However, we find that although the domains S and B (Bierman's domain) are, with the addition of a least element $\bot$, boolean lattices, the definitions of the operators $+$ and $\times$ over these domains are such that they do not satisfy the laws of boolean algebra. Furthermore, we find that the resource use domain L does not even satisfy the properties of a boolean lattice. We leave as an open problem the definition of a unitary unification procedure, i.e., one that produces a single most general unifier, for these domains, and instead simplify the S domain to a two-point strictness and non-strictness domain N in which the operators $+$ and $\times$ satisfy the laws of boolean algebra.

We describe a typing algorithm in the style of the type algorithm GA of Mitchell [73] for type inference with subtypes, together with auxiliary functions to infer resource use coercion sets. We show that the type algorithm is sound with respect to the type inference system F_{rc} parameterised over N.

6.1 Unification over resource use domains

In this section, we briefly review material on boolean unification from Martin and Nipkow [68] [69], and discuss its application to unification over expressions from the three resource use domains considered in Chapter 3, i.e., domains S, L, and Bierman's domain (here called B).

6.1.1 Distributive lattices, boolean algebras and unification in boolean rings

A *lattice* is defined as a pair $(P, \leq)$, where P is a set and $\leq$ is a partial order (a reflexive, transitive, anti-symmetric relation) over P, such that

$$\forall x, y \in P. x \wedge y \text{ and } x \vee y \in P$$

where $x \wedge y$ and $x \vee y$ are the least upper and greatest lower bounds induced by the ordering relation $\leq$ over P. Furthermore, a distributive lattice $L = (P, \leq)$ is defined as a lattice if it satisfies the following laws:

$$\forall x, y, z \in L. x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

and

$$\forall x, y, z \in L. x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

For example, the following lattice is distributive

$$\begin{tikzcd}
 & a & \\
b \arrow[ur] & & c \arrow[ul] \\
 & d \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

whereas the following lattice

$$\begin{tikzcd}
 & a & \\
b \arrow[ur] & & \\
 & & d \arrow[uul] \\
c \arrow[u] & & \\
 & e \arrow[ul] \arrow[uur] &
\end{tikzcd}$$

is not distributive. In fact, in Davey and Priestley [27], it is shown that if this lattice is a sub-lattice of any other lattice L', then L' is also non-distributive (Theorem 6.10, Chapter 6 of [27]). A special kind of distributive lattice is a *boolean lattice.* In [27], a lattice L is defined as a boolean lattice if L is distributive, L has 0 and 1, and each $a \in L$ has a unique complement a', i.e., an a' such that $a \vee a' = 1$ and $a \wedge a' = 0$.

A *boolean algebra* is a set B with distinguished elements 0 and 1, and operators $\wedge$, $\vee$, and $\neg$, which satisfies axioms concerning, for example, the commutativity, associativity, and distributivity of $\wedge$ and $\vee$. Moreover, a boolean algebra $(B, \wedge, \vee, \neg, 0, 1)$ can be represented as a *boolean lattice* (see Davey and Priestley [27] or Cooke and Bez [20]).

A *boolean ring* is a set B containing two distinguished elements 0 (zero element) and 1 (unit) with operations $\otimes$ and $\oplus$ and axioms defining their behaviour defined over B. These axioms induce the following rewrite system $\mathcal{R}$ under associative-commutative rewriting

$$\begin{array}{lcl@{\qquad\qquad}lcl}
0 \oplus x & \Rightarrow & x & x \oplus x & \Rightarrow & 0 \\
0 \otimes x & \Rightarrow & 0 & 1 \otimes x & \Rightarrow & x \\
x \otimes x & \Rightarrow & x & x \otimes (y \oplus z) & \Rightarrow & x \otimes y \oplus x \otimes z
\end{array}$$

The set of rules comprising $\mathcal{R}$ rewrite boolean ring terms into a *polynomial normal form*. Unification over boolean ring expressions attempts to solve equations of the form $f(x_1,\ldots,x_n) = g(x_1,\ldots,x_n)$, which as a consequence of the laws of boolean rings is equivalent to $f(x_1,\ldots,x_n) \oplus g(x_1,\ldots,x_n) = 0$. One unification method is due to Boole and is known as "successive variable elimination" (see Martin and Nipkow [69] for further discussion). A recursive algorithm in a functional notation of Boole's method is given by Martin and Nipkow in [69].

Expressions in a boolean algebra can be converted into boolean ring expressions and vice versa, as follows: for boolean algebra expressions into boolean ring expressions we use the following translation of operators

$$\begin{aligned} x \vee y &= x \oplus y \oplus x \otimes y \\ x \wedge y &= x \otimes y \\ \neg x &= x \oplus 1 \end{aligned}$$

and for boolean ring expressions into boolean algebra expressions, we use

$$\begin{aligned} x \oplus y &= (x \wedge \neg y) \vee (\neg x \vee y) \\ x \otimes y &= x \wedge y \end{aligned}$$

6.1.2 Unification over resource use domain S

Wright has shown in [93] that the set of arrow expressions induced over the set of arrow variables and the strict and constants arrows together with the operators $\wedge$ and $\vee$ can be represented as a boolean algebra $(B, \vee, \wedge, \neg, \nrightarrow, \Rightarrow)$, in which $\nrightarrow$ takes the role of 0 (zero) and $\Rightarrow$ takes the part of 1 (unit). and B represents the set of arrow expressions (the negation $\neg$ has no role in the arrow expressions inferred by the type systems described in [93]).

Since the resource use domain S is very close to Wright's set of ground arrow $\{\nrightarrow, \Rightarrow\}$, we apply Wright's approach in using boolean ring unification to unification over expressions induced over variables, values in S, and the operators $\times$ and $+$ (the definitions of $\times$ and $+$ over S are reproduced in Figure 6.1). However, as we will demonstrate below, the definition of the operators $+$ and $\times$ over S are such that they do not correspond to the $\wedge$

$\times$	S	A	N
S	S	A	N
A	A	A	A
N	N	A	N

$+$	S	A	N
S	S	S	S
A	S	A	N
N	S	N	N

Table 6.1: Operators over S

and $\vee$ operators, and that consequently, we are unable to define a boolean algebra on S using $+$ and $\times$.

In Wright's algebra of function arrows, $\not\rightarrow$ and $\Rightarrow$ take the roles of 0 and 1, respectively. Similarly, we might take A and S to be 0 and 1, respectively. Also, we might assume that $+$ and $\times$, which play similar roles to $\vee$ and $\wedge$ in Wright's type system, can be used as operators in the boolean algebra on S.

Two of the laws of boolean algebra are

$$\begin{aligned} x \vee \neg x &= 1 \\ x \wedge \neg x &= 0 \end{aligned}$$

For $x \equiv$ A, the complement $\neg x$ is S, and for $x \equiv$ S, $\neg x$ is S. However, there is no suitable complement for N. In fact, by examining the definition of $+$ and $\times$, we see that there is no value in the lattice corresponding to $\neg$N such that the equations

$$\begin{aligned} \mathrm{N} \vee \neg\mathrm{N} &= \mathrm{S} \\ \mathrm{N} \wedge \neg\mathrm{N} &= \mathrm{A} \end{aligned}$$

are satisfied. Even if we extend S with a least element $\perp \equiv \mathrm{S} \sqcap \mathrm{A}$ so that it is properly a boolean lattice, the same problem occurs (even with different values from S taken to be 0 and 1).

We therefore conclude that we are unable to directly define a boolean algebra from S with operators $+$ and $\times$. In Martin and Nipkow [69], the possibility of different operators over a set in terms of the boolean ring operators is discussed. It remains an open problem

$$\begin{tikzcd} \mathrm{N} \\ \mathrm{S} \arrow[u] \end{tikzcd}$$

Figure 6.1: Resource use domain $\mathbb{N}$

$\times$	S	N
S	S	N
N	N	N

$+$	S	N
S	S	S
N	S	N

Table 6.2: Operators over $\mathbb{N}$

whether or not $\times$ and $+$ over $\mathbb{S}$ can be translated in this way. Siekmann's survey of unification theory [84] shows that, generally, in the presence of associative and commutative operators, unification is not unitary but produces a finite set of most general unifiers. Wand has described a type inference algorithm in [91] for record types in which unification is not unitary, but is finite. It may be possible, therefore, to develop a unification procedure for resource use expressions over $\mathbb{S}$ which is finitary, though not unitary.

We also note that, since $\mathbb{S}$ is a sub-lattice of $\mathbb{B}$ (Bierman's lattice of resource use annotations), that similar problems concerning boolean ring unification will apply to $\mathbb{B}$.

Another solution to the problem we face in unifying over $\mathbb{S}$ is to simplify the domain so that the definitions of $\times$ and $+$ satisfy the laws of boolean algebra. Figure 6.1 presents a simplification of $\mathbb{S}$ to domain $\mathbb{N}$ which describes only strictness and non-strictness. In this domain, the domain point A, representing absence, is subsumed into the non-strict domain point N. The definitions of $\times$ and $+$ over $\mathbb{N}$ are given in Table 6.2.

In this resource use domain, the roles of the distinguished constants 0 and 1 are taken by N and S respectively. Also, $\neg\mathrm{N} = \mathrm{S}$ and $\neg\mathrm{S} = \mathrm{N}$. Hence, the laws

$$\begin{aligned} x \vee \neg x &= 1 \\ x \wedge \neg x &= 0 \end{aligned}$$

are satisfied when $\vee$ is interpreted as $+$ and $\wedge$ as $\times$. For example, we have

$$\begin{aligned} \mathrm{N} + \neg\mathrm{N} &= \mathrm{S} \\ \mathrm{N} \times \neg\mathrm{N} &= \mathrm{N} \end{aligned}$$

Given that $(N, \times, +, \mathrm{N}, \mathrm{S})$ is a boolean algebra we can use the unification algorithm given by Martin and Nipkow for unification in boolean rings (see Section 6.1.1 above). We will denote this algorithm by $\mathrm{RUNIFY}_{\mathsf{N}}$, given as

```
$\mathrm{RUNIFY}_{\mathsf{N}}(f(x_1, \ldots, x_n)$
  $=$ *if* $f(x_1, \ldots, x_n) =_B \mathrm{N}$
    *then id*
    *else let* $G = \mathrm{RUNIFY}_{\mathsf{N}}(f(\mathrm{N}, x_2, \ldots, x_n) \times f(\mathrm{S}, x_2, \ldots, x_n))$
    *in*
    $G[((f(\mathrm{N}, G(x_2, \ldots, x_n)) + f(\mathrm{N}, G(x_2, \ldots, x_n)) + \mathrm{S}) \times x_1 + f(\mathrm{N}, G(x_2, \ldots, x_n)))/x_1]$
```

6.1.3 Unification over resource use domain L

Even if resource use expressions over S could be unified using Wright's technique using boolean unification, we find that boolean unification cannot be used over the resource use domain L. We reproduce this domain in Figure 6.2.

When we add a least point $\perp$ to the lattice, the structure of L is clearly isomorphic to the lattice in Section 6.1.1 above that stands as a canonical example of a non-distributive lattice. Therefore, we can expect not to be able to define a boolean algebra over this domain. However, since the operators $\times$ and $+$ over L are associative and commutative, we may still be able to unify to produce a finite set of most general unifiers in the manner suggested in Section 6.1.2 above for resource use domain S.

$$\begin{tikzcd}
 & N & \\
S \arrow[ur] & & A \arrow[ul] \\
L \arrow[u] & &
\end{tikzcd}$$

Figure 6.2: Resource use domain L

6.2 A type inference algorithm for resource use

In this section, we describe a resource-aware type inference algorithm in the style of Mitchell's GA algorithm [73] for the simply typed λ-calculus. The strategy followed by this algorithm is to type λ-terms with the minimum of context, i.e., without a type environment parameter assigning types to free variables and a type for the term determined by the surrounding context in which it appears. (By contrast, Milner's W algorithm passes a type environment parameter.) This makes the algorithm conceptually simpler and makes it more applicable to type systems that use type coercion[1].

6.2.1 Unification

In order to produce a single most general unifier for resource use expressions, we use the restricted domain N for strictness and non-strictness and the unification algorithm RUNIFY$_{\mathsf{N}}$.

We follow a two-stage approach to unification over type expressions with resource use annotations. Given a set of equations involving resource type expressions of the form $E \equiv \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$, we first unify E using the standard unification algorithm,

[1]See Leivant [60] for a discussion of the advantages and disadvantages of different typing strategies.

called here UNIFY$_C$. In other words, UNIFY$_C$ finds a substitution of type expressions for type variables, if one exists, such that E is *Curry-unified*, defined as follows.

Definition 6.1 *Let E be a set of equations $\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$ where σ_i, τ_i are resource types. Then E is Curry-unified by a substitution S mapping type variables to type expressions iff*

$$\forall \sigma_i = \tau_i \in E. \Upsilon(S\sigma_i) = \Upsilon(S\tau_i)$$

(where Υ is as defined in Section 5.1.3).

The unification algorithm UNIFY$_C$ is presented in Figure 6.3. Note that, given two functional resource types $\sigma_1 \xrightarrow{u} \tau_1$ and $\sigma_2 \xrightarrow{v} \tau_2$, UNIFY$_C$ does *not* unify the resource use expressions u and v. This algorithm is in the style of Robinson's original unification algorithm described in [80], which is known to be exponential in both time and space complexity. More efficient unification algorithms, such as Paterson and Wegman's linear algorithm [76], have been discovered, which employ special graph-based data structures and sharing via pointers to increase efficiency. (See Knight's survey on unification in [57] for a discussion on implementations of unification algorithms.)

Given an equation $\sigma = \tau$ which is Curry-unified, we define the algorithm UNIFY$_R$, inductive over the structure of σ and τ, which applies the algorithm RUNIFY$_{\mathsf{N}}$ given in Section 6.1.2 above to unify over resource-use expressions. Note that as a consequence of Curry-unification, the structure of σ and τ match and hence we can pursue a joint induction over the two type expressions.

The overall unification algorithm over type expressions is defined in terms of UNIFY$_C$ and UNIFY$_R$ as follows:

$$\begin{aligned}
\text{UNIFY}(\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}) = {} & \text{UNIFY}_R(\{S\sigma_1 = S\tau_1, \ldots, S\sigma_n = S\tau_n\}) \\
& \text{where} \\
& S = \text{UNIFY}_C((\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\})
\end{aligned}$$

```
$\mathrm{UNIFY}_C(\alpha, \tau) = $ if $\alpha = \tau$
                    then $\mathit{IdSubst}$
                    else if $\alpha \in \mathit{typevars}(\tau)$
                        then $\bot$
                        else $[\tau/\alpha]$
$\mathrm{UNIFY}_C(\tau, \alpha) = \mathrm{UNIFY}_C(\alpha, \tau)$
$\mathrm{UNIFY}_C(\sigma_1 \xrightarrow{u} \tau_1, \sigma_2 \xrightarrow{v} \tau_2) = $ let $S = \mathrm{UNIFY}_C(\sigma_1, \sigma_2)$
                                        in let $T = \mathrm{UNIFY}_C(S\tau_1, S\tau_2)$
                                          in $T \circ S$
```

Figure 6.3: Algorithm UNIFY_C for unification over types

The separation of the unification process has the advantage of modularising distinct unification algorithms. This is used to advantage in the definition of the inference algorithm $\mathcal{R}$, where unification of the type of the function and the function type created from the argument type and a fresh type variable uses only UNIFY_C.

6.2.2 Algorithm $\mathcal{R}$ for resource-aware type inference

In the type system F_{rc}[2], types of variables in the type environment are restricted in the sense that resource use in functional types is restricted to N, which represents the top point $\top$ of the resource use domain. In the type inference algorithm, we must maintain this restriction. The fact that functional types in the type environment of a typing contain only N resource use values (a property we shall define below as $\top$-*conformance*) has

[2]To keep the presentation reasonably simple, we do not consider the let-polymorphic extension to F_{rc}, described in Section 5.3 of Chapter 5. However, extending the algorithm to perform let-polymorphic typing should not pose significant problems, other than to ensure that the type inferred for the local definition and the type of the let-bound variable are ordered with respect to $\sqsubseteq$.

$$
\begin{aligned}
&\mathrm{UNIFY}_R(\alpha, \alpha) = \mathit{Idsubst} \\
&\mathrm{UNIFY}_R(\sigma_1 \xrightarrow{u} \tau_1, \sigma_2 \xrightarrow{v} \tau_2) = \text{let } S = \mathrm{UNIFY}_R(\sigma_1, \sigma_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{in let } T = \mathrm{UNIFY}_R(S\tau_1, S\tau_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } \mathrm{RUNIFY}_{\mathsf{N}}((T \circ S)u, (T \circ S)v) \circ T \circ S
\end{aligned}
$$

Figure 6.4: Algorithm UNIFY_R

$$
\begin{aligned}
&\mathrm{ATOMIZE}(\alpha, \alpha) = \{\} \\
&\mathrm{ATOMIZE}(\sigma_1 \xrightarrow{u} \tau_1, \sigma_2 \xrightarrow{v} \tau_2) = \{u \sqsubseteq v\} \cup \mathrm{ATOMIZE}(\sigma_2, \sigma_1) \cup \mathrm{ATOMIZE}(\tau_1, \tau_2)
\end{aligned}
$$

Figure 6.5: Algorithm ATOMIZE

certain consequences for type inference. For example, any functional type inferred from such a type environment only has resource values below N in the resource use domain for functional arrows present in the type as a result of applying the inference rule $\rightarrow$ Intro, i.e., for first-order resource use. A second consequence concerns the typing of applications MN: if N has a functional type, then the resource use annotating the arrow constructors within the type must be updated to N in order to unify with the type of M.

The typing algorithm also makes use of algorithm ATOMIZE to extract resource use values from type coercions and produce a resource coercion set. The algorithm is defined in Figure 6.5.

The resource-aware type inference algorithm $\mathcal{R}$, presented in Figure 6.6 takes as argument a λ-term to be typed, and returns a triple (R, A, σ), such that (R, σ) is the principal type of M in the context of the types of free variables in the type environment A. The

```
$\mathcal{R}(x)$
        $= (\{\mathrm{S} \sqsubseteq i\}, \{x : \alpha^i\}, \alpha)$
          where $\alpha$ is a fresh type variable
          and $i$ is a fresh resource use variable

$\mathcal{R}(\lambda x.M)$
        $=$ let $(R', A', \tau) = \mathcal{R}(M)$
          in
            if $x : \sigma^i \in A$
            then
                $(R', A - \{x : \sigma^i\}, \sigma \xrightarrow{i} \tau)$
            else
                $(R' \cup \{\mathrm{A} \sqsubseteq j\}, A', \alpha \xrightarrow{j} \tau)$
          where $\alpha$ is a fresh type variable
          and $j$ is a fresh resource use variable
$\mathcal{R}(MN)$
        $=$ let $(R_1, A_1, \sigma) = \mathcal{R}(M)$
            $(R_2, A_2, \phi) = \mathcal{R}(N)$
            $S = \mathrm{UNIFY}(\{\tau_1 = \tau_2 | x : \tau_1^m \in A_1, x : \tau_2^n \in A_2\})$
            $T = \mathrm{UNIFY}_C(\{\sigma = \phi \downarrow \xrightarrow{\mathrm{N}} \alpha\}) \circ S$
            $\psi_1 \xrightarrow{u} \psi_2 = T\sigma$
            $R_3 = \mathrm{ATOMIZE}(T\sigma \subseteq T(\phi \xrightarrow{u} \alpha))$
          in
            $(R_1 \cup R_2 \cup R_3, TA_1 + (TA_2)^{\times u}, T\alpha)$
          where $\alpha$ is a fresh type variable
```

Figure 6.6: Algorithm $\mathcal{R}$ for resource-aware type inference

algorithm is defined inductively over the structure of λ-terms.

Definition 6.2 *A type σ is said to be* $\top$-conformant *iff* $\top(\sigma)$, *where*

$$\begin{array}{lcl} \top(\alpha) & = & \textit{True} \\ \top(\sigma \xrightarrow{u} \tau) & = & \textit{if } u \equiv \mathrm{N} \\ & & \textit{then } \top(\sigma) \wedge \top(\tau) \\ & & \textit{else False} \end{array}$$

Moreover, a type environment A is said to be $\top$-conformant if $\forall x : \sigma^i \in A.\top(\sigma)$.

Hence, the types $\alpha \xrightarrow{\mathrm{N}} \beta$ and $(\alpha \xrightarrow{\mathrm{N}} \beta) \xrightarrow{\mathrm{N}} \beta$ are $\top$-conformant, while $\alpha \xrightarrow{\mathrm{S}} \alpha$ is not $\top$-conformant.

In Section 4.3 of Chapter 4, we introduced the $\downarrow i$ operator over propositions, defined as

$$\begin{array}{lcl} \alpha \downarrow & = & \alpha \\ (\sigma \xrightarrow{j} \tau) \downarrow & = & \sigma \downarrow \xrightarrow{\mathrm{N}} \tau \downarrow \end{array}$$

The following lemma shows that for any type σ, $\sigma \downarrow$ is $\top$-conformant.

Lemma 6.1 *Let σ be any type. Then $\sigma \downarrow$ is $\top$-conformant.*

Proof Directly from the definition of $\downarrow$.

□

The following definition defines as *first-order conformant* any type, which if functional, only has resource use annotations $u \sqsubseteq \mathrm{N}$ only for the top-level function arrows.

Definition 6.3 *A type σ is said to be* first-order conformant *iff* $foc(\sigma)$, *where*

$$\begin{array}{lcl} foc(\alpha) & = & \textit{True} \\ foc(\sigma \xrightarrow{u} \tau) & = & \top(\sigma) \wedge foc(\tau) \end{array}$$

Therefore, the type $(\alpha\xrightarrow{\mathbf{N}}\beta)\xrightarrow{\mathbf{S}}\gamma$ is first-order conformant, while $\alpha\xrightarrow{\mathbf{N}}(\alpha\xrightarrow{\mathbf{S}}\beta)\xrightarrow{\mathbf{S}}\beta$ is not. It is also the case that any $\top$-conformant type is first-order conformant.

The operator $+$ over type environments A_1 and A_2 is defined as

$$\{x : \sigma^{u+v} | x : \sigma^u \in A_1, x : \sigma^v \in A_2\} \cup (A_1 \bigtriangleup A_2)$$

where u and v range over resource use expressions and $\bigtriangleup$ is symmetric set difference. The $\times$ operator is defined as follows:

$$A^{\times i} = \{x : \sigma^{j \times i} | x : \sigma^j \in A\}$$

6.3 Soundness of resource-aware type inference

In this section, we show that the inference algorithm in Section 6.2.2 above is sound with respect to the type inference system presented in Section 5.3 of Chapter 5.

The following theorem and its proof demonstrate the soundness of the type inference algorithm $\mathcal{R}$.

Theorem 6.1 *If $\mathcal{R}(M) = (R, A, \sigma)$, then $R; A \vdash M : \sigma$ is a provable typing.*

Proof The proof proceeds by structural induction over the λ-term M to be typed.

Case 1: $\mathcal{R}(x) = (\{\mathsf{S} \sqsubseteq i\}, \{x : \alpha^i\}, \alpha)$. Directly from definition of Axiom rule in F_{rc}.

Case 2: $\mathcal{R}(\lambda x.M)$. By induction,

$$\mathcal{R}(M) = (R, A, \tau) \Rightarrow R; A \vdash M : \tau \text{ is provable}$$

If $x : \sigma^i$ is in A, then by the $\rightarrow$ Intro rule we can derive

$$R; A - \{x : \sigma^i\} \vdash \lambda x.M : \sigma \xrightarrow{i} \tau$$

Case 3: $\mathcal{R}(MN)$. By induction

$$\begin{aligned}\mathcal{R}(M) = (R_1, A_1, \sigma) &\Rightarrow R_1; A_1 \vdash M : \sigma \text{ is provable}\\ \mathcal{R}(N) = (R_2, A_2, \sigma) &\Rightarrow R_2; A_2 \vdash N : \phi \text{ is provable}\end{aligned}$$

Since T is a substitution of $\top$-conformant types for type variables (this is a result of the curry-unification of σ with the $\top$-conformant type $\phi \downarrow \xrightarrow{\mathsf{N}} \alpha$ composed with unification over A_1 and A_2 which are also $\top$-conformant), and unifies A_1 and A_2, then TA_1 and TA_2 are well-formed $\top$-conformant type environments. Also, since T is a curry-unifier of σ and $\phi \downarrow \xrightarrow{\mathsf{N}} \alpha$, then $\sigma \equiv \psi_1 \xrightarrow{u} \psi_2$, and also that R_3 is a well-formed resource coercion set. It follows that both

$$\begin{aligned}R_1 \cup R_2 \cup R_3; TA_1 \vdash M : T\sigma\\ R_1 \cup R_2 \cup R_3; TA_2 \vdash N : T\phi\end{aligned}$$

are provable. Therefore, by rule $\rightarrow$ Elim, we have

$$R_1 \cup R_2 \cup R_3; TA_1 + TA_2^{\times u} \vdash MN : T\alpha$$

$\square$

6.4 Summary

In this chapter, we have described an algorithm to implement the type inference system F_{rc} with resource coercion sets, described in Section 5.2.4 of Chapter 5. We have found significant difficulties in adapting Wright's method of boolean unification to the resource use domains S, L, and B. In the case of S and B, this was due to the definition of the $+$ and $\times$ operators, which we discovered did not conform to the definition of boolean algebra operators. Nor could we easily discover a translation in the style of that for boolean ring operators into boolean algebra operators presented in Section 6.1.1 above. In the case of L, the difficulty was more serious, since the shape of the domain was exactly that of the canonical non-distributive lattice.

Chapter 7

Summary and Conclusions

In this chapter, we provide a summary of the thesis, and assess its contributions. We also give some suggestions for future work.

7.1 Summary and assessment

In this thesis, we have studied resource use in the λ-calculus and in intuitionistic logic (which in this thesis referred to the implicational fragment of intuitionistic propositional logic) with the aim of deriving a type system in which it can be shown that the resource use of term variables inferred by the type system were properties of the variables in the term being typed. Previous work on resource-aware type systems by Wright and Baker-Finch has shown the soundness and completeness of type inference with respect to a type semantics that incorporates reduction information using needed redexes. However, we have argued that while this establishes the consistency of inference of annotations with respect to a given interpretation, it does not justify the inference of resource use for λ-terms.

In Chapter 3, we gave a first-order definition initially of a simple resource use domain of strictness, absence and non-strictness based on the concept of needed and head-needed redexes from Barendregt *at al* [7]. We also defined an inference system $\triangleright_S$, similar to the context analysis of Wadler and Hughes [90] but which requires reduction to head normal form, which infers the resource use of free variables in a λ-term. We demonstrated the soundness of this system and conjectured its completeness.

We then extended the definitions of neededness and head-neededness to *degrees* of neededness and head-neededness by measuring the number of descendants of redexes contracted on leftmost and head reduction paths. On this basis, we were able to define more sophisticated resource use domains, such as L which incorporated a linear resource use. Inference systems similar to $\rhd_{\mathsf{S}}$ were defined.

Chapter 4 dealt with resource use in intuitionistic logic. We defined analogous concepts of neededness and head-neededness, and consequently strict, absent, and non-strict resource use in intuitionistic logic, and gave a definition of a resource-aware system of intuitionistic logic IL_{S}, in which resource use values were attached to open hypotheses and annotated implication arrows. However, it was discovered that IL_{S} was not sound with respect to the definition of resource use, since it is able to infer stronger resource use values for hypotheses than is the case according to the semantics. Consequently, we defined a version of IL_{S}, called $\mathrm{IL}_{\mathsf{S}f}$, in which the resource use annotating implicative axioms is restricted to N. This system was shown to be sound with respect to resource use inference, in that it could not infer a more specific resource use than that given by the semantics.

We demonstrated the equivalence of our definitions of neededness and head-neededness with those for the λ-calculus under the Curry-Howard isomorphism. When it came to relating resource use in $\mathrm{IL}_{\mathsf{S}f}$ to that in the λ-calculus, however, we discovered that resource use in this system was weaker than in $\rhd_{\mathsf{S}}$. The problem lies with the coercion premise in the $\rightarrow$ App rule, which does not take account of more specific resource use in the derivation of the resource use of hypotheses in the conclusion. It would appear that Baker-Finch's system $R_{\supset}$ also suffers from this problem, since it also uses coercion as a premise in typing applications.

In Chapter 5, we derived a resource-aware type system from $\mathrm{IL}_{\mathsf{S}f}$ for λ-terms. We showed that in this system F_r, a resource use v inferred for a term variable in the type environment was related via the resource use domain ordering to the resource use u defined for the variable in the λ-term being typed, such that $u \sqsubseteq v$. As a consequence, the Subject Reduction theorem held only in a modified form in this system, i.e., that if $\Gamma \vdash M : \sigma$ and $M \twoheadrightarrow_{\beta} N$, then $\Gamma \vdash N : \sigma'$ such that $\sigma \subseteq \sigma'$ (where $\subseteq$ is an ordering relation over types induced by the ordering over resource use). The soundness of this type system was demonstrated. Also in this chapter, we gave representations for principal types and showed how let-polymorphism could be incorporated into the type system, using an adapted form of bounded quantification.

Chapter 6 discussed the implementation of resource-aware type inference. Wright has shown in [93] how unification over boolean rings can be used to unify type expressions with resource use information (in his case, function arrows for strictness and absence) by formulating a boolean algebra from the resource use values and the operators over them. We have shown that resource use domains and their operators only form a boolean algebra in very restricted cases, because of the definition of the operators. We gave a algorithm for typing λ-terms which implemented the type inference system F_{rc}, with resource coercion sets, over a restricted resource use domain N for strictness and non-strictness only, and demonstrated the soundness of this algorithm.

7.2 Further work

The following are areas where further work is required.

7.2.1 The semantics of higher-order resource use

The definition of resource use for the λ-calculus in Chapter 3 is first-order. A higher-order treatment of resource use is a subject for further work. This may prove difficult, as a result of the similarity between, for example, $\triangleright_{\mathsf{S}}$ and Wadler and Hughes' context analysis, and the known difficulty in extending backwards analyses such as context analysis to the higher-order case. Wright and Baker-Finch have both defined the neededness of an argument to a function in the context of all possible arguments, but a higher-order definition of resource use of a function in its parameter in the style of Section 3.2 in

Chapter 3, and an operational semantics in the style of $\rhd_{\mathsf{S}}$ is also required.

7.2.2 A resource-typed λ-calculus

Also left to further work is the investigation of a resource-typed λ-calculus. The resource-aware system of intuitionistic logic IL_{S} appears to be isomorphic to a typed λ-calculus in which the types of terms contain resource-use information. This area also seems related to λ-calculi with operators or annotations indicating strictness.

7.2.3 Definition of resource-use domains

In Chapter 6, we saw that the resource use domains S, L, and B, together with operators $+$ and $\times$ defined over them, could not be defined as boolean algebras. Despite the fact that both S and B were boolean lattices, the definition of $+$ and $\times$ over these domains did not conform to the definition of the boolean operators $\vee$ and $\wedge$. Furthermore, the domain L was isomorphic to a lattice known to be non-distributive, a key characteristic of boolean lattices (and hence, of boolean algebras). However, boolean ring unification may not be the only method for unifying expressions over these domains. Further work, therefore, would involve the investigation of other methods for unifying over these domains, either by finding translations of the operators $+$ and $\times$ into the boolean ring operators for those resource use domains which are boolean lattices, or by unification algorithms to find finite sets of most general unifiers.

7.2.4 Extensions to the type system F_r

In this thesis, the semantics of resource use and the resource-aware type system are developed only for the pure λ-calculus. However, if the type system is to be used for practical analysis of functional programming languages, then both the semantics of resource use and the type system should be extended to recursion, conditionals, data structures such as lists and tuples, and constants. Wright has already outlined suggestions in [93] on how to incorporate recursion, data structures and constants, including a conditional, into his type system, but does not consider the semantics in terms of reduction of head-needed redexes. Baker-Finch in [3] has shown in greater detail how recursion can be represented and has considered the semantics in terms of descendants of redexes that occur on the head reduction path. However, the method of typing is iterative, in order to find the limit, in terms of the ordering over type annotations. It is not clear how costly this may be, especially in those cases of functions with several arguments. Typing of tuple data structures, but not semantics, is also considered.

We should also investigate the implementation of let-polymorphism in F_r, which currently degrades the quality of the resource-use information inferred for λ-terms.

7.2.5 Completeness of F_r

The type system F_r has been shown to be sound, but lacks a proof of completeness. Wright has already shown the completeness of his reduction type system in [93] using Plotkin's model of reduction [78], although Wright's proof relies on having established the soundness of the type system. For F_r, proving completeness may be difficult, primarily because the modified Subject Reduction theorem for F_r implies that types of λ-terms are not maintained by reduction, although they are still ordered. However, the fact that the types of a term across reduction steps are ordered means that we may be able to associate some minimal type (in terms of the ordering) with the equivalence class of the λ-term in a semantic model, which can then be used to establish completeness.

7.2.6 Relationship with Linear Logic

We should further look at the connection between the resource use in systems such as $\mathrm{IL}_{\mathsf{L}f}$ and resource control in Linear Logic, perhaps with a view to defining an efficient translation between λ-terms and terms in a linear term calculus. A standard translation exists based on the translation of propositions in intuitionistic logic into linear logic (see Girard [36], but this applies the "non-linear" modal ! operator to all propositions. It may prove possible to define a translation from propositions in $\mathrm{IL}_{\mathsf{L}f}$ into linear logic that produces some linearity in the result. We can then take advantage of implementations of linear term calculi that optimise space use. Other work in the area of translations of the λ-calculus into a linear logic-based term calculus includes Mackie [65], Maraist, Odersky, Turner, and Wadler [67]. Lincoln [63] [62], and Schellinx [82], have also studied translations of intuitionistic logic proofs into linear logic proofs with the aim of preserving their structure, i.e., translations that minimise the application of the modal operator ! in the translated proofs.

7.2.7 Relationship with other semantics

It is also interesting to consider the relationship between resource-aware type systems such as F_r, and the type systems of Wright and Baker-Finch, with other analyses such as abstract interpretation (Jensen has already commented on this point in his discussion of future work in his thesis [55]). Baker-Finch has recently shown a connection between resource-aware type systems and *projections* in [5]. Since the semantics of resource use embodied by $\rhd$ appear close to Hughes and Wadler's context analysis [90] which is based on projections, it seems possible to develop an understanding of the relationship between the two.

7.3 Conclusion

Resource-aware type inference is a promising field of research for static analysis. It combines the analysis of the reduction behaviour of programs with conventional static type-checking, thus obviating the need for further work within a compiler or interpreter. In this thesis, we have expanded upon the work of Wright and Baker-Finch into resource-aware type systems in order to establish a correspondence between the resource use inferred by the type system and the resource use of λ-terms.

For our definition of the semantics of resource use in the λ-calculus, and in intuitionistic logic, we have found that a precise equivalence between these two definitions does not exist, although we have been able to define a correspondence based on approximation. The main reason for this is the use of coercion between types, based on the ordering of the resource use domains, as a premise to the logical rule $\rightarrow$ Elim. While this premise is necessary to ensure, in terms of the set of typable terms, full expressiveness, while avoiding the need for intersection types, it has a severe impact on the power of the analysis of the reduction behaviour of those terms.

We conclude, therefore, that much more remains to be done, in terms of increasing the power of its analysis without sacrificing its efficiency, before resource-aware type inference will be of significant use to functional programming language development.

Bibliography

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, April 1993.

[2] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd., 1987.

[3] C. Baker-Finch. Relevance and contraction : A logical basis for strictness and sharing analysis. Technical Report ISE RR 34/94, University of Canberra, 1993.

[4] C. A. Baker-Finch. Relevant logic and strictness analysis. In *Workshop on Static Analysis, LaBRI, Bordeaux*, pages 221–228. Bigre 81–82, September 1992.

[5] C. A. Baker-Finch. Type theory and projections for higher-order static analysis. In *ACM Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Florida*, 25 June 1994. Technical Report 94/9, Dept. of Computer Science, University of Melbourne.

[6] H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. North Holland, 1984.

[7] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[8] N. Benton, G. Bierman, M. Hyland, and V. de Paiva. Term assignment for intuitionistic linear logic. Technical Report 262, University of Cambridge Computer Laboratory, August 1992.

[9] P. N. Benton. Strictness logic and polymorphic invariance. In *Proc. of 2nd International Symposium on Logical Foundations of Computer Science*. Springer-Verlag LNCS Vol. 620, 1992.

[10] G. Bierman. Type systems, linearity and functional languages, 1992. Slides from talk given at CLICS Workshop, Aarhus University, Denmark.

[11] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *FPCA '89 Conference Proceedings*, pages 26–38. ACM Press, 1989.

[12] G. L. Burn. A logical framework for program analysis. *Functional Programming, Glasgow 1990, Workshops in Computing*, pages 30–42, 1992.

[13] G. L. Burn. A logical framework for program analysis. Technical report, Imperial College, London, 1992.

[14] G. L. Burn, C. L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[15] G.L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *Proceedings of the 17th Symposium on Principles of Programming Languages*, pages 151–156. ACM SIGACT-SIGPLAN, January 1990.

[16] G.L. Burn. The abstract interpretation of functional languages. In G.L. Burn, S.J. Gay, and M.D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29–31 March 1993. Springer-Verlag Workshops in Computer Science.

[17] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[18] C. Clack and S. L. Peyton Jones. *Finding fixpoints in abstract interpretation*, chapter 11. In Abramsky and Hankin [2], 1987.

[19] C.D. Clack and S. L. Peyton Jones. Strictness analysis — a practical approach. In *Proceedings of FPCA Conference*, pages 35–49. ACM, Springer Verlag, September 1985. LNCS 201.

[20] D. J. Cooke and H. E. Bez. *Computer Mathematics.* Cambridge Computer Science Texts. Cambridge University Press, 1989. ISBN 0-521-27324-2.

[21] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[22] S. A. Courtenage and C. D. Clack. Analysing resource use in the λ-calculus by type inference. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, June 1994.

[23] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252. ACM, New York, NY, 1977.

[24] H.B. Curry, W. Craig, and R. Feys. *Combinatory Logic, volume 1.* North-Holland, Amsterdam, NL, 1958.

[25] H.B. Curry, J.R. Hindley, and J.P. Seldin. *Combinatory Logic, volume 2.* North-Holland, Amsterdam, NL, 1972.

[26] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, New York, 1982.

[27] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press, 1992. ISBN 0-521-36766-2.

[28] K. Davis. Analysing Functions by Projection-Based Backwards Abstraction. *Functional Programming, Glasgow 1990, Workshops in Computing*, pages 43–56, 1992.

[29] K. Davis and P. Wadler. Backwards strictness analysis : Proved and improved. In *Functional Programing, Glasgow 1989.* Springer-Verlag, 1990.

[30] K. Davis and P. Wadler. Strictness analysis in 4D. In *Functional Programing, Glasgow 1990.* Springer-Verlag, 1991.

[31] J. M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 3. D. Reidel, 1986.

[32] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, Berkshire, 1988. ISBN 0-201-19249-7.

[33] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 70:155–175, 1990.

[34] J. H. Gallier. *Logic for Computer Science.* Harper & Row, New York, 1986.

[35] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'èlimination des coupres dans l'analyse et la thèorie des types. In J. E. Fenstad, editor, *2nd. Scandanavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[36] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[37] J.-Y. Girard. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[38] B. Goldberg. Detecting sharing of partial applications in functional languages. In *Functional Programming and Computer Architecture*, pages 408–425. Springer-Verlag, 1987. LNCS Vol. 274.

[39] C. Hankin. *Lambda Calculi: A Guide for Computer Scientists.* Graduate Texts in Computer Science. Oxford University Press, 1994.

[40] C. L. Hankin and D. Le Métayer. Deriving algorithms from type inference systems. In *21th ACM Symposium on Principles of Programming Languages*, 1994.

[41] C. L. Hankin and D. Le Métayer. Lazy type inference for the strictness analysis of lists. In D. Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, volume 788 of *LNCS*. Springer-Verlag, 1994.

[42] C. L. Hankin and D. Le Métayer. A type-based framework for program analysis. In *Proceedings of the First Static Analysis Symposium*, volume 864 of *LNCS*. Springer-Verlag, 1994.

[43] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[44] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[45] R. Hindley. Curry's type rules are complete with respect to the F-semantics too. *Theoretical Computer Science*, 22:127–133, 1983.

[46] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[47] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[48] J. Hughes. *Analysing Strictness by Abstract Interpretation of Continuations*, chapter 4. In Abramsky and Hankin [2], 1987.

[49] J. Hughes. Backwards analysis of functional programs. *Partial Evaluation and Mixed Computation*, pages 187–208, 1988.

[50] J. Hughes. Compile-time analysis of functional programs. *Research topics in Functional Programming*, pages 117–153, 1990.

[51] J. Hughes and J. Launchbury. Towards relating forwards and backwards analyses. *Functional Programming, Glasgow 1990, Workshops in Computing*, pages 101–113, 1991.

[52] S. Hughes. *Static Analysis of Store Use in Functional Programs.* PhD thesis, Dept. of Computing, Imperial College, London, 1991.

[53] L. S. Hunt. *Abstract Interpretation: From Theory to Practice.* PhD thesis, Dept. of Computing, Imperial College, London, 1991.

[54] T. P. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional programmig Languages and Computer Architecture.* LNCS vol. 523. Springer Verlag, 1991.

[55] T. P. Jensen. *Abstract Interpretation In Logical Form.* PhD thesis, Imperial College, London, November 1992.

[56] J. W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127, 1980.

[57] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[58] T. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. of 4th ACM Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1989.

[59] J. Launchbury. *Projections Factorisations in Partial Evaluation.* PhD thesis, Dept. of Computing, University of Glasgow, 1989.

[60] D. Leivant. Polymorphic type inference. In *10th ACM Symposium on Principles of Programming Languages*, 1983.

[61] J.-J. Lévy. An algebraic interpretation of the $\lambda\beta k$-calculus and a labelled λ-calculus. In C. Böehm, editor, *Proceedings of the Symposium on λ-Calculus and Computer Science Theory, Rome.* LNCS Vol. 37, Springer Verlag., March 25-27th, 1975.

[62] P. Lincoln, A. Scedrov, and N. Shankar. Linearizing intuitionistic implication. In *Proc. 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 51–62. IEEE Computer Society Press, Los Alamitos, California, July 1991.

[63] P. D. Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Stanford University, August 1992.

[64] I. Mackie. Lilac. Master's thesis, Imperial College, Dept of Computing, September 1991.

[65] I. Mackie. *The Geometry of Implementation (Applications of the geometry of Implementation to language implementation).* PhD thesis, Dept. of Computing, Imperial College, London, 1994.

[66] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[67] J. Maraist, M. Odersky, D. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11th International Conference on the Mathematical Foundations of Programming Semantics*, April 1995.

[68] U. Martin and T. Nipkow. Unification in boolean rings. *Journal of Automated Reasoning*, 4:381–396, 1988.

[69] U. Martin and T. Nipkow. Boolean unification - the story so far. *Journal of Symbolic Computation*, 7:275–293, 1989.

[70] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth and Brooks/Cole, 1987.

[71] A. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, January 1982.

[72] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[73] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[74] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* Department of Computer Science, University of Edinburgh, Edinburgh, GB, 1981. PhD Thesis.

[75] M. Neuberger and P. Mishra. A precise relationship between the deductive power of forward and backward strictness analysis. In *ACM Conference on Lisp and Functional Programming*, 1992.

[76] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[77] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 184–201. ACM, New York, NY, 1989.

[78] G. Plotkin. A semantics for type checking. In *Theoretical Aspects of Computer Software.* LNCS Vol. 526, Springer Verlag, 1991.

[79] J. C. Reynolds. Towards a theory of type structure. In *Colloquium sur la Programmation*, volume 19 of *LNCS*. Springer-Verlag, 1974.

[80] J. A. Robinson. A machine orientated logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[81] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[82] H. Schellinx. *The noble art of linear decorating.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, 1994.

[83] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, Department of Computer Science, University of Copenhagen, October 1991.

[84] J. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[85] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[86] A. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Springer Verlag, Berlin, 1973.

[87] D. van Dalen. *Logic and Structure.* Springer-Verlag, 1989.

[88] P. Wadler. *Strictness analysis on non-flat domains (by Abstract Interpretation over Finite domains)*, chapter 12. In Abramsky and Hankin [2], 1987.

[89] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science.* Springer Verlag, 1993. Invited talk.

[90] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Functional Programming and Computer Architecture*, pages 385–407. Springer-Verlag, 1987. LNCS Vol. 274.

[91] M. Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, 1987. Corrigendum in *Proc. 3rd IEEE Symposium on Logic in Computer Science.*

[92] S. Wray. *Implementation and Programming Techniques for Functional Languages.* PhD thesis, University of Camdridge, 1986.

[93] D. A. Wright. *Reduction Types and Intensionality in the Lambda-Calculus.* PhD thesis, University of Tasmania, 1992.

[94] D. A. Wright and C. A. Baker-Finch. Usage analysis with natural reduction types. In *Workshop on Static Analysis.* Springer-Verlag LNCS Vol. 724, September 1993.