An approach to overloading with polymorphism

Stephen Blott

A Thesis
Submitted for the Degree of
Doctor of Philosophy
at the
Department of Computing Science,
University of Glasgow,
September 1991.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

One of the principal characterising features of a programming language is its type system. Many recent functional programming languages adopt a Hindley-Milner style type system facilitating parametric polymorphism.

One of the forms of polymorphism found most commonly in programming languages is overloading. Whereas one may consider the Hindley-Milner system an off-the-shelf package for parametric polymorphism, there is no similar uniformity in the approaches taken to overloading.

This thesis extends the standard Hindley-Milner system. A type system incorporating parametric polymorphism and overloading is presented both formally and informally, and it is shown to satisfy a principal type theorem. The Hindley-Milner type inference algorithm is extended for the new system. This algorithm is shown to be sound and complete.

The characteristic feature of parametric polymorphism is that the same code can be used at many different types. The corresponding characterisation rule for overloading is that different code is used at different types. As such, meaning is assigned to terms on the basis of their typing.

The semantics of the form of overloading described herein is assigned by means of a derivation to derivation translation scheme. This scheme is shown to be sound and, under certain well-defined conditions, coherent.

This approach to overloading is closely related to the lazy functional programming language Haskell's *type class* mechanism. Some discussion of matters related to the current system, and arising through that project, is given.

Preface and overview

Approaches to *ad-hoc* polymorphism, or *overloading*, are many and varied; what can one deduce from this alone? Since many, or possibly most, languages include overloading to some extent, it seems reasonable to assert that overloading of some form is regarded as a desirable programming language feature—not many language designers fail to include an overloaded equality operator in their language.

But what of the *variety* of approaches? Languages differ greatly in the power and extensibility of the overloading mechanisms they provide: Can users define their own overloaded operators? Can definitions be extended to operate over user defined types? If it is not possible to identify which meaning of an operator is intended, can overloading be propagated, or is it defaulted or erroneous? In summary, from "many and varied," it appears reasonable to conclude "desirable but frequently *ad-hoc*."

This thesis presents an approach to overloading within *one* increasingly popular type system. The type system is that discovered originally by Roger Hindley, developed independently by Robin Milner for ML, and studied subsequently by Luis Damas in his thesis. The Damas-Milner type system, as it is referred to throughout, is now extensively used within functional programming languages: it is the basis of languages such as Miranda, Orwell, LML, and most recently, Haskell.

The Damas-Milner type system utilises a type inference algorithm based on unification. The approach to *ad-hoc* polymorphism in this thesis is to embed overloading within the Damas-Milner type system. Thus the system is polymorphic; uses unification to drive the selection of the appropriate version of an overloaded operator; and allows user defined identifiers to be simultaneously, and largely orthogonally, polymorphic in both the parametric and *ad-hoc* senses. Further, an implementation strategy is given for the approach.

Chapter 2 presents the language informally along with the type system, and the type inference algorithm. A translation scheme is introduced suggesting an implementation strategy. The system is referred to as the OL system.

Chaper 3 describes the OL language and type system formally. In addition, the chapter presents the type inference algorithm and a formal translation into the Damas-Milner system.

Chapter 4 establishes some admissable rules. These follow directly from the formal rules and are conveniently abstracted here for subsequent use in proving more substantial results.

Chapter 5 shows the inference algorithm to be sound and complete. As such, the OL system satisfies a principal type theorem. Further, the translation semantics is shown to be sound with respect to the Damas-Milner system.

For technical reasons, the informal inference algorithm described in Chapter 2 and the formal algorithm described in Chapter 3 are different. Chapter 6 motivates this dichotomy and provides an analysis of when and how the two approaches can be unified.

The following Chapter, Chapter 7, describes a canonicalisation process for typing derivations. Apart from being useful in subsequent proofs, this process suggests a simplified version of the type system.

Since semantics is assigned to typed terms via a derivation to derivation translation scheme, it is necessary to ensure this translation scheme is coherent. Under certain conditions, the translation scheme is coherent and Chapter 8 motivates these conditions and presents the coherence result.

Chapter 9 presents the design of type classes in Haskell, a realisation of the approach. This implementation raises several problems and, naturally, solutions. Chapter 10 re-examines the approach to *ad-hoc* polymorphism with respect to the initial discussion set out in the survey, technical issues raised, and the experience gained with Haskell.

Acknowledgements

I am deeply indebted to many people for their technical help and advice in the preparation of this thesis: particularly, my supervisor Phil Wadler. For reading and commenting on an earlier version of this thesis, I am grateful to Mark Jones.

This work was funded by the Science and Engineering Research Council under a Quota Studentship awarded to the Computing Science Department, University of Glasgow.

At Glasgow I was lucky enough to work with the members of a very active functional programming group; I owe particular thanks to the two Johns, that is, John Launchbury and John Hughes. Further thanks is due to Kieran Clenaghan, Dave Harper, and others, in the department and outwith, who have helped me in the past few years.

Finally, I owe a great deal to Fiona, to my parents, and to all the members of Glasgow University Mountaineering Club.

Stephen Blott
September 1991

Contents

Chapter 1

Type systems—a selective survey

The type system associated with a language design trades-off expressiveness with reliability. A rigid type discipline ensures a certain large class of programming errors may be detected at compile time. On the other hand, a language with a loose type system, or no type system at all, may be considerably more expressive. A goal of modern type system research is to increase the expressive power of a language while maintaining the security associated with a strict type discipline.

This survey chapter reviews type systems associated mainly with extended lambda calculi examining the trade-offs that are made with the choice of type discipline. The early parts deal with well-understood branches of the lambda calculus and the latter systems represent more recent programming language research. Since this thesis presents a type system for overloading, particular emphasis is placed on systems addressing issues associated with *ad-hoc* polymorphism.

Why types? The type system adopted by a programming language influences greatly the character of that language: programmers are required to be aware of the types of the objects in their programs, and they will be confronted by problems which cannot be solved simply because the demands of the type system are too great. Further, compilers are required to enforce the type discipline, thus complicating the compilation process and, if the compiler is to be useful, requiring simple meaningful error messages to be generated from possibly subtle programming errors.

Many modern programming languages, however, adopt increasingly sophisticated type systems. The principal reasons for this are as follows.

Reliability. Type systems are adopted by programming languages to increase reliability. A common class of programming error can be detected at compile-time allowing more time and effort to be devoted to fixing *real* bugs, and more faith to be placed in the robustness of a final product.

Discipline. Type systems provide an implicit and unavoidable form of documentation. Programmers, or groups of programmers, employed on a task are

required to ensure that the interface between components of the entire system are type correct. This requires an awareness of the types of objects both in the small and in the large.

Modelling. Many type systems facilitate a higher-level style of programming. For example, C++ incorporates classes encouraging programmers to impose a particular object-oriented structure in their program designs.

On the other hand, the following disadvantage of type systems leads to their being inappropriate for certain programming tasks, typically systems-oriented applications.

Expressiveness. Type systems *approximate* the dynamic semantics of a programming language, they can be viewed as a form of abstract interpretation. In order for a well-typed program to be robust it is necessary that this approximation be conservative: type-correct programs may *not* result in a type error at runtime, whereas type-incorrect programs may possibly evaluate perfectly acceptably. This approximation can be too restrictive for certain programming tasks.

It should be noted that the adoption of programming languages with sophisticated type systems by the computing community in general is a slow process. The existence of a large programming base—programmers and programs—in languages such as Fortran and Cobol makes the adoption of new systems committing both financially and logistically. Over time, however, it appears that language features which have in the past been prevalent only in the academic community are being adopted more generally.

The rest of this chapter is organised as follows. Sections 1.1 through 1.3 present the untyped, simply typed, and polymorphically typed lambda calculi respectively. Section 1.4 discusses the Damas-Milner type system—this system is fundamental to subsequent chapters. Some significant theoretical properties of type systems are discussed in general in Section 1.5 providing some background to the technical results of Chapters 5 and 8.

Section 1.6 presents a brief categorisation of four significant forms of polymorphism. One of these, overloading, is then discussed in a more detailed and pragmatic way in Section 1.7 where a critique is given of the approaches taken by several programming languages.

1.1 The untyped lambda calculus

Alonzo Church developed the lambda calculus [Chu41*, Bar81*, HS86] to express functions as terms—as opposed to the more traditional model based on sets of argument and result pairs. Terms, including function terms, can thus be argued about at a single level and computed upon by a simple set of rewrite rules.

The lambda calculus (and the closely related combinatory logic) is in fact a collection of formal systems constructed in different ways for different applications: some include constants, some have type restrictions, etc. This section gives an outline of the basic *untyped* lambda calculus, the abstract syntax and semantics of which will recur throughout this chapter.

1.1.1 Terms, variables and substitutions

Terms of the pure untyped lambda calculus are constructed from the context-free grammar

$$e ::= x \mid (\lambda x.e) \mid (e_1\ e_2)$$

where x is drawn from some finite set of variable names, $(\lambda x.e)$ denotes a function abstraction, and $(e_1\ e_2)$ denotes an application. Brackets may be omitted from terms under the usual convention that application binds more tightly than λ and associates to the left.

A variable x is referred to as *bound* in a lambda term if it appears within the e part of a sub-term of the form $\lambda x.e$; a variable is referred to as *free* in a term if it appears in a non-bound position. Bound variables may be renamed. The set of free variables in a term is denoted by $fv(e)$. Notice that a variable may have both free and bound occurrences within a given term: for example, the first occurrence of x in $(f\ x\ (\lambda x.\ x))$ is free, and the other occurrences bound. The x immediately succeeding the lambda does not constitute a free or a bound occurrence.

Substitutions are functions mapping variables to terms, they are denoted by lists of term/variable pairs, e.g. $[e_1/x_1; \cdots; e_n/x_n]$. When a substitution is applied to a term, each free variable of the term appearing in the domain of the substitution is replaced simultaneously in the term by the result of applying the substitution to that variable. For example, if the substitution $[y/x;\ g/f]$ is applied to the term above, then the result is $(g\ y\ (\lambda x.\ x))$; notice that only the free occurrences of variables are substituted.

1.1.2 Reduction rules

Computation in the lambda calculus is defined by a set of rewrite or *reduction* rules. Each rule allows a sub-term matching a particular pattern to be replaced with a term defined by the applicable rule and the term at hand. The most significant reduction rule is the β-reduction rule representing function application.

A β-redex is any sub-term of the form $(\lambda x.e_1)\ e_2$, that is, the application of an explicit function abstraction to a specific argument. A β-reduction is the action of replacing a β-redex with a term representing the result of the application. Thus the β-reduction rule is

$$(\lambda x.e_1)\ e_2 \stackrel{\beta}{\Longrightarrow} [e_2/x]e_1$$

where the substitution $[e_2/x]e_1$ replaces each free occurrence of x in e_1 with e_2.

Other common rewrite rules include the η- and α-reduction rules. The η rule allows the reduction of redundant function abstraction and application,

$$\lambda x.\ (e\ x)$$

where it is required that x is not free in e; and the α rule allows the renaming of a bound variable,

$$\lambda x.e \stackrel{\alpha}{\Longrightarrow} \lambda y.\ [y/x]e$$

where it is required that y is not free in e.

Sequences of reductions using the β-, η- and α-reduction rules are labelled throughout with the general $\Longrightarrow$ notation: that is, $\Longrightarrow$ is the transitive closure of $\stackrel{\beta}{\Longrightarrow}$, $\stackrel{\eta}{\Longrightarrow}$ and $\stackrel{\alpha}{\Longrightarrow}$. Although the term *reduction* is used, no decrease in the lexical size of the term need be implied. Indeed, the term may grow considerably.

A term in *normal form* is one for which no reduction rule is applicable; it may be thought of as the result of a computation. Arbitrary terms do not necessarily have normal forms, that is, there need not be a reduction sequence from an arbitrary term to a term in normal form. Such terms correspond to non-terminating computations. Further, there cannot be an algorithm to detect such and only such terms: if there were, it would solve the halting problem.

Detailed presentation and discussions of the pure lambda calculus may be found in Barendregt's [Bar81*] and Hindley and Seldin's [HS86].

1.1.3 The Church-Rosser and Standardisation theorems

The Church-Rosser and Standardisation theorems are the main technical results rendering the lambda calculus an appropriate model of computation. The definition of reduction given thus far is non-deterministic: given an arbitrary term there may be many reduction rules applicable. The Church-Rosser theorem states, in effect, that this does not matter.

Church-Rosser Theorem. Given a term e, if $e \Longrightarrow e_1$ and $e \Longrightarrow e_2$ then there exists a term e' such that $e_1 \Longrightarrow e'$ and $e_2 \Longrightarrow e'$.

That is, there always exists a term which unites any two reduction sequences. Since termination is not guaranteed, this result does not suffice in itself.

A *normal order* reduction sequence is one in which the redex chosen for reduction is always the leftmost outermost redex. The Standardisation theorem is as follows.

Standardisation Theorem. Given e, if there exists a term e' such that $e \Longrightarrow e'$ and e' is in normal form, then there exists a normal order reduction sequence from e to e'.

That is, if there exists a reduction sequence from a term to a term in normal form, then a normal order reduction sequence will reach the term in normal form. Specifically, normal order reductions guarantee termination if termination is possible.[1]

Proofs of the theorems above are given in [HS86].

1.2 The simply typed lambda calculus

Adding simple types to the lambda calculus represents a considerable reduction in expressiveness: it is required that every term be statically and uniquely typed. Simple types are defined by the grammar

$$\tau ::= \chi \mid \tau \rightarrow \tau'$$

representing atomic types and function types respectively. All the types that can be expressed are monotypes, there is no polymorphism. Further, the syntax of lambda abstractions is augmented with a type annotation. That is, lambda abstractions are of the form $\lambda x_\tau.e$ indicating that the abstracted variable x is of type τ.

An *assumption set*, denoted A, is a set of bindings of types to identifiers:

$$x_1 : \tau_1;\ \ldots;\ x_n : \tau_n$$

It is required that the identifiers bound in an assumption set be distinct. In general, free variables are assigned types by assumption sets and bound variables are assigned types by the lambda term binding them. The notation A_x denotes the assumption set A with any binding of x removed.

Simple typing rules. A *typing judgement* is a formal statement of the form

$$A \vdash e : \tau$$

asserting that: "Under the assumptions in A, the term e has type τ". There are three rules in the calculus of types for deriving such judgements. The rules are given in Figure 1.1.

These, or similar, rules appear in many type systems and embody the typing of variables, and function abstraction and application. The **Taut** rule asserts that

[1] Strict languages, such as LISP [McC78*] and ML [HMM86], employ rightmost innermost, or *applicative order*, reduction since this is considered more efficient in general. However, examples which terminate under normal order reduction may loop under applicative order reduction.

$$\textbf{Taut}\quad \frac{}{A;\ x:\tau \vdash x:\tau}$$

$$\textbf{Abs}\quad \frac{A_x;\ x:\tau' \vdash e:\tau}{A \vdash (\lambda x_{\tau'}.e):\tau' \to \tau}$$

$$\textbf{Comb}\quad \frac{A \vdash e_1:\tau' \to \tau \quad A \vdash e_2:\tau'}{A \vdash (e_1\ e_2):\tau}$$

Figure 1.1: Simple typing rules

under the assumption $x:\tau$, the variable x has type τ; the **Abs** rule augments the environment with a binding of the identifier to the given type, types the body of the abstraction, and constructs a function type for the term; and the **Comb** rule ensures the type of the argument matches that in the function type and yields a typing at the result type.

This language is not as expressive as its untyped counterpart: no terms have been added, many have been removed. One significant term which has been disallowed is the fixed point combinator. In the untyped language, the fixed point combinator *fix* represents recursion[2].

$$\begin{aligned} \mathit{fix} &= \lambda x.(\lambda y.x\,(y\,y))\,(\lambda y.x\,(y\,y)) \\ \mathit{fix}\ f &= f\ (\mathit{fix}\ f) \\ &= f\ (f\ (f\ \cdots (f\ (\mathit{fix}\ f))\ \cdots)) \end{aligned}$$

However, the definition above is disallowed in the simply typed calculus: types cannot be assigned to the identifiers in such a way as to construct a well typed term.

A system is strongly normalisable if every term has a normal form, and any reduction sequence converges on that normal form.

Strong normalisability theorem. The simply typed lambda calculus is strongly normalisable.

[2] *fix* here denotes Curry's fixed point combinator. There are other terms with essentially the same property, for example Turing's combinator ZZ where $Z = \lambda z.\lambda x.x\,(z\,z\,x)$, all of which embody self application, the essence of recursion. As one might expect, *no* fixed point combinator can be defined in the simply typed calculus.

$$\textbf{Gen}\quad \frac{A \vdash e : \tau}{A \vdash \Lambda\alpha.e : \forall\alpha.\tau} \quad \alpha \notin fv(A)$$

$$\textbf{Spec}\quad \frac{A \vdash e : \forall\alpha.\tau}{A \vdash e[\tau'] : [\tau'/\alpha]\tau}$$

Figure 1.2: Polymorphic typing rules

That is, all programs terminate regardless of evaluation order. The proof of this, again, is given in [HS86].

Checking the type correctness of terms in the simply typed calculus is trivial: the type of every term can be generated from the types of the variables it contains.

1.3 The polymorphic lambda calculus

The polymorphic—or *second order*—lambda calculus is due independently to Jean-Yves Girard and John Reynolds [Gir72*, Rey74, Rey85]. It regains much of the expressibility lost with the introduction of types to the simply typed calculus while retaining strong normalisability. The key step is to allow variables and quantification in types.

Type variables. *Type variables* are placeholders for types in the same way program variables are placeholders for terms. The definition of types is extended as follows. Lower case Greek letters from the start of the alphabet denote type variables.

$$\tau ::= \alpha \mid \chi \mid \tau \to \tau' \mid \forall\alpha.\tau$$

That is, type variables are types, and type variables may be universally quantified over in types to create new types. The quantifier $\forall$ does for type variables what the quantifier λ does for program variables.

A type containing universal quantification is a polymorphic type and terms with polymorphic types are polymorphic. For example, the polymorphic identity function has type $\forall\alpha.\alpha \to \alpha$.

Free and bound occurrences of type variables are defined in the same way as they were for program variables, further, the same notation is used for substitutions of types for type variables; for example, $[\tau'/\alpha]\tau$.

Term syntax. Following the previous section, lambda terms retain the annotation of their variables with their type: this type may now, however, include type variables and quantification. Two new forms of term are added: introduction and elimination terms for polymorphism.

$$e ::= x \mid (\lambda x_\tau.e) \mid (e_1\ e_2) \mid \Lambda\alpha.e \mid e[\tau]$$

Terms of the form $\Lambda\alpha.e$ introduce polymorphism by abstracting over a type variable, and terms of the form $e[\tau]$ instantiate a polymorphic variable at type τ. For example, the polymorphic identity function is defined by the term

$$\Lambda\alpha.\lambda x_\alpha.\, x : \forall\alpha.\alpha \to \alpha;$$

whereas $\lambda x_\alpha.\, x$ defines the monomorphic identity function at type α. The polymorphic identity function may only be applied to a specific value, say the free variable 3 of type *Int*, after type specialisation.

$$(\Lambda\alpha.\lambda x_\alpha.\, x)[Int]\, 3 : Int$$

The rewrite rule for type variable abstraction and application is

$$(\Lambda\alpha.e)[\tau] \Longrightarrow [\tau/\alpha]e$$

illustrating the similarity between Λ and λ abstraction.

Polymorphic typing rules. The type rules for the polymorphic lambda calculus are the same as for the simply typed calculus with the addition of an introduction and an elimination rule for Λ abstractions. The two new rules are given in Figure 1.2. **Gen**, for generalisation, introduces Λ abstractions; and **Spec**, for specialisation, is the application rule for Λ abstractions.

The **Gen** rule requires a side condition asserting that α is not free in the type context. This ensures the semantic soundness of the system. For example, without this condition one can change the type of an object.

$$3 : Int \vdash (\Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\ \widehat{(\Lambda\alpha.x)}[\beta])[Int][Bool]\ 3 : Bool$$

The typing of the annotated Λ abstraction is invalid because α is free in the type context.

Discussion. The polymorphic lambda calculus is strongly normalisable: terms such as the fix-point combinator cannot be assigned valid types. However, the polymorphic system admits many terms that are invalid in the simply typed calculus.

The polymorphic lambda calculus has proved an excellent tool for research into both modern type systems and programming languages. Luca Cardelli and Peter Wegner use it as the basis of the language *Fun* [CW85] which combines many modern type features in a single language.

As with the simply typed calculus it is relatively easy to ensure a raw term of the polymorphic calculus is well formed. However, it is not known whether type inference is decidable or not.

Higher-order polymorphic programming. The polymorphic lambda calculus admits a rather unusual style of programming. The approach taken is to represent traditional data values with function terms [Rey85].

Data types are modelled with polymorphic function types, and values with functions of that type. For example, the Boolean type is modelled by the following polymorphic type.

$$Bool \stackrel{\text{def}}{=} \forall\alpha.\alpha \to \alpha \to \alpha$$

Boolean values are then modelled by selector functions.

$$true \stackrel{\text{def}}{=} \Lambda\alpha.\lambda x_\alpha.\lambda y_\alpha.\, x$$
$$false \stackrel{\text{def}}{=} \Lambda\alpha.\lambda x_\alpha.\lambda y_\alpha.\, y$$

Jon Fairbairn [Fai85] uses the polymorphic lambda calculus as the basis for *Ponder* making extensive use of this polymorphic programming style.

1.4 The Damas-Milner type system

The Damas-Milner type system is due independently to Roger Hindley [Hin69] originally, and later to Robin Milner [Mil78]. It eliminates the need to express all quantification and type specialisation explicitly, indeed, it eliminates the need to incorporate type information in terms at all. Hindley's work is on the lambda calculus and combinatory logic, while Milner's system was designed for the programming language ML. It is Milner's system, therefore, which is considered in this section. More specifically, Milner's system was studied in detail by Luis Damas [DaMi82, Dam85] and it is their system which is presented here. Hence, the system is referred to throughout as the Damas-Milner system and a full description can be found in Chapter 2 of Damas' thesis [Dam85].

The type information incorporated in terms of the polymorphic lambda calculus is undesirably detailed and unwieldy; some system whereby it can be omitted would be desirable for a practical programming language. The Damas-Milner system provides this at some cost in terms of expressibility. In particular, under a restriction on the form of terms, no type information need be included in a program. A type inference algorithm, utilising Robinson's unification algorithm [Rob65*], reconstructs type information if possible.

1.4.1 A restriction on polymorphism

Polymorphism is restricted to appear only at specific syntactic positions. These positions are represented by the polymorphic `let`.

Firstly, all references to types are removed from the terms: that is, type annotations are removed from lambda abstractions; and the constructs for explicit type abstraction and application are removed. Secondly, types are partitioned into two syntactic classes: simple types and type schemes.

$$\tau ::= \chi(\tau_1, \ldots, \tau_n) \mid \tau' \rightarrow \tau \mid \alpha$$
$$\sigma ::= \forall\alpha.\sigma \mid \tau$$

Notice that only type schemes, denoted by σ, may be polymorphic. Type constructors are also parametrised to allow more interesting types, such as $List(Int)$, to be discussed. Assumption sets now bind identifiers to type schemes (which may be types by projection).

The key difference is that now all quantifiers are at the top level of type schemes and simple types cannot be polymorphic.

1.4.2 The polymorphic `let`

The final syntactic change is to introduce a new form of expression term, the polymorphic `let`. The only bound variables which may be assigned polymorphic types are those introduced by `let` declarations. The new grammar for terms is therefore as follows.

$$e ::= x \mid e_1\ e_2 \mid \lambda x.e \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

Notice that type annotation is now omitted from lambda abstractions.

The meaning of a `let` term is defined by the following reduction rule

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \implies [e_1/x]e_2$$

where each occurrence of x in e_2 is replaced with e_1.

1.4.3 The Damas-Milner typing rules

Though the Damas-Milner typing rules repeat much of what is given in previous sections, they are included here in full since they are particularly significant for later chapters. The typing rules are given in Figure 1.3. The first five rules are similar to those presented previously with the following qualifications: the **Taut** rule is applicable to both types and type schemes; the **Spec** and **Gen** rules are

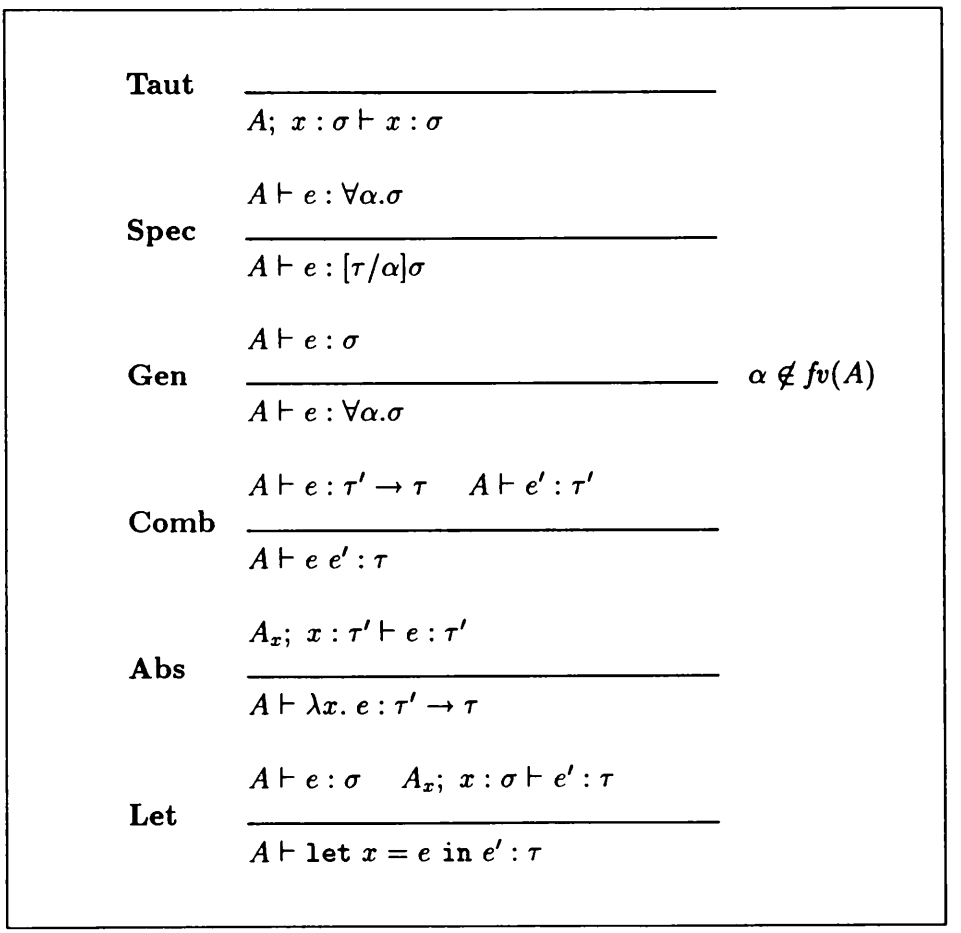

Figure 1.3: The Damas-Milner typing rules

$\mathcal{W}((\Gamma;\ x:\sigma),\ x) =$	If $\sigma = \forall\alpha_1.\cdots.\forall\alpha_n.\tau$, and S_β is the substitution $[\beta_1/\alpha_1,\cdots,\beta_m/\alpha_m]$ where $\beta_1,\cdots,\beta_n$ are fresh, then the typing is $(Id,\ S_\beta\tau)$.
$\mathcal{W}(\Gamma,\ (e_1\ e_2)) =$	Make the recursive calls $(S_1,\tau_1) = \mathcal{W}(\Gamma, e_1)$ $(S_2,\tau_2) = \mathcal{W}(S_1\Gamma, e_2)$ and set S_u to be $U(S_2\tau_1,\ \tau_2 \rightarrow \beta)$ where β is fresh. The typing is then $(S_uS_2S_1,\ S_u\beta)$.
$\mathcal{W}(\Gamma_x,\ (\lambda x.\ e)) =$	Make the recursive call $(S_1,\tau_1) = \mathcal{W}((\Gamma;\ x:\beta),\ e)$ where β is fresh. The typing is then $(S_1,\ S_1\beta \rightarrow \tau_1)$.
$\mathcal{W}(\Gamma_x,\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)) =$	Make the recursive calls $(S_1,\tau_1) = \mathcal{W}(\Gamma, e_1)$ $(S_2,\tau_2) = \mathcal{W}((S_1\Gamma;\ x:\overline{S_1\Gamma}(\tau_1)),\ e_2)$ and return the typing $(S_2S_1,\ \tau_2)$.

Figure 1.4: The Damas-Milner type inference algorithm

implicit (not driven by the form of the term at hand); and the **Comb** and **Abs** rules are applicable only to types.

Finally, the **Let** rule allows the typing of `let` terms. The variable bound by a `let` term may be bound to a type or a type scheme and thus may be polymorphic. Notice that this is *not* the case for λ-bound variables. It is this distinction that characterises the Damas-Milner system.

1.4.4 The Damas-Milner type inference algorithm

As mentioned above, with Damas-Milner type system, type information can be reconstructed by an inference algorithm. The algorithm, referred to as algorithm $\mathcal{W}$, is based on Robinson's unification algorithm U [Rob65*], and is presented in full in Damas' thesis [Dam85].

The algorithm takes an assumption set A and an expression e as arguments, and produces a substitution S and type τ as results. If

$$(S, \tau) = \mathcal{W}(A, e)$$

then

$$SA \vdash e : \tau.$$

Figure 1.4 presents the algorithm. Therein, the notation $\overline{A}(\tau)$ denotes

$$\forall\alpha_1.\cdots.\forall\alpha_n.\tau$$

where $\{\alpha_1, \ldots, \alpha_n\}$ are the type variables free in τ but not free in A. All type variables named β are *fresh*, that is, do not appear free in A and are not generated as fresh elsewhere.

The first clause of the algorithm types identifiers on the basis of a binding in the assumption set A. A substitution S' is constructed to rename all the generic variables to *fresh* variables. The substitution returned is the identity and the typing is the type bound in the environment after renaming by S'.

The second clause types lambda terms. A fresh type variable β is assumed for the bound variable. The resultant substitution of the recursive call then provides the actual type of the parameter.

The third clause types application terms making two recursive calls to type the sub-terms. Unification of the function type with the type of a function mapping the actual argument type to a fresh type variable β then produces a unifying substitution S'. Applying S' to β produces the result type.

The final clause types `let` terms by first making a recursive call for the type of the defining expression. This type is generalised, $\overline{A}(\tau)$, and bound to x for the typing of the body. The type of the body is the resulting type for the `let` term.

$\mathcal{W}$ fails under two circumstances: if no binding is found in the environment when typing an identifier, or if unification fails at an application term. If the typing of any sub-term of a term fails then the typing of the entire term fails.

Discussion. The main restriction in expressiveness that the Damas-Milner system suffers with respect to the polymorphic calculus is that lambda bound variables may not be polymorphic. For example, it is not possible to pass a polymorphic function (such as $\lambda x.x$) as an argument to a function and use it polymorphically in the body. This limitation, however, is in general more than offset by the ability to do type inference.

Damas' thesis presents the main technical results with respect to the calculus[3]. In particular, the system is shown to be semantically sound with respect to a simple denotational semantics; the algorithm is shown to be syntactically sound and complete with respect to the type calculus; and, as such, the algorithm is shown to compute principal types.

As mentioned above, Milner's type system was originally developed for ML; it has subsequently been used as the basic type system for many modern functional programming languages such as Miranda[4], Orwell, and most recently, Haskell.

In [Car84a] and [Han87], Luca Cardelli and Peter Hancock, respectively, discuss the implementation of a Damas-Milner style type checker. Reynolds discusses the system in [Rey85] relating it to the polymorphic calculus.

An alternative four-rule version of the system is given in [Cle86]. Therein, the choice of rule at any point in a typing is driven by the form of the term at hand. This represents a more appropriate scheme for implementation in a logic programming language such as Prolog. Further, since the structure of the typing derivation is derived from the structure of the term, this system provides more intuition as to the form of typings under the system.

Unfortunately, and perhaps surprisingly, the Damas-Milner type system does not extend trivially to languages with an assignment operator[5]. The difficulty comes with references to polymorphic objects such as [], the empty list. If a polymorphic object is created in store, then one must ensure that no two different monomorphic types are subsequently assigned to the object. Chapter 3 of Damas' thesis [Dam85] proposes a solution. It has since been shown to be slightly erroneous[6] and Mads Tofte [Tof88] has more recently presented a solution. Tofte's paper proposes two classes of type variables, one of which may *not* be quantified over. The types

[3]Some of these results were originally established by Milner [Mil78].

[4]Miranda is a trademark of Research Software Ltd.

[5]This is ironic since the type system was developed in association with ML—a language with an assignment operator!

[6]The author does not know the details of the problem with Damas' solution.

of objects in store may not contain polymorphic type variables, that is, all type variables they contain are of the class which may not be quantified over.

1.5 Properties of type systems

This section surveys some formal properties one may wish to establish of a type system. As subsequent chapters extend the Damas-Milner type system, wherever possible results with respect to the Damas-Milner system are illustrated.

The systems discussed in the preceding sections are purely syntactic: they consist of a syntax for terms, a syntax for types, a set of type calculus rules for deriving typing judgements, and in some cases a type checking or inference algorithm. Throughout, and particularly in examples, some notion of meaning, or semantics, is implicit. Since informal semantics are notoriously hard to reason about safely, it is traditional to also present a formal semantics for programming languages and their associated type disciplines.

Thus, in the case of the Damas-Milner system, there are three formal systems of interest: the type calculus rules, the formal semantics, and the type inference algorithm. The first group of properties discussed below are semantic and syntactic soundness and completeness results: "semantic" results relate the type system to the formal semantics; "syntactic" results relate the type inference algorithm to the type systems.

1.5.1 Semantic soundness

Damas adopts a denotational semantics for terms and types [Dam85]: the semantic functions T and E assign meaning to types and expressions respectively. The collection of all possible values forms a domain. Following MacQueen, Plotkin and Sethi [MPS84], types are modelled by ideals, or downward-closed partially ordered sets, within that domain.

Take η and ρ to be environments mapping expression and type variables to values and ideals respectively. Then $E(e)\eta$ is the value denoted by e under environment η; and $T(\sigma)\rho$ is the ideal denoted by σ under environment ρ.

A typing of the form $\Gamma \vdash e : \tau$ asserts that e has type τ. Semantic soundness is the property the system satisfies whereby the assertion also ensures that the valuation $E(e)\eta$ is a member of $T(\tau)\rho$, the set of values in the type τ.

> **Definition of semantic soundness.** If $\forall x : \tau' \in \Gamma.\ \eta(x) \in T(\tau')\rho$ and $\Gamma \vdash e : \tau$ then $E(e)\eta \in T(\tau)\rho$.

Typically, there exists a special element in the domain of values, denoted $\{.\}$ and referred to as *wrong*, which represents the result of evaluating an erroneous program. Further, *wrong* is not an element of any of the ideals assigned to types.

The semantic soundness property is often given as follows: "Well-typed programs cannot go *wrong*". More formally, if $\Gamma \vdash e : \tau$, then $E(e)\eta \neq \mathit{wrong}$.

This result was established by Milner in [Mil78] and is given in full by Damas in [Dam85].

The corresponding semantic completeness result is not ordinarily required of a type system. A semantic completeness result would be of the form:

> **Definition of semantic completeness.** If $\forall x : \tau' \in \Gamma.\ \eta(x) \in T(\tau')\rho$ and $E(e)\eta \in T(\tau)\rho$ then $\Gamma \vdash e : \tau$.

To see why most type systems do not have this property, consider the ill-typed expression

if true then 3 *else* '*a*'

which clearly evaluates to 3. In this case semantic completeness may appear to be a desirable property. However, the condition term, the literal *true* here, may be an arbitrarily complicated *Bool* valued term in general. As such, to allow the typing of such terms corresponds to a requirement that the the type checking or inference process compute the value of the condition. This is an impractical proposition and generally outwith the scope of a type system.

1.5.2 Syntactic soundness

Syntactic soundness is a property relating the type inference algorithm to the type calculus. In general terms, the result asserts the following: "If the type inference algorithm computes a typing, then that typing is provable within the type calculus". In the case of the Damas-Milner calculus, the result is stated formally as follows:

> **Definition of syntactic soundness.** If $\mathcal{W}(A, e) = (S, \tau)$ terminates, then $SA \vdash e : \tau$.

This result is given by Milner in [Mil78] and by Damas in [Dam85].

1.5.3 Syntactic completeness

The syntactic completeness result for the Damas-Milner system also relates the type inference algorithm to the type calculus. In this case, however, the assertion is inverted: "Given a typing of a term under a type context, then the inference algorithm will compute a typing. Further, computed typing relates to the given typing in a particular way." Taking "$\geq$" to be an "is more general than" relationship between types, the result is as follows.

Definition of syntactic completeness. Given a type context A and some instance A' thereof, if $A' \vdash e : \tau'$ then

- $\mathcal{W}(A, e)$ succeeds with (S, τ),
- A' is an instance of SA,
- and $\tau \geq \tau'$.

That is, the inference algorithm computes a most general typing. The proof of this result is given in [Dam85].

1.5.4 Principal types

The principal type property is a property of the type calculus alone.

Principal types. If $A \vdash e : \tau$ then there exists a type σ such that $A \vdash e : \sigma$; and for any other typing $A \vdash e : \tau'$, it is the case that $\sigma \geq \tau'$.

In general, the existence of principal types follows as a simple corollary to the syntactic completeness result. This is the case in [Dam85].

1.5.5 Coherence

Coherence is a property of any derivation based semantics: that is, where the semantics is defined as a function of a typing derivation, as opposed to a term or typing judgement. As such, this result is not applicable to the Damas-Milner system. It is appropriate, however, for the system described in the body of this thesis. The result is described here in general terms.

The coherence property is of interest when the system at hand exhibits the following two properties: meaning is assigned as a functions of typing derivations; and it is possible to have two *different* derivations of the same typing judgement.

Using Δ to denote a derivation, the property may be defined as follows.

Definition of coherence. A type system is *coherent* if given any two derivations Δ_1 and Δ_2 of the *same* typing judgement, then the meanings assigned to each are provably equal.

That is the form of the derivation of a typing judgement does not effect the meaning assigned to the term at the given typing. The meaning of the term "provably equal", of course, depends on the system under discussion.

The importance of coherence with respect to an extended polymorphic lambda calculus similar to *Fun* is discussed in [BCGS88]. The principal extension is the

addition of an inclusion relation between types. This relation $\geq$ is assigned meaning by coercion functions; these are introduced by a derivation to derivation translation. Coherence is required to ensure the meaning of a term is defined by its typing judgement, as opposed to the form of the derivation at hand.

John Reynolds in [Rey80] discusses an essentially equivalent property with respect to coercions. Therein, category theory is used to define when a system of coercions is coherent, i.e., when all ways of applying coercions within a term to yield a particular typing are equivalent.

1.6 Universal and *ad-hoc* polymorphism

In the polymorphic calculus and the Damas-Milner calculus only one form of polymorphism is considered, *parametric* polymorphism. Though parametric polymorphism may be considered the purest form of polymorphism, several other forms have been identified and prove useful in programming languages. Christopher Stratchey [Str67*] was the first to identify different forms of polymorphism and an extensive classification can be found in [CW85]. Following [CW85], four principal forms are discussed in detail here as they relate closely to the system discussed in subsequent chapters. These four forms can be grouped into two classes: ***universal*** polymorphism and ***ad-hoc*** polymorphism.

1.6.1 Universal polymorphism

As the name implies, universal polymorphism can be characterised by the fact that the same code can be used *universally* on objects of different types—that the code generated makes no, or limited, assumptions about the types of the objects it operates upon. Universal polymorphism may be further sub-divided into *parametric* and *inclusion* polymorphism.

Parametric polymorphism. This is the form of polymorphism employed by the polymorphic and Damas-Milner calculi, it is characterised by unrestricted universal quantification. The instantiation rule, such as **Spec** in Figure 1.2 or Figure 1.3, allows *any* type to be instantiated for a bound type variable.

Inclusion polymorphism. Inclusion polymorphism, which is closely related to parametric polymorphism and is frequently referred to as "bounded quantification", is characterised by universal quantification in the context of an inclusion relation and a restriction on the instantiating type. The instantiating type is required to be an instance of the given type.

$$\textbf{Spec}\quad \frac{A \vdash e : \forall \alpha \leq \tau.\sigma}{A \vdash e : [\tau'/\alpha]\sigma} \quad \tau' \leq \tau$$

In the rule above, the binding of the type variable α includes a requirement that α be instantiated *only* to types τ' which are instances of τ.

Inclusion polymorphism is common in type systems characterising object-oriented programming systems, for example, [Mit84], [Car84b, Car88], and [JM88]. In particular, *objects* may be modelled by records and the inclusion relation $\leq$ may state that some record type is an instance of some other record type if the fields of the latter form a subset of the fields of the former. Inclusion polymorphism models inheritance.

1.6.2 *Ad-hoc* polymorphism

In contrast to universal polymorphism, *ad-hoc* polymorphism can be characterised by implementations which *vary* with the type under consideration. Again, there are two main forms of *ad-hoc* polymorphism.

Overloading. An overloaded operator assumes different values depending on the type at which it is used. For example, numeric operators, such as $(+)$ and $(*)$, typically apply to both integer and floating point values. When $(+)$ is applied to an integer, one implementation is used; when $(+)$ is applied to a floating point value, another implementation is used. Different object code is inserted for the operator at different types.

In a type system, this is typically represented by several bindings of the same operator at different types. For example, an appropriate assumption set may be

$$A = \cdots;\ (+) : \mathit{Int} \to \mathit{Int} \to \mathit{Int};\ (+) : \mathit{Float} \to \mathit{Float} \to \mathit{Float};\ \cdots$$

allowing typings $A \vdash (+) : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and $A \vdash (+) : \mathit{Float} \to \mathit{Float} \to \mathit{Float}$; that is, two essentially independent typings of the same identifier. The choice of binding used to type the identifier is significant, it typically corresponds to the intended implementation.

Since discussion of a specific type system for overloading is the principal aim of this thesis, several type systems designed to handle overloading are discussed in more detail in Section 1.7.

Coercion. Coercion is the process of taking a value of one type and converting it to a value of another type. For example, if an integer were given as an argument to a function expecting a floating point number, it may be possible to apply a coercion function to the integer value constructing a floating point value as required. If a term is typed at type *Int* but the required typing is at type *Float*, then code to implement the coercion at run-time must be inserted at compile time. The rules for allowable coercions in programming language are typically complicated and *ad hoc*—see C++ [Str86] for a modern example.

Coercions are frequently used in conjunction with overloading to achieve the full range of operator typings a programmer may desire. This is the case in many traditional programming languages such as Pascal and C.

1.7 Approaches to overloading

Many and varied approaches have been taken to the problem of typing and implementing overloading within programming languages; this section examines several existing approaches. Particular attention is paid to the most commonly overloaded operators—the numeric and equality operators.

How, or whether, a system propagates overloading is an issue which is handled differently by different systems. Frequently, definitions may not provide enough type information to allow a single instance of an overloaded operator to be selected: for example, the definition of *double* below.

$$double\ x = x + x$$

Some systems may default *double* to apply to integers only, others may declare it erroneous. One may prefer that the identifier *double* itself become implicitly overloaded at each type at which there is an instance of $(+)$.

Programmers may wish to declare new overloaded operators or extend the definitions of existing ones to apply to user defined abstract types. Two systems providing this functionality are described in Sections 1.7.2 and 1.7.3.

1.7.1 Miranda and ML

Miranda [Tur85] and ML [HMM86] are commonly used functional programming languages incorporating the Damas-Milner type system. Miranda employs normal order reduction, and ML employs applicative order reduction. Both adopt relatively *ad hoc* approaches to overloading. Neither language allows user-defined overloaded operators.

Numeric operators. Miranda has a single numeric type *num* containing both integer and floating point values. This side-steps the problem of overloading arithmetic operators. Though integer and floating point values may be mixed, this approach is insufficiently expressive to meet many of the demands of programmers wishing to perform serious numerical computation.

ML, on the other hand, has distinct numeric types and overloaded arithmetic operators. Integer and floating point values may not be mixed arbitrarily in arithmetic expressions. Definitions such as that of *double* above are erroneous—overloading implicit in the definition of an identifier is not propagated. Further, it is not possible to extend the overloading of arithmetic operators to user defined types, for example, a type representing complex numbers.

The equality operator. The equality operator is also treated differently in both ML and Miranda. An early version of ML [Mil84*] defined equality on all monotypes that *admit equality*, that is, do not contain abstract or function types. Further, it was required that the type be resolved locally. Thus, functions such as *member*

$$\begin{aligned} &member\ []\ y &&= false \\ &member\ (x : xs)\ y &&= (x == y)\ or\ (member\ xs\ y) \end{aligned}$$

could not be defined and equality could not be applied to user-defined abstract types.

Miranda, on the other hand, considers equality to be a fully parametric polymorphic function for the purpose of type checking. Only at run-time is an error generated if equality is applied to a function type or a constructed type containing a function type. Further, if equality is applied to a user defined abstract data type, then the *representation* is tested for equality contravening data abstraction.

A more sophisticated approach is taken in Standard ML [HMM86]. Equality is made polymorphic in a limited way. In particular, the equality operator is given the typing

$$(==) : ''a \to ''a \to Bool$$

where $''a$ is a special type variable that may only be instantiated to types that *admit equality*. Such type variables allow overloadings in definitions such as *member* above to be propagated.

$$member : List(''a) \to ''a \to Bool$$

The definition of the equality operator may not be extended to apply to user defined abstract types. That is, the function *member* above may not be used to test for membership of a list of objects of some abstract type.

In summary, both Miranda and ML use a collection of *ad hoc* techniques to provide some of the facilities a programmer requires. Different approaches are taken even within the same language and no attempt is made to allow the user to define overloaded operators or extend the definitions of existing operators. ML's equality type variables provide some structure to the use of overloading, though only for a single predefined operator.

1.7.2 Kaes' system

Stefan Kaes generalises ML's equality type variable approach [Kae88]. Kaes' system allows overloaded identifiers and instances thereof to be declared by the programmer, indeed such declarations may be arbitrarily nested. Each overloaded operator

has associated with it a signature, or a pattern, into which each instance typing must fit. For example, the following declares the equality operator over integer and list types. The definition at list types admits equality only when the type of the elements admit equality. (The notation <...> denotes some appropriate, though omitted, implementation.)

$$\begin{array}{l}\texttt{letop}\ (==) : \$ \rightarrow \$ \rightarrow \mathit{Bool}\ \texttt{in} \\ \texttt{extend}\ (==) : \mathit{Int}\ = <...>\ \texttt{in} \\ \texttt{extend}\ (==) : \mathit{List}(\alpha_{==})\ = <...>\ \texttt{in}\end{array}$$

Firstly, the **letop** declaration declares (==) to be an overloaded operator. The special symbol $ can be thought of as a universally quantified type variable. The signature $\$ \rightarrow \$ \rightarrow \mathit{Bool}$ is the pattern into which all instances must fit; that is, all instances must be functions mapping two values of the same type to a *Bool*. Notice that the $ notation in effect restricts Kaes' overloaded operators to be polymorphic only in a single variable.

The first **extend** declares equality to have an instance at type *Int*. Type variables subscripted with overloaded operators may only be instantiated to types admitting those operators. Thus, the instance of (==) at type $\mathit{List}(\alpha_{==})$ declares equality applicable to any list of equality types; for example,

$\mathit{List}(\mathit{Int}) \rightarrow \mathit{List}(\mathit{Int}) \rightarrow \mathit{Bool}$; and
$\mathit{List}(\mathit{List}(\mathit{Int})) \rightarrow \mathit{List}(\mathit{List}(\mathit{Int})) \rightarrow \mathit{Bool}$

but not

$$\mathit{List}(\mathit{Int} \rightarrow \mathit{Bool}) \rightarrow \mathit{List}(\mathit{Int} \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$$

as $\alpha_{==}$ cannot be instantiated to type $\mathit{Int} \rightarrow \mathit{Bool}$.

As a further example, if (+) is an overloaded operator with signature $\$ \rightarrow \$ \rightarrow \$$, then the *member* and *double* functions declared previously have typings

$$\begin{array}{ll}\mathit{member} & : \mathit{List}(\alpha_{==}) \rightarrow \alpha_{==}\mathit{Bool} \\ \mathit{double} & : \alpha_{+} \rightarrow \alpha_{+}\end{array}$$

where α_{+} is a type variable instantiable only to types admitting addition.

Type variables are in fact annotated with *sets* of overloaded operators—the empty set denotes no annotation. Thus, assuming an appropriate definition of *map* and the definition

$$\mathit{memdbl}\ l\ e\ =\ \mathit{member}\ (\mathit{map}\ \mathit{double}\ l)\ e,$$

memdbl has type

$$\mathit{List}(\alpha_{==,+}) \rightarrow \alpha_{==,+} \rightarrow \mathit{Bool}.$$

That is, *memdbl* is applicable only to lists whose elements admit both equality and addition.

Kaes' defines two forms of semantics for the system: one static and one dynamic. The static form views propagated overloading as an abbreviation: each use of an implicitly overloaded identifier at a known type is macro expanded at compile time. This process is potentially exponential both in terms of the efficiency of the compiler and the amount of object code generated.

The second dynamic approach delays the choice of actual overloaded operator until run-time. Every potentially overloaded value is tagged with its type which is matched at run-time to select the appropriate implementation of the overloaded operator. This, naturally, results in a run-time overhead.

Though Kaes claims a principal type theorem, the current author believes this is erroneous for his system in its full generality. In particular, nested overloading and instance declarations appear to contravene the principal type property. The two examples below illustrate the problem.

Expression terms containing overloading declarations do not satisfy a principal type theorem. For example, under some suitable environment, the term

$$\begin{array}{l}\texttt{letop } x : \$ \to \$ \texttt{ in} \\ \texttt{extend } x : \mathit{Int} = \lambda x.\, 1 \texttt{ in} \\ \texttt{extend } x : \mathit{Char} = \lambda x.\, \texttt{'}a\texttt{'} \texttt{ in} \\ x\end{array}$$

has types $\mathit{Int} \to \mathit{Int}$ and $\mathit{Char} \to \mathit{Char}$ but the more general type $\forall\alpha_x.\alpha_x \to \alpha_x$ is disallowed (since x has no meaning outside the scope of its class declaration).

A similar problem with respect to principal types occurs when one allows **extend** expressions to appear in arbitrary positions. Assume the initial context contains the assumptions and structure generated by the `letop` declaration

$$\texttt{letop } x : \$ \to \$ \texttt{ in}$$

and instances of x at *Int* and *Float*. Then the term

$$\begin{array}{l}\texttt{extend } x : \mathit{Char} = \lambda x.\, \texttt{'}a\texttt{'} \texttt{ in} \\ x\end{array}$$

has types $\mathit{Char} \to \mathit{Char}$ and $\forall\alpha_x.\alpha_x \to \alpha_x$ which are not related by the instance relation. Further, there is no typing more general than either.

The type system described in Chapter 3 avoids the problems with respect to principal types by ensuring that all overloading and instance declarations are at the top level.

1.7.3 Nipkow and Snelting's system

Another, though more recent, system addressing a similar problem is that developed by Nipkow and Snelting [NS91]. The Nipkow-Snelting system extends the Damas-Milner system with an order-sorted algebraic description of overloaded operators. Order-sorted unification then replaces Robinson's unification algorithm for type inference.

Following [WaBl89], operators are grouped into named classes: for example,

$$\begin{array}{l}\texttt{class } \textit{Num} \texttt{ where} \\ \quad (+) : \forall \alpha_{Num}.\alpha_{Num} \rightarrow \alpha_{Num} \rightarrow \alpha_{Num}, \\ \quad (*) : \forall \alpha_{Num}.\alpha_{Num} \rightarrow \alpha_{Num} \rightarrow \alpha_{Num}\end{array}$$

declares $(+)$ and $(*)$ to be in the class *Num*. Type variables are annotated with *class* names, as opposed to operator names. Again following [WaBl89], classes may be structured to form a hierarchy. The declarations

$$\begin{array}{l}\texttt{class } \gamma_1 \texttt{ where } <\cdots> \texttt{ in} \\ \texttt{class } \gamma_2 \leq \gamma_1 \texttt{ where } <\cdots>\end{array}$$

ensure that no instance may be declared at type τ in class γ_2 unless an instance at type τ has been declared in class γ_1. Such class hierarchies are transitive and may be directed acyclic graphs: one class may be a sub-class of another in more than one way, though no class may be a sub-class of itself. Class declarations syntactically require the types of the operators of the class to be polymorphic only in a single variable.

Instances of classes are declared with **inst** declarations. For example,

$$\texttt{inst } \chi : (\gamma_1, \ldots, \gamma_n)\gamma \texttt{ where } x_1 = e_1, \ldots, x_m = e_m$$

declares an instance of class γ at type χ with its appropriate implementations. Classes γ_1 to γ_m represent class requirements on the constituent types of χ: for example, they may be used to specify that equality on lists is applicable only when the elements of the list admit equality.

Class signatures $(\gamma \leq \gamma_1, \ldots \gamma_n)$ and instance restrictions $(\chi : (\gamma_1, \ldots, \gamma_n)\gamma)$ define an order-sorted algebra. Where Robinson's unification algorithm is used in the Damas-Milner algorithm $\mathcal{W}$, Nipkow and Snelting's system utilises an order-sorted unification algorithm. Thus, the order-sorted algebraic signature generated by the declarations is respected by unification.

The semantics of Nipkow and Snelting's system is based on a derivation to derivation translation. This follows [WaBl89] in using parametrisation to implement propagated overloading.

Chapter 2

A language for overloading

This Chapter presents OL, a simple language with parametric polymorphism and overloading. Robin Milner and Luis Damas [Mil78, DaMi82, Dam85] used a small language, the lambda calculus with `let`, to illustrate their type system for parametric polymorphism; OL extends that system. OL is perhaps a rather uninspired name, it stands for Overloading Language.

The presentation in this chapter is informal, subsequent chapters provide a more rigorous definition of OL. To facilitate the discussion, several assumptions and abbreviations are made: for example, predefined definitions of operations such as *eqInt*, *eqList*, *plusInt* and *plusFloat* are assumed representing equality and addition on base types. Literals such as 3, 3.0 and $(3, 3.0)$, are used with types *Int*, *Float* and $Pair(Int, Float)$ respectively.

The term *operator* is used to refer to overloaded names, and the term *identifier* is used to refer to lambda- and `let`-bound names. These are not distinguished in the syntax of OL. Symbols, such as (==), are usually chosen to represent overloaded operators.

2.1 Terms and Types

This section presents the syntax of terms and types and introduces the typing rules by example. OL subsumes the Damas-Milner system: anything which is expressible and typable in the Damas-Milner system is expressible and typable in OL.

Raw terms are generated by the context-free grammar in Figure 2.1. Variables and terms are represented by x and e respectively, with α, χ, τ, π and σ representing type variables, type constructors, types, predicated types and type schemes respectively. Type schemes are of the general form

$$\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

where predicates (such as $x_1 : \tau_1$) act as a form of bounded quantification. In the absence of predicates, type schemes reduce to Damas-Milner type schemes.

Identifiers	x		
Type Variables	α		
Type Constructors	χ		
Types	τ	::=	$\alpha \mid \tau \rightarrow \tau' \mid \chi(\tau_1, ..., \tau_n)$
Predicated types	π	::=	$(x : \tau).\pi \mid \tau$
Type schemes	σ	::=	$\forall\alpha.\sigma \mid \pi$
Terms	e	::=	$x \mid e_0\ e_1 \mid \lambda x.\ e \mid$ **let** $x = e_0$ **in** e_1
Declarations	d	::=	**over** $x : \sigma;\ d \mid$
			inst $x : \sigma = e;\ d \mid$
			ϵ
Programs	p	::=	(d, e)
Sets of type variables	δ		

Figure 2.1: Raw terms and types

Similarly, terms are the same as Damas-Milner expressions; however declarations, which allow overloadings to be declared, extend the Damas-Milner system. Declarations form a list—**over** declaring overloaded operators, and **inst** declaring instances thereof. The empty declaration list is denoted by ϵ or simply omitted.

Programs are composed of a declaration part and an expression term. The approach adopted by Kaes [Kae88] is to allow declarations to appear arbitrarily within terms. For technical reasons, however, it is better to partition programs into two parts.[1]

2.1.1 Declaring overloaded operators

Overloaded operators are declared with **over** terms in the declaration part of a program. The **over** declaration

over $x : \sigma$;

declares x to be an overloaded operator with *signature* σ. The signature of an overloaded operator is a type scheme which characterises all the instance types. Such signatures may *not* contain predicates. This contrast with the system presented in [WaBl89]; the change simplifies technical considerations.

Consider the equality operator (==). In general this takes two objects of the same type, compares them for equality, and returns a *Bool*. As such, the type scheme $\forall\alpha.\alpha \rightarrow \alpha \rightarrow Bool$ characterises all instances of (==), therefore the following declaration is appropriate.

over $(==) : \forall\alpha.\alpha \rightarrow \alpha \rightarrow Bool$;

[1]The motivation for this dichotomy is given in Section 10.1.1.

```
over (+) : ∀α.α → α → α;
over (*) : ∀α.α → α → α;
over (==) : ∀α.α → α → Bool;
inst (+) : Int → Int → Int = plusInt;
inst (+) : Float → Float → Float = plusFloat;
inst (*) : Int → Int → Int = timesInt;
inst (*) : Float → Float → Float = timesFloat;
inst (==) : Int → Int → Bool = eqInt;
inst (==) : Float → Float → Bool = eqFloat;
inst (==) : Char → Char → Bool = eqChar;
```

Figure 2.2: Example `over` and `inst` declarations

In the context of such a declaration, (==) is an overloaded operator for which instances may be declared at the types on the left below, but not at the types on the right.

$Int \to Int \to Bool$	$Int \to Int \to Int$
$Char \to Char \to Bool$	$Char \to Int \to Bool$

By a similar argument, overloaded addition accepts two arguments of the same type and returns a result, the sum, which is also of the same type. The declaration

```
over (+) : ∀α.α → α → α;
```

reflects this.

Figure 2.2 contains an example declaration d, in this case d declares three overloaded operators: (+), (*) and (==).

For simplicity, it is convenient to restrict lambda and `let` bound variables such that they may not redeclare (or hide) overloaded identifiers.

2.1.2 Declaring instances of overloaded operators

An `inst` declaration declares an instance of an overloaded operator. Since the characterisation of *ad-hoc* polymorphism entails that different implementations are used at different types, `inst` declarations give a new type for an overloaded operator and the implementation at that type. The general `inst` declaration

```
inst x : σ = e;
```

declares a new instance of overloaded operator x at type σ with implementation e.

Following the equality and addition examples from the previous section (and assuming the appropriate definition for *plusInt* etc.) instances

$$\texttt{inst}\ (+) : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} = \mathit{plusInt};$$
$$\texttt{inst}\ (+) : \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} = \mathit{plusFloat};$$

declare addition on integers and floating point numbers, respectively. Wherever addition is required on integers, the implementation *plusInt* is appropriate; and wherever addition is required on floating point numbers, the implementation *plusFloat* is appropriate.

Further examples of `inst` declarations are given in Figure 2.2. There are three overloaded operators: (+) and (*), with instances at *Int* and *Float*; and (==) with instances at *Int*, *Float* and *Char*. The overloadings defined in Figure 2.2 are used as the context for the rest of this chapter.

2.1.3 Using overloaded operators

This section describes how overloaded operators, as defined in the previous two sections, may be used in expression terms. In particular, how *predicates* model the propagation of overloading from one `let` declaration to another.

Simple typings. Overloaded operators may be applied to values and expressions in the ordinary way: the difference lies in the typing and semantics. Types, in general, are of the form

$$\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

where $(x_i : \tau_i)$ are *predicates*. Predicates represent a form of bounded quantification, a requirement that overloaded operator x_i has an instance at type τ_i. They record the typing information required of overloaded operators. A predicate is associated with each instance at which an overloaded operator is used within an expression. For example, consider the expressions $(3 + 4)$ using integer addition. The type of this expression is *Int*, with the requirement that (+) have an instance at type $(\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int})$. The following typing expresses this.

$$3 + 4 \ : ((+) : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}).\mathit{Int}$$

Since there *is* an integer instance of (+) over integers in scope—see Figure 2.2—the predicate is satisfied and may be discharged. The final typing is, therefore, *Int*.

$$3 + 4 \ : \mathit{Int}$$

The type inference algorithm first computes the typing containing the predicate; then, observing that an instance in the environment satisfies the predicate, the predicate is discharged. Another example of a simple typing is given below.

$$\begin{aligned}\lambda x.\ x == \text{`}z\text{`} \ &: ((==) : \mathit{Char} \to \mathit{Char} \to \mathit{Bool}).\mathit{Char} \to \mathit{Bool} \\ &: \mathit{Char} \to \mathit{Bool}\end{aligned}$$

The function $(\lambda x.\ x == \text{`}z\text{`})$ requires an instance of equality at type *Char*. There is such an instance in scope allowing the predicate to be discharged.

Terms which fail to type. Simple typings fail when an overloaded operator is used at a type which fails to satisfy its signature. For example, $(2 * 3.142)$ fails to type since it requires multiplication to accept arguments of differing types.

Sometimes the type variables in predicates are instantiated but no appropriate instance is in scope.

$$\text{`}a\text{'} + \text{`}b\text{'} \ : ((+) : \mathit{Char} \to \mathit{Char} \to \mathit{Char}).\mathit{Char}$$

This requirement is noted as a predicate in the typing as before—it is not considered erroneous—and the term receives a sensible semantics. This approach is unusual, most systems deem such terms untypable or erroneous. The main justification is technical: the substitution rule[2] is not valid if such typings are disallowed since a predicate on a type variable could be mapped to a predicate on an arbitrary type.

Implicitly overloaded declarations. It is not always possible to eliminate predicates from a typing—the type information implicit in the term may be insufficient to establish that there is an instance in scope at the given type. This is the case for the following three terms.

$$\begin{aligned}(==) \quad &: \forall\alpha.((==) : \alpha \to \alpha \to \mathit{Bool}).\alpha \to \alpha \to \mathit{Bool} \\ \lambda x.\ \lambda y.\ x == y \quad &: \forall\alpha.((==) : \alpha \to \alpha \to \mathit{Bool}).\alpha \to \alpha \to \mathit{Bool} \\ \lambda x.\ x + x \quad &: \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha\end{aligned}$$

The first is the unapplied equality operator; the second, a lambda reconstruction of the equality operator; and the third, a lambda term to double a numeric value. These terms inherit a form of *implicit overloading* from the overloaded operators they contain.

As with parametric polymorphism, identifiers which are not explicitly overloaded by `over` declarations may only be overloaded if they are bound by a `let` declaration. When implicitly overloaded terms appear in `let` definitions, the identifier being defined is itself implicitly overloaded. For example, given the definition of *double*,

$$\texttt{let}\ \mathit{double} = \lambda x.\ x + x\ \texttt{in}\ e,$$

[2]Given as $\mathbf{Rule}_1$ in Chapter 4.

in which the defining term is implicitly overloaded at type

$$\lambda x.\ x + x : \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha,$$

the identifier *double* itself becomes implicitly overloaded.

$$\mathit{double}\ : \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha$$

The identifier *double* may then appear in terms just as could an explicitly overloaded operators. Since *double*'s type contains a predicate, this predicate reappears in the typing of any use of *double*.

$$\begin{aligned}\mathit{double}\ 3\ &: ((+) : \mathit{Int} \to \mathit{Int} \to \mathit{Int}).\mathit{Int}\\ &: \mathit{Int}\end{aligned}$$

In this case, an instance in scope satisfies the predicate so it may be discharged. Elsewhere, *double* may be applied to a floating point value.

$$\begin{aligned}\mathit{double}\ 3.4\ &: ((+) : \mathit{Float} \to \mathit{Float} \to \mathit{Float}).\mathit{Float}\\ &: \mathit{Float}\end{aligned}$$

Notice that *double* is now effectively overloaded itself; no instances can be declared for it, but all instances of (+) imply types at which *double* is applicable.

It is convenient to adopt some abbreviations for predicates.

Pl_τ abbreviates $((+) : \tau \to \tau \to \tau)$
Ti_τ abbreviates $((*) : \tau \to \tau \to \tau)$
Eq_τ abbreviates $((==) : \tau \to \tau \to \mathit{Bool})$

These abbreviations allow more complicated types to be expressed concisely. For example, *double*'s type is written

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

A term's type may contain more than one predicate. The term

$$\lambda x.\ \lambda y.\ (x + y) == (x * y)$$

contains three different overloaded operators and as such has a typing with three predicates.

$$\forall\alpha.(Pl_\alpha).(Ti_\alpha).(Eq_\alpha).\alpha \to \alpha \to \mathit{Bool}$$

It is the use of predicates which characterises the way in which overloading is handled in OL. Predicates record the use of overloaded operators and the types at which instances are required. Each predicate is propagated and replicated until a specific application provides enough type information to discharge it.

2.2 Some further uses of predicates

So far, predicates have only appeared in the types of expressions and the types of `let` bound identifiers. Significant expressive power is gained by allowing predicates to appear in **over** and **inst** declarations. In the case of **over** expressions, this facility is not included in the current work. Section 10.3 describes how it can be added.

Instances over constructed types. Operators such as equality should be applicable to constructed types, such as *List* and *Pair*, as well as simple types. For instance, equality should be applicable to the following types

$$Pair(Int, Float)$$
$$List(Pair(Int, Float))$$

but not to a pair or list with a component not of an equality types—such as

$$Pair((Int \rightarrow Int), Int).$$

Consider a definition of equality on pairs. Two pairs are equal if their respective components are equal. The term

$$\lambda p_1.\ \lambda p_2.\ ((fst\ p_1) == (fst\ p_2))\ \textbf{and}\ ((snd\ p_1) == (snd\ p_2))$$

reflects this with the typing

$$\forall\alpha.\forall\beta.(Eq_\alpha).(Eq_\beta).Pair(\alpha,\beta) \rightarrow Pair(\alpha,\beta) \rightarrow Bool.$$

Therefore, taking <*eqPair*> and <*eqPairType*> to abbreviate the term and type above respectively, an `inst` declaration of the form

$$\texttt{inst}\ (==) : <eqPairType> = <eqPair>;$$

makes $Pair(\tau_1, \tau_2)$ an instance of equality whenever both τ_1 and τ_2 are instances of equality.

A similar technique may be used to define equality over lists. Two lists are equal if they are of the same length and their corresponding elements are equal. Hence, their elements must be of the same type. Assuming <*eqList*> stands for an appropriate term, the following declaration achieves this.

$$\texttt{inst}\ (==) : \forall\alpha.(Eq_\alpha).List(\alpha) \rightarrow List(\alpha) \rightarrow Bool = <eqList>;$$

The equality operator is now applicable to an infinite number of types:

$$List(Int)$$
$$List(List(Bool))$$
$$List(Pair(Float, List(Float)))$$

In typing such uses of overloaded operators, when an instance is used to discharge a predicate, all predicates in the instance type are introduced into the typing. Two examples of such typings are as follows.

$$\begin{array}{ll} [1,2,3] == [2,6,4] & : (Eq_{List(Int)}).Bool \\ & : (Eq_{Int}).Bool \\ & : Bool \end{array}$$

$$\begin{array}{ll} \lambda p.\ \lambda x.\ p == (x, 4.6) & : (Eq_{Pair(\alpha,Float)}).Pair(\alpha, Float) \rightarrow \alpha \rightarrow Bool \\ & : (Eq_{\alpha}).(Eq_{Float}).Pair(\alpha, Float) \rightarrow \alpha \rightarrow Bool \\ & : (Eq_{\alpha}).Pair(\alpha, Float) \rightarrow \alpha \rightarrow Bool \end{array}$$

The first contains enough typing information to discharge all the predicates; initially with the instance at type $List(Int)$ and then with the instance at type Int. For the second, discharging the predicate at type $Pair(\alpha, Int)$ introduces two predicates; only one of these, at Int, can be subsequently discharged.

A hierarchy of overloaded operators. Predicates in the type signature of an overloaded operator induce a *subclass* relation on overloaded operators[3]. An `over` declaration of the form

$$\texttt{over}\ x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau;$$

requires statically that, for each instance of x, there be appropriate instances of x_i in scope. For example, it may be the case that the *less-than-or-equal-to* operator $(<=)$ only has instances at types at which $(==)$ has instances. The `over` declaration

$$\texttt{over}\ (<=) : \forall\alpha.(Eq_{\alpha}).\alpha \rightarrow \alpha \rightarrow Bool;$$

declares this to be the case. It is now possible to declare instances of $(<=)$ at types Int, $Char$ and $Pair(Int, Float)$, but not at types $(Int \rightarrow Int)$ or $List(Int \rightarrow Int)$ for which equality is not defined.

An extension of the current work including relationships of this sort between overloaded operators is discussed in Section 10.3.

[3]As mentioned previously, this feature was included in [Waß89] but is not included in the scope of the current work. The feature is included in Haskell and is outlined here for completeness.

Grouping operators together. Often a particular group of overloaded operators are related and have instances at the same types. This relationship may be modelled in OL by grouping overloaded operators together into tuples.

For example, the numeric functions $(+)$ and $(*)$ are closely related. Rather than declare them individually, as was done previously, they may be declared together, as follows.

$$\begin{array}{ll}
\texttt{over } \mathit{plusTimes} & : \forall\alpha.\mathit{Pair}\ (\alpha \to \alpha \to \alpha, \alpha \to \alpha \to \alpha); \\
\texttt{inst } \mathit{plusTimes} & : (\mathit{Int} \to \mathit{Int} \to \mathit{Int}, \mathit{Int} \to \mathit{Int} \to \mathit{Int}) \\
& = (\mathit{plusInt}, \mathit{timesInt}); \\
\texttt{inst } \mathit{plusTimes} & : (\mathit{Float} \to \mathit{Float} \to \mathit{Float}, \mathit{Float} \to \mathit{Float} \to \mathit{Float}) \\
& = (\mathit{plusFloat}, \mathit{timesFloat}); \\
 & \\
\texttt{let } (+) & = \mathit{fst}\ \mathit{plusTimes}\ \texttt{in} \\
\texttt{let } (*) & = \mathit{snd}\ \mathit{plusTimes}\ \texttt{in}
\end{array}$$

Operators, under such a scheme, become selector functions over overloaded tuples: $(+)$ and $(*)$ are, thus, implicitly overloaded.

$$\begin{array}{ll}
(+) & : \forall\alpha.(\mathit{plusTimes} : \mathit{Pair}\ (\alpha \to \alpha \to \alpha, \alpha \to \alpha \to \alpha)).\alpha \to \alpha \to \alpha \\
(*) & : \forall\alpha.(\mathit{plusTimes} : \mathit{Pair}\ (\alpha \to \alpha \to \alpha, \alpha \to \alpha \to \alpha)).\alpha \to \alpha \to \alpha
\end{array}$$

2.3 Associating Meaning with Typed Terms

Meaning is assigned to OL terms by a translation process eliminating overloading from terms. Each well-typed OL term is translated into a well-typed term containing no overloading. The translation is based on the type assigned to the term. Implicit overloading is modelled by explicit parametrisation.

- Each predicate introduced during typing yields a new parameter in the translation. This parameter denotes the implementation of the overloaded operator associated with the predicate.
- Each predicate discharged during typing yields an application in the translation. The translation of an implicitly overloaded term is applied to the implementation associated with the instance that discharges the predicate.

2.3.1 Translation of declarations

The translation eliminates all **over** declaration terms. Each `inst` declaration translates to a `let` expression.

$$\texttt{inst } x : \sigma = e; \quad \text{becomes} \quad \texttt{let } d\sigma_x = \overline{e}\ \texttt{in } \cdots$$

```
let dInt(+) = plusInt in
let dFloat(+) = plusFloat in
let dInt(*) = timesInt in
let dFloat(*) = timesFloat in
let dInt(==) = eqInt in
let dFloat(==) = eqFloat in
let dChar(==) = eqChar in
...
```

Figure 2.3: Example **over** and **inst** translations

A new and unique name is associated with each instance, in the case $d\sigma_x$; this is referred to as the instance's *implementation identifier*. The **let** term binds the implementation identifier to the translation of the the implementation term. As an example, the **inst** expression

$$\texttt{inst}\ (+) : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} = \mathit{plusInt};\ \cdots$$

translates to the **let** expression

$$\texttt{let}\ \mathit{dInt}_{(+)} = \mathit{plusInt}\ \texttt{in}\ \cdots$$

The translation of an identifier whose type contains no predicates is simply the identifier itself. So, the translation of *plusInt* is *plusInt*. The identifier $\mathit{dInt}_{(+)}$ now denotes the instance of (+) over integers. The prefix notation *d* stands for *dictionary*, and derives from object-oriented programming systems.

The translation of the **over** and **inst** declarations from Figure 2.2 are given in Figure 2.3.

2.3.2 Translation of overloaded expressions

This section presents the translation of overloaded and implicitly overloaded identifiers within expressions. Firstly, the resolution of overloaded operators *in situ* is considered, then a more general scheme for the propagation of implicit overloading is discussed.

Simple overloading resolution. Consider first the situation where the type of an expression is sufficient to resolve the overloading *in situ*. In this case, the identifier

associated with the appropriate instance replaces the overloaded operator. For example, given the typing

$$\begin{array}{ll} 3+4 & : ((+) : \mathit{Int} \to \mathit{Int} \to \mathit{Int}).\mathit{Int} \\ & : \mathit{Int} \end{array}$$

the integer instance of (+) is used to discharge the predicate, therefore, in the translation the implementation identifier associated with the integer instance, $dInt_{(+)}$, is inserted for the overloaded operator. The translation—including a change from infix operator to prefix identifier—is as below.

$$dInt_{(+)}\ 3\ 4$$

This method of overloading resolution is only applicable when the appropriate instance binding for an occurrence of an overloaded operator is known directly.

Resolution for implicitly overloaded identifiers. Consider the term,

$$\texttt{let}\ \mathit{double} = \lambda x.\ x + x\ \texttt{in}\ \cdots$$

where the defining expression is implicitly overloaded. The first step is to assume *some* implementation identifier, say $da_{(+)}$, to implement addition at the appropriate type. This identifier is then inserted in place of the overloaded (+), and the assumed implementation identifier is lambda-abstracted over yielding a parametrised translation.

$$\texttt{let}\ \mathit{double} = \lambda da_{(+)}.\ \lambda x.\ da_{(+)}\ x\ x\ \texttt{in}\ \cdots$$

This extra parameter corresponds to the implementation of (+) which will be used in each instantiation of *double*. When *double* is applied to an *Int*, it will first be applied to the implementation identifier for addition over integers; when it is applied to a *Float*, it will first be applied to the implementation identifier for addition over floating points.

The translation of instances with predicated types is done in the same way. The translation of the instance of equality over pairs is given below.

$$\begin{array}{l} \texttt{let}\ dPair_{(==)} = \lambda da_{(==)}.\ \lambda db_{(==)}.\ \lambda p_1.\ \lambda p_2. \\ \qquad (da_{(==)}\ (\mathit{fst}\ p_1)\ (\mathit{fst}\ p_2))\ \texttt{and}\ (db_{(==)}\ (\mathit{snd}\ p_1)\ (\mathit{snd}\ p_2))\ \texttt{in} \end{array}$$

Two new parameters are added, one for each predicate in the type. The implementation identifier $da_{(==)}$ represents equality between the left components, and $db_{(==)}$ represents equality between the right components.

The OL type of a term and its translated type are closely related. Where the OL type of *double* is

$$\forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha,$$

the translation's type is

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

This reflects the relationship between predicates in the type and parametrisation in the translation. The general rule is that if an OL term has the type

$$\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

then its translation has the type

$$\forall\alpha_1 \cdots \forall\alpha_n.\tau_1 \to \cdots \to \tau_m \to \tau.$$

Discharging predicates. Lambda abstractions are used in the translation to model predicates on implicitly overloaded identifiers. The discharge of predicates is, therefore, modelled by application to the appropriate implementation identifier.

For example, an application of *double* to a specific value, *double* 3, is modelled by first applying double to a specific instances implementation identifier.

$$(double\ dInt_{(+)})\ 3$$

This achieves the required instantiation of the occurrence of (+) in the defining term for *double*.

The same rule applies to instances with predicates in their types. The term

$$(3, true) == (4, false) : Bool$$

translates to

$$(dPair_{(==)}\ dInt_{(==)}\ dBool_{(==)})\ (3, true)\ (4, false)$$

where $dPair_{(==)}$ is first applied to appropriate implementation identifiers achieving the required instantiation.

Chapter 3

A type system for overloading

This chapter presents the formal type system and semantics the language OL. The OL type system extends the Damas-Milner system [DaMi82, Dam85] described in Section 1.4. There follows an overview of the main concepts introduced in this chapter. Terms and types are as previously defined in Figure 2.1.

A *type context*, denoted Γ, binds identifiers and operators to type schemes. A restriction is required on type contexts to ensure they represent valid overloadings. This restriction is denoted by *valid type context judgements* of the form $\rhd \Gamma$.

An ordering relation, relating type schemes in a given type context, is introduced. The *instance judgement*

$$\Gamma \rhd \sigma \geq \sigma'$$

is read: "In context Γ, the type scheme σ has an instance at type σ'". In the absence of overloading, this degenerates to the Damas-Milner ordering. The two forms of judgement above form a well-founded mutual recursion.

The type system is a set of rules for deriving *typing judgements*, or *typings* for short. A typing judgement

$$\Gamma \rhd e : \sigma$$

is read: "Under type context Γ, the term e has type scheme σ". There are nine rules for the construction of such judgements: the six rules of the Damas-Milner calculus, one rule for typing instances of overloaded operators, and two rules for predicates (introduction and elimination).

Two other forms of judgement are required. A judgement of the form $\Gamma \rhd d \rightsquigarrow \Gamma'$ is an *extraction judgement*. It is read: "In context Γ, the declaration d yields the type context Γ'". Finally, programs are typed by *program typing judgements* of the form $\Gamma \rhd (d, e) : \sigma$ which denote the typing of an entire program.

A type inference algorithm $\mathcal{O}$ infers typings for the calculus of OL; it corresponds to Damas-Milner's algorithm $\mathcal{W}$. It is *sound* and *complete* with respect to the type

calculus. As such, it computes *principal typings*, that is, typings which characterise all other typings of the term at hand under the given type context. These results are given in full in Chapter 5.

Finally, the semantics of typed OL is given. As in Chapter 2, this is done by a translation into the Damas-Milner system. *Derivations* in the OL calculus are translated into *derivations* in the Damas-Milner calculus.

3.1 Syntax

This section introduces the rest of the notation that is required to present the type calculus. It is in two main parts: Section 3.1.1 introduces type contexts, substitutions and the instance relation; and Section 3.1.2 presents a notion of *validity* for type contexts ensuring overloadings make sense.

3.1.1 The raw notation

Terms and types are as defined in Figure 2.1 of Chapter 2. The notation $fv(\sigma)$ denotes the set of free variables in σ; and the notation $fv(\Gamma)$ denotes the set of type variables which are free in Γ.

Identifiers and operators are *not* syntactically distinct. For simplicity in presentation and in proofs, no identifiers or operators may be redeclared within terms. This poses no real restriction as α-conversion can be used to re-name bound variables.

Substitutions. A substitution is a function mapping type variables to types. Substitutions may be applied to types, type schemes and type contexts under the natural extensions. S denotes a substitution in general, and Id denotes the identity substitution. The notation $dom(S)$ and $fv(S)$ denote the domain of S and the type variables contained in the range of S respectively. All substitutions are considered to be the identity on type variables for which they are not otherwise defined. The domain refers to all type variables for which the substitution is not the identity, so $dom(Id) = \{\}$.

Composition of substitutions is denoted S_1S_2—that is $S_1S_2\tau = S_1(S_2\tau)$. If the domains of S_1 and S_2 do not overlap then, $(S_1 + S_2)$ denotes the substitution which applies S_1 to type variables in $dom(S_1)$ and S_2 to those in $dom(S_2)$. Given a substitution S, the notation $S\backslash_\delta$ denotes the substitution identical to S on all type variables except that it is the identity on type variables in δ.

Robinson's unification algorithm U [Rob65*] is used to find the most general substitution that unifies two types, if such a substitution exists. That is, given τ_1 and τ_2, the call $S = U(\tau_1, \tau_2)$ produces S such that $S\tau_1 = S\tau_2$. If no such substitution exists then U fails. Algorithm U computes the minimal most general unifying solution; that is, if S' is some other unifying substitution, then there exists a substitution S_0 such that $S' = S_0S$.

Type contexts. Type contexts bind identifiers and operators to types. Type contexts are the same, in the case of lambda- and `let`-bound identifiers as Damas-Milner assumption sets: each such identifier is bound exactly once to a type scheme. Overloaded operators, however, require more information: they are bound once to their signature, and further bindings record each type at which there is an instance. Hence, there are three forms of binding in a type context:

- $x : \sigma$ for lambda- and `let`-bound identifiers;
- $x :_o \sigma$ for the signature of an overloaded operator; and
- $x :_i \sigma$ for the instances of overloaded operators.

Type contexts are written Γ. Entries in type contexts are separated by semi-colons, and two type contexts can be joined by a semi-colon. Type contexts are unordered. The empty type context is denoted by ϵ, or frequently simply omitted.

Frequently, a *section* of a type context is required which contains only instance bindings of the form $x :_i \tau$. These sections are denoted by Π since they are closely related to predicates.

The notation $\Gamma \backslash \Gamma'$ denotes the type context derived from Γ by removing all bindings which also appear in Γ'.

A type context Γ' is said to be an instance of another type context Γ if there exists a substitution S and a section Π such that

$$S\Gamma;\ \Pi = \Gamma'.$$

That is, some substitution can be applied to Γ, and some instance bindings added, yielding Γ'. The pair (S, Π) is referred to as a *witness* to Γ' being an instance of Γ. In comparing type contexts, the order of the bindings is not significant.

The type context that is generated by the declarations in Figure 2.2 is given in Figure 3.1.

Note that, where the Damas-Milner system uses assumption sets, the system for OL uses bags or multisets. When the translation is discussed, type contexts are translated correctly into assumption sets.

Instance relation on type schemes. *Instance judgements*, of the form

$$\Gamma \rhd \sigma \geq \sigma',$$

denote when one type scheme, σ, is more general than another, σ', under a give type context. The set of rules for deriving instance judgements are given in Figure 3.2.

In the absence of predicates, instance judgements are equivalent to Damas-Milner's generic instances—the three rules **Taut**, **Gen** and **Spec** embody this. The **Pred**

$$
\begin{array}{lll}
\Gamma_0 = & (+) & :_o \forall\alpha.\alpha \to \alpha \to \alpha; \\
& (*) & :_o \forall\alpha.\alpha \to \alpha \to \alpha; \\
& (==) & :_o \forall\alpha.\alpha \to \alpha \to \mathit{Bool}; \\
& (+) & :_i \mathit{Int} \to \mathit{Int} \to \mathit{Int}; \\
& (+) & :_i \mathit{Float} \to \mathit{Float} \to \mathit{Float}; \\
& (*) & :_i \mathit{Int} \to \mathit{Int} \to \mathit{Int}; \\
& (*) & :_i \mathit{Float} \to \mathit{Float} \to \mathit{Float}; \\
& (==) & :_i \mathit{Int} \to \mathit{Int} \to \mathit{Bool}; \\
& (==) & :_i \mathit{Float} \to \mathit{Float} \to \mathit{Bool}; \\
& (==) & :_i \mathit{Char} \to \mathit{Char} \to \mathit{Bool}
\end{array}
$$

Figure 3.1: Example type context—Γ_0

and **Rel** rules are the introduction and elimination rules for predicates. **Pred** allows an instance of an overloaded operator to be *assumed.* This assumption is reflected with a predicate in the instance type scheme. **Rel** allows a predicate to be discharged. Given an instance judgement with a predicate on the right-hand side, the predicate can be removed if, and only if, there is an instance binding of the appropriate operator in the environment whose type is more general than the type of the predicate.

Though the rule based definition of the instance relation is used throughout, the definition may also be given in the style of Damas. Assuming the typing judgement form $\Gamma \rhd e : \sigma$ (which is defined below), the judgement

$$
\begin{aligned}
\Gamma \rhd \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \\
\geq \forall\beta_1 \cdots \forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'
\end{aligned}
$$

holds if

- There exists a substitution S of types for α_1 through α_n such that $S\tau = \tau'$,
- for each i in 1 to m the judgement

  $$\Gamma;\ x'_1 :_i \tau'_1;\ \cdots;\ x'_q :_i \tau'_q \rhd x_i : S\tau_i$$

  holds, and
- β_i, for i in 1 to p, are not free in Γ.

There is a subtle difference between the approach taken here and Damas' approach: in the last clause, Damas restricts β_i to be not free in

$$\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau.$$

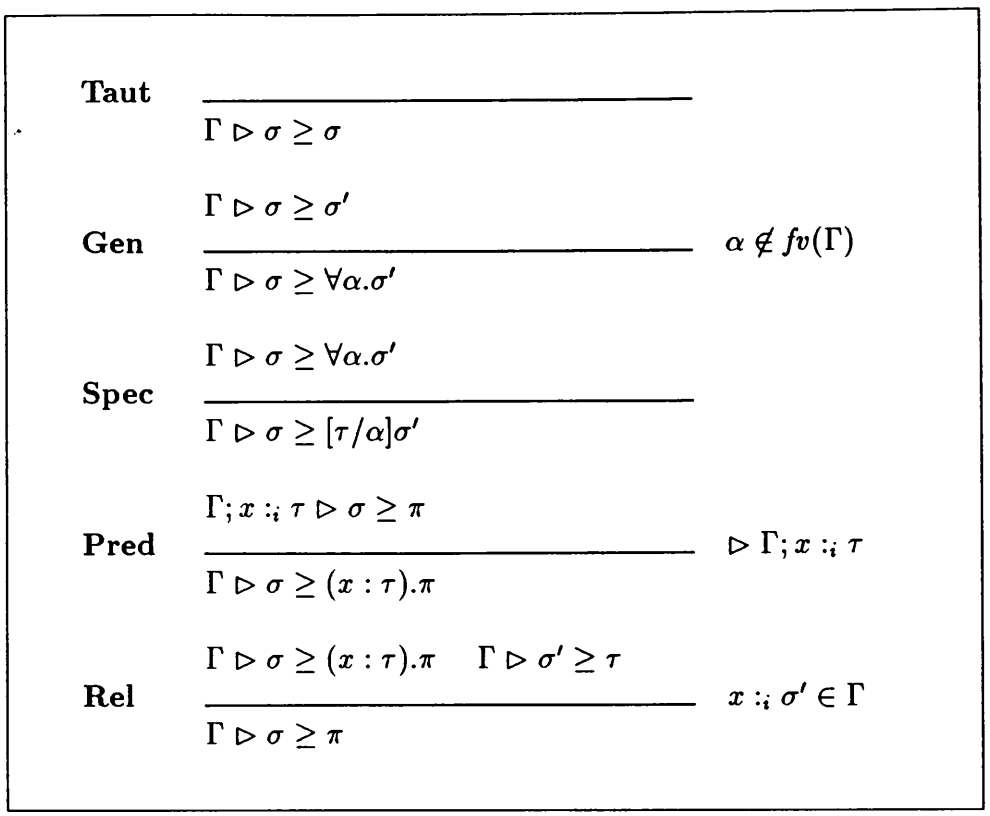

Figure 3.2: Instance judgement rules

This change is consistent with the approach taken to predicates and renders the judgement form to be a preorder as oppossed to a partial order.

Two types schemes are said to be trivial variants of each other if each is an instance of the other. That is, type schemes σ and σ' are trivial variants under a particular type context Γ if $\Gamma \rhd \sigma \geq \sigma'$ and $\Gamma \rhd \sigma' \geq \sigma$.

Figure 3.3 shows the derivation for the instance judgement below.

$$\Gamma_0 \rhd \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha \geq \mathit{Int} \to \mathit{Int}$$

The type on the left is the type assigned to *double* in Section 2.1.3. This instance judgement indicates that *double* can be used at type $\mathit{Int} \to \mathit{Int}$.

3.1.2 The validity condition

In order for a typing judgement or an instance judgement to make sense, it is required that the type context involved be *valid.* A valid type context satisfies the following:

- Each overloading binding is of the correct form—it contains no predicates or free type variables.

$$\boxed{Pl_\tau \stackrel{\text{def}}{=} (+) : \tau \to \tau \to \tau}$$

$$\textbf{Taut}\ \frac{}{\Gamma_0 \rhd \forall\alpha.(Pl_\alpha).\alpha \to \alpha \geq \forall\alpha.(Pl_\alpha).\alpha \to \alpha}$$

$$\textbf{Spec}\ \frac{\Gamma_0 \rhd \forall\alpha.(Pl_\alpha).\alpha \to \alpha \geq (Pl_{Int}).Int \to Int \quad (1)}{\Gamma_0 \rhd \forall\alpha.(Pl_\alpha).\alpha \to \alpha \geq Int \to Int}\ \textbf{Rel}$$

where (1) stands for the proof tree:

$$\textbf{Taut}\ \frac{(+) :_i Int \to Int \to Int \in \Gamma_0}{\Gamma_0 \rhd Int \to Int \to Int \geq Int \to Int \to Int}$$

Figure 3.3: Example instance judgement derivation

- For each instance binding, there is a corresponding overloaded operator binding characterising the type scheme bound for the instance.
- For every predicate which appears in a type scheme in the context there must be a corresponding overloaded operator whose signature matches the type in the predicate.

Validity judgements, of the form $\rhd \Gamma$, embody these requirements. The rules for deriving validity judgements are given in Figure 3.4. Therein, there are two forms of judgement: the first four rules are for type contexts and the last rule is for type schemes.

The first rule states, simply, that the empty context is valid. The **Over** rule asserts that, if x is not bound in Γ, and the given type scheme is of the appropriate shape—that is, contains no predicates or free type variables—then an overloading binding can be placed in the environment. The **Inst** rule ensures that every instance of an overloaded operator satisfies its signature.

The **Bind** rule, for lambda- and `let`-bound identifiers, asserts that the identifier must not previously be bound in Γ. Further, there is a requirement that the type scheme bound be valid. There is only one rule for judgements of the form $\Gamma \rhd \sigma$, this is the **Sig** rule. Judgements of this form are required only for bindings of the form $x : \sigma$ as types bound in $:_o$-bindings contain no predicates; and the required condition is subsumed by $\Gamma \rhd \sigma \geq \sigma'$ in the case of $:_i$-bindings.

Instance and typing judgements can only be made under valid type contexts. This can be considered to be a side condition of each instance and typing judgement

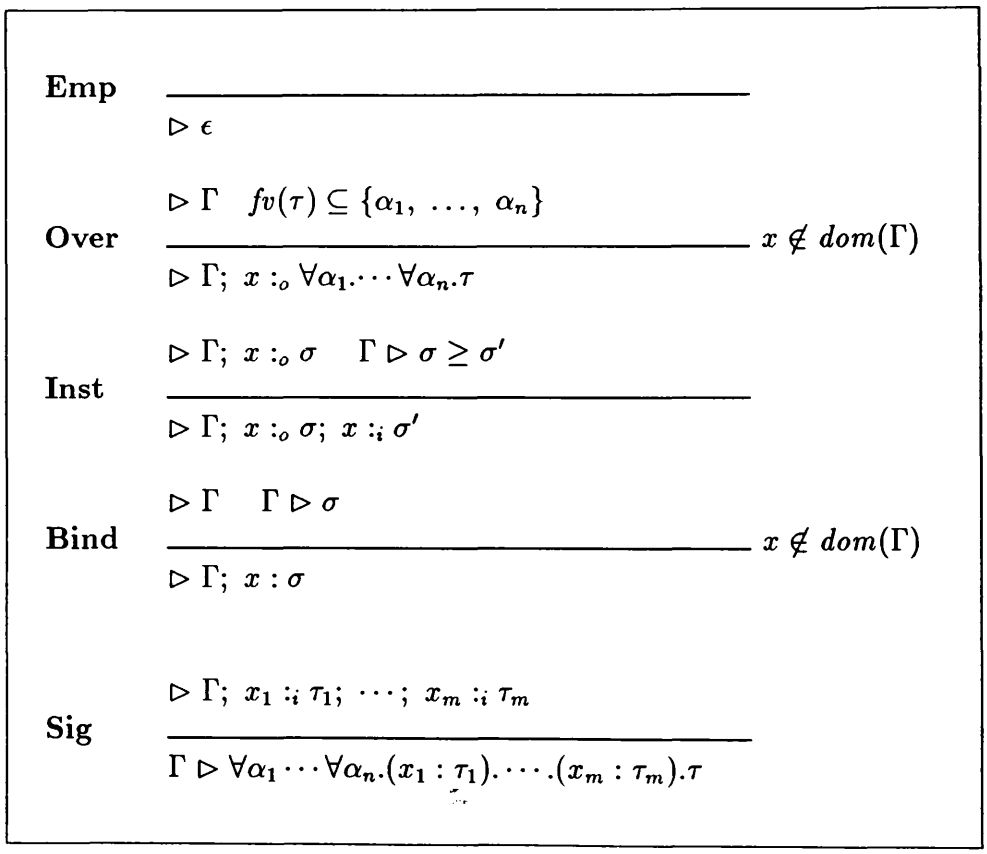

Figure 3.4: Validity judgement rules

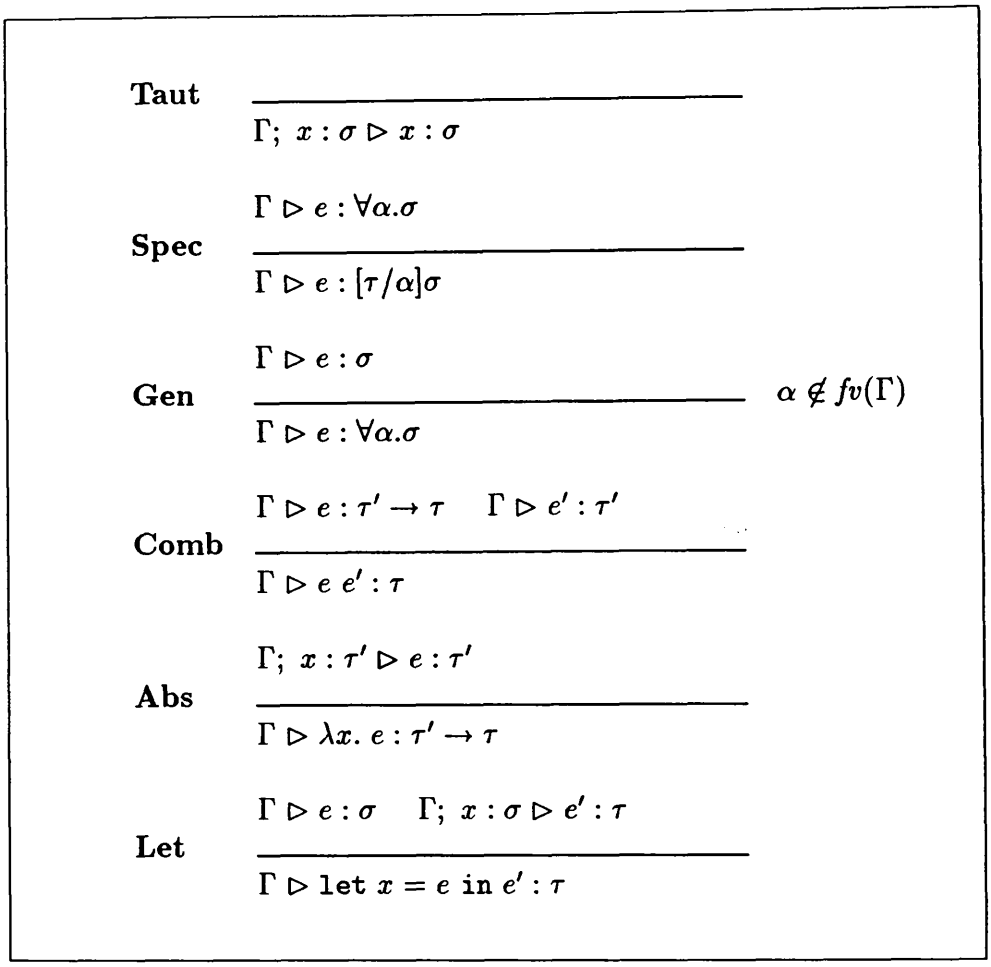

Figure 3.5: The first group of rules (the Damas-Milner rules)

rule. Since the definition of validity judgements depends on instance judgements, a recursion is formed. This recursion is well-founded: that is, there are no infinite chains of dependency between the two groups of rules.

3.2 Typing Judgements

The type rules for OL are based on the Damas-Milner rules [DaMi82, Dam85] which are presented in Section 1.4.

There are three groups of rules. The first group (Figure 3.5) are the Damas-Milner rules—the difference lies only in the underlying syntax of expressions, types, and type contexts. The second group (Figure 3.6) contains a tautology rule for instances of overloaded operators, and introduction and elimination rules for predicates. Finally, the third group (Figure 3.7) give rules for declarations and programs. All the rules apply only to valid type contexts.

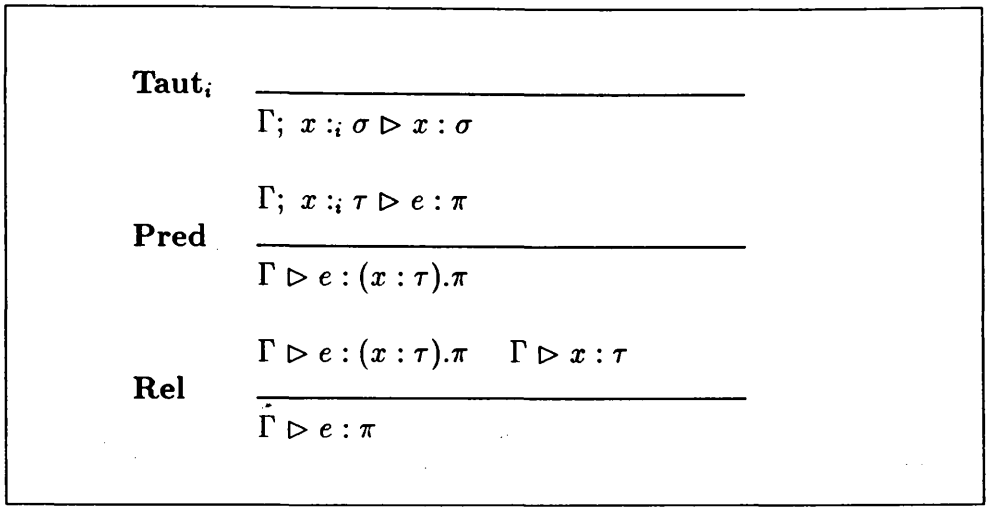

Figure 3.6: The second group of rules (handling overloading)

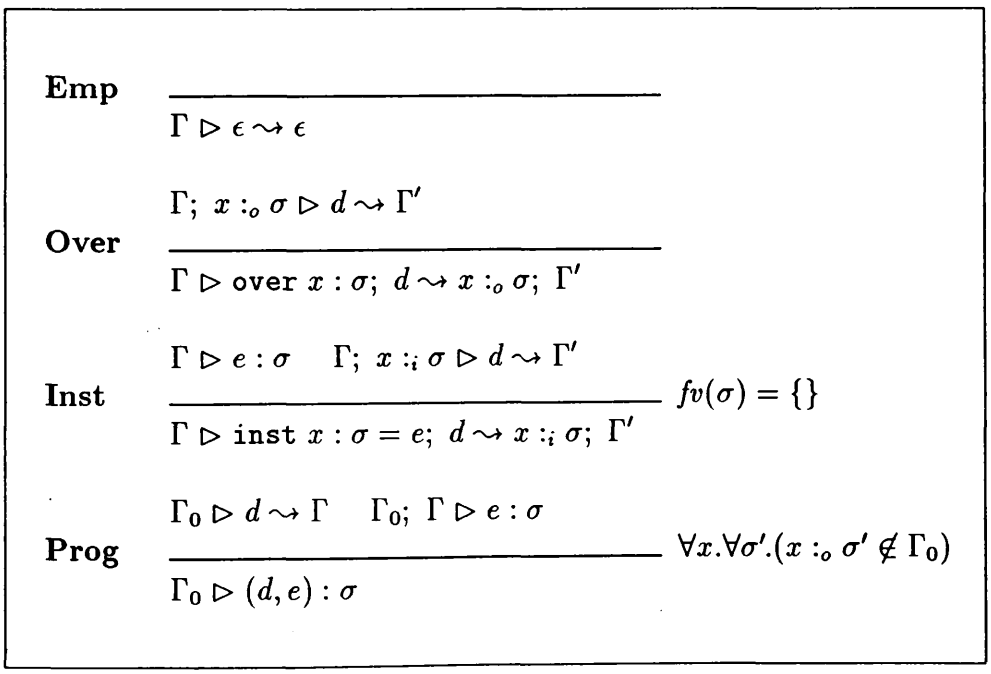

Figure 3.7: The third group of rules (declarations and programs)

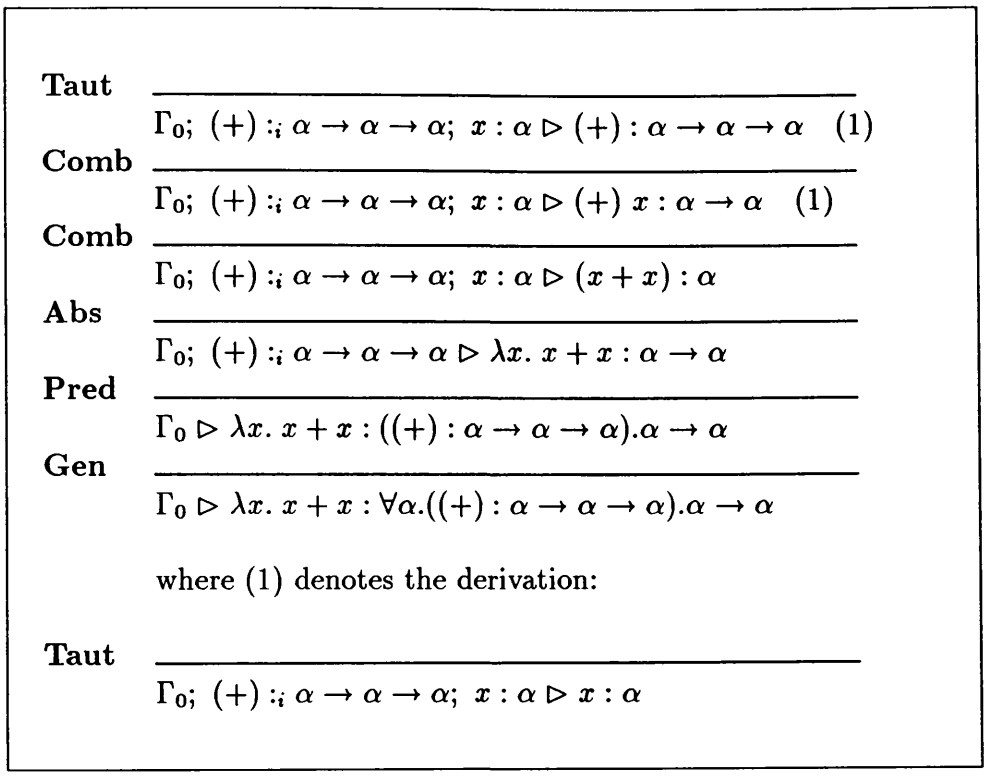

Figure 3.8: Example derivation—body of *double*

The Damas-Milner rules require no further explanation.

The second group of rules handle overloading. The **Taut**$_i$ rule types overloaded operators on the basis of an instance binding in the environment. The **Pred** rule allows an instance of an overloaded operator to be assumed, this assumption is then reflected with a predicate in the conclusion. The **Rel** rule discharges predicates. Given a derivation with a predicate in the type and a proof that the typing of the predicate is derivable, the predicate can be discharged.

The third group of rules handle declarations and programs. Extraction judgements are novel: they extract the overloading information from the declarations deterministically. Further, they ensure that the implementations given for instance declarations are at the correct type.

Two examples of type derivations in the OL calculus are given in Figures 3.8 and 3.9. The former shows the typing of the body of the *double* function from Section 2.1.3.

$$\Gamma_0 \rhd \lambda x.\ x + x : \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha$$

Notice, particularly, how the **Pred** rule is used to assume an instance of $(+)$ at the appropriate type. Figure 3.9 shows an application of *double* to an integer. Take

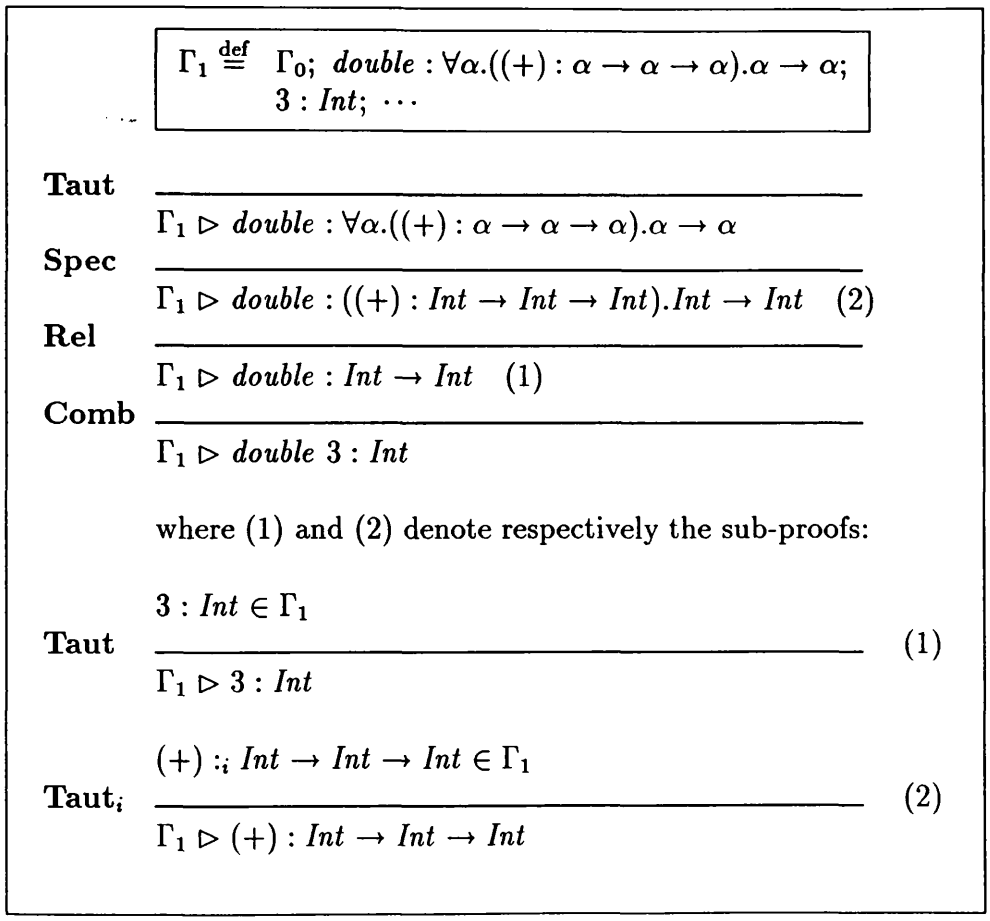

Figure 3.9: Example typing judgement derivation—application of *double*

Γ_1 to be Γ_0 with a binding for *double* at the type derived above, and bindings for integer literals. The derivation of the judgement

$$\Gamma_1 \rhd \mathit{double}\ 3 : \mathit{Int}$$

is given in the figure; it illustrates the use of the **Rel** rule to discharge a predicate. It is possible to combine these two proofs with the **Let** typing rule as follows.

$$\frac{\begin{array}{l}\Gamma_0 \rhd \lambda x.\ x + x : \forall\alpha.((+) : \alpha \to \alpha \to \alpha).\alpha \to \alpha \\ \Gamma_1 \rhd \mathit{double}\ 3 : \mathit{Int}\end{array}}{\Gamma_0 \rhd \mathtt{let}\ \mathit{double} = \lambda x.\ x + x\ \mathtt{in}\ \mathit{double}\ 3 : \mathit{Int}}\ \textbf{Let}$$

3.3 The type inference algorithm

This section presents the type inference algorithm for OL, algorithm $\mathcal{O}$. It corresponds to algorithm $\mathcal{W}$ of the Damas-Milner system. Note that algorithm $\mathcal{O}$ infers typings for expressions, not for programs.

Given a valid type context and a term, algorithm $\mathcal{O}$ infers a typing for the term under the context—if a typing exists. The typing is represented by three results:

- S, a substitution restricting the type context;
- Π, the instance bindings assumed in typing the term; and
- τ, the type of the term.

That is, if $\mathcal{O}$ is applied to a valid type context and an expression $\mathcal{O}(\Gamma, e)$, then the result is of the form (S, Π, τ). The typing thus inferred is

$$S\Gamma;\ \Pi \rhd e : \tau$$

which is the *characterisation rule* for $\mathcal{O}$. Algorithm $\mathcal{O}$ differs from $\mathcal{W}$ only with respect to predicates.

Some further notation is required to present $\mathcal{O}$, in particular, the notation of a *generalisation of a typing.* Given a section Π and a type τ, the notation $\Pi.\tau$ denotes the predicated type containing a predicate for each instance binding in Π. That is,

$$\Pi.\tau \stackrel{\text{def}}{=} (x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

where $\Pi = x_1 :_i \tau_1;\ \cdots;\ x_m :_i \tau_m$.

Given a type context Γ, a section Π and a type τ, the notation $\overline{\Gamma}(\Pi.\tau)$ denotes the generalisation of a typing: that is, the type scheme based on τ that contains a predicate for each instance binding in Π, and a quantified type variable for every type variable free in Π or τ but not free in Γ. The definitions is as follows

$$\overline{\Gamma}([x_1 :_i \tau_1;\ \cdots;\ x_m :_i \tau_m].\tau) \stackrel{\text{def}}{=} \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

where $\{\alpha_1, \cdots, \alpha_n\}$ are $fv(\tau, \tau_1, \cdots, \tau_m)$ without $fv(\Gamma)$.

Algorithm $\mathcal{O}$ is given in Figure 3.10; it extends $\mathcal{W}$ to handle predicates and type overloaded identifiers.

Algorithm $\mathcal{O}$ fails under the following three conditions: if a call of U fails to find a unifying substitution; if a variable is not bound in the type context when required; or if an invalid type context is constructed. On encountering one of these conditions, the entire activation fails and no typing exists.

$\mathcal{O}$ does *not* implement identically the algorithm outlined in Chapter 2. Predicates are treated passively by $\mathcal{O}$, they are merely propagated. There is no attempt to discharge predicates as was done previously. Under this simplification, $\mathcal{O}$ computes principal typings—as is discussed in Section 5.3. An extension, modelling the algorithm outlined in Chapter 2, is given in Chapter 6 along with some discussion of the need for this dichotomy.

3.4 A translation semantics

This section presents the semantics of OL. This is given by means of a translation into the Damas-Milner calculus [DaMi82, Dam85]. The translation maps *typing derivations* in the calculus of OL into *typing derivations* in the Damas-Milner calculus. The version of the Damas-Milner calculus appearing in Chapter II of Damas' thesis [Dam85] is used; this is the version discussed informally in Section 1.4. Damas-Milner type schemes and assumption sets are denoted by σ and A respectively; and expressions are denoted by e as before. The Damas-Milner typing judgement form is denoted by $A \vdash e : \sigma$.

A mapping to the Damas-Milner system. As a preliminary, it is necessary to define a mapping for type schemes and type contexts from the OL calculus to those of the Damas-Milner system. This mapping, denoted $\lfloor\sigma\rfloor$ and $\lfloor\Gamma\rfloor$, maps type schemes to Damas-Milner type schemes, and type contexts to Damas-Milner assumption sets, respectively. The mapping for type schemes is defined as follows:

$$\lfloor\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau\rfloor \stackrel{\text{def}}{=} \forall\alpha_1 \cdots \forall\alpha_n.\tau_1 \to \cdots \to \tau_m \to \tau$$

that is, each predicate in the type scheme adds a function type in the Damas-Milner system.

$\mathcal{O}((\Gamma;\ x:\sigma),\ x) =$	If $\sigma = \forall\alpha_1 \cdots \forall\alpha_n.(x_1:\tau_1).\cdots.(x_m:\tau_m).\tau$, and S_β is the substitution $[\beta_1/\alpha_1, \cdots, \beta_m/\alpha_m]$ where $\beta_1, \cdots, \beta_n$ are fresh, then the typing is $(Id,\ [x_1 :_i S_\beta\tau_1;\ \cdots;\ x_m :_i S_\beta\tau_m],\ S_\beta\tau)$.
$\mathcal{O}((\Gamma;\ x :_o \sigma),\ x) =$	If $\sigma = \forall\alpha_1.\cdots\forall\alpha_n.\tau$, and S_β is the substitution $[\beta_1/\alpha_1, \cdots, \beta_m/\alpha_m]$ where $\beta_1, \cdots, \beta_n$ are fresh, then the typing is $(Id,\ [x :_i S_\beta\tau],\ S_\beta\tau)$.
$\mathcal{O}(\Gamma,\ (e_1\ e_2)) =$	Make the recursive calls $(S_1, \Pi_1, \tau_1) = \mathcal{O}(\Gamma, e_1)$ $(S_2, \Pi_2, \tau_2) = \mathcal{O}(S_1\Gamma, e_2)$ and set S_u to be $U(S_2\tau_1,\ \tau_2 \to \beta)$ where β is fresh. The typing is then $(S_uS_2S_1,\ (S_u\Pi_2;\ S_uS_2\Pi_1),\ S_u\beta)$.
$\mathcal{O}(\Gamma,\ (\lambda x.\ e)) =$	Make the recursive call $(S_1, \Pi_1, \tau_1) = \mathcal{O}((\Gamma;\ x:\beta),\ e)$ where β is fresh. The typing is then $(S_1,\ \Pi_1,\ S_1\beta \to \tau_1)$.
$\mathcal{O}(\Gamma,\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)) =$	Make the recursive calls $(S_1, \Pi_1, \tau_1) = \mathcal{O}(\Gamma, e_1)$ $(S_2, \Pi_2, \tau_2) = \mathcal{O}((S_1\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1)),\ e_2)$ and return the typing $(S_2S_1,\ \Pi_2,\ \tau_2)$.

Figure 3.10: Algorithm $\mathcal{O}$

It is slightly more complicated to describe the mapping of type contexts. It is required that there exist an injective function assigning an identifier to each $:_i$-binding appearing in the derivation at hand. Call this function $\lfloor . \rfloor$: that is, $\lfloor x :_i \sigma \rfloor$ denotes some unique variable to be used as the name of the given instance. Further, the identifiers in the range of $\lfloor . \rfloor$ may not appear elsewhere in the derivation at hand. Two instance bindings of the same type to the same operator are considered to be different if they were created separately. That is, it is *not* the case that

$$x :_i \tau;\ x :_i \tau = x :_i \tau;$$

and $\lfloor . \rfloor$ distinguishes such bindings.

As an example, if the type context at hand were Γ_0 of Figure 3.1, then $\lfloor . \rfloor$ could be defined by the partial function

$$\begin{array}{lcl}
(+) :_i \mathit{Int} \to \mathit{Int} \to \mathit{Int} & \longmapsto & (+_I) \\
(+) :_i \mathit{Float} \to \mathit{Float} \to \mathit{Float} & \longmapsto & (+_F) \\
(*) :_i \mathit{Int} \to \mathit{Int} \to \mathit{Int} & \longmapsto & (*_I) \\
(*) :_i \mathit{Float} \to \mathit{Float} \to \mathit{Float} & \longmapsto & (*_F) \\
(==) :_i \mathit{Int} \to \mathit{Int} \to \mathit{Bool} & \longmapsto & (==_I) \\
(==) :_i \mathit{Float} \to \mathit{Float} \to \mathit{Float} & \longmapsto & (==_F) \\
(==) :_i \mathit{Char} \to \mathit{Char} \to \mathit{Char} & \longmapsto & (==_C)
\end{array}$$

assuming identifiers $(+_I)$ etc. appear nowhere else in the derivation at hand. It is assumed throughout that an appropriate $\lfloor . \rfloor$ is defined.

The mapping for valid type contexts then maps :-bindings to :-bindings, removes $:_o$-bindings, and replaces $:_i$-bindings with :-bindings of the appropriate identifier. It is defined inductively as follows.

$$\begin{array}{lcll}
\lfloor (x : \sigma;\ \Gamma) \rfloor & \stackrel{\text{def}}{=} & x : \lfloor \sigma \rfloor;\ \lfloor \Gamma \rfloor & \\
\lfloor (x :_o \sigma;\ \Gamma) \rfloor & \stackrel{\text{def}}{=} & \lfloor \Gamma \rfloor & \\
\lfloor (x :_i \sigma;\ \Gamma) \rfloor & \stackrel{\text{def}}{=} & \overline{x} : \lfloor \sigma \rfloor;\ \lfloor \Gamma \rfloor & \text{where } \overline{x} = \lfloor x :_i \sigma \rfloor
\end{array}$$

The translation rules. The translation rules are given in Figure 3.11. Therein, each application of a rule on the left in a derivation is replaced with an application of the corresponding rule on the right. Expressions denoted $\overline{e}$ are the synthesised translations of the term at hand. Given a derivation $\Gamma \rhd e : \sigma$, if the translation is $A \vdash \overline{e} : \lfloor \sigma \rfloor$, then the translation $\overline{e}$ denotes the meaning of e under the given derivation; $\overline{e}$ may now be interpreted by some other appropriate technique, for example, a denotational semantics.

The translation is straight-forward in all rules except **Pred** and **Rel**. As is seen in the informal translation in Chapter 2 and in the mapping of type schemes defined above, predicates in a typing correspond to parametrisation in the translation.

$$\textbf{Taut}\quad \frac{}{\Gamma;\ x:\sigma \rhd x:\sigma} \quad\rightsquigarrow\quad \frac{}{\lfloor\Gamma\rfloor;\ x:\lfloor\sigma\rfloor \vdash x:\lfloor\sigma\rfloor}$$

$$\textbf{Spec}\quad \frac{\Gamma \rhd e:\forall\alpha.\,\sigma}{\Gamma \rhd e:[\tau/\alpha]\sigma} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor \vdash \overline{e}:\forall\alpha.\,\lfloor\sigma\rfloor}{\lfloor\Gamma\rfloor \vdash \overline{e}:[\tau/\alpha]\lfloor\sigma\rfloor}$$

$$\textbf{Gen}\quad \frac{\Gamma \rhd e:\sigma \qquad \alpha \notin fv(\Gamma)}{\Gamma \rhd e:\forall\alpha.\,\sigma} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor \vdash \overline{e}:\lfloor\sigma\rfloor \qquad \alpha \notin fv(\lfloor\Gamma\rfloor)}{\lfloor\Gamma\rfloor \vdash \overline{e}:\forall\alpha.\,\lfloor\sigma\rfloor}$$

$$\textbf{Comb}\quad \frac{\Gamma \rhd e:(\tau' \to \tau) \qquad \Gamma \rhd e':\tau'}{\Gamma \rhd (e\ e'):\tau} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor \vdash \overline{e}:(\tau' \to \tau) \qquad \lfloor\Gamma\rfloor \vdash \overline{e}':\tau'}{\lfloor\Gamma\rfloor \vdash (\overline{e}\ \overline{e}'):\tau}$$

$$\textbf{Abs}\quad \frac{\Gamma;\ x:\tau' \rhd e:\tau}{\Gamma \rhd (\lambda x.\,e):(\tau' \to \tau)} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor;\ x:\tau' \vdash \overline{e}:\tau}{\lfloor\Gamma\rfloor \vdash (\lambda x.\,\overline{e}):(\tau' \to \tau)}$$

$$\textbf{Let}\quad \frac{\Gamma \rhd e:\sigma \qquad \Gamma;\ x:\sigma \rhd e':\tau}{\Gamma \rhd (\mathtt{let}\ x = e\ \mathtt{in}\ e'):\tau} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor \vdash \overline{e}:\lfloor\sigma\rfloor \qquad \lfloor\Gamma\rfloor;\ x:\lfloor\sigma\rfloor \vdash \overline{e}':\tau}{\lfloor\Gamma\rfloor \vdash (\mathtt{let}\ x = \overline{e}\ \mathtt{in}\ \overline{e}'):\tau}$$

$$\textbf{Taut}_i\quad \frac{}{\Gamma;\ x :_i \sigma \rhd x:\sigma} \quad\rightsquigarrow\quad \frac{}{\lfloor\Gamma;\ x :_i \sigma\rfloor \vdash \lfloor x :_i \sigma\rfloor : \lfloor\sigma\rfloor}$$

$$\textbf{Pred}\quad \frac{\Gamma;\ x :_i \tau \rhd e:\pi}{\Gamma \rhd e:(x:\tau).\,\pi} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma;\ x :_i \tau\rfloor \vdash \overline{e}:\lfloor\pi\rfloor \qquad \overline{x} = \lfloor x :_i \tau\rfloor}{\lfloor\Gamma\rfloor \vdash \lambda\overline{x}.\,\overline{e}:(\tau \to \lfloor\pi\rfloor)}$$

$$\textbf{Rel}\quad \frac{\Gamma \rhd e:(x:\tau).\,\pi \qquad \Gamma \rhd x:\tau}{\Gamma \rhd e:\pi} \quad\rightsquigarrow\quad \frac{\lfloor\Gamma\rfloor \vdash \overline{e}:\tau \to \lfloor\pi\rfloor \qquad \lfloor\Gamma\rfloor \vdash \overline{e}':\tau}{\lfloor\Gamma\rfloor \vdash \overline{e}\ \overline{e}':\lfloor\pi\rfloor}$$

Figure 3.11: Translation rules

Thus each application of the **Pred** rule is translated to an application of the Damas-Milner **Abs** rule.

Similarly, discharge of a predicate in Chapter 2 is modelled by an application to an appropriate dictionary. Applications of the **Rel** rule in a derivation are replaced by applications of the **Comb** rule in the translation. The derivation of the typing of the instance corresponds to the dictionary or implementation of the overloaded operator.

The type inference algorithm $\mathcal{O}$ can be extended to yield a term representing the translation of the derivation it computes.

Extracting the meaning from a derivation. It is convenient to define a short-hand notation for the process of extracting the meaning from a given derivation.

If Δ is some derivation of $\Gamma \rhd e : \sigma$, then $\lfloor \Delta \rfloor$ denotes e^* where e^* is extracted from the judgement

$$\lfloor \Gamma \rfloor \vdash e^* : \lfloor \sigma \rfloor,$$

the translation of Δ in the Damas-Milner calculus.

Chapter 4

Admissable rules

Hindley and Seldin define the concept of *admissable rules* on page 70 of [HS86]. Several admissable rules are implied by the material of the previous chapter. Before continuing, it is convenient to present some of these. These rules add nothing new to the system but are conveniently collected together for subsequent use in proofs.

Since the proofs of the rules are trivial, they are omitted. Some indication of the appropriate proof method is given in each case.

Admissable Rule$_1$ – If a substitution is applied to the type context and type part of a derivable typing judgement, then the resulting judgement is derivable.

$$\textbf{Rule}_1 \qquad \frac{\Gamma \rhd e : \sigma}{S\Gamma \rhd e : S\sigma}$$

The proof is by induction on the typing rules. The only subtlety is to ensure, by renaming, that no generic variables appear in S.

Admissable Rule$_2$ – The second substitution rule is similar to the first though applicable to instance, rather than typing, judgements.

$$\textbf{Rule}_2 \qquad \frac{\Gamma \rhd \sigma \geq \sigma'}{S\Gamma \rhd S\sigma \geq S\sigma'}$$

The proof is very similar to the proof of the previous substitution rule.

Admissable Rule$_3$ – Non-structural typing derivations and instance judgements are equivalent. Notice that e is unchanged in the conclusion.

$$\textbf{Rule}_3 \qquad \frac{\Gamma \rhd e : \sigma \qquad \Gamma \rhd \sigma \geq \sigma'}{\Gamma \rhd e : \sigma'}$$

This rule embodies the similarity between the non-structural typing and the instance judgement rules. There are two other useful forms of this rule:

$$\textbf{Rule}_{3'}\quad \frac{\dfrac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'}}{\Gamma \rhd \sigma \geq \sigma'}$$

That is: if the upper derivation holds, then the lower judgement holds.

$$\textbf{Rule}_{3''}\quad \frac{\dfrac{\Gamma \rhd e : \sigma' \quad \Gamma \rhd \sigma \geq \sigma'}{\Gamma \rhd e : \sigma}}{\Gamma \rhd e : \sigma'}$$

That is: if the upper judgements hold, then the lower derivation holds.

The proof in each case is by induction, and hinges on the similarity between the non-structural typing rules and the instance judgement rules.

Admissable Rule$_4$ – The instance judgement relation is transitive.

$$\textbf{Rule}_4\quad \frac{\begin{array}{c}\Gamma \rhd \sigma \geq \sigma' \\ \Gamma \rhd \sigma' \geq \sigma''\end{array}}{\Gamma \rhd \sigma \geq \sigma''}$$

The proof is by induction on the structure of the derivation of $\Gamma \rhd \sigma \geq \sigma'$.

Admissable Rule$_5$ – This rule parallels for predicates the behaviour of the substitution rule for type variables. This and the following rule are forms of weakening.

$$\textbf{Rule}_5\quad \frac{\Gamma \rhd e : \sigma}{\Gamma;\ \Gamma' \rhd e : \sigma}$$

That is, the type context of a typing judgement, may be augmented with an arbitrary type context yielding a valid typing judgement. Note that, as ever, the composition $(\Gamma;\ \Gamma')$ is required to be a valid type context. The proof is by induction, though trivial.

Admissable Rule$_6$ – The type context for an instance judgement may also be augmented with an arbitrary type context.

$$\textbf{Rule}_6\quad \frac{\Gamma \rhd \sigma \geq \sigma'}{\Gamma;\ \Gamma' \rhd \sigma \geq \sigma'}$$

Admissable Rule$_7$ – The generalisation process is sound for typing judgements.

$$\textbf{Rule}_7 \qquad \frac{\Gamma;\ \Pi \rhd e : \tau}{\Gamma \rhd e : \overline{\Gamma}(\Pi.\tau)}$$

The proof is by construction: so many applications of **Pred** followed by so many applications of **Gen**. The reverse of this rule also holds.

Admissable Rule$_8$ – The generalisation process is sound for instance judgements.

$$\textbf{Rule}_8 \qquad \frac{\Gamma;\ \Pi \rhd \sigma \geq \tau}{\Gamma \rhd \sigma \geq \overline{\Gamma}(\Pi.\tau)}$$

The reverse of this rule also holds.

Admissable Rule$_9$ – Under specific circumstances, a section which can be discharged in one context can also be discharged in another.

$$\textbf{Rule}_9 \qquad \frac{\begin{array}{l}\Gamma \rhd \pi \geq \sigma \\ \Gamma \rhd \Pi.\tau \geq \tau\end{array}}{\Gamma \rhd \Pi.\pi \geq \sigma} \quad (fv(\Pi) \setminus (fv(\tau) \cup fv(\Gamma))) \cap fv(\pi) = \{\}$$

If Π contains $x_i :_i \tau_i$ for $i = 1, \ldots, n$, then from the $\Gamma \rhd \Pi.\tau \geq \tau$ it follows $\Gamma \rhd x_i : S\tau_i$ for some S mapping only type variables not free in τ or Γ. If those type variables do not appear in π, then $\Gamma \rhd \Pi.\pi \geq \pi$ giving the required conclusion by transitivity.

Admissable Rule$_{10}$ – Predicates may be dropped from the left-hand side of an instance judgement.

$$\textbf{Rule}_{10} \qquad \frac{\Gamma \rhd (x : \tau).\pi \geq \sigma'}{\Gamma \rhd \pi \geq \sigma'}$$

The proof is by construction of $\Gamma \rhd \pi \geq (x : \tau).\pi$ and transitivity.

This rule also holds in the reverse direction under the side condition that x is an overloaded operator and $\Gamma \rhd x : \tau$.

Admissable Rule$_{11}$ – This rule makes an assertion about the form of instance judgements. Specifically, if

$$\Gamma \rhd \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \geq \tau'$$

then there exists a substitution S of types for $\{\alpha_1, \ldots, \alpha_n\}$ such that:

- $S\tau = \tau'$; and
- $\Gamma \rhd x_i : S\tau_i$ for each $i \in \{1, \ldots, m\}$.

As a corollary, $\Gamma \rhd (x_1 : S\tau_1).\cdots(x_m : S\tau_m).S\tau \geq \tau'$ also holds since each of the predicates may be discharged and $S\tau = \tau'$.

The proof of this result relies on the existence of canonical derivations. This form is given in Chapter 7. Specifically, any instance judgement of the given form can be constructed using only the **Taut**$_i$, **Spec** and **Rel** rules.

Admissable Rule$_{12}$ – If a derivation holds under a particular context, then a binding can be replaced with a binding of the same identifier to a more general type and the derivations still holds.

$$\textbf{Rule}_{12} \quad \frac{\Gamma;\ x : \sigma' \rhd e : \sigma'' \qquad \Gamma \rhd \sigma \geq \sigma'}{\Gamma;\ x : \sigma \rhd e : \sigma''}$$

This follows from the relationship between non-structural typing rules and instance judgement rules (**Rule**$_3$).

Chapter 5

Soundness, completeness and principal types

This Chapter presents the main technical results and proofs thereof. In particular, algorithm $\mathcal{O}$ is shown to be syntactically sound and complete; as such it computes *principal typings*; finally, the translation semantics are shown to be syntactically sound. The main results are as follows

Syntactic soundness theorem. $\mathcal{O}$ is syntactically sound. That is, given a type environment and an expression (Γ, e), if $\mathcal{O}$ computes a typing (S, Π, τ) then a derivation of $S\Gamma \rhd e : \tau$ is provable in the calculus.

Syntactic completeness theorem. $\mathcal{O}$ is syntactically complete. That is, given a type environment and expression (Γ, e), if any typing (S', Π', τ') exists, then $\mathcal{O}$ will compute a typing. Further, the computed typing is more general, with respect to the instance relation, than the given typing.

Principal type theorem. The type calculus satisfies a principal type theorem, further, $\mathcal{O}$ computes principal types.

Translation soundness theorem. Given an arbitrary derivation in the OL calculus, the translation of that derivation into the Damas-Milner calculus yields a valid derivation.

5.1 $\mathcal{O}$ is syntactically sound

$\mathcal{O}$ is sound: if $\mathcal{O}$ computes a typing then a derivation of the typing exists within the type calculus. Specifically:

Theorem 1 Syntactic soundness. *Given a valid type context* Γ *and an expression* e*, if the evaluation of* $\mathcal{O}(\Gamma, e)$ *succeeds with* (S, Π, τ)*, then the judgement* $S\Gamma; \Pi \rhd e : \tau$ *is derivable.*

Proof

The proof is by structural induction on the term e requiring two base cases and three inductive cases. In each case, the typing derivation corresponding to the result of the application of $\mathcal{O}$ at hand is construct. The names of the outer-most invocation of $\mathcal{O}$, see Figure 3.10, are used throughout.

Identifier terms. There are two mutually exclusive cases: firstly lambda- and `let`-bound identifiers, and secondly overloaded operators. If $\mathcal{O}$ succeeds, then there is an appropriate binding in the environment. In the lambda- and `let`-bound case the following derivation holds.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \in \Gamma}}{\Gamma; \Pi \rhd x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau}\ \textbf{Taut}}{\Gamma; \Pi \rhd x : [\beta_1/\alpha_1]\forall\alpha_2.\cdots\forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau}\ \textbf{Spec}}{\vdots}}{\Gamma; \Pi \rhd x : S_\beta(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau}\ \begin{array}{l}\textbf{Spec } [\beta_i/\alpha_i] \\ \text{for } i = 2, \ldots, n\end{array}}{\Gamma; \Pi \rhd x : S_\beta(x_2 : \tau_2).\cdots(x_m : \tau_m).\tau}\ \textbf{Rel } x_1 :_i S_\beta\tau_1 \in \Pi}{\vdots}}{\Gamma; \Pi \rhd x : S_\beta\tau}\ \begin{array}{l}\textbf{Rel } x_i :_i S_\beta\tau_i \in \Pi \\ \text{for } i = 2, \ldots, m\end{array}}{S\Gamma;\ \Pi \rhd x : S_\beta\tau}\ S = Id$$

For the second case, overloaded operators, the derivation is simpler.

$$\dfrac{\dfrac{}{x :_i S_\beta\tau \in \Pi}}{\Gamma; \Pi \rhd x : S_\beta\tau}\ \textbf{Taut}_i}{S\Gamma;\ \Pi \rhd x : S_\beta\tau}\ S = Id$$

Lambda terms. Starting from the inductive hypothesis, the following derivation holds.

$$\dfrac{\dfrac{S_1(\Gamma;\ x : \beta);\ \Pi_1 \rhd e : \tau_1}{S_1\Gamma;\ x : S_1\beta;\ \Pi_1 \rhd e : \tau_1}}{S_1\Gamma;\ \Pi_1 \rhd \lambda x.\ e : S_1\beta \to \tau_1}\ \textbf{Abs}$$

This typing is as required by the assignment to S, Π and τ of S_1, Π_1 and $S_1\beta \to \tau_1$ respectively, as $\mathcal{O}$ does.

Application terms. For application terms, there are two inductive hypotheses. For each, a short derivation achieves the correct form for an application of the **Comb** rule to construct the required typing. Below are the two sub-derivations from the two inductive hypotheses followed by an application of the **Comb** rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_1\Gamma;\ \Pi_1 \rhd e_1 : \tau_1}{S_uS_2S_1\Gamma;\ S_uS_2\Pi_1 \rhd e_1 : S_uS_2\tau_1}\ \textbf{Rule}_1\ S_uS_2}{S_uS_2S_1\Gamma;\ S_uS_2\Pi_1 \rhd e_1 : S_u\tau_2 \to S_u\beta}\ S_uS_2\tau_1 = S_u\tau_2 \to S_u\beta}{S_uS_2S_1\Gamma;\ S_uS_2\Pi_1;\ S_u\Pi_2 \rhd e_1 : S_u\tau_2 \to S_u\beta}\ \textbf{Rule}_5\ S_u\Pi_2}{(1)}$$

$$\dfrac{\dfrac{S_2S_1\Gamma;\ \Pi_2 \rhd e_2 : \tau_2}{S_uS_2S_1\Gamma;\ S_u\Pi_2 \rhd e_2 : S_u\tau_2}\ \textbf{Rule}_1\ S_u}{S_uS_2S_1\Gamma;\ S_u\Pi_2;\ S_uS_2\Pi_1 \rhd e_2 : S_u\tau_2}\ \textbf{Rule}_5\ S_uS_2\Pi_1$$

(2)

$$\dfrac{(1) \quad (2)}{S_uS_2S_1\Gamma;\ S_u\Pi_2;\ S_uS_2\Pi_1 \rhd e_1\ e_2 : S_u\beta}\ \textbf{Comb}$$

This final typing being as required by the assignment to S, Π and τ of $S_uS_2S_1$, $S_uS_2\Pi_1; S_u\Pi_2$, and $S_u\beta$ respectively, as $\mathcal{O}$ does.

Let terms. As with the previous case, the two inductive hypotheses lead to two sub-derivations. The first is again named (1) and combined with the second by an application of the structural rule for `let` terms.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_1\Gamma;\ \Pi_1 \rhd e_1 : \tau_1}{S_1\Gamma \rhd e_1 : \overline{S_1\Gamma}(\Pi_1.\tau_1)}\ \textbf{Rule}_7}{S_2S_1\Gamma \rhd e_1 : S_2\overline{S_1\Gamma}(\Pi_1.\tau_1)}\ \textbf{Rule}_1\ S_2}{S_2S_1\Gamma;\ \Pi_2 \rhd e_1 : S_2\overline{S_1\Gamma}(\Pi_1.\tau_1)}\ \textbf{Rule}_5\ \Pi_2}{(1)}$$

$$\dfrac{(1) \quad S_2S_1\Gamma;\ x : S_2\overline{S_1\Gamma}(\Pi_1.\tau_1);\ \Pi_2 \rhd e_2 : \tau_2}{S_2S_1\Gamma \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \textbf{Let}$$

This final typing being as required by the assignment to S, Π and τ of S_2S_1, Π_2, and τ_2 respectively, as $\mathcal{O}$ does.

It remains, finally, to show that all the type contexts constructed in such a derivation are valid. To see this, observing that Γ is valid and that all the alterations to Γ preserve its validity. It is necessary to examine every way in which Γ is altered.

- Substitutions are applied to the type context. Since all type variables in overloaded operator signatures are generic, validity before implies validity after.
- Lambda bound identifiers are added to the context, since these cannot contain predicates, validity is trivially maintained.
- Instance bindings are added to the environment. These are from two sources: the predicate part of a binding in the environment, or a specialisation of an overloaded operator's signature. The latter is trivially valid and the validity of the former derives from the validity of the predicates within the type scheme from which they originate.
- Let bindings are added to the environment. Since the predicates in such type schemes are generated from instance bindings in a valid type context, the resulting type context is also valid.

□

5.2 $\mathcal{O}$ is syntactically complete

Recall that one type context Γ' is an instance of another type context Γ if, and only if, there exists a substitution S and a section Π such that $\Gamma' = S\Gamma;\ \Pi$. The pair (S, Π) is referred to as a *witness* to the instance relation.

Theorem 2 Syntactic completeness. *Given a valid type context Γ and an expression e, if there exists a type context Γ', an instance of Γ, and a type scheme σ' such that $\Gamma' \rhd e : \sigma'$ then:*

1. *$\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ succeeds;*
2. *Γ' is an instance of $S\Gamma$ with witness (S', Π'); and*
3. *taking $\sigma = \overline{S\Gamma}(\Pi.\tau)$, it is the case that $\Gamma' \rhd S'\sigma \geq \sigma'$.*

Proof

The proof is by structural induction on the derivation of $\Gamma' \rhd e : \sigma'$. This is taken in two parts: firstly, establish the results if the last rule applied were one of **Gen**, **Spec**, **Pred** or **Rel** (in these cases termination comes trivially from the inductive hypotheses); and secondly, establish the results when the structural rule corresponding to the form of the expression has been used. The base cases **Taut** and $\mathbf{Taut}_i$ are considered in the second half of the proof.

As before, the names of the results and the variables of the outermost call of $\mathcal{O}$ are used.

The Gen rule. If the last step of the derivation is an application of the **Gen** rule then it must be of the form

$$\frac{\Gamma' \rhd e : \sigma''}{\Gamma' \rhd e : \sigma'} \quad \alpha \notin fv(\Gamma')$$

where $\sigma' = \forall\alpha.\sigma''$. That Γ' is an instance of Γ follows trivially from the inductive hypotheses. Further, the inductive hypotheses give that $\Gamma' \rhd S'\sigma \geq \sigma''$. Since this application of the **Gen** rule is applicable, it must be the case that $\alpha \notin fv(\Gamma')$. Therefore, the **Gen** instance rule is also applicable

$$\frac{\dfrac{\Gamma' \rhd S'\sigma \geq \sigma''}{\Gamma' \rhd S'\sigma \geq \forall\alpha.\sigma''}\ \mathbf{Gen}}{\Gamma' \rhd S'\sigma \geq \sigma'} \quad \forall\alpha.\sigma'' = \sigma'$$

giving the required instance judgement.

The Spec rule. If the last step of the derivation is an application of the **Spec** rule then it must be of the form

$$\frac{\Gamma' \rhd e : \forall\alpha.\sigma''}{\Gamma' \rhd e : \sigma'}$$

where $\sigma' = [\tau/\alpha]\sigma''$. That Γ' is an instance of Γ follows trivially from the inductive hypotheses. Further, the inductive hypotheses give that $\Gamma' \rhd S'\sigma \geq \forall\alpha.\sigma''$. A single application of the **Spec** instance rule gives $\Gamma' \rhd S'\sigma \geq \sigma'$, the required instance judgement.

The Pred rule. If the last step of the derivation is an application of the **Pred** rule then it must be of the form

$$\frac{\Gamma';\ x :_i \tau \rhd e : \pi}{\Gamma' \rhd e : \sigma'}$$

where $\sigma' = (x : \tau).\pi$. The inductive hypotheses give that $\Gamma';\ x :_i \tau$ is an instance of $S\Gamma$ so $S'S\Gamma;\ \Pi' = \Gamma';\ x :_i \tau$. Further, the statement of the theorem gives that Γ' is an instance of Γ. For both these to be the case requires $x :_i \tau \in \Pi'$. Therefore, Γ' is an instance of $S\Gamma$ as witnessed by $(S',\ \Pi' \setminus x :_i \tau)$.

The instance relation $\Gamma';\ x :_i \tau \rhd \sigma \geq \pi$ also follows from the inductive hypotheses. A single application of the **Pred** instance rule then yields $\Gamma' \rhd \sigma \geq (x : \tau).\pi$, the required judgement.

The Rel rule. If the last step of the derivation is an application of the **Rel** rule then it must be of the form

$$\frac{\Gamma' \rhd e : (x : \tau).\pi \qquad \Gamma' \rhd x : \tau}{\Gamma' \rhd e : \sigma'}$$

where $\sigma' = \pi$. That Γ' is an instance of $S\Gamma$ follows directly from the inductive hypotheses. Further, the inductive hypotheses give $\Gamma' \rhd \sigma \geq (x : \tau).\pi$. An application of the **Rel** instance rule yields $\Gamma' \rhd \sigma \geq \pi$, the required judgement if, and only if, $\Gamma' \rhd \sigma'' \geq \tau$ for some σ'' such that $x :_i \sigma'' \in \Gamma'$. Since $\Gamma' \rhd x : \tau$, it must be the case that there is a derivation of the form

$$\frac{\dfrac{x :_i \sigma'' \in \Gamma'}{\Gamma' \rhd x : \sigma''}}{\Gamma' \rhd x : \tau}$$

giving, by $\mathbf{Rule}_{3'}$, the judgement $\Gamma' \rhd \sigma'' \geq \tau$ as required.

Lambda- and `let`-bound identifiers. In this case the derivation is an instance of the **Taut** rule.

$$\frac{x : \sigma' \in \Gamma'}{\Gamma' \rhd x : \sigma'}$$

With $x : \sigma \in \Gamma$ the evaluation of $\mathcal{O}(\Gamma, x)$ proceeds to produce a derivation $S\Gamma;\ \Pi \rhd x : \tau$ such that S is the identity substitution and $\overline{S\Gamma}(\Pi.\tau)$ is σ—or a trivial variant

thereof. Therefore, $\mathcal{O}$ succeeds. Further, since Γ' is an instance of Γ and S is the identity substitution, Γ' is trivially an instance of $S\Gamma$.

It is only left to show $\Gamma' \rhd S'\sigma \geq \sigma'$; this also is trivial. Since $S'S\Gamma;\ \Pi' = \Gamma'$ it must be the case that $S'S\sigma = \sigma'$. Since S is the identity substitution, $S'\sigma = \sigma'$ implying $\Gamma' \rhd S'\sigma \geq \sigma'$ trivially by the **Taut** instance rule.

Overloaded operators. In this case the derivation must be an instance of the **Taut**$_i$ rule.

$$\frac{x :_i \sigma' \in \Gamma'}{\Gamma' \rhd x : \sigma'}$$

By a similar argument as was used in the previous case, it can be seen here that if $x :_o \forall\alpha_1.\cdots.\forall\alpha_n.\tau \in \Gamma$ then $S = Id$ and $\sigma = \forall\alpha_1.\cdots.\forall\alpha_n.(x : \tau).\tau$—or, again, a trivial variant thereof. Again $\mathcal{O}$ succeeds and Γ' is seen to be an instance of $S\Gamma$ since it is an instance of Γ.

It remains only to establish the judgement $\Gamma' \rhd S'\sigma \geq \sigma'$ given $x :_i \sigma' \in \Gamma'$. Since Γ is valid, $S'\sigma = \sigma$ since overloaded operator signatures contain no free type variables. Since $x :_i \sigma' \in \Gamma'$ and Γ' is valid, it must be the case that $\Gamma' \rhd \forall\alpha_1.\cdots.\forall\alpha_n.\tau \geq \sigma'$. That is, if σ' is of the form

$$\forall\beta_1\cdots\forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'$$

then there exists a substitution $S_{\sigma'}$ (of types for $\{\alpha_1, \ldots, \alpha_n\}$) such that $S_{\sigma'}\tau = \tau'$. Further, take $\Pi_{\sigma'}$ to be $\{x'_1 :_i \tau'_1;\ \cdots;\ x'_q :_i \tau'_q\}$. The following derivation holds.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Gamma';\ \Pi_{\sigma'} \rhd \sigma \geq \forall\alpha_1.\cdots.\forall\alpha_n.(x : \tau).\tau}{\Gamma';\ \Pi_{\sigma'} \rhd \sigma \geq (x : S_{\sigma'}\tau).S_{\sigma'}\tau}\ n \text{ applications of } \mathbf{Spec}}{\Gamma';\ \Pi_{\sigma'} \rhd \sigma \geq (x : \tau').\tau'}\ S_{\sigma'}\tau = \tau'}{\Gamma';\ \Pi_{\sigma'} \rhd \sigma \geq \tau'}\ \mathbf{Rel}\ \ \Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq \tau'}{\Gamma' \rhd \sigma \geq (x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'}\ q \text{ applications of } \mathbf{Pred}}{\Gamma' \rhd S'\sigma \geq \forall\beta_1\cdots\forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'}\ p \text{ applications of } \mathbf{Gen}$$

The substitution S' is introduced in the final line since $S'\sigma = \sigma$.

It remains to verify that **Rel** and **Gen** are indeed applicable above. Firstly, for the **Rel** case, it is required to show that $\Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq \tau'$. The following proof

establishes this.

$$\frac{\Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq \forall\beta_1 \cdots \forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'}{\Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq (x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'}\ \textbf{Spec}\ [\beta_i/\beta_i]\ \text{for}\ i = 1,\ \ldots,\ p$$

$$\frac{\Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq (x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'}{\Gamma';\ \Pi_{\sigma'} \rhd \sigma' \geq \tau'}\ \textbf{Rel},\ x'_j :_i \tau'_j \in \Pi_{\sigma'}\ \text{and}\ \Gamma';\Pi_{\sigma'} \rhd \tau'_j \geq \tau'_j$$

Secondly, the applications of **Gen** are valid trivially since one can ensure $\beta_j \notin fv(\Gamma')$ by renaming of bound type variables.

Lambda terms. In this case the **Abs** rule must have been applied

$$\frac{\Gamma';\ x : \tau' \rhd e : \tau''}{\Gamma' \rhd \lambda x.\ e : \tau' \to \tau''}$$

with $\sigma' = \tau' \to \tau''$. The application of $\mathcal{O}$ makes the recursive call $(S_1, \Pi_1, \tau_1) = \mathcal{O}(\Gamma;\ x : \beta, e)$ yielding the inductive hypotheses:

- that the recursive call succeeds;
- that $\Gamma';\ x : \tau'$ is an instance of $S_1(\Gamma;\ x : \beta)$ (witness (S'', Π''), say); and
- that $\Gamma';\ x : \tau' \rhd S''\overline{S_1(\Gamma;\ x : \beta)}(\Pi_1.\tau_1) \geq \tau''$.

$\mathcal{O}$ fails for lambda terms in general if the recursive call fails, or the variable concerned already appears in the type context. In the case at hand, the inductive hypotheses give that the recursive call succeeds. It is required only to show that x cannot be bound in Γ. Since $\Gamma';\ x : \tau'$ is valid, Γ' cannot contain x. Since Γ' is an instance of Γ it follows that Γ cannot contain x. Thus the call at hand succeeds.

Γ' is shown to be an instance of $S\Gamma$ by the following argument.

$$\frac{S''S_1(\Gamma;\ x : \beta);\ \Pi'' = \Gamma';\ x : \tau'}{S''S_1\Gamma;\ x : S''S_1\beta;\ \Pi'' = \Gamma';\ x : \tau'}$$

$$\frac{S''S_1\Gamma;\ x : S''S_1\beta;\ \Pi'' = \Gamma';\ x : \tau'}{S''S_1\Gamma;\ \Pi'' = \Gamma'}\ \text{remove}\ x : S''S_1\beta = \tau'\ \text{bindings}$$

$$\frac{S''S_1\Gamma;\ \Pi'' = \Gamma'}{S''S\Gamma;\ \Pi'' = \Gamma'}\ S = S_1$$

Thus the witness to Γ' being an instance of $S\Gamma$ is (S', Π') where Π' is Π'', and S' is the same as S'' on all variables except those free in $fv(S_1\beta) \setminus fv(S_1\Gamma)$ on which it is the identity. It remains only to show that $\Gamma' \rhd S'\sigma \geq \sigma'$, or more specifically,

$$\Gamma' \rhd S'\overline{S_1\Gamma}(\Pi_1.S_1\beta \to \tau_1) \geq \tau' \to \tau''.$$

Starting from the inductive hypothesis, the following derivation holds. The numbered steps are justified subsequently.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \rhd S''\overline{S_1(\Gamma;\ x : \beta)}(\Pi_1.\tau_1) \geq \tau''}{\Gamma' \rhd S''\overline{(S_1\Gamma;\ x : S_1\beta)}(\Pi_1.\tau_1) \geq \tau''}}{\Gamma' \rhd S''\overline{(S_1\Gamma;\ x : S_1\beta)}(\Pi_1.S_1\beta \to \tau_1) \geq \tau' \to \tau''}(1)}{\Gamma' \rhd S''\overline{S_1\Gamma}(\Pi_1.S_1\beta \to \tau_1) \geq \tau' \to \tau''}(2)}{\Gamma' \rhd S'\overline{S_1\Gamma}(\Pi_1.S_1\beta \to \tau_1) \geq \tau' \to \tau''}(3)}{\Gamma' \rhd S'\overline{S\Gamma}(\Pi.\tau) \geq \tau' \to \tau''}\text{def. of } S,\ \Pi, \text{ and } \tau$$

1. The free type variables of $S_1\beta$ all appear in the environment; as such they cannot be polymorphic. Thus S'' is applicable directly. As was noted above, $S''S_1\beta = \tau'$ so the change to the structure is symmetric.

2. The effect of removing $S_1\beta$ from the environment is to decrease the number type variables free in the environment. Therefore, more type variables become polymorphic. Specifically, those type variables in $fv(S_1\beta) \setminus fv(S_1\Gamma)$ become polymorphic. Call this set δ and S'' restricted to this set S_δ. Since δ are now polymorphic they are not effected by S''. Polymorphic variables may be assigned any type by the **Spec** instance rule; specifically, they may be assigned types in accordance with S_δ. Thus the instance judgement is still derivable.

3. The set δ contains those type variables which were previously free in the type context but are no longer. Thus, none of δ are free in the type scheme. As such, none of the mappings defined in S'' for δ can be applicable. This is the same restriction as was made in the definition of S' above. It therefore suffices to replace S'' with S'.

Application terms. In this case the **Comb** rule must have been applied

$$\dfrac{\Gamma' \rhd e_1 : \tau' \to \tau'' \quad \Gamma' \rhd e_2 : \tau'}{\Gamma' \rhd e_1\ e_2 : \tau''}$$

with $\sigma' = \tau''$. The two recursive calls of $\mathcal{O}$ are

$$\begin{aligned}(S_1, \Pi_1, \tau_1) &= \mathcal{O}(\Gamma, e_1)\\ (S_2, \Pi_2, \tau_2) &= \mathcal{O}(S_1\Gamma, e_2)\end{aligned}$$

which yield the following inductive hypothesis: that both recursive calls succeed;

- that Γ' is an instance of $S_1\Gamma$ (witness (S_1', Π_1'), say);
- that $\Gamma' \rhd S_1'\overline{S_1\Gamma}(\Pi_1.\tau_1) \geq \tau' \rightarrow \tau''$;
- that Γ' is an instance of $S_2S_1\Gamma$ (witness (S_2', Π_2'), say); and
- that $\Gamma' \rhd S_2'\overline{S_2S_1\Gamma}(\Pi_2.\tau_1) \geq \tau'$.

The inductive hypotheses hold for the second recursive call since Γ' is an an instance of $S_1\Gamma$ by the inductive hypotheses of the first recursive call.

From $S_1'S_1\Gamma;\ \Pi_1' = \Gamma' = S_2'S_2S_1\Gamma;\ \Pi_2'$ it follows that $\Pi_1' = \Pi_2'$ and $S_1' = S_2'S_2$ for $fv(S_1\Gamma)$.

To show that the call of $\mathcal{O}$ at hand succeeds it is necessary to show that $S_u = U(S_2\tau_1, \tau_2 \rightarrow \beta)$ succeeds. Let δ_1 be the generic variables of $\overline{S_1\Gamma}(\Pi_1.\tau_1)$ and δ_2 be the generic variables of $\overline{S_2S_1\Gamma}(\Pi_2.\tau_1)$. Now δ_1 and δ_2 are unmoved by S_1' and S_2' since the latter are minimal. Since $\Gamma' \rhd S_1'\overline{S_1\Gamma}(\Pi_1.\tau_1) \geq \tau' \rightarrow \tau''$ there must exist a substitution S_{δ_1} of types for δ_1 such that $(S_1' + S_{\delta_1})\tau_1 = \tau' \rightarrow \tau''$.

Similarly, from $\Gamma' \rhd S_2'\overline{S_2S_1\Gamma}(\Pi_2.\tau_1) \geq \tau'$ there must exist a substitution S_{δ_2} of types for δ_2 such that $(S_2' + S_{\delta_2})\tau_2 = \tau'$. The substitution

$$S_0 = S_2' + S_{\delta_1} + S_{\delta_2} + [\tau''/\beta]$$

is shown to be a unifying substitution. To do this, it is required to show that

$$S_0(S_2\tau_1) = \tau' \rightarrow \tau'' = S_0(\tau_2 \rightarrow \beta)$$

The following two derivations show this.

$$\begin{array}{lll}
 & S_0(\tau_2 \rightarrow \beta) & \\
= & (S_0\tau_2) \rightarrow (S_0\beta) & \\
= & (S_0\tau_2) \rightarrow \tau'' & S_0\beta = \tau'' \\
= & ((S_2' + S_{\delta_1} + S_{\delta_2} + [\tau''/\beta])\tau_2) \rightarrow \tau'' & \\
= & ((S_2' + S_{\delta_2})\tau_2) \rightarrow \tau'' & (\delta_1 \cup \{\beta\}) \cap fv(\tau_2) = \{\} \\
= & \tau' \rightarrow \tau'' & \text{def. of } S_{\delta_2}
\end{array}$$

$$\begin{array}{lll}
 & S_0(S_2\tau_1) & \\
= & (S_2' + S_{\delta_1} + S_{\delta_2} + [\tau''/\beta])S_2\tau_1 & \\
= & (S_2' + S_{\delta_1})S_2\tau_1 & (\delta_2 \cup \{\beta\}) \cap fv(S_2\tau_1) = \{\} \\
= & (S_2'S_2 + S_{\delta_1})\tau_1 & dom(S_2) \cap \delta_1 = \{\} \\
= & (S_1' + S_{\delta_1})\tau_1 & S_2'S_2 = S_1' \\
= & \tau' \rightarrow \tau'' & \text{def. of } S_{\delta_1}
\end{array}$$

Thus, S_0 is a unifying substitution. If any unifying substitution exists, $S_u = U(S_2\tau_1, \tau_2 \rightarrow \beta)$ succeeds, and the call of $\mathcal{O}$ at hand succeeds in turn.

Γ' is shown to be an instance of $S_u S_2 S_1 \Gamma$ by the following argument. Since both S_0 and S_u are unifying substitution and S_u is the most general unifying substitution, it must be the case for some S' that $S' S_u = S_0$. Taking Π' to be Π'_2 gives

$$\begin{array}{lcll} S' S_u S_2 S_1 \Gamma;\ \Pi' & = & S_0 S_2 S_1 \Gamma;\ \Pi' & \\ & = & S'_2 S_2 S_1 \Gamma;\ \Pi' & fv(S_2 S_1 \Gamma) \cap (\delta_1 \cup \delta_2 \cup \{\beta\}) = \{\} \\ & = & \Gamma' & \text{inductive hypothesis} \end{array}$$

That is, the pair (S', Π') bears witness to Γ' being an instance of $S_u S_2 S_1 \Gamma$.

It remains only to show that $\Gamma' \rhd S' \overline{S_u S_2 S_1 \Gamma}(S_u(S_2 \Pi_1;\ \Pi_2).S_u \beta) \geq \tau''$. This follows from the initial two judgements of the following derivation. The two premises are proved subsequently.

$$\dfrac{\dfrac{\dfrac{\Gamma' \rhd S' S_u S_2 \Pi_1 . S' S_u \beta \geq \tau'' \quad \Gamma' \rhd S' S_u \Pi_2 . \tau' \geq \tau'}{\Gamma' \rhd S' S_u S_2 \Pi_1;\ S' S_u \Pi_2 .\ S' S_u \beta \geq \tau''}\ \textbf{Rule}_9}{\Gamma' \rhd S'(S_u S_2 \Pi_1;\ S_u \Pi_2 .\ S_u \beta) \geq \tau''}}{\Gamma' \rhd S' \overline{S_u S_2 S_1 \Gamma}(S_u S_2 \Pi_1;\ S_u \Pi_2 .\ S_u \beta) \geq \tau''}$$

The section elimination (**Rule**$_9$) step is applicable since those free variables of $S' S_u \Pi_2$ which are not in Γ or τ' must be generic. Such variables, if they are to be in $S' S_u S_2 \Pi_1 . S' S_u \beta$, then they must also be in τ' due to unification. As stated above, however, they are not in τ'.

Only the two premises above remain undischarged, the first is proved by the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \rhd S'_1 \overline{S_1 \Gamma}(\Pi_1 . \tau_1) \geq \tau' \to \tau''}{\Gamma' \rhd (S'_1 + S_{\delta_1})(\Pi_1 . \tau_1) \geq \tau' \to \tau''}\ \textbf{Rule}_{11}}{\Gamma' \rhd (S'_2 S_2 + S_{\delta_1})(\Pi_1 . \tau_1) \geq \tau' \to \tau''}\ S'_1 = S'_2 S_2 \text{ on } fv(S_1 \Gamma)}{\Gamma' \rhd (S'_2 + S_{\delta_1})(S_2 \Pi_1 . S_2 \tau_1) \geq \tau' \to \tau''}\ \delta_1 \notin S_2}{\Gamma' \rhd (S'_2 + S_{\delta_1} + S_{\delta_2} + [\tau''/\beta])(S_2 \Pi_1 . S_2 \tau_1) \geq \tau' \to \tau''}}{\Gamma' \rhd S_0(S_2 \Pi_1 . S_2 \tau_1) \geq \tau' \to \tau''}\ S_0 = S' S_u}{\Gamma' \rhd S' S_u S_2 \Pi_1 . S' S_u S_2 \tau_1 \geq \tau' \to \tau''}\ S_u S_2 \tau_1 = S_u(\tau_2 \to \beta)}{\Gamma' \rhd S' S_u S_2 \Pi_1 . S' S_u(\tau_2 \to \beta) \geq \tau' \to \tau''}\ \textbf{Rule}_9}{\Gamma' \rhd S' S_u S_2 \Pi_1 . S' S_u \beta \geq \tau''}$$

And finally, $\Gamma' \rhd S'S_u\Pi_2.\tau' \geq \tau'$ is proved as follows.

$$\dfrac{\dfrac{\dfrac{\Gamma' \rhd S'_2\overline{S_2S_1\Gamma}(\Pi_2.\tau_2) \geq \tau'}{\Gamma' \rhd (S_0)\Pi_2.\tau_2 \geq \tau'}\text{as above}}{\Gamma' \rhd S'S_u\Pi_2.S'S_u\tau_2 \geq \tau'}}{\Gamma' \rhd S'S_u\Pi_2.\tau' \geq \tau'}\,S'S_u\tau_2 = \tau'$$

Let terms. In this case the **Let** rule must have been applied

$$\dfrac{\Gamma' \rhd e_1 : \sigma'' \quad \Gamma';\ x : \sigma'' \rhd e_2 : \tau'}{\Gamma' \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'}$$

with $\sigma' = \tau'$. The two recursive calls of $\mathcal{O}$ are

$$(S_1, \Pi_1, \tau_1) = \mathcal{O}(\Gamma, e_1); \text{ and}$$
$$(S_2, \Pi_2, \tau_2) = \mathcal{O}(S_1\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1), e_2)$$

yielding the following inductive hypotheses:

- that the first recursive call succeeds;
- that Γ' is an instance of $S_1\Gamma$ (witness (S'_1, Π'_1), say); and
- that $\Gamma' \rhd S'_1\overline{S_1\Gamma}(\Pi_1.\tau_1) \geq \sigma''$.

From $\Gamma';\ x : \sigma' \rhd e_2 : \tau'$ and $\Gamma' \rhd S'_1\overline{S_1\Gamma}(\Pi_1.\tau_1) \geq \sigma''$ it follows by $\mathbf{Rule}_{12}$ that

$$\Gamma';\ x : S'_1\overline{S_1\Gamma}(\Pi_1.\tau_1) \rhd e_2 : \tau'.$$

Since

$$\Gamma';\ x : S'_1\overline{S_1\Gamma}(\Pi_1.\tau_1) = S'_1(S_1\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1));\ \Pi'_1$$

the conditions for the applicability of the theorem in the second recursive call are satisfied and the following inductive hypotheses hold:

- the second recursive call succeeds;
- that $\Gamma';\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1)$ is an instance of $S_2(S_1\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1))$, witness (S'_2, Π'_2), say; and

- that $\Gamma';\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1) \rhd S_2'\overline{S_2S_1(\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1))}(\Pi_2.\tau_2) \geq \tau'$.

Typings for `let` terms fail if, and only if, either recursive call fails or the identifier concerned is already bound in the type context. Neither recursive call fails, and since $\Gamma';\ x : \sigma''$ is valid, x is not bound in Γ'. Further, since Γ' is an instance of Γ, x cannot be bound in Γ. Therefore, the call of $\mathcal{O}$ at hand succeeds.

Next, it is required to show that Γ' is an instance of $S_2S_1\Gamma$. This follows from the inductive hypothesis.

$$\begin{array}{ll} S_2'S_2(S_1\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1)); \Pi_2' & = \Gamma';\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1) \\ S_2'S_2(S_1\Gamma); \Pi_2' & = \Gamma' \end{array}$$

Therefore, Γ' is an (S', Π') instance of $S_2S_1\Gamma$ where S' is S_2' and Π' is Π_2'.

Finally, it is required to show that $\Gamma' \rhd S'\overline{S_2S_1\Gamma}(\Pi_2.\tau_2) \geq \tau'$. This follows from the inductive hypotheses as shown below.

$$\dfrac{\dfrac{\dfrac{\Gamma';\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1) \rhd S_2'\overline{S_2S_1(\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1))}(\Pi_2.\tau_2) \geq \tau'}{\Gamma' \rhd S_2'\overline{S_2S_1(\Gamma;\ x : \overline{S_1\Gamma}(\Pi_1.\tau_1))}(\Pi_2.\tau_2) \geq \tau'}}{\Gamma' \rhd S_2'\overline{S_2S_1\Gamma}(\Pi_2.\tau_2) \geq \tau'}}{\Gamma' \rhd S'\overline{S_2S_1\Gamma}(\Pi_2.\tau_2) \geq \tau'}\ S' = S_2'$$

□

5.3 $\mathcal{O}$ computes principal types

The specification and proof of the completeness result above is messy and largely opaque. It is convenient to conclude the discussion with two simple corollaries.

A type scheme σ is the *principal type scheme* of e under Γ if, and only if,

- $\Gamma \rhd e : \sigma$; and
- if, for some σ', $\Gamma \rhd e : \sigma'$, then $\Gamma \rhd \sigma \geq \sigma'$.

The following two corollaries are consequences of the syntactic soundness and completeness results.

Corollary 3 $\mathcal{O}$ computes principal types. *If $\mathcal{O}(\Gamma, e)$ succeeds with (S, Π, τ) then $\overline{S\Gamma}(\Pi.\tau)$ is a principal type scheme for e under $S\Gamma$.*

One need no longer be concerned with the completeness result when considering principal types. If a typing $\Gamma \rhd e : \sigma'$ exists then take (S, Π, τ) to be $\mathcal{O}(\Gamma, e)$ and it follows that $\overline{\Gamma}(\Pi.\tau)$ is the principal type of e under Γ. Notice that S must be *Id*.

Corollary 4 Principal types. *If it is possible to derive a type scheme for e under* Γ *then there is a principal type scheme for e under* Γ.

Notice that this corollary makes no mention of $\mathcal{O}$. Specifically, the existence of principal types is a property, not of the type inference algorithm, but of the type system alone.

5.4 The translation is syntactically sound

The translation assigning meaning to type derivations is syntactically sound: that is, given a derivation in the OL calculus, the translated derivation in the Damas-Milner system is a valid derivation. The formal statement of the theorem is as follows.

Theorem 5 Syntactic soundness of translation. *Given a derivation* $\Gamma \rhd e : \sigma$, *then the translation of that derivation,* $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\sigma\rfloor$, *is a valid derivation in the Damas-Milner calculus.*

It follows trivially from the syntactic soundness and the semantic soundness of the Damas-Milner system, that this translation is semantically sound.

Proof

The proof is by structural induction on the derivation of $\Gamma \rhd e : \sigma$. It establishes that the translation of each rule in the OL calculus is valid in the Damas-Milner calculus. Once again, the base cases are the **Taut** and **Taut**$_i$ rules.

Firstly however, it is required to show that if Γ is a valid type context, then $\lfloor\Gamma\rfloor$ is a sensible Damas-Milner assumption set. A simple examination of the rules for $\lfloor\Gamma\rfloor$ establishes this. It is clear that all type schemes $\lfloor\sigma\rfloor$ are valid Damas-Milner type schemes. There are bindings in $\lfloor\Gamma\rfloor$ for all the :-bound identifiers in Γ, there are also bindings for all the :$_i$-bound operators. All these identifiers are distinct—by the definition of valid type contexts and $\lfloor . \rfloor$—and as such, the assumption set is sensible.

The Taut rule. From $x : \sigma \in \Gamma$ if follows $x : \lfloor\sigma\rfloor \in \lfloor\Gamma\rfloor$. Therefore, the typing is trivially true in the Damas-Milner system.

The Taut$_i$ rule. From $x :_i \sigma \in \Gamma$ it follows $\overline{x} : \lfloor\sigma\rfloor \in \lfloor\Gamma\rfloor$ where $\overline{x} = \lfloor x :_i \sigma\rfloor$. Again, the typing is trivially true by the Damas-Milner translation of the **Taut$_i$** rule.

The Gen rule. The inductive hypothesis is $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\sigma\rfloor$. It is easy to see that $fv(\lfloor\Gamma\rfloor) \subseteq fv(\Gamma)$, therefore, from $\alpha \notin fv(\Gamma)$ it follows $\alpha \notin fv(\lfloor\Gamma\rfloor)$. As such **Gen** is applicable in the Damas-Milner system. Therefore $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\forall\alpha.\sigma\rfloor$ is shown, as required.

The Spec rule. The inductive hypothesis gives $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\forall\alpha.\sigma\rfloor$. From the definition of $\lfloor\forall\alpha.\sigma\rfloor$, it follows that α is also the outermost bound variable of $\lfloor\forall\alpha.\sigma\rfloor$. Further $\lfloor\tau\rfloor$ is τ. As such **Spec** is indeed applicable in the Damas-Milner system giving $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor([\tau/\alpha]\sigma)\rfloor$.

The Abs rule. The inductive hypothesis is $\lfloor(\Gamma;\ x : \tau)\rfloor \vdash \overline{e} : \lfloor\tau'\rfloor$. That is, for $A = \lfloor\Gamma\rfloor$, the judgement $A;\ x : \tau \vdash \overline{e} : \tau'$ is valid. Clearly the Damas-Milner rule **Abs** is applicable giving $\lfloor\Gamma\rfloor \vdash \lambda x.\ \overline{e} : \lfloor\tau \to \tau'\rfloor$ as required.

The Comb rule. In this case there are two inductive hypotheses: $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\tau' \to \tau\rfloor$ and $\lfloor\Gamma\rfloor \vdash \overline{e}' : \lfloor\tau'\rfloor$. Now $\lfloor(\tau' \to \tau)\rfloor$ and $\lfloor\tau'\rfloor$ are $\tau' \to \tau$ and τ' respectively so **Comb** is applicable in the Damas-Milner system giving $\lfloor\Gamma\rfloor \vdash (\overline{e}\ \overline{e}') : \lfloor\tau\rfloor$ as required.

The Let rule. Again there are two inductive hypotheses:

$$\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\sigma\rfloor \quad \text{and} \quad \lfloor(\Gamma;\ x : \sigma)\rfloor \vdash \overline{e}' : \lfloor\tau\rfloor.$$

It is obvious that the Damas-Milner **Let** rule is applicable giving

$$\lfloor\Gamma\rfloor \vdash \texttt{let}\ x = \overline{e}\ \texttt{in}\ \overline{e}' : \lfloor\tau\rfloor$$

as required.

The Pred rule. In this case the inductive hypothesis is $\lfloor(\Gamma;\ x :_i \tau)\rfloor \vdash \overline{e} : \lfloor\pi\rfloor$. Taking A to be $\lfloor\Gamma\rfloor$ that is $A;\ \overline{x} : \tau \vdash \overline{e} : \lfloor\pi\rfloor$ (where $\overline{x}$ is $name^*(x :_i \tau)$). As such, the Damas-Milner **Abs** rule is applicable giving $A \vdash \lambda\overline{x}.\ \overline{e} : \tau \to \lfloor\pi\rfloor$. By an obvious property of the mapping, $\tau \to \lfloor\pi\rfloor = \lfloor((x : \tau).\pi)\rfloor$ which gives the required result.

The Rel rule. The final case yields the inductive hypotheses $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor((x : \tau).\pi)\rfloor$ and $\lfloor\Gamma\rfloor \vdash \overline{e}' : \lfloor\tau\rfloor$. The former of these is $\lfloor\Gamma\rfloor \vdash \overline{e} : \lfloor\tau \to \pi\rfloor$ which makes the Damas-Milner **Comb** rule applicable yielding $\lfloor\Gamma\rfloor \vdash (\overline{e}\ \overline{e}') : \lfloor\pi\rfloor$ as required. □

Chapter 6

Discharging predicates

As is noted in Chapter 3, the algorithm presented informally in Chapter 2 is not modelled exactly by algorithm $\mathcal{O}$. This Chapter addresses the differences and presents an appropriate extension to $\mathcal{O}$.

6.1 The difference between the formal and informal algorithms

The informal algorithm and algorithm $\mathcal{O}$ differ only in their handling of predicates, that part inherited from the Damas-Milner algorithm is unchanged. When the informal algorithm establishes that a predicate is satisfied by a instance binding in the type context, that predicate is discharged; when it discovers that two predicates are the same, one or other is discharged—algorithm $\mathcal{O}$, on the other hand, makes no attempt to discharge predicates.

The motivation for this dichotomy is technical: $\mathcal{O}$ computes principal types whereas the informal algorithm does not. To see this, it is best to consider an example. If [] is used as shorthand for the empty list, and equality is overloaded on lists as it is in Chapter 2, that is,

$$(==) :_i \forall\alpha.((==) : \alpha \to \alpha \to \mathit{Bool}).\mathit{List}(\alpha) \to \mathit{List}(\alpha) \to \mathit{Bool} \in \Gamma$$

then the term $(\lambda x.\ (x == []))$ is assigned different types by the two algorithms—the first below by $\mathcal{O}$, and the second by the informal algorithm.

$$\Gamma \rhd (\lambda x.\ (x == [])) : \forall\alpha.((==) : \mathit{List}(\alpha) \to \mathit{List}(\alpha) \to \mathit{Bool}).\alpha \to \mathit{Bool}$$
$$\Gamma \rhd (\lambda x.\ (x == [])) : \forall\alpha.((==) : \alpha \to \alpha \to \mathit{Bool}).\alpha \to \mathit{Bool}$$

That is, the typing of the term generated a predicate on equality over $\mathit{List}(\alpha)$ as seen on the $\mathcal{O}$ version. The informal algorithm discharges this predicate with the instance of equality over lists in the type context and introduces the predicate on

$\mathcal{R}(\Gamma, \Pi)$ = If there are bindings

$$x :_i \tau' \in \Pi$$
$$x :_i \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \in \Gamma$$

such that there is a substitution S of types for $\{\alpha_1, \ldots, \alpha_n\}$ with

$$S\tau = \tau',$$

then return

$$\mathcal{R}(\Gamma, x_1 :_i S\tau_1;\ \cdots;\ x_m :_i S\tau_m; \Pi \setminus x :_i \tau')$$

otherwise return Π.

Figure 6.1: Algorithm $\mathcal{R}$: discharging predicates (II)

equality at type α as required by the instance. It is asserted above that the typing computed by $\mathcal{O}$ is the principal typing. To see this, observe that, taking Eq_τ to abbreviate $\tau \to \tau \to Bool$,

$$\Gamma \rhd \forall\alpha.((==) : Eq_{List(\alpha)}).\alpha \to Bool \geq \forall\alpha.((==) : Eq_\alpha).\alpha \to Bool$$

is a valid judgement, as one would expect since $\mathcal{O}$ computes principal types. The reverse, however, is not the case: the following is not a valid instance judgement.

$$\Gamma \rhd \forall\alpha.((==) : Eq_\alpha).\alpha \to Bool \geq \forall\alpha.((==) : Eq_{List(\alpha)}).\alpha \to Bool$$

Thus, the typing computed by the informal algorithm is not principal. This chapter presents an algorithm $\mathcal{R}$ unifying the two approaches. After $\mathcal{R}$ is discussed below, Section 6.7 returns to this question illustrating the sense in which it is sensible to use $\mathcal{R}$ despite the observation above.

6.2 Algorithm $\mathcal{R}$: discharging predicates

This section defines an algorithm $\mathcal{R}$ which implements predicate discharge, formalising the method presented in Chapter 2. The section begins by defining $\mathcal{R}$ and outlining its operation. There then follows a brief discussion on how to integrate $\mathcal{R}$ with $\mathcal{O}$ to achieve the same effect as that of the informal algorithm. The subsequent sections describe when it is safe to apply $\mathcal{R}$, and show that $\mathcal{R}$ is sound.

6.2.1 Algorithm $\mathcal{R}$

If under a given type context Γ a section Π is generated by $\mathcal{O}$, then $\mathcal{R}$ discharges those predicates in Π that are satisfied by instance bindings in Γ. As with the informal algorithm, if an instance binding is itself predicated, then $\mathcal{R}$ introduces the required predicates into the section. This procedure is then iterated until no further predicates can be discharged. Specifically, calls to $\mathcal{R}$ are of the form

$$\hat{\Pi} = \mathcal{R}(\Gamma, \Pi)$$

where $\hat{\Pi}$ is the predicate set after all predicates in Π which are satisfied by Γ are discharged. Algorithm $\mathcal{R}$ is given in Figure 6.1.

As an example of the application of $\mathcal{R}$, if Γ is

$$\begin{array}{lll} \Gamma = \{ & (+) & :_o \forall\alpha.\alpha \to \alpha \to \alpha; \\ & (==) & :_o \forall\alpha.\alpha \to \alpha \to \mathit{Bool}; \\ & (+) & :_i \mathit{Int} \to \mathit{Int} \to \mathit{Int}; \\ & (==) & :_i \forall\alpha.((==) : \alpha \to \alpha \to \mathit{Bool}).\mathit{List}(\alpha) \to \mathit{List}(\alpha) \to \mathit{Bool}; \\ \} & & \end{array}$$

and Π is the set of predicates

$$\begin{array}{lll} \Pi = \{ & (+) & :_i \ \mathit{Int} \to \mathit{Int} \to \mathit{Int}; \\ & (==) & :_i \ \mathit{List}(\gamma) \to \mathit{List}(\gamma) \to \mathit{Bool}; \ \} \end{array}$$

generated during an application of $\mathcal{O}$, then $\hat{\Pi} = \mathcal{R}(\Gamma, \Pi)$ is as follows.

$$\hat{\Pi} = \{(==) :_i \gamma \to \gamma \to \mathit{Bool}; \ \}$$

The instance of $(+)$ over integers is used to discharge the first predicate, and the instance of $(==)$ over $\mathit{List}(\alpha)$ is used to discharge the second predicate. A predicate requiring $(==)$ at type α is introduced to allow the latter discharge above.

6.2.2 The use of $\mathcal{R}$ with $\mathcal{O}$

As noted above, $\mathcal{R}$ unifies the two approaches to type inference taken by $\mathcal{O}$ and the informal algorithm. What remains to be answered, however, is how $\mathcal{O}$ and $\mathcal{R}$ may be combined to achieve the action of the informal algorithm. This section looks briefly at the choices available.

Two approaches are outlined below: these are motivated by practical and technical considerations respectively. Under the first approach, $\mathcal{R}$ is called immediately prior to any generalisation of a typing—specifically, at `let` terms.

If a typing for the defining term of a `let` declaration is of the form[1]

$$S_1\Gamma;\ \Pi_1 \rhd e_1 : \tau_1$$

then $\mathcal{O}$ binds x to $\overline{S_1\Gamma}(\Pi_1.\tau_1)$. To incorporate $\mathcal{R}$ with $\mathcal{O}$, the variable x is bound to

$$\overline{S_1\Gamma}(\mathcal{R}(S_1\Gamma, \Pi_1).\tau_1)$$

that is, $\mathcal{R}$ is called to discharge predicates in Π_1 immediately prior to the generalisation of the typing in a `let` term. Under this approach, $\mathcal{R}$ should further be applied after the entire activation of $\mathcal{O}$.

The second approach, which is simpler to discuss, is simply to compose $\mathcal{O}$ and $\mathcal{R}$. That is, apply $\mathcal{O}$ as before yielding some typing $S\Gamma;\ \Pi \rhd e : \tau$; then apply $\mathcal{R}$ to achieve the final typing.

$$S\Gamma;\ \mathcal{R}(S\Gamma, \Pi) \rhd e : \tau$$

This approach allows $\mathcal{O}$ and $\mathcal{R}$ to be discussed independently and will be assumed throughout the rest of this document. The two approaches are essentially equivalent: the proof observes that, for each clause of $\mathcal{O}$, the two approaches yield the same result.

A final point should be made about duplicate predicates: the informal algorithm ensures duplicate predicates do not arise; $\mathcal{O}$ and $\mathcal{R}$ make no effort to eliminate duplicate predicates. Though one would ensure in practice that no duplicate predicates arose, for current purposes it is not necessary to discharge them. For example, taking Pl_τ to abbreviate $\tau \to \tau \to \tau$, consider the following two instance judgements.

$$\Gamma \rhd \forall\alpha.((+) : Pl_\alpha).\alpha \to \alpha \geq \forall\alpha.((+) : Pl_\alpha).((+) : Pl_\alpha).\alpha \to \alpha$$
$$\Gamma \rhd \forall\alpha.((+) : Pl_\alpha).((+) : Pl_\alpha).\alpha \to \alpha \geq \forall\alpha.((+) : Pl_\alpha).\alpha \to \alpha$$

The number of occurrences of a predicate in a typing is no more than a trivial variance in the same sense that the renaming of bound type variables is a trivial variance. $\mathcal{R}$ is easily extended to eliminate duplicate predicates if required.

6.3 When is it sensible to use $\mathcal{R}$?

This section examines two cases in which it is *not* sensible to apply $\mathcal{R}$: when it may not have a unique result, and when it may fail to terminate. $\mathcal{R}$ is safe to use under

[1]The variable names used here are consistent with those used in the `let` clause of the definition of $\mathcal{O}$(see Figure 3.10).

the conditions outlined in the following two sections. Both these restrictions are adhered to in the Haskell implementation of type classes described in Chapter 9; and both are assumed throughout the rest of this chapter.

At a more technical level, the two restrictions outlined below ensure decidability. The omission of these restrictions in [WaBl89] led to a degree of controvercy over the decidability of the system [Lil91]. It is important to note that the system satisfies a principal type theorem without the restrictions, but, for any practical purpose, predicates must be discharged. At this stage, these restrictions ensure that $\mathcal{R}$ terminates and has a unique result (up to trivial variance).

6.3.1 Ensuring $\mathcal{R}$ has a unique result

On examination of the definition of $\mathcal{R}$, it is clear that if the type context Γ contains two instance bindings which may discharge the same predicate, then the result of a call to $\mathcal{R}$ is not uniquely defined[2]. For example, if Γ contains

$$(==) :_i \forall\alpha.(==: \alpha \to \alpha \to \mathit{Bool}).\mathit{List}(\alpha) \to \mathit{List}(\alpha) \to \mathit{Bool} \quad \text{and}$$
$$(==) :_i \forall\alpha.\mathit{List}(\alpha) \to \mathit{List}(\alpha) \to \mathit{Bool},$$

then the result of

$$\mathcal{R}(\Gamma,\ [(==) :_i \mathit{List}(\gamma) \to \mathit{List}(\gamma) \to \mathit{Bool}])$$

could be either

$$[(==) :_i \gamma \to \gamma \to \mathit{Bool}],$$

if the first instance is chosen, or empty. For $\mathcal{R}$ to be appropriate, therefore, it is essential to ensure that no two instance bindings in the type context overlap.

Definition 6.1 Non-overlapping type contexts. *A type context Π is described as* non-overlapping *if for no pair of instance bindings $x :_i \sigma_1$ and $x :_i \sigma_2$ in Π does there exist a type τ and predicate set Π, such that both*

$$\Gamma;\ \Pi \rhd \sigma_1 \geq \tau \quad \textit{and} \quad \Gamma;\ \Pi \rhd \sigma_2 \geq \tau.$$

That is, two instance bindings in a non-overlapping type context cannot have the same type as an instance.

To ensure $\mathcal{R}$ behaves in a sensible manner it should only be applied to non-overlapping type contexts. This restriction does not imply that all type contexts appearing within a derivation must be non-overlapping, merely that the top-level type context must be non-overlapping.

[2]The meaning of *uniquely defined* here is intended to be up to trivial variance: that is, renaming of bound type variables, reorganisation of the order in which predicates appear, and possible duplicate predicates.

6.3.2 Ensuring $\mathcal{R}$ terminates

Since R is recursive, it is necessary to ensure that all calls of $\mathcal{R}$ terminate. A further restriction on type contexts is required, that they be strictly decreasing.

Throughout the examples in Chapter 2, all instance bindings satisfy a particular property—if they are used to discharge a predicate, then any predicates they introduce are smaller, in some sense, than those which they discharge. The instance binding for equality on lists is given in Section 2.2 as follows.

$$\texttt{inst}\ (==) : \forall\alpha.(Eq_\alpha).List(\alpha) \rightarrow List(\alpha) \rightarrow Bool = eqList;$$

The types involved in the predicate are "smaller" than the types in the instance itself. If this instance is used to discharge a predicate on $List(\alpha)$, then the predicate introduced is on α.

This property, in the case of the informal algorithm, is essential to guarantee termination. In Haskell, the equivalent condition is imposed syntactically. Below, the definition of *strictly decreasing* type contexts captures this property.

Firstly, it is necessary to abstract some information from instance bindings. Given a binding $x :_o \forall\beta_1.\cdots\forall\beta_p.\tau'$ every instance binding of the form

$$x :_i \forall\alpha_1\cdots\forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

defines two sets: its *primary* and *secondary* sets. Since the type context involved is valid, there must exists a substitution of types for type variables such that

$$\tau = [\tau_1'/\beta_1;\ \cdots;\ \tau_p'/\beta_p]\tau'$$

for some types τ_1' to τ_p'. The set $\{\tau_1',\ \ldots,\ \tau_p'\}$ is the instance binding's *primary* set.

In the same way, each predicate $x_i : \tau_i$ for i in 1 to m must be a substitution instance of its signatures in the type context. The *secondary* set associated with this instance is the union of these sets of substituted types.

In the equality-over-lists instance above, the primary set is $\{List(\alpha)\}$, and the secondary set is $\{\alpha\}$. The primary set contains the list type substituted for the single type variable; and the secondary set contains the type substituted to validate the predicate. In more general examples the sets involved need not be so small.

Definition 6.2 Strictly decreasing instance bindings. *An instance binding is strictly decreasing if every type in its secondary set is a proper subterm of a type in its primary set.*

A type τ' is a proper subterm of a type τ if, and only if: τ is of the form $\chi(\tau_1, \ldots, \tau_n)$ and τ' is either equal to, or a proper subterm of τ_i for some $i \in \{1, \ldots, m\}$; or τ is of the form $\tau_1 \rightarrow \tau_2$ and τ' is either equal to, or a proper subterm of, one of τ_1 or τ_2. Finally, the required definition can be given for type contexts.

Definition 6.3 Strictly decreasing type contexts. *A type context is strictly decreasing if all the instance bindings it contains are strictly decreasing.*

6.4 $\mathcal{R}$ terminates

All the restrictions have now been presented to allow a result to be given with respect to the termination of $\mathcal{R}$.

Theorem 6 Termination of $\mathcal{R}$. *Given a call $\mathcal{R}(\Gamma, \Pi)$ with all sets and types involved finite and Γ strictly decreasing, then the call terminates.*

Proof

The proof is informal, the recursion is shown to be well-founded by giving a partial ordering between the recursive calls. This partial ordering has no infinitely descending chains.

Reviewing the definition of strictly decreasing type contexts, each instance binding defines a primary and a secondary set. Every element of the secondary set is a subterm of an element of the primary set.

Now take $\mathcal{T}$ to be the union of the primary sets associated with the instance bindings in Π. Then take $\mathcal{T}_i$ to be the union of the primary sets in Π on the ith recursive call of $\mathcal{R}$. Then $\mathcal{T}_{i+1}$ is computed from $\mathcal{T}_i$ by replacing some instance of the primary set of some instance binding in Γ with an instance of the corresponding secondary set. The types in the secondary set are required to be proper subterms of the types in the primary set. As such, each iteration of $\mathcal{R}$ results in smaller types in the $\mathcal{T}$ sets. Since all the sets and types involved are of finite size, this process must terminate. □

6.5 $\mathcal{R}$ is deterministic

The dynamic behaviour of $\mathcal{R}$ is non-deterministic. It is necessary to ensure that the choice of predicate to discharge at any stage is not significant, and the result of an application is deterministic up to trivial variance. An informal proof is given below.

Take Π to be $\{x_1 :_i \tau_1;\ \cdots;\ x_n :_i \tau_n\}$ and $\hat{\Pi}_i$ to be $\mathcal{R}(\Gamma, [x_i :_i \tau_i])$ for each i in $\{1, \ldots, n\}$. Then

$$\mathcal{R}(\Gamma, \Pi) = \hat{\Pi}_1;\ \cdots;\ \hat{\Pi}_n$$

This equivalence follows from the observation that there are no dependencies between predicates in the action of $\mathcal{R}$. Each clause $\hat{\Pi}_i$ is deterministic in its first step. By an inductive argument, $\mathcal{R}(\Gamma, \Pi)$ is deterministic as required.

6.6 $\mathcal{R}$ is sound

This section provides a proof that $\mathcal{R}$ is sound: that is, a typing specialised by $\mathcal{R}$ is indeed a derivable typing. This can be stated formally by the following theorem.

Theorem 7 Soundness of $\mathcal{R}$. *If* $\Gamma;\ \Pi \rhd e : \tau$, *and* $\hat{\Pi} = \mathcal{R}(\Gamma, \Pi)$, *then*

$$\Gamma;\ \hat{\Pi} \rhd e : \tau.$$

That is, predicates are discharged by $\mathcal{R}$ in accordance with the type rules.

Proof

The proof is by induction on the number of recursive calls. Since $\mathcal{R}$ terminates, this must be finite.

Base case. If the number of calls is zero, then the result follows trivially since $\Pi = \hat{\Pi}$.

Inductive case. In the inductive case, it suffices to show that an arbitrary application of the body is sound. That is, if the there exists a predicate and instance binding as required, then

$$\frac{\Gamma;\ \Pi \rhd e : \tau}{\Gamma;\ x_1 : S\tau_1;\ \cdots;\ x_m : S\tau_m;\ \Pi \setminus x : \tau' \rhd e : \tau}$$

The following derivation establishes this.

$$\dfrac{\dfrac{\dfrac{\Gamma;\ \Pi \rhd e : \tau}{\Gamma;\ \Pi;\ x_1 : S\tau_1;\ \cdots;\ x_m : S\tau_m \rhd e : \tau}\ \textbf{Rule}_5}{\Gamma;\ x_1 : S\tau_1;\ \cdots;\ x_m : S\tau_m;\ \Pi \setminus x : \tau' \rhd e : (x : \tau').\tau \qquad \Gamma;\ x_1 : S\tau_1;\ \cdots;\ x_m : S\tau_m;\ \Pi \setminus x : \tau' \rhd x : \tau'}\ \textbf{Pred}}{\Gamma;\ x_1 : S\tau_1;\ \cdots;\ x_m : S\tau_m;\ \Pi \setminus x : \tau' \rhd e : \tau}\ \textbf{Rel}$$

The **Rel** case is applicable since the required instances are specifically added to the type context involved. □

As a simple corollary to this result, $\mathcal{R}$ is also sound with respect to the instance relation.

$$\frac{\Gamma;\ \Pi \rhd \sigma \geq \sigma'}{\Gamma;\ \hat{\Pi} \rhd \sigma \geq \sigma'}\ \hat{\Pi} = \mathcal{R}(\Gamma, \Pi)$$

This follows from the relationship between typing and instance judgements, that is, $\mathbf{Rule}_3$ in Chapter 4.

6.7 $\mathcal{R}$ preserves instance judgements

Having shown the differences between $\mathcal{O}$ and the informal algorithm, and a sound algorithm $\mathcal{R}$ unifying the two approaches, it is now necessary to establish a theoretical basis justifying the use of an algorithm which appears *not* to compute principal types. This section describes the sense in which the informal algorithm, or more specifically $\mathcal{O}$ with $\mathcal{R}$, represents a sensible choice of algorithm.

The approach is to show that $\mathcal{R}$ preserves instance judgements: if a type scheme σ is related to another σ' by an instance judgement, then the corresponding types after the application of $\mathcal{R}$ are also related.

If $\Gamma \rhd e : \sigma$ then there exists Π and τ such that the following all hold.

$$\Gamma \rhd e : \overline{\Gamma}(\Pi.\tau)$$
$$\Gamma \rhd \overline{\Gamma}(\Pi.\tau) \geq \sigma$$
$$\Gamma \rhd \sigma \geq \overline{\Gamma}(\Pi.\tau)$$

In particular, any typing at σ has a trivial variant $\overline{\Gamma}(\Pi.\tau)$. It suffices to establish a result between type schemes of the latter form since $\mathcal{R}$ is applicable to sections, as opposed to type schemes.

Theorem 8 **$\mathcal{R}$ preserves instance judgements.**

$$\frac{\Gamma \rhd \overline{\Gamma}(\Pi.\tau) \geq \overline{\Gamma}(\Pi'.\tau')}{\Gamma \rhd \overline{\Gamma}(\hat{\Pi}.\tau) \geq \overline{\Gamma}(\hat{\Pi}'.\tau')} \quad \hat{\Pi} = \mathcal{R}(\Gamma, \Pi),\ \hat{\Pi}' = \mathcal{R}(\Gamma, \Pi')$$

That is, given two types of the form shown related by an instance judgement, if $\mathcal{R}$ is applied to both the sections used in their construction, then the instance judgement is preserved.

As an important corollary, this result provides a sense in which $\mathcal{R}$ preserves principal types. Since any term one would consider evaluating would be at a ground type (containing no predicates), this result asserts that the ground typing will be an instance of both the principal typing and $\mathcal{R}$ applied to the principal typing.

Proof

The proof is in two stages: first a general derivation is made connecting the premise to the conclusion. This derivation assumes a step $(*)$ which is subsequently shown to be valid.

From the premise, the following derivation holds.

$$\frac{\Gamma \rhd \overline{\Gamma}(\Pi.\tau) \geq \overline{\Gamma}(\Pi'.\tau')}{\Gamma;\ \Pi' \rhd \overline{\Gamma}(\Pi.\tau) \geq \tau'}\ \textbf{Rule}_5\text{, Spec and Rel}$$

$$\frac{\Gamma;\ \Pi' \rhd \overline{\Gamma}(\Pi.\tau) \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\Pi.\tau) \geq \tau'}\ \text{Soundness of } \mathcal{R}$$

$$\frac{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\Pi.\tau) \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\hat{\Pi}.\tau) \geq \tau'}\ (*)$$

$$\frac{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\hat{\Pi}.\tau) \geq \tau'}{\Gamma \rhd \overline{\Gamma}(\hat{\Pi}.\tau) \geq \overline{\Gamma}(\hat{\Pi}'.\tau')}\ \textbf{Rule}_8$$

It remains only to show that the step marked $(*)$ holds. Again, this is split into a simple and a hard part: the derivation $(*)$ above can be made in the following way

$$\frac{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\Pi.\tau) \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd S'\Pi.S'\tau \geq \tau'}\ S'\tau = \tau' \text{ and } \textbf{Rule}_{11}$$

$$\frac{\Gamma;\ \hat{\Pi}' \rhd S'\Pi.S'\tau \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd S'\hat{\Pi}.S'\tau \geq \tau'}\ (**)$$

$$\frac{\Gamma;\ \hat{\Pi}' \rhd S'\hat{\Pi}.S'\tau \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd \overline{\Gamma}(\hat{\Pi}.\tau) \geq \tau'}\ dom(S') \cap fv(\Gamma) = \{\}$$

assuming the step $(**)$ is valid. The substitution S' is some substitution of types for the generic variables of $\overline{\Gamma}(\Pi.\tau)$.

Finally it is possible to consider $(**)$ which represents the crux of the problem. This is shown by induction on the number of calls of $\mathcal{R}$ in the computation of $\hat{\Pi}$.

For some k, $\hat{\Pi}$ is the result of k applications of the body of $\mathcal{R}$. Take Π_i to be the result of i applications of the body of $\mathcal{R}$. The required result, therefore, is the following:

$$\frac{\Gamma;\ \hat{\Pi}' \rhd S'\Pi.S'\tau \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd S'\Pi_k.S'\tau \geq \tau'}\ (**)$$

with $\Pi_k = \hat{\Pi}$.

Base case. The base case is trivial. Since Π_0 is Π, the required result is simply the premise.

Inductive case. For the inductive case, establishing the result for Π_{k+1} when Π_k is known, one can conclude from that action of $\mathcal{R}$ that there is some binding $x :_i S'\tau_0 \in \Pi_k$ and a binding $x :_i \sigma' \in \Gamma$. If σ' is of the form

$$\sigma' = \forall\alpha_1.\cdots\forall\alpha_n.(x_1 : \tau_1).\cdots(x_m : \tau_m).\tau_0'$$

then there is some substitution S of types for $\{\alpha_1, \ldots, \alpha_n\}$ such that $S\tau_0' = S'\tau_0$.

Now, from the premise, there must be some instance binding in Γ; $\hat{\Pi}'$ discharging the predicate $x :_i S'\tau_0$ from Π_k. This instance binding cannot be in $\hat{\Pi}'$ since, as such, it would be discharged during the computation of $\hat{\Pi}'$. Since Γ is non-overlapping, the only applicable instance is the one discussed above. Therefore,

$$\Gamma;\ \hat{\Pi}' \rhd \sigma' \geq S'\tau_0,$$

and more importantly,

$$\exists (x_j :_i \sigma_j) \in \Gamma;\ \hat{\Pi}' \text{ s.t. } \Gamma;\ \hat{\Pi}' \rhd \sigma_j \geq S\tau_j$$

for $j \in \{1, \ldots, m\}$. Therefore, from the premise and the above discussion, the following derivation holds.

$$\frac{\Gamma;\ \hat{\Pi}' \rhd S'\Pi_k.S'\tau \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd S'\Pi_k \setminus x :_i S'\tau_0.S'\tau \geq \tau'}\ \textbf{Rule}_{10}$$

$$\frac{\Gamma;\ \hat{\Pi}' \rhd S'\Pi_k \setminus x :_i S'\tau_0.S'\tau \geq \tau'}{\Gamma;\ \hat{\Pi}' \rhd S'\Pi_{k+1}.S'\tau \geq \tau'}\ \textbf{Rule}_{10}\ m \text{ times in reverse}$$

This is the required result. □

Chapter 7

Canonical derivations

This chapter presents a normalisation process on typing judgement derivations. The process transforms an arbitrary derivation Δ deriving $\Gamma \rhd e : \sigma$ to a new derivation Δ^* deriving $\Gamma \rhd e^* : \sigma$, where the form of e^* is determined by the form of e. The resulting derivation is referred to as a *canonical* derivation. Such derivations satisfy several useful properties which are discussed below.

As well as being interesting in their own right, canonical derivations allow a *translation completeness* theorem to be established. This is useful for the subsequent discussion of coherence in Chapter 8.

Finally, the form of canonical derivations suggests a specialised version of the typing rules given previously. Specifically, a new reduced version of the type system is presented. The equivalence between this and the original system is illustrated.

7.1 Equality of translations

To begin with, it is necessary to define *provably equal* within the Damas-Milner system; as a consequence, provably equal is implied for derivations in the OL calculus.

Equality is defined by two re-write rules: the β- and `let`-reduction rules. Figure 7.1 contains the rules, they are as previously discussed in Chapter 1.

The first rule, β-reduction, specifies how to apply a lambda term: specifically, each occurrence of the formal parameter is replaced in the body by the actual argument. The second rule, `let`-reduction, performs a similar operation for `let` terms: each occurrence of the bound identifier in the body is replaced by the defining term. The transformation $\Longrightarrow$ maintains type correctness withing the Damas-Milner system.

Two terms, e_1 and e_2, are defined to be equal if there exists a term e such that $e_1 \Longrightarrow^* e$ and $e_2 \Longrightarrow^* e$ where $\Longrightarrow^*$ denotes the transitive closure of $\Longrightarrow$.

$$(\lambda x.\ e')\ e \quad \Longrightarrow \quad [e/x]e'$$
$$\texttt{let}\ x = e\ \texttt{in}\ e' \quad \Longrightarrow \quad [e/x]e'$$

Figure 7.1: Provably equal terms in the Damas-Milner calculus

7.2 Canonical OL derivations

This section presents a re-write system translating any derivation in the OL calculus into a canonical form. The system takes the form of three rules: these are applied repeatedly—in any order—until no rule is applicable.

The rules mutate a derivation tree. In the process, the terms involved change, as may the type contexts (with the exception of the top level type context). The types involved, however, remain the same. The three rules are presented below in turn.

Chapter 3 requires that no program variable be redeclared. In addition it is convenient to assume that no type variables are redeclared. That is, all derivations are α-converted prior to canonicalisation.

The Let elimination rule. The first rule can be used to eliminate all uses of the **Let** typing rule. A use of the **Let** rule contains a derivation, say Δ, of a typing of the defining term and introduces a binding into the type context recording this information. Whenever an instance of the **Taut** rule uses this binding, that instance of the **Taut** rule is replaced with the entire derivation of the defining term—that is, it is replaced with Δ. This is repeated for all uses of the binding in question.

The rule is given in Figure 7.2. Notice that *all* such uses of the **Taut** rule are replaced by Δ. The binding in question, $x : \sigma$, is also removed from all the type contexts involved.

Notice that the term in the new derivation is $[e_1/x]e_2$ which is determined by the form of the term in the given derivation. Thus, the particular form of the given derivation does not effect the resulting term. Further, this is the only rule affecting the form of the term. Hence, the statement above applies equally to the canonicalisation process as a whole.

The Gen/Spec elimination rule. Figure 7.3 contains the second rule for constructing derivations in canonical form. This rule is used to remove all uses of the **Gen** rule which are followed immediately by an application of the **Spec** rule.

The notation $[\tau/\alpha]\Delta$ denotes the derivation Δ with every free occurrence of the variable α replaced, simultaneously, with τ.

This rule is used to remove all consecutive uses of **Gen** and **Spec**. As such, a canonical derivation contains uses of these rules only at the leaves and root of the

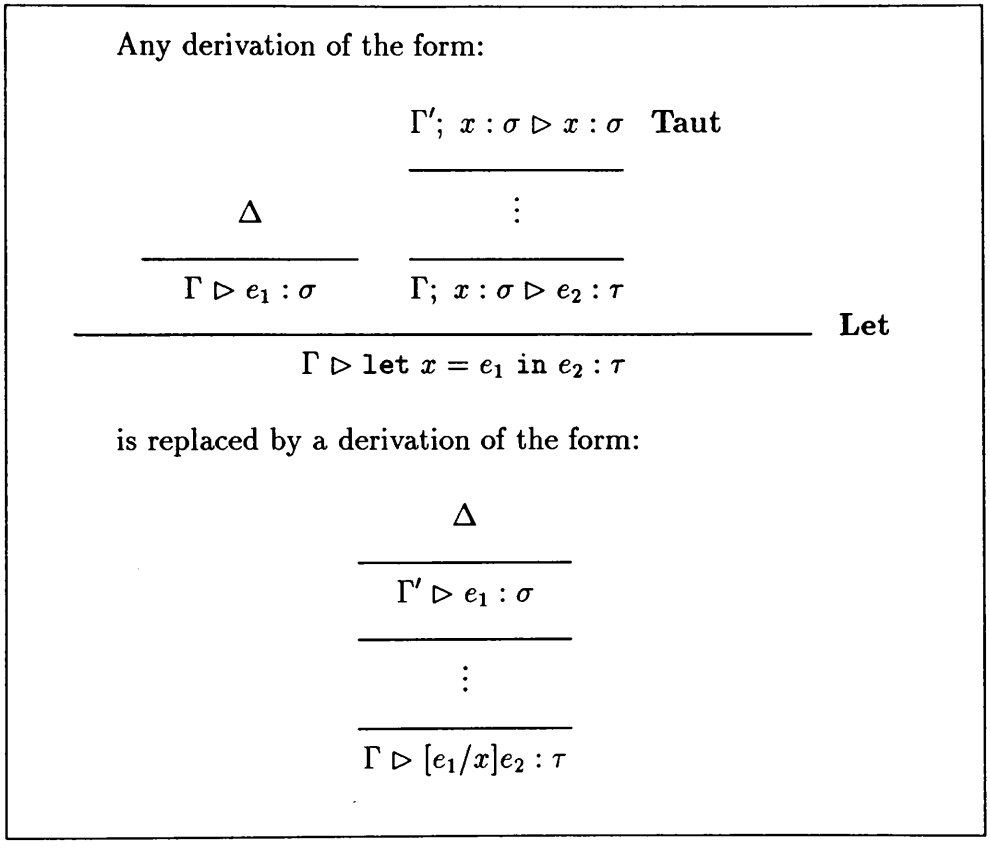

Figure 7.2: (1) Rules for canonical forms

Any derivation of the form:

$$\dfrac{\dfrac{\dfrac{\Delta}{\Gamma \rhd e : \sigma}}{\Gamma \rhd e : \forall\alpha.\sigma}\ \textbf{Gen}}{\Gamma \rhd e : [\tau/\alpha]\sigma}\ \textbf{Spec}$$

is replaced by a derivation of the form:

$$\dfrac{[\tau/\alpha]\Delta}{\Gamma \rhd e : [\tau/\alpha]\sigma}$$

Figure 7.3: (2) Rules for canonical forms

derivation: specifically, **Spec** is used at the leaves and **Gen** is used at the root. Any other occurrences of either of these rules must be adjacent to an application of the other and, as such, can be eliminated.

The Pred/Rel elimination rule. The third rule, in Figure 7.4, performs a similar function as the second rule but for consecutive applications of the **Pred** and **Rel** rules. In this case, however, the entire derivation of the judgement allowing the predicate to be discharged is replaced for *every* use of the instance binding at hand.

By this method, *all* such applications of these rules can be removed. Specifically, if a derivation is in canonical form, then all the uses of the **Rel** rule are at the leaves, and all the uses of the **Pred** rule are at the root.

Moving derivations within a proof. In the rules above, sub-derivations and types from one part of a proof tree are replicated in other parts of the tree. It is required that these sub-derivations and types maintain their meaning.

The assumption that terms and types are α-converted ensures that no binding on which the sub-derivation or type at hands depends is removed from the type context. In addition, if a derivation holds given a particular type context, then the derivation holds under any extension of that type context. This is a form of weakening. Together these properties ensure the relocated derivations and types are valid.

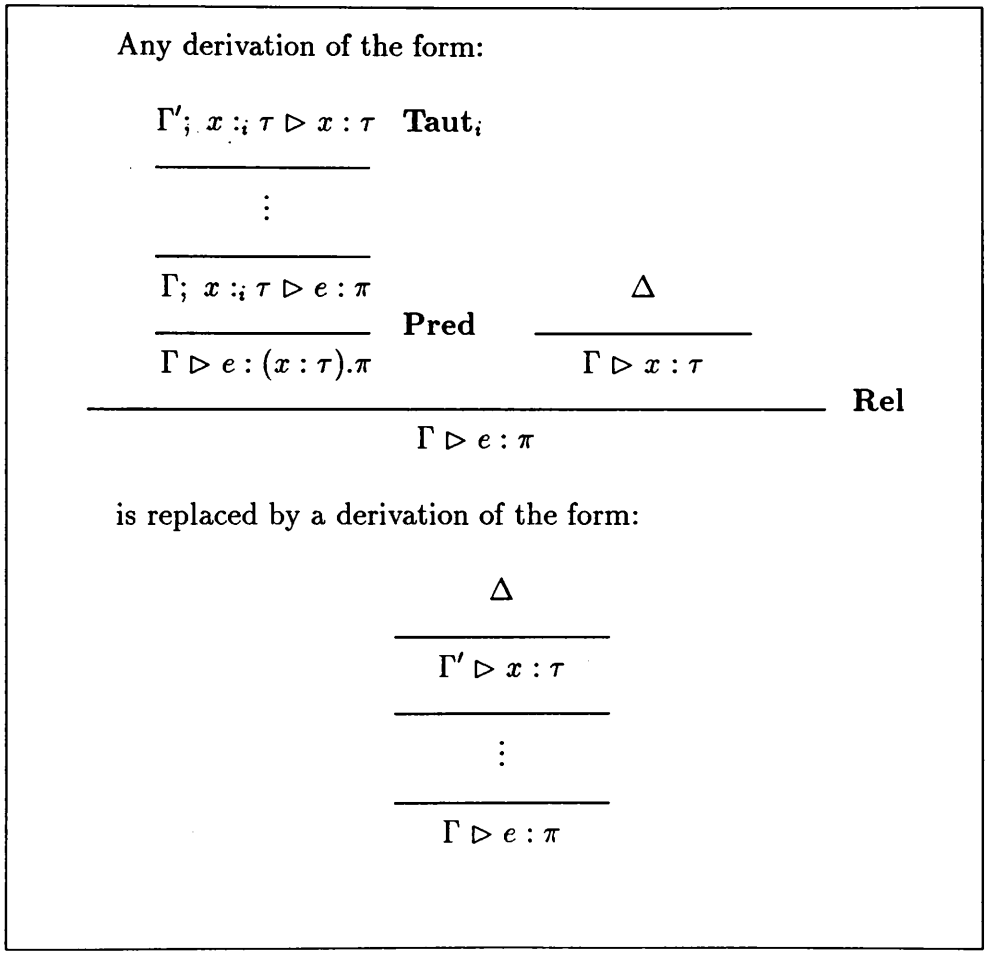

Figure 7.4: (3) Rules for canonical forms

7.3 Discussion of Canonical derivations

Canonical derivations have several interesting and useful properties. Some of these are discussed below.

Termination and convergence. There is an injection encoding derivations in the OL calculus in the polymorphic lambda calculus. For most judgement forms the translation is obvious. For **Let** judgements the translation is to directly applied λ-abstractions; for **Pred** and **Rel** judgements the translation is to λ-abstractions and applications; and for **Abs** and **Comb** judgements an *apply* operator is introduced contriving that the real β-reductions are not possible.

Under such a scheme, applications of the canonicalisation rules are in one-to-one correspondence with β-reductions in the polymorphic calculus.

- The **Let** elimination rule coresponds to β-reduction;
- the **Gen/Spec** elimination rule corresponds to β-reduction for type abstraction and application; and
- the **Pred/Rel** elimination rule also corresponds to β reduction.

Since the polymorphic lambda calculus is strongly normalisable and confluent, it follows that the canonicalisation process terminates and converges.

Maintaining typings. Each rule for transformation to a canonical derivation maintains the judgement that the derivation as a whole yields: that is, if Δ derives $\Gamma \rhd e : \sigma$, and Δ^* is the corresponding derivation in canonical form, then Δ^* derives $\Gamma \rhd e^* : \sigma$. Further, as one would expect, Δ^* is a valid typing judgement derivation.

Equality of meaning. Notice that each of the rules for transforming a derivation maintains the meaning of the derivation as a whole. This is clearly true for the elimination of **Gen/Spec** pairs since the applications of such rules do not effect the translation of the term.

The elimination of **Pred/Rel** pairs maintains meaning since the resulting derivation is simply the β-reduction of the meaning of the initial derivation.

Finally, the **Let** elimination rule obviously satisfies the `let`-reduction rule required of the semantics.

The form of canonical derivations. If Δ is a derivation of the judgement $\Gamma \rhd e : \sigma$, and Δ reduces to the derivation Δ^* by the above process, then the following statements are true of Δ^*.

- Derivation Δ^* contains no `let` terms.

- Derivation Δ^* contains no uses of the **Gen** or **Pred** rules apart from at the root, that is, they must be the very last rules applied. Any other occurrences of these rules must be adjacent to their corresponding elimination rules and as such will be eliminated in the final canonical form.
- All uses of the **Spec** and **Rel** rules are at the leaves. This is the case for the same reason given immediately above.
- If σ is a *type* with no quantification or predicates, then there are no uses of the **Gen** or **Pred** rules. In particular, since there are no uses of the **Pred** rule, the instance bindings in scope throughout the derivation cannot change.

7.4 Restricted translation completeness

This section presents a completeness result with respect to the translations associated with typing derivations. In particular, the meaning assigned to a derivation of the principal typing is related to the meaning assigned to derivations of instances thereof.

7.4.1 Canonical derivations and principal types

Any application of $\mathcal{O}$ defines a typing judgement derivation Δ: that is, if

$$\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$$

then a derivation Δ is defined deriving $S\Gamma;\ \Pi \rhd e : \tau$. The proof of the soundness of $\mathcal{O}$, in Chapter 5, constructs a derivation tree. That tree is taken to be the derivation tree *defined by* the application of $\mathcal{O}$ at hand.

If $\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ defining Δ, and Δ^* is the corresponding canonical derivation deriving $S\Gamma;\ \Pi \rhd e^* : \tau$, then

$$\mathcal{O}(\Gamma, e^*) = (S', \Pi', \tau') \text{ defining } \Delta'$$

such that $\overline{S\Gamma}(\Pi.\tau)$ is a trivial variant of $\overline{S'\Gamma}(\Pi'.\tau')$. Further, all the derivations involved have equal meaning.

$$\lfloor\Delta\rfloor = \lfloor\Delta^*\rfloor = \lfloor\Delta'^*\rfloor = \lfloor\Delta'\rfloor$$

The proof of both these assertions follows from observing the effect of each canonicalisation rule on the action of $\mathcal{O}$. In particular, the **Let** elimination rule is the only one which can effect the action of $\mathcal{O}$. The action of $\mathcal{O}$, both in terms of type and meaning assigned, is unchanged by canonicalisation.

7.4.2 Translation completeness for canonical derivations

It is now possible to present a translation completeness result: from the derivation computed by $\mathcal{O}$, any possible meaning can be derived. The result comes in two parts: this section presents a limited result with respect to canonical derivations, Section 7.5 generalises this to arbitrary typings.

Theorem 9 Translation completeness for canonical derivations. *If* $\Gamma \rhd e : \tau'$ *is a derivation by* Δ' *in canonical form, and* $\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ *defines* Δ *with*

$$\Pi = \{x_1 :_i \tau_1, \ \ldots, \ x_n :_i \tau_n\}$$

then for some substitution S' *of types for the generic variables of* $\overline{\Gamma}(\Pi.\tau)$ *with* $S'\tau = \tau'$, *there are derivations*

$$\frac{\Delta_1}{\Gamma \rhd x_1 : S'\tau_1} \quad \cdots \quad \frac{\Delta_n}{\Gamma \rhd x_n : S'\tau_n}$$

such that $[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor; \ \cdots; \ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor$ *is equal to* $\lfloor\Delta'\rfloor$.

Proof

From the syntactic completeness result, it follows that $S\Gamma$ is a trivial renameing of Γ. It can be safely assumed that S is the identity substitution. Also from the proof of the syntactic completeness result and $\mathbf{Rule}_{11}$, it follows that derivations Δ_1 to Δ_n exist. It is required to show that the final equality holds.

The proof is by induction of the structure of e. Since Δ' is in canonical form, there can be no uses of the **Gen** or **Pred** rules, all uses of the **Spec** and **Rel** rules must be at the leaves, and the only other rules used must be **Abs**, **Comb** and the two **Taut** rules.

Identifiers. If e is x and

$$x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \in \Gamma$$

then, by examination of the action of $\mathcal{O}$,

$$\begin{array}{lcl} \tau & = & \tau \\ \Pi & = & \{x_1 :_i \tau_1; \ \cdots; \ x_n :_i \tau_n\} \\ \lfloor\Delta\rfloor & = & x \ \lfloor x_1 :_i \tau_1\rfloor \ \cdots \ \lfloor x_n :_i \tau_n\rfloor \end{array}$$

or a trivial variant thereof. Taking S'' such that $S''\tau = \tau'$ and Δ'_i to derive $\Gamma \rhd x_i : S''\tau_i$, for $i \in \{1, \ldots, n\}$, The derivation Δ' must be of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \rhd x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau}{\vdots}\ \textbf{Spec}}{\Gamma \rhd x : (x_1 : S''\tau_1).\cdots(x_n : S''\tau_n).S''\tau}}{\vdots}\ \textbf{Rel}\ \Delta'_i}{\Gamma \rhd x : S''\tau}$$

The translation of Δ', therefore, is $x\ \lfloor\Delta'_1\rfloor\ \cdots\ \lfloor\Delta'_n\rfloor$.

Taking S' to be S'' and Δ_i to be Δ'_i gives

$$\begin{aligned}\lfloor\Delta'\rfloor &= x\ \lfloor\Delta'_1\rfloor\ \cdots\ \lfloor\Delta'_n\rfloor \\ &= x\ \lfloor\Delta_1\rfloor\ \cdots\ \lfloor\Delta_n\rfloor \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor](x\ \lfloor x_1 :_i \tau_1\rfloor\ \cdots\ \lfloor x_n :_i \tau_n\rfloor) \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor\end{aligned}$$

as required.

Overloaded identifiers. The case for overloaded identifiers is similar to that above. For Γ to be valid it must contain a binding of the form $x :_o \forall\alpha_1.\cdots\forall\alpha_m.\tau$. Again by examination of $\mathcal{O}$, Δ is a derivation of the form

$$\dfrac{\Delta}{\Gamma;\ x :_i \tau \rhd x : \tau}$$

or a trivial variant thereof. In particular, $\lfloor\Delta\rfloor$ is $\lfloor x :_i \tau\rfloor$. There must be a binding

$$x :_i \forall\beta_1 \cdots \forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau'' \in \Gamma$$

such that the derivation Δ' is of the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \rhd x : \forall\beta_1 \cdots \forall\beta_p.(x'_1 : \tau'_1).\cdots.(x'_q : \tau'_q).\tau''}{\vdots}\ \textbf{Spec}\ S''\tau'' = \tau'}{\Gamma \rhd x : (x_1 : S''\tau'_1).\cdots(x_n : S''\tau'_q).S''\tau''}}{\vdots}\ \textbf{Rel}\ \Delta'_i}{\Gamma \rhd x : S''\tau}$$

where Δ'_i derive $\Gamma \rhd x_i : S''\tau'_i$, for $i \in \{1, \ldots, q\}$.

Since Γ is valid, there must be some substitution S_0 such that $S_0\tau = \tau''$ and $S''S_0\tau = \tau'$. Take S' to be $S''S_0$.

Now, taking Δ_1 to be Δ', which proves the required judgement, gives

$$\begin{aligned} \lfloor\Delta'\rfloor &= \lfloor\Delta_1\rfloor \\ &= [\lfloor\Delta_1\rfloor/\lfloor x :_i \tau\rfloor](\lfloor x :_i \tau\rfloor) \\ &= [\lfloor\Delta_1\rfloor/\lfloor x :_i \tau\rfloor]\lfloor\Delta\rfloor \end{aligned}$$

as required.

Abstraction terms. The case for abstraction terms (and application terms) is slightly more complicated. The final step of the derivation Δ' must be of the form

$$\dfrac{\dfrac{\Delta''}{\Gamma;\ x : \tau'' \rhd e : \tau'''}}{\Gamma \rhd \lambda x.\ e : \tau'' \to \tau'''}\ \textbf{Abs}$$

where $\tau' = \tau'' \to \tau'''$. Since $\Gamma;\ x : \tau'' \rhd e : \tau'''$, it must be the case that

$$\mathcal{O}(\Gamma;\ x : \tau'',\ e) = (S^0, \Pi^0, \tau^0)$$

succeeds by derivation Δ^0 yielding the inductive hypothesis that

$$[\lfloor\Delta^0_1\rfloor/\lfloor x_1 :_i \tau^0_1\rfloor;\ \cdots;\ \lfloor\Delta^0_n\rfloor/\lfloor x_n :_i \tau^0_n\rfloor]\lfloor\Delta^0\rfloor = \lfloor\Delta''\rfloor$$

for some $\lfloor\Delta^0_1\rfloor$ to $\lfloor\Delta^0_n\rfloor$ with substitution S'^0 of types for the generic variables of $\overline{\Gamma;\ x : \tau''}(\Pi^0.\tau^0)$.

Since $\mathcal{O}(\Gamma;\ x : \tau'',\ e) = (S^0, \Pi^0, \tau^0)$ succeeds, it follows that $\mathcal{O}(\Gamma;\ x : \beta,\ e) = (S^1, \Pi^1, \tau^1)$ succeeds by derivation Δ^1. Further, since the meaning of a derivation computed by $\mathcal{O}$ does not depend on the types associated with λ-bound identifiers, it follows that:

$$\lfloor\Delta^1\rfloor = \lfloor\Delta^0\rfloor.$$

Now, since both $\lfloor\Delta\rfloor$ and $\lfloor\Delta'\rfloor$ are constructed from $\lfloor\Delta^1\rfloor$ and $\lfloor\Delta''\rfloor$, respectively, by applications of the **Abs** rule, the following are the case:

$$\lfloor\Delta\rfloor \text{ is } \lambda x.\ \lfloor\Delta^1\rfloor \quad \text{and} \quad \lfloor\Delta'\rfloor \text{ is } \lambda x.\ \lfloor\Delta''\rfloor.$$

Take Π to be of the form

$$\Pi = \{x_1 :_i \tau_1;\ \cdots;\ x_n :_i \tau_n\}.$$

From the syntactic completeness result, S' exists such that $S'\tau = \tau'$, and $S'\tau_i = S'^0\tau_i^0$ and $\Gamma \rhd x_i : S'\tau_i$ for $i \in \{1, \ldots, n\}$. Now, since $S'\tau_i = S'^0\tau_i^0$, taking Δ_i to be Δ_i^0 yield derivations of the required judgements. The following equalities hold:

$$\begin{aligned}
\lfloor\Delta'\rfloor &= \lambda x.\ \lfloor\Delta''\rfloor \\
&= \lambda x.\ [\lfloor\Delta_1^0\rfloor/\lfloor x_1 :_i \tau_1^0\rfloor;\ \cdots;\ \lfloor\Delta_n^0\rfloor/\lfloor x_n :_i \tau_n^0\rfloor]\lfloor\Delta^0\rfloor \\
&= [\lfloor\Delta_1^0\rfloor/\lfloor x_1 :_i \tau_1^0\rfloor;\ \cdots;\ \lfloor\Delta_n^0\rfloor/\lfloor x_n :_i \tau_n^0\rfloor]\lambda x.\ \lfloor\Delta^0\rfloor \\
&= [\lfloor\Delta_1^0\rfloor/\lfloor x_1 :_i \tau_1^0\rfloor;\ \cdots;\ \lfloor\Delta_n^0\rfloor/\lfloor x_n :_i \tau_n^0\rfloor]\lambda x.\ \lfloor\Delta^1\rfloor \\
&= [\lfloor\Delta_1^0\rfloor/\lfloor x_1 :_i \tau_1^0\rfloor;\ \cdots;\ \lfloor\Delta_n^0\rfloor/\lfloor x_n :_i \tau_n^0\rfloor]\lfloor\Delta\rfloor \\
&= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor
\end{aligned}$$

as required.

Application terms. Finally, the derivation Δ' must be of the form

$$\dfrac{\dfrac{\Delta_1'}{\Gamma \rhd e_1 : \tau_1' \to \tau_2'} \qquad \dfrac{\Delta_2'}{\Gamma \rhd e_2 : \tau_1'}}{\Gamma \rhd e_1\ e_2 : \tau_2'}$$

where $\tau_2' = \tau'$. Similarly, the proof Δ constructed by $\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ is of the form

$$\dfrac{\dfrac{\Delta_1^0}{\Gamma \rhd e_1 : \tau_1 \to \tau_2} \qquad \dfrac{\Delta_2^0}{\Gamma \rhd e_2 : \tau_1}}{\Gamma \rhd e_1\ e_2 : \tau_2}$$

where many of the substitutions have been omitted as they represent essentially the same argument that is made in the syntactic completeness proof of Chapter 5.

From the left-hand recursive call the inductive hypothesis yields the existence of Δ_1 to Δ_{m-1} such that

$$[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_{m-1}\rfloor/\lfloor x_{m-1} :_i \tau_{m-1}\rfloor]\lfloor\Delta_1^0\rfloor = \lfloor\Delta_1'\rfloor$$

Similarly, the right recursive call yields Δ_m to Δ_n such that

$$[\lfloor\Delta_m\rfloor/\lfloor x_m :_i \tau_m\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta_2^0\rfloor = \lfloor\Delta_2'\rfloor$$

Since every instance binding in Π is used in one or other, but not both, of the derivations $\lfloor\Delta_1^0\rfloor$ and $\lfloor\Delta_2^0\rfloor$, it follows that

$$[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta_1^0\rfloor = \lfloor\Delta_1'\rfloor$$
$$[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta_2^0\rfloor = \lfloor\Delta_2'\rfloor$$

as the extra substitutions are irrelevant in each case. The translation of Δ is the application term $(\lfloor\Delta_1^0\rfloor\ \lfloor\Delta_2^0\rfloor)$ giving

$$\begin{aligned}\lfloor\Delta'\rfloor &= \lfloor\Delta_1'\rfloor\ \lfloor\Delta_2'\rfloor \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor c_n :_i \tau_n\rfloor](\lfloor\Delta_1^0\rfloor\ \lfloor\Delta_2^0\rfloor) \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor c_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor\end{aligned}$$

as required.

Let abstractions. Since the derivation under consideration is in canonical form, no `let` terms can occur. □

7.5 Translation completeness

It is now possible to present a more general semantic completeness result. In particular, the restriction to canonical derivations can be removed.

Theorem 10 Translation completeness. *If* $\Gamma \rhd e : \tau'$ *by a derivation* Δ' *and* $\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ *defines* Δ *with*

$$\Pi = \{x_1 :_i \tau_1,\ \ldots,\ x_n :_i \tau_n\}$$

then for some substitution S' *of types for the generic variables of* $\overline{\Gamma}(\Pi.\tau)$ *with* $S'\tau = \tau'$, *there are derivations*

$$\frac{\Delta_1}{\Gamma \rhd x_1 : S'\tau_1} \quad \cdots \quad \frac{\Delta_n}{\Gamma \rhd x_n : S'\tau_n}$$

such that $[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor$ *is equal to* $\lfloor\Delta'\rfloor$.

This is the same as the previous result except for the omission of the requirement that Δ' be in canonical form.

Proof

Take Δ'^* to be the canonical form of Δ'. As discussed previously, $\lfloor\Delta'^*\rfloor = \lfloor\Delta'\rfloor$ and Δ'^* derives $\Gamma \rhd e^* : \tau$.

Now, if $\mathcal{O}(\Gamma, e^*) = (S^*, \Pi^*, \tau^*)$ by Δ^*, then $\overline{S\Gamma}(\Pi.\tau)$ and $\overline{S^*\Gamma}(\Pi^*.\tau^*)$ are trivial variants, and the meaning of canonical form of Δ is the same as Δ^*. Further, by the previous lemma, appropriate Δ_i exists such that

$$[\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta^*\rfloor = \lfloor\Delta'^*\rfloor.$$

The following equivalences hold

$$\begin{aligned}\lfloor\Delta'\rfloor &= \lfloor\Delta'^*\rfloor \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor]\lfloor\Delta^*\rfloor \\ &= [\lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_1\rfloor/\lfloor x_1 :_i \tau_1\rfloor]\lfloor\Delta\rfloor\end{aligned}$$

giving the required result. □

7.6 An equivalent reduced type system

Finally, the canonicalisation process suggests an equivalent reduced type system for OL. The reduced system is strongly related to the four-rule version of the Damas-Milner system presented in [Cle86].

Consider, for the time being, the canonicalisation process without the **Let** elimination rule: only the **Gen/Spec** and **Pred/Rel** elimination rules are applicable.

Any derivation in the OL type calculus can be mapped to a derivation in this limited canonical form. All derivations in the reduced calculus have this limited canonical structure.

The reduced version of the system is given in Figure 7.5. Notice that the **Gen**, **Pred**, **Spec** and **Rel** rules are omitted. The functionality of **Gen** and **Pred** is now captured by the **Para** rule, for *parametrisation*; and the functionality of the **Spec** and **Rel** rules is now captured by instantiation within the **Taut** rules.

The new rule, **Para**, allows for the generalisation of a typing. It corresponds to a sequence of applications of the **Pred** rule followed by a sequence of applications of the **Gen** rule.

Observing the form of canonical derivations, the system could further be restricted such that the instance derivations in the two **Taut** rules need never use the **Gen** or **Pred** rules.

The full and the reduced versions of the OL system are equivalent. A typing in the full version of the system implies a typing in the reduced version, and *vice versa*. In one direction the equivalence is

$$\Gamma \rhd e : \sigma \Longrightarrow \Gamma \rhd_r e : \sigma' \text{ where } \Gamma \rhd \sigma' \geq \sigma$$

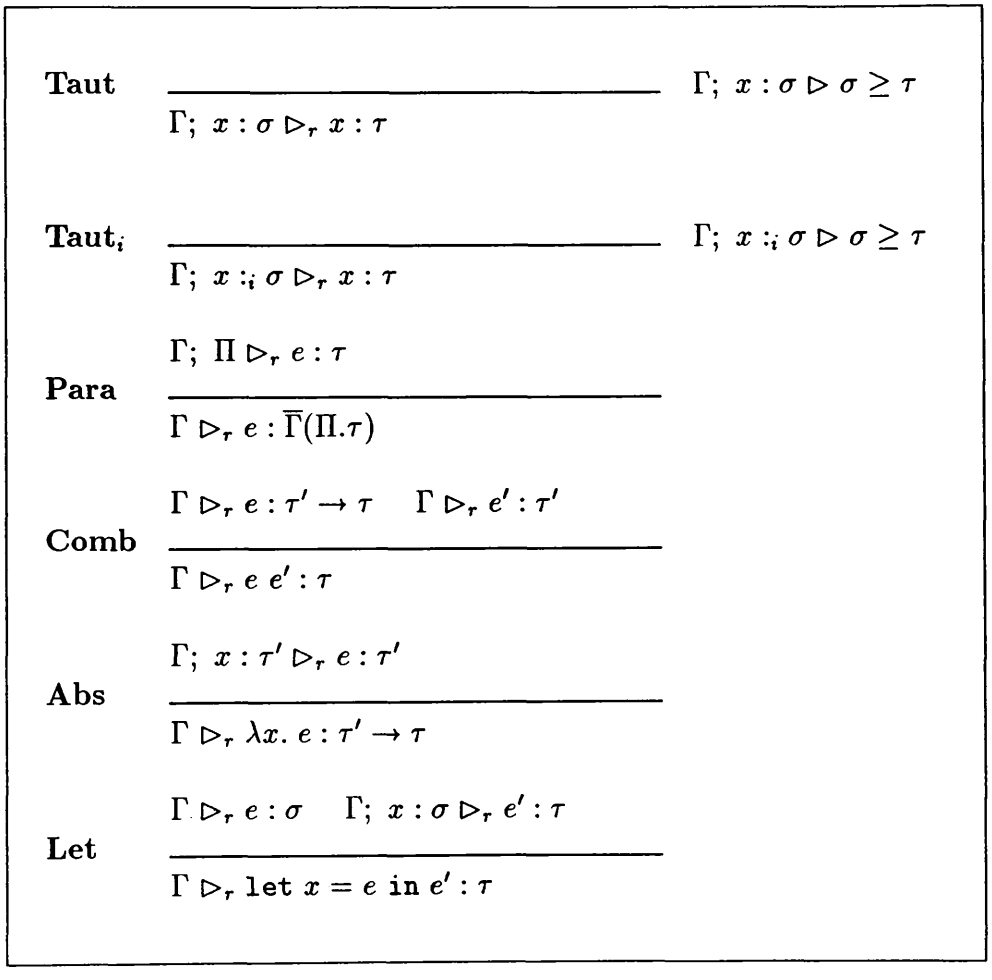

Figure 7.5: The reduced type system for OL

whereas in the other direction it is

$$\Gamma \rhd_r e : \sigma \Longrightarrow \Gamma \rhd e : \sigma$$

The existence of canonical derivation, without the **Let** elimination rule, implies these equivalences.

Chapter 8

Ambiguity and coherence

Ambiguity and coherence are two important technical considerations with practical significance: *ambiguity* is a syntactic property of a typing; and *coherence* is a semantic property of translation based semantics. They are closely related and, as such, merit being discussed in the same chapter. This Chapter defines both ambiguity and coherence with respect to OL; shows how they are related; and proposes an approach to the problems exposed.

8.1 Ambiguity

Ambiguity is a syntactic property of a typing, and an undesirable property. If the principal typing of a term is ambiguous, then the term does not contain enough type information to define the instances appropriate to discharge the predicates in its typing. In particular, since unification is used during type inference, if $\mathcal{O}$ computes an ambiguous typing at any point, then it is clear that unification is not able to determine the type at which a particular occurrence of an overloaded operator should be used.

Fortunately a simple syntactic test identifies ambiguous typings.

Definition 8.1 Ambiguity. *A typing* $\Gamma \rhd e : \sigma$ *where*

$$\sigma = \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$$

is ambiguous if there exists a type variable α *such that*

- $\alpha \in fv(\tau_i)$ *for some* $i \in \{1, \ldots, m\}$;
- $\alpha \notin fv(\tau)$; *and*
- *either* $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$, *or* $\alpha \notin fv(\Gamma)$.

That is, a typing is ambiguous if there is a type variable α appearing in the predicate part but not in the type part, and α is either generic or not free in the type context. In either of these cases the type variable could not possibly be disambiguated by unification.

By extension to ambiguity, an i-unambiguous type context is one containing no ambiguous instance bindings.

Definition 8.2 An i-unambiguous type context. *A type context Γ is said to be i-unambiguous if for all bindings $x :_i \sigma \in \Gamma$, the judgement*

$$\Gamma \rhd x : \sigma$$

is unambiguous.

8.1.1 Examples of ambiguous typings

This section gives some examples of the situations in which ambiguities arise; it is not exhaustive. As a general rule, ambiguities arise whenever overloaded data is used in expressions but does not appear in the final results.

Redundant data. The most obvious example of ambiguity is when data is abstracted from a structure; this may be viewed as the rest of the structure being discarded. For example, the expression

$$snd(square, true)$$

clearly evaluates to an object of type *Bool*. The principal type of the application is, however,

$$\forall\alpha.((*) : \alpha \to \alpha \to \alpha).Bool$$

which is ambiguous since α is a generic variable appearing in the predicate part but not in the type part. Extracting a part of the tuple structure amounts to discarding the rest; in the case above, that part which is overloaded is discarded. The overloading, represented by the predicate $(*) : \alpha \to \alpha \to \alpha$, still appears in the principal typing of the term as a whole. This ambiguity is semantically unimportant.

Ambiguity in tests. A common example of data which does not appear in the result of evaluating an expression is the condition part of an 'if' or 'case' expression. In this case, the value not appearing in, but determining, the result is clearly semantically important.

As an example of this form of ambiguity, consider the case where the literals *zero* and *unit* are overloaded to represent the zero and unit values of various numeric types. That is, they have the signatures

$$\begin{aligned} zero &:_o \forall\alpha.\alpha \\ unit &:_o \forall\alpha.\alpha \end{aligned}$$

and appropriate instances at all numeric types. Then

$$(zero == unit) : \forall\alpha.(zero : \alpha).(unit : \alpha).((==) : \alpha \to \alpha \to Bool).Bool$$

is an ambiguous typing. Further, the ambiguity is semantically significant. Though in practice one would always expect this term to evaluate to false, this is not imposed by the language. For some some contrived numeric type, one may wish the unit and zero values to be identified.

Ambiguous intermediate values. Function composition creates intermediate values; the types of these values, however, need not appear in the type of the term as a whole. If such a value is overloaded then it may cause a semantically significant ambiguity.

For example, given the overloaded input/output functions

$$\begin{aligned} read &:_o \forall\alpha.List(Char) \to \alpha \\ write &:_o \forall\alpha.\alpha \to List(Char) \end{aligned}$$

the composition $\lambda x.\ write\ (read\ x)$ is a function with an overloaded intermediate value and a resulting ambiguous type. If LC abbreviates $List(Char)$, then its type is as follows.

$$\forall\alpha.(write : \alpha \to LC).(read : LC \to \alpha).LC \to LC$$

To see that this is semantically significant, consider the lengths of the results of the above function specialised to both *Int* and *Char* when applied to the string "345". One would expect the results to be of lengths 3 and 1 respectively—clearly not provably equal.

Discussion. In the examples above, it is implied that there are two classes of ambiguities—those which are, and those which are not, semantically significant. Unfortunately, there does not appear to be a simple test distinguishing the two. As such, all ambiguities are treated as undesirable.

It is tempting to declare all ambiguities illegal. Experience with Haskell, however, has shown that such cases are rather too common to be eliminated—an alternative is required. Haskell adopts a *default* mechanism to define how ambiguous predicates are discharged; the default system is discussed briefly in Chapter 9.

The remainder of this Chapter deals primarily with unambiguous typings.

8.2 Coherence

Coherence is a property of a translation semantics [BCGS88] (as opposed to the translation of a particular typing). A translation semantics is *coherent* if the meaning assigned to all possible derivations of a particular typing judgement are the same. Specifically, it is the property whereby one may know from a typing judgement alone—without its derivation—the meaning of the term. That is not to say the semantics are not based on the derivation, merely that all derivations of a given typing yield the *same* meaning.

Taking the definition of "provably equal" from Figure 7.1, the definition of coherence is as follows.

Definition 8.3 Definition of coherence. *The translation semantics of OL is* coherent *if, and only if, for any two derivations* Δ_1 *and* Δ_2 *of a typing judgement* $\Gamma \rhd e : \sigma$, *if* $\lfloor \Delta_1 \rfloor$ *is the meaning assigned to* Δ_1, *and* $\lfloor \Delta_2 \rfloor$ *is the meaning assigned to* Δ_2, *then* $\lfloor \Delta_1 \rfloor$ *and* $\lfloor \Delta_2 \rfloor$ *are provably equal.*

That is, if OL is coherent, then given two derivations of the same typing their associated meanings will be provably equal.

8.2.1 Examples of incoherence

OL is coherent only when certain restrictions are satisfied. There are two principal sources of incoherence which are explained below. A further minor source of incoherence is also discussed, though this arises primarily for technical reasons.

Overlapping instance bindings. It is easy to see that two instance bindings at the same type result in incoherent typings. For example, if a typing context contains two instance bindings for equality over integers

$$==:_i \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$==:_i \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$

then the term $(3 == 4)$ has two simple derivations to type *Bool*—one for each instance. Since the translation rules demand that distinct instance bindings are implemented by distinct implementation identifiers, these two derivations lead to unequal translations.

The restriction of type contexts to be non-overlapping has already been used in Chapter 6. Non-overlapping type contexts are required to ensure coherence.

Ambiguity and coherence. The second cause of incoherence is ambiguity. Ambiguity, a syntactic condition, reflects overloaded intermediate values in a computation which may be semantically significant. Semantically significant ambiguities lead directly to incoherence.

For example, following on the example in Section 8.1.1, it is easy to see that there are at least two derivations of a typing

$$(zero == unit) : Bool;$$

using instances at types *Int* and *Float* respectively. These need not be equal[1].

Frivolous use of Pred. A third, though less worrying, source of ambiguity is the frivolous use of the **Pred** rule. This is most easily observed in the typing of a term consisting only of an overloaded identifier. For example, the term *zero* has, among others, the following typing

$$\Gamma \rhd zero : \forall\alpha.(zero : \alpha).(zero : \alpha).\alpha$$

with at least two possible derivations leading to the following translations.

$$\lambda da_1.\ \lambda da_2.\ da_1 \qquad \text{or} \qquad \lambda da_1.\ \lambda da_2.\ da_2$$

The implementation identifier corresponding to either of the predicates could be used as the actual implementation. These two terms are not, in general, provably equal.

8.3 OL, ambiguity and coherence

As discussed above, one of the principal sources of incoherence with OL is ambiguity. An appealing approach may be, therefore, to impose some restriction on the system prohibiting ambiguity. Several alterations to the system have been considered aiming to achieve this goal.

An early approach was to restrict overloaded operators signatures to be of the form

$$\forall\alpha_1.\cdots\forall\alpha_n.(x_1 : \tau_1).\cdots(x_m : \tau_m).\alpha_1 \to \cdots \to \alpha_n \to \tau$$

where an "example" of each type parameter is required in each instance. Ambiguity is still possible under this approach, recursion can be used to construct an ambiguous (and incoherent) typing.

The counter examples in the cases above rely upon recursion to achieve ambiguity. The next approach was to require that each instance be strict in the "example" arguments. However, it is possible to construct counter examples in this case too.

[1]It is possible to argue that one would expect the results of the expressions to be equal. There are two reasons why this argument is not considered here: firstly, it requires a semantic check (as opposed to syntactic) on a property of all instances of all the overloaded identifiers involved; and secondly because what appears like a reasonable semantic restriction in one numeric type may not be so in another. As such, semantic arguments of this form are omitted from the current discussion.

In both the cases above, counter examples are constructed by utilising functions of the form

$$\begin{array}{ll} \mathit{fix} & : \forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha \\ \mathit{bottom} & : \forall\alpha.\alpha \\ \mathit{coerce} & : \forall\alpha.\forall\beta.\alpha \rightarrow \beta \end{array}$$

that is, types for which no term exists in the pure lambda calculus. Further, these types need not be assigned to overloaded operators in order to facilitate the construction of examples of ambiguity. Though it represents a significant reduction in expressiveness, the restriction of all bindings in a type context to be "lambda definable" may eliminate ambiguity.

8.4 The coherence of typings of overloaded identifiers

This section presents a preliminary coherence result for the typing of overloaded identifiers. This result is then used in the proof of the coherence theorem. Under certain restrictions, two derivations of a typing of an overloaded identifier at a particular type, have the same translations.

A further restriction is required on type contexts. In particular, for coherence the type context concerned must be i-unambiguous as defined previously. That is, all the instance bindings it contains must be unambiguous.

The coherence lemma is stated formally as follows:

Lemma 11 Coherence of typings of overloaded identifiers. *If Γ is non-overlapping and i-unambiguous, and $\Gamma \rhd x : \tau'$ for some overloaded identifier x by two derivations Δ_1 and Δ_2, then the translations associated with the two derivations are equal.*

Proof

The proof observes that the meaning of the derivations are determined by the type context and type involved. It suffices to consider derivations using only the **Taut**$_i$, **Spec** and **Rel** rules since all derivations have associated canonical forms with meaning equal to the given derivations. Given the form of the typing, the rules the canonical forms utilise are determined.

Since Γ is non-overlapping, there can be only one instance applicable, say the following.

$$x :_i \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \in \Gamma$$

Now, since each instance must be unambiguous, there must exist only one substitution S of types for $\{\alpha_1, \ldots, \alpha_n\}$ such that

$$\frac{\Gamma \rhd x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau}{\Gamma \rhd x : (x_1 : S\tau_1).\cdots(x_m : S\tau_m).S\tau}\ \textbf{Spec}$$

$$\frac{\Gamma \rhd x : (x_1 : S\tau_1).\cdots(x_m : S\tau_m).S\tau}{\Gamma \rhd x : S\tau}\ \textbf{Rel}$$

with $S\tau = \tau'$. S is unique since all instance bindings are unambiguous.

Now, the instance discharging the predicate and S are uniquely determined. The lemma may now be used inductively on each predicate $(x_i : S\tau_1)$ in turn yielding the required result. The inductive argument terminates since the two derivations Δ_1 and Δ_2 have been given.

Since the form of the canonical derivations are determined, the meaning in the associated translation is determined. The given derivations, Δ_1 and Δ_2, have meaning equal to the meaning of their associated canonical derivations, and these have a uniquely determined meaning. This is as required. □

8.5 The coherence theorem

This section presents a limited coherence result: that is, under a specific set of restrictions motivated by the preceding discussion, the system is coherent. To reiterate, the restrictions are as follows.

Overlapping instance bindings: the coherence result is predicated upon non-overlapping type contexts.

Ambiguities: the coherence result applies only when the principal typing is unambiguous.

Frivolous use of Pred: to prohibit the overuse of the **Pred** rule, the coherence result applies to the coherence of typing judgements at *types*, as opposed to typing judgements at more general type schemes.

Type contexts must be i-unambiguous: the system is coherent only when the type context concerned is i-unambiguous. This allows the preceding lemma on the coherence of typings of overloaded identifiers to be applied at leaf nodes.

The proof of the coherence theorem makes use of the translation completeness result presented in the previous chapter.

Theorem 12 Coherence. *If Γ is a non-overlapping, i-unambiguous type context, $\Gamma \rhd e : \tau_0$ by two derivations Δ_1 and Δ_2 with translations $\lfloor\Delta_1\rfloor$ and $\lfloor\Delta_2\rfloor$ respectively, and the principal type of e under Γ is unambiguous, then $\lfloor\Delta_1\rfloor$ is equal to $\lfloor\Delta_2\rfloor$.*

Proof

The proof is by showing that meanings of Δ_1 and Δ_2 can both be constructed from the meaning of the derivation of the principal type computed by $\mathcal{O}$, and then showing there is only *one* meaning, at the correct type, that can be extracted from this principal typing.

Take $\mathcal{O}(\Gamma, e) = (S, \Pi, \tau)$ to define Δ. Notice that $S\Gamma = \Gamma$. Now, by the translation completeness result, if Π is of the form

$$\Pi = \{x_1 :_i \tau_1;\ \cdots;\ x_n :_i \tau_n\},$$

then there exists a substitution S'_1 of types for the generic variable of $\overline{\Gamma}(\Pi.\tau)$ giving

$$\frac{\Delta_1^1}{\Gamma \rhd x_1 : S'_1\tau_1} \quad \cdots \quad \frac{\Delta_n^1}{\Gamma \rhd x_n : S'_1\tau_n}$$

and $S'_1\tau = \tau_0$ such that

$$[\lfloor\Delta_1^1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n^1\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor = \lfloor\Delta_1\rfloor.$$

Similarly, there exists a substitution S'_2 of types for the generic variable of $\overline{\Gamma}(\Pi.\tau)$ giving

$$\frac{\Delta_1^2}{\Gamma \rhd x_1 : S'_2\tau_1} \quad \cdots \quad \frac{\Delta_n^2}{\Gamma \rhd x_n : S'_2\tau_n}$$

and $S'_2\tau = \tau_0$ such that

$$[\lfloor\Delta_1^2\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n^2\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor = \lfloor\Delta_2\rfloor.$$

Now since the principal type of e under Γ is unambiguous, there can be only one substitution S' such that $S'\tau = \tau_0$. That is, $S' = S'_1 = S'_2$.

By the lemma for the coherence of typings of overloaded identifiers, for which the conditions are satisfied, since $S'_1\tau_i = S'_2\tau_i$ it follows that $\lfloor\Delta_i^1\rfloor = \lfloor\Delta_i^2\rfloor$ for $i \in \{1, \ldots, n\}$. Therefore,

$$\begin{aligned} & \lfloor\Delta_1\rfloor \\ = \ & [\lfloor\Delta_1^1\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n^1\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor \\ = \ & [\lfloor\Delta_1^2\rfloor/\lfloor x_1 :_i \tau_1\rfloor;\ \cdots;\ \lfloor\Delta_n^2\rfloor/\lfloor x_n :_i \tau_n\rfloor]\lfloor\Delta\rfloor \\ = \ & \lfloor\Delta_2\rfloor \end{aligned}$$

as required. □

Chapter 9

Type classes in Haskell

Haskell is a recent functional programming language designed primarily for two purposes: to act as a unified basis for future research; and to be suitable for a wider range of programming applications than existing lazy functional programming languages. That is, Haskell is intended to be both a research-tool, and a "working" programming language.

In the main, Haskell adopts established programming features: such as the Hindley-Milner type systems, list comprehensions, and pattern matching—[HuWa90]. However, since a principal Haskell design goal was to produce a working programming language, some mechanism facilitating overloading was essential. No generally accepted solution, however, was available. The solution adopted by Haskell is that of type classes.

The OL type system was developed in parallel with the type class system: OL is small and appropriate for theoretical analysis; and type classes is its realisation in a full scale programming language. This chapter describes type classes with reference to OL illustrating the differences and providing some background to the design decisions adopted by Haskell. Some of the syntax used, for example that of types, is a hybrid between that of Haskell and that of OL.

A first significant difference between Haskell and OL is that whereas OL allows only declarations for use in a given term,

$$\texttt{let } id = \lambda x.x \texttt{ in } e$$

in Haskell, modules consist of a list of declarations. Further, the syntax of lambda abstractions is less verbose.

```
id x = x
```

This largely superficial difference can be viewed as similar to the difference between the Damas-Milner system and ML, the language for which that system was developed. Most such features of Haskell, which are independent of type classes, are glossed over throughout this chapter; the notation should, in the main, be self-explanatory.

9.1 Declaring type classes and instances thereof

A principal difference between the two approaches is that, whereas OL focuses on the overloading of individual operators, type classes group related operators together into a single `class`. For example, the numeric operators `+`, `*` and `negate` may be grouped in the class `Num` by the following declaration[1].

```
class Num a
   +      :: a -> a -> a
   *      :: a -> a -> a
   negate :: a -> a -> a
```

Type variables are denoted by small letters from the start of the Roman alphabet. The identifier `Num` is the name of the class and `+`, `*` and `negate` are the operators of the class. The type variable `a` is implicitly universally quantified over the entire class declaration. The term `Num a` is referred to as a *predicate*: it represents the same property as the binding predicates of Chapter 2 (for example, $(+) : \forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$).

Instances of type classes are declared with `instance` declarations. For example, `Int` and `Float` are declared to be members of the numeric class `Num` by the following two `instance` declarations (identifiers are assumed here, as they were in Chapter 2, to represent the implementation expression).

```
instance Num Int where
   +      = addInt
   *      = MultInt
   negate = addInt

instance Num IntFloat where
   +      = addFloat
   *      = MultFloat
   negate = addFloat
```

The first declaration declares `Int` to be a member of the class `Num`, that is, the predicate `Num Int` is true. Further, the implementations of `Num`'s operators at type `Int` are given to be `addInt`, `multInt` and `negInt`. The second `instance` declarations declares `Float` to be an instance of the numeric class giving the appropriate instances of the overloaded operators for floating point values.

Related operators are grouped together. Operators are grouped together in Haskell for two main reasons. Firstly, this allows groups of operators to be referred to by a single class name; programmers need not consider the individual overloaded operators but may focus on the abstract concept of a type being, or not being, a

[1]This is a simplification of the actual numeric class defined in the Haskell report.

member of a particular class. Secondly, grouping operators in this way frequently reduces the number of dictionary parameters introduced during the translation process.

The distinction between individual and grouped overloaded identifiers, however, is not as significant as one might think. Either approach can easily be implemented in the other. For example, classes can trivially model the single overloaded identifier approach by having only a single overloaded identifier in each class. The operators of the class **Num** above could equally have been separated as is shown below.

```
class NumPlus a where
   + :: a -> a -> a
class NumMult a where
   * :: a -> a -> a
class NumNeg a where
   negate :: a -> a -> a
```

Equally, operators can be grouped together in OL by declaring a single overloaded identifier to contain the values of a group of overloaded identifiers. Again, the *Num* class above could be declared as follows

```
( over Nums : ∀α.(α → α → α, α → α → α, α → α);
  let (+) = fst Nums in
  let (*) = snd Nums in
  let negate = thrd Nums in
  ...
)
```

in OL.

Classes are polymorphic in a *single* type variable. Type classes may be polymorphic only in a single type variable. For example, whereas it is possible in OL to declare an operator

```
over coerce : ∀α.∀β.α → β;
```

type classes in Haskell are syntactically restricted to refer to only a single polymorphic type variable. Hence, the coercion example above is not possible.

Haskell's instance declarations are further restricted to apply only to a single data type constructor. Thus, if the operator `==` is a member of the class `Eq`, then the declaration

```
instance Eq a => Eq [a] where
   (==) = eqList
```

is allowed (meaning [a] is an equality type whenever a is an equality type), but the declaration

```
instance Eq [Int] where
   (==) = eqListInt
```

is illegal. Each instance declaration may declare an instance for only, and exactly, a single data constructor. This restriction is enforced by Haskell's concrete syntax. By this restriction, Haskell ensures that no two declared instances overlap. As such, the system avoids certain incoherence problems (see Chapter 8). In the context of OL, it is convenient to factor out the issue of overlapping instances to maintain certain technical properties[2].

9.2 Type inference for Haskell type classes

Haskell's basic type inference algorithm is essentially the same as the informal algorithm presented in Chapter 2; it is discussed in detail in [HaBl89].

In OL, if an overloaded operator appears in a term, then a predicate is generated in the type. This predicate may be discharged if, and only if, there is an instance binding in the environment which satisfies it. If predicates are generated on the right-hand side of a definition and the predicates may not be discharged, then the definition itself becomes implicitly overloaded. When typing uses of the implicitly overloaded identifier, all associated predicates are introduced and the process continues.

Haskell's type classes adopt essentially the same approach. At each use of an overloaded operator from a class, a predicate is introduced into the typing. Predicates, in the Haskell context, are a class name applied to a type. For example, a use of the operator (+) receives the typing below.

```
+ :: Num a => a -> a -> a
```

The part before the => is a list of predicates, in Haskell terminology a *context*; the part after the => is the type.

If + is applied to a specific values, say the integers 3 and 4, then the typing is as follows.

```
3 + 4 :: Num Int => Int
```

At this point it is possible to discharge the predicate, as is the case with OL, since Int is a member of the class Num—as given by the previous instance declaration[3].

[2]Notably the substitution rule of Chapter 4.

[3]In Haskell, it is not possible to refer to a type of the form Num Int, such predicates are restricted to be applicable only to type variables.

```
3 + 4 :: Int
```

The final typing is thus `Int`.

Implicitly overloaded identifiers are also handled in a similar way to that in which they are handled in OL. The overloading in the definition of an identifier is carried over into its type. For example, the definition of `square` below

```
square x = x * x
square :: Num a => a -> a
```

where the type variable `a` is implicitly universally quantified. Such predicates are inserted in the typing at instances of implicitly overloaded identifiers and discharged if possible, or propagated.

```
squares x y z = (square x, square y, square z)
squares :: Num a, Numb, Num c => a -> b -> c -> (a,b,c)
```

This approach to typing is essentially the same as the $\mathcal{O}$ with $\mathcal{R}$ approach for OL described in Chapter 6.

9.3 Translation of Haskell type classes

As one might expect, Haskell's type classes are implemented by an optimised version of essentially the same translation mechanism that is described in Chapter 2 for OL[4]. A translation is applied to a typing derivation to yield a typing derivation in a Hindley/Milner based language; in the process, all type classes and overloading is replaced with explicit parametrisation.

There is however one slight difference resulting from the choice to group related operators together in classes. Whereas in OL it suffices to insert the the implementation identifier directly, in Haskell a selection operator is required to extract the correct operator from the class.

Every instance declaration translates to the construction of a *dictionary.* A dictionary is a tuple containing an implementation of each of the operations in the class. For example, the instance declarations at types `Int` and `Float` in the example above translate to the following dictionary definitions.

```
NumDInt = (addInt,multInt,negInt)
NumDFloat = (addFloat,multFloat,negFloat)
```

[4]This is the case for the current Glasgow implementation. How type classes are implemented is not defined in the language definition. Other, or subsequent, implementations may adopt different approaches.

When an operator, such as * is applied to an argument of a known type a selector function must be used to extract the appropriate operator definition from the dictionary. Thus, the translation of `3*4` is

```
(snd NumDInt) 3 4
```

where `NumDInt` is the dictionary of numeric operators at the type `Int`, and `snd` selects the second element, that is, the multiplication operator. For implicitly overloaded identifiers the situation is similar. An identifier is chosen to represent the as yet unknown dictionary and parametrised upon. Thus, the definition of `square` above translates as follows.

```
square NumDa x = (snd NumDa) x x
```

9.4 Super-classes in Haskell

Chapter 2 mentioned, in passing, the possibility of allowing predicates to appear in the signature of overloaded operators; the equivalent feature is included in Haskell.

A `class` declaration may be of the form

```
class C1 a, ..., Cn a => C a where
   ...
```

indicating the static requirement that, for a type to be declared a member of class `C`, it must be the case that the type is already a member of classes `C1` through `Cn`.

This facility is used extensively in the definition of the predefined classes in Haskell: there are several numeric classes in Haskell, these are organised into a hierarchy of sub- and super-classes. The relational operators, such as `==` and `<=`, are also related in this way.

9.5 But! Haskell is a working programming language

The impression given thus far is that the differences between Haskell and OL are merely syntactic sugar. A principal design goal for Haskell, however, is that it should be a real programming language: it should be applicable to large applications which may be maintained over a significant period of time. Thus, many issues have arisen in the design of Haskell which must be addressed in order to make type classes as unobtrusive as possible. This section discusses these problems and presents Haskell's solutions.

What to do with ambiguous typings. As one might expect following the discussion of ambiguity in Section 8, ambiguity poses a significant problem in Haskell. To recap, an ambiguous typing is one of the form

```
e :: ..., ClassName a, ... => t
```

where the type variable a does not appear in the type t or in the type environment. As such, unification cannot resolve the overloading.

Ambiguous typings can arise in several ways, principally when overloaded data is discarded. The sources of ambiguity in Haskell are essentially the same as those in OL.

If ambiguity were not of semantic importance the solution would be trivial: one could merely discharge the offending predicates with arbitrary instance. As seen in Chapter 8, however, this is not the case. An obvious alternative is to declare ambiguous typings illegal. As well as contravening the principal type theorem, this possibility was deemed impracticable on the grounds that, particularly in the context of overloaded numeric literals, there are too many instances of ambiguity in most programs.

Haskell adopts a relatively *ad hoc* but deterministic solution to the problem. *Default declarations* are used to specify the instance which should be used to discharge predicates in ambiguous typings[5]. A default declaration takes the form

```
default (t1, ..., tn)
```

and only one such declaration may appear in each module. When the compiler encounters definitions which are ambiguous in a numeric class, the offending predicates are discharged at the first type t such that *all* the the classes of the offending variable have instances at type t. This guarantees that, for "reasonable" instances of the Num class, changing default will not change the answer (modulo overflow and round off).

As mentioned above, this approach is clearly *ad hoc* though it at least provides a deterministic and usable solution to the problem. It is hoped that the second version of Haskell will incorporate a better solution.

Efficiency problems. One feature of the presentation of type classes in both OL and in Haskell is the implementation technique: typing derivations are translated, during this process implicitly overloaded definitions are replaced by explicitly parametrised definitions. At each use of an implicitly overloaded identifier, an actual dictionary containing the required overloaded operator definitions is passed as an argument.

[5]Notice that this solution does not contravene the principal type theorem. This, however, is mis-leading since defaults render ambiguities *incoherent*, though in a deterministic way.

One must pay a run-time penalty for this parametrisation. This cost can be significant: early Haskell implementations were slowed down by a factor of ten due to translator introduced parametrisation. The most recent version of the compiler is slowed down by a factor of four to eight in general. This issue is discussed more fully in [HaBl89].

The monomorphism restriction. There is, however, a rather more subtle and potentially dangerous loss of efficiency associated with type classes. Consider the following examples as they are treated by a language with a Hindley-Milner type discipline and no type classes.

```
globalValue = <expensive_to_compute>
f x y = (a,b) where
        tempValue  = g x y
        a = h tempValue x
        b = h tempValue y
```

In the first case, `globalValue` is expensive to compute; graph re-writing, however, ensures it is computed at most once. In the second case, `tempValue` is used to factor out a potentially expensive common subexpression; as such, it is also computed at most once. Specifically, since both the identifiers represent values, as opposed to functions, we can ensure they are evaluated at most once.

The parametrisation introduced by type classes potentially causes `globalValue` and `tempValue` to be computed many times. If both `globalValue` and `tempValue` are implicitly overloaded, then their translations become parametrised and they no longer represent values; they become functions which may be computed many times at the same argument.

```
globalValue Da = <expensive_to_compute>
f Da x y = (a,b) where
        tempValue Da  = g Da x y
        a = h (tempValue Da) x
        b = h (tempValue Da) y
```

Type classes offer extra power in this case (*values* may be overloaded), but only at the expense of the efficiency of a program being highly unpredictable in practice. This was considered unacceptable for Haksell.

The solution adopted by Haskell is to disallow polymorphism in certain places which would potentially suffer from this efficiency problem[6]. In particular, only identifiers bound directly to lambda expression may be polymorphic. Since `globalValue` and `tempValue` are not bound directly to a lambda term they may not be polymorphic. As such they are monomorphic and may be instantiated only to a single type. Being

[6]An alternative would be to regain much of the lost efficiency by the use of memo functions. This, however, requires sophisticated compiler technology to achieve anticipated efficiency levels.

instantiated to at most one type implies they are evaluated at most once. Notice that this does *not* imply that the function f above is monomorphic. It is bound directly to a lambda term and so may be polymorphic. The variable `tempValue` is restricted to be monomorphic only within each instantiation of `f`.

Derived instances. One of the most frequently used overloaded operators is, unsurprisingly, the equality operator ==. For current purposes, it suffices to assume == is declared by a class declaration of the form below.

```
class Eq a where
   == :: a -> a -> a Bool
```

Functions such as **member**, the membership function on lists, are implicitly overloaded at type `Eq a => [a] -> a -> Bool`. When new abstract data types are declared, a new instance of `Eq` must be given to render == and **member** applicable to the new type. This process can be tedious for two reasons: firstly, there are several classes in addition to `Eq` of which the type will almost certainly have to be an instance; and secondly, because the structure of the instance declarations frequently follows the same pattern based on the structure of the type.

Haskell provides a mechanism for constructing *derived instances* of particular prelude classes, thus avoiding the need for the programmer to write the instance declarations required. Naturally, the programmer may specify for which classes, if any, instances are to be included. For example, given the data type declaration

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Eq
```

an instance declaration for type `Tree` and class `Eq` is included based on the structure of the type `Tree`. That instance declaration is of the form below.

```
instance Eq a => Eq (Tree a) where
   Leaf m == Leaf n          = (m == n)
   Tree m1 n1 == Tree m2 n3 = (m1 == m2) && (n1 == n2)
   _ == _                    = False
```

The programmer could alternatively provide a non-structural definition of equality; this may be required to maintain data abstraction.

Chapter 10

Review

The OL language and type system represent an approach to overloading within a Damas-Milner style programming language. This chapter reviews OL outlining the advantages and weaknesses of the system.

Though closely related, the OL system and Haskell's type class mechanism were developed for different purposes. This thesis presents a minimal formal language, type system and inference algorithm with particular emphasis on technical considerations. The design of type classes, however, is motivated by the need to produce a general purpose, unobtrusive and usable overloading mechanism for Haskell. Hence OL and Haskell solve different aspects of the same question.

The Haskell type class mechanism has been commented upon and discussed at length in papers and over electronic mail. There is little need to add anything here. Some discussion is due, however, of the OL system.

10.1 The OL language

OL is a small and remarkably unwieldy language! It is not intended for programming real applications. In addition to the lambda calculus with `let`, OL allows overloaded operators and instances thereof to be declared and used. Several simple OL programming examples are given in Chapter 2.

10.1.1 Declarations and expressions

The dichotomy between declarations and expression terms ensures the existence of principal types. Consider a program (d, e) in which d declares the overloading of $(+)$ and two instances thereof, say *Int* and *Float*. Consider also e to be implicitly overloaded, say the body of the *double* function discussed in Chapter 2. Now the principal type of e under the type context generated by d is

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

which has instances at type $Int \rightarrow Int \rightarrow Int$ and $Float \rightarrow Float \rightarrow Float$ (since there are instance bindings at those types). If this typing were to propagate outwith the scope of the corresponding **over** declaration, then the two instance typings would be valid but the principal typing would no longer hold.

The example above concerns **over** declarations, a similar problem arises for **inst** declarations. If a typing is allowed to propagate such that an instance binding is no longer in scope, then a typings exists utilising that instance binding, and a principal typing as computed exists, but the principal typing is not related to the specific typing.

In Haskell, which is discussed in Section 9, this issue is accommodated by an unusual scoping condition which ensures that all classes and instances thereof are exported in such a way that predicates may not leave the scope of their corresponding overloading bindings.

10.1.2 Coercions

OL is presented throughout as an approach to overloading, it may also be used to describe coercions. Every overloaded operator must be bound to a signature. This signature, however, serves only to limit the types of instances of the identifier in question. For example, the overloaded operator (+) is assigned the signature $(+) :_o \forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ restricting instances to map two objects of a given type to another object of the same type. Thus, (+) may be defined at types

$$Int \rightarrow Int \rightarrow Int \qquad \text{and}$$
$$Float \rightarrow Float \rightarrow Float,$$

but not at type

$$Int \rightarrow Float \rightarrow Float$$

which does not match the given signature. Many languages facilitate such typings with the use of coercions.

Below, two approaches are given illustrating how such typings may be achieved in OL. Since both are simply different ways of approaching the same problem, it should come as little surprise that each exhibits the same disadvantages. Specifically, both approaches generate a large number of ambiguous typings.

Generalise type signatures. One approach is to assign (+) a more general signature; both the following seem reasonable:

$$(+) :_o \forall\alpha.\forall\beta.\forall\gamma.\alpha \rightarrow \beta \rightarrow \gamma \quad \text{or}$$
$$(+) :_o \forall\alpha.\alpha.$$

The former maintains the restriction that $(+)$ is a binary operation; and the latter, more radically, asserts nothing about the form of the instance types[1]. Both these signatures allow the declaration of the more interesting instances of $(+)$. Specifically, instances such as

$$(+) :_i \mathit{Int} \to \mathit{Float} \to \mathit{Float}$$

are now valid. Terms such as $(10 + 3.14)$ may now be typed and evaluated giving 13.14, as one expects.

Under this approach, however, too many terms would be overloaded and a number of these may also be ambiguous. For example, the principal type of the term above would be

$$(10 + 3.14) : \forall\gamma.((+) : \mathit{Int} \to \mathit{Float} \to \gamma).\gamma$$

which is overloaded. Some mechanism, perhaps similar to Haskell's default mechanism, would be required.

Overloaded coercions. The second approach is, in some ways, more elegant. The first step is to declare an overloaded low level addition operation $(+')$ and an overloaded coercion function.

$$\begin{array}{ll} (+') & :_o \forall\alpha.\alpha \to \alpha \to \alpha \\ \mathit{coerce} & :_o \forall\alpha.\forall\beta.\alpha \to \beta \end{array}$$

Instances of $(+')$ may declared as previously at types *Int* and *Float*; and the *coerce* operator may receive the following instances.

$$\begin{array}{l} \mathit{coerce} :_i \forall\alpha.\alpha \to \alpha \\ \mathit{coerce} :_i \mathit{Int} \to \mathit{Float} \end{array}$$

Notice that these instances do not overlap.

Now, if the real addition operator $(+)$, that which intended to be used by a programmer, is declared by the following implicitly overloaded declaration,

$$\texttt{let}\ (+) = \lambda x.\ \lambda y.\ (\mathit{corce}\ x) +' (\mathit{corce}\ y)\ \texttt{in}\ \cdots$$

then the principal typing of $(+)$ is

$$\begin{array}{l} (+) : \forall\alpha.\forall\beta.\forall\gamma.(\mathit{coerce} : \alpha \to \gamma). \\ \qquad\qquad (\mathit{coerce} : \beta \to \gamma). \\ \qquad\qquad ((+') : \gamma \to \gamma \to \gamma).\alpha \to \beta \to \gamma \end{array}$$

[1] A system in which all overloaded operators had the fully general signature $\forall\alpha.\alpha$ may be interesting in its own right. In such a system, predicates record all, and no more than, the typing information implicit in the term.

which has instances at all the general types required of $(+)$.

Now consider the term $(6 + 4 + 3.14)$. There is a choice in the typing of such a term. Should the 6 and 4 be added as integers then coerced to be added to the 3.14; or should the 6 and the 4 be coerced immediately to floating point values and all addition be done between floating point numbers?

Reynolds, in [Rey80], addresses this issue suggesting a coherence property should be required. Specifically, Reynolds suggests the coercions should be constructed such that the choice of where to apply coercions is irrelevant—all choices lead to the same result.

Under the type context above, the term $(6 + 4 + 3.14)$ has an ambiguous principal typing: hence the coherence result of Chapter 8 is not applicable. Further, as with the previous implementation of coercions, the above approach leads to an unacceptable number of unexpected overloadings and ambiguities.

10.2 The OL type system

The principal motivation for the current thesis is the examination of technical aspects of the OL type system. This section reviews some of the issues raised by the presentation.

The need for both $\mathcal{O}$ and $\mathcal{R}$

Chapter 2 presents an informal type inference algorithm for OL extending the Damas-Milner inference algorithm. The current Glasgow Haskell compiler employs a similar algorithm. This informal algorithm, however, does not compute principal types. The formal inference algorithm $\mathcal{O}$, on the other hand, does compute principal types and algorithm $\mathcal{R}$ is required to unify the two approaches.

This dichotomy is unfortunate and inelegant. The current author, however, is not aware of a better approach.

Volpano and Smith [VoSm91] have, more recently, investigated the complexity of the original [WaBl89] system. They identify and solve the same problems addressed and solved by Chapter 6 herein. Where the OL system restricts type contexts to be strictly decreasing, the [VoSm91] system requires that they be parametrically recursive. These properties are similar.

Coherence

The other principal inelegance in the system is that restrictions are required to render it coherent; an implicitly coherent system would be more satisfactory.

Chapter 8 discusses the relationship between ambiguity and coherence: An ambiguous typing may, in general, indicate the possibility of incoherence. Short of

outlawing ambiguities explicitly (an expensive restriction in terms of expressibility), the current author is not aware of a way of avoiding this problem.

The two systems most closely related to OL are that of Kaes [Kae88] and Nipkow and Snelting [NS91]. Though neither of these papers address the issue directly, both these systems exhibit the same behaviour with respect to ambiguity and coherence.

The translation based implementation strategy

Meaning is assigned to OL terms, or more specifically OL typing derivations, by means of a derivation to derivation translation scheme. Predicates in typings are mapped to parametrisation in the translation. Used naively, this approach is implicitly inefficient. Related systems have used other semantic approaches.

A rule based static semantics of Haskell is given in [PJWa90]. Dictionary parameters are used to propagate overloading. Haskell's super-class relationships are modelled explicitly within the implementation: specifically, the dictionary for a super-class is incorporated as part of the dictionary for the sub-class. As such, less dictionary parameters are required.

A categorical semantics of Haskell's type classes is given by Hilken and Rydeheard in [HiRy91]. Therein, type classes are related to categorical schemata.

The separation of declarations and programs

Declarations of overloaded operators and instances thereof are separated syntactically from the terms in which they are used. This separation ensures the system satisfies a principal type theorem. Section 10.1.1 motivates this distinction and Nipkow and Snelting [NS91] adopt a similar approach.

10.3 Operator hierarchies in OL

A feature of early versions of the OL system [WaBl89] and Haskell's type classes are super-class hierarchies. Other related work, for example [PJWa90] and [NS91] incorporate this feature. For technical simplicity, however, the current system syntactically excludes such relationships.

Sub-classes serve two purposes in Haskell: they provide a way of specifying static restrictions on admissable instance judgements, and they facilitate a more efficient implementation technique. The OL system may be extended relatively simply to incorporate static super-class requirements. It is inappropriate to consider the efficiency issue here as that is an implementation question.

Consider firstly the meaning of a declaration of the form

$$\texttt{over}\ x : \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau;$$

Whenever an instance is declared at some specific type, each predicate in the operators signature imposes a condition on instances that a particular operator have a particular instance in scope. Notice two things: firstly, that this is an entirely static requirement; and secondly, for any instance declaration of x at type σ', this requirement is implied by the relationship

$$\Gamma \rhd \forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau \geq \sigma$$

which, in turn, is required by the validity condition on type contexts. Thus, simply allowing predicates to appear in overloaded operator's signatures imposes the required relationship between instances declarations.

A problem arises, however, when one considers applications of the **Pred** typing rule. Inconvenient dependencies arise between instance bindings and predicates. Given the type context Γ

$$x :_o \forall\alpha.\alpha;$$
$$y :_o \forall\alpha.(x : \alpha).\alpha;$$

the term y has principal type $\forall\alpha.(x : \alpha).(y : \alpha).\alpha$. Further, the ordering of the predicates is significant. This is an implication of the requirement that all the type contexts in a derivation are valid.

However, it is reasonable to consider the sup- and super-class relationships between overloaded operators to be an entirely static requirement. The following changes augment the OL language and type system to handle super-classes while maintaining all the results presented in this thesis.

- Define $<\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau>$ to be $\forall\alpha_1 \cdots \forall\alpha_n.\tau$.
- Define $<\Gamma>$ to be Γ with all bindings of the form $x :_o \sigma$ replaced with bindings of the form $x :_o <\sigma>$.
- Allow types of the form $\forall\alpha_1 \cdots \forall\alpha_n.(x_1 : \tau_1).\cdots.(x_m : \tau_m).\tau$ to appear in `over` declarations.
- Replace the typing rule for instance declarations in Figure 3.7 with the following.

$$\textbf{Inst} \quad \frac{<\Gamma>\rhd e : \sigma \quad \Gamma;\ x :_i \sigma \rhd d \rightsquigarrow \Gamma'}{\Gamma \rhd \texttt{inst}\ x : \sigma = e;\ d \rightsquigarrow x :_i \sigma;\ \Gamma'}\ fv(\sigma) = \{\}$$

- Replace the typing rule for programs in Figure 3.7 with the following.

$$\textbf{Prog} \quad \frac{\Gamma_0 \rhd d \rightsquigarrow \Gamma \quad \Gamma_0;\ <\Gamma>\rhd e : \sigma}{\Gamma_0 \rhd (d, e) : \sigma}\ \forall x.\forall\sigma'.(x :_o \sigma' \notin \Gamma_0)$$

That is, all typing judgements are made from typing contexts without super-class information. The validity requirement on type contexts at the top level, however, enforces the appropriate relationships statically.

Bibliography

Bibliography entries marked with an asterisk, for example [Bar81*], have not been studied by the author. They are included primarily for completeness.

[Bac78] J. Backus, Can programming be liberated form the von Neumann style? A functional style and its algebra of programs. Communications of the ACM, 21(8), pp. 613-641, August 1978.

[Bar81*] H. P. Barendregt, The lambda calculus, its syntax and semantics, North-Holland 1981.

[BCGS88] V. Breazu, T. Coquand, C. A. Gunter, and S. Scedrov. *Inheritance and explicit coercion (preliminary report)*, V. Breazu, Department of Computer and Information Sciences, University of Pensylvania, Philidelphia, PA 19104, USA. October 1988.

[Car84a] Luca Cardelli, Basic polymorphic type-checking, Computing Science tech. report 119, AT and T Bell laboratories, Murray Hill, NJ, 1984.

[Car84b] Luca Cardelli, A semantics of multiple inheritance. In *Semantics of data types,* LNCS 173, Springer-Verlag, New York, pp. 51-67.

[Car88] Luca Cardelli, Structural sub-typing and the notion of power type, In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Car89a] Luca Cardelli, *A Quest Preview.* DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301. (Circulated at POPL 89, Austin Texas).

[Car89b] Luca Cardelli, *Typeful programming.* DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301.

[Chu41*] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, NJ, 1941.

[CW85] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4, December 1985.

[CM89] Luca Cardelli and John C. Mitchell, Operations on records. Digital Equipment Corporation technical report 48, Paolo Alto, August 1989.

[Cle86] Dominique Clèment *et. al.*, A simple applicative language: Mini-ML. In *Proceedings of the 13'th Annual Symposium on Principles of Programming Languages*, January 1986. This reference is wrong!

[CuGh89] Pierre-Louis Curien and Giorio Ghelli, Coherence of subsumption, Dipartimento di Informaticà, Universita di Pisa, Corso, Italia 40, 1989.

[DaMi82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, NM., January 1982.

[Dam85] L. Damas, Type assignment in programming languages, Thesis, University of Edinburgh, 1985.

[Fai85] Jon Fairbairn, *Design and implementation of a simple typed language based on the lambda calculus.* Ph.D. Thesis, University of Cambridge, May 1985. Available as Computer Laboratory technical report No. 75.

[FGJM84] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer, Principles of OBJ2. In *Conference of the twelfth annual ACM Symposium on the Principles of Programming Languages*, pp. 52-66, ACM, 1984.

[FW81] William Findlay and David A. Watt, PASCAL: an introduction to methodological programming (2nd edition). Pitman International Press, 1981.

[GMM78] M, Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, A meta-language for interactive proof in LCF. In *Conference of the fifth annual ACM Symposium on the Principles of Programming Languages*, pp. 119-130, ACM, 1978.

[Gir72*] J-Y. Girard, "Interpretation functionelle et elimination des coupers de l'arithmetic d'ordre superior", Ph.D. Theses, University of Paris VII, 2, Place Jussieu, Paris.

[Han87] Peter Hancock, Chapter 9 of [Pey87].

[HaBl89] Kevin Hammond and Stephen Blott, Type inference for type classes in Haskell, in *Proceedings of the 2nd Glasgow FP group workshop*, Fraserburgh, Scotland, (Springer-Verlag) 1989.

[HMM86] R, Harper, D. MacQueen, and R. Milner, Standard ML, Report ECS-LFCS-86-2, Edinburgh University, Computer Science Dept., 1986

[HiRy91] Barney Hilken and David Rydeheard, Towards a categorical semantics of type classes, Dept. of Computer Science, University of Manchester, Oxford Road, Manchester, England, 1991.

[HS86] R. Hindley and J. P. Seldin, Introduction to Combinators and λ-calculus, London Mathematical Society Student Texts 1, Cambridge University Press, 1986 (reprinted 1988).

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. In *Trans. Am. Math. Soc. 146*, pp. 29–60, December 1969.

[HuWa90] Paul Hudak, and Phil Wadler (editors), Report on the programming language Haskell, a non-strict purely functional language. Technical report, Dept. of Computing Science, University of Glasgow, Scotland, and Yale University, Department of Computer Science, April 1990.

[JM88] Lalita A. Jategaonkar and John C. Mitchell, ML with extended pattern matching and sub-types, In *ACM Symposium on Lisp and functional programming*, 1988.

[Jat89] Lalita A. Jategaonkar, ML with extended pattern matching and subtypes. M.Sc. thesis, Department of Electrical Engineering and Computer Science, MIT, Boston, 1989.

[Kae88] S. Kaes, Parametric polymorphism. In *Proceedings of the 2'nd European Symposium on Programming: LNCS 300*, Nancy, France, March 1988.

[Lan66] P. J. Landin, The next 700 programming languages, Communications of the ACM, 9(3), pp. 157-166, 1966.

[Lil91] Mark Lillibridge, Series of notes on Haskel mailing list, 1990-1991.

[Mac84] D. B. MacQueen, Modules for Srandard ML. In *Proceedings of the symposium on Lisp and functional programming*, ACM, Austin, Texas, pp. 198-207, 1984.

[MPS84] D. MacQueen, G. Plotkin and R. Sethi, An *ideal* model for recursive polymorphic types. In *Proceedings of the 11'th Annual Symposium on Principles of Programming Languages*, ACM, 1984.

[McC78*] J. McCarthy, History of Lisp, In *Preprints of Proceedings of ACM SIGPLAN History of Programming Languages Conference*, pages 217-223,1978. Published as SIGPLAN notices 13(8), August 1978.

[MH88] John C. Mitchell and Robert Harper, The essence of ML. In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Mit84] John C. Mitchell, Coercion and type inference (summary), In *Proceedings of the 11'th Annual Symposium on Principles of Programming Languages*, pp. 175-185, January 1984.

[Mil78] R. Milner, A theory of type polymorphism in programming. In *J. Comput. Syst. Sci. 17*, pp. 348–375, 1978.

[Mil84*] R. Milner, A proposal for Standard ML. In *ACM Symposium on Lisp and functional programming*, Austin, Texas, 1984.

[MP85] John C. Mitchell and Gordon D. Plotkin, Abstract types have existential type. In *Proceedings of the 12'th Annual Symposium on Principles of Programming Languages*, New Orleans, La, January 1985.

[NS91] Tobias Nipkow and Gregor Snelting, Type classes and overloading resolution via order-sorted unification. *Functional programming and computer architecture*, 1991.

[Pey87] Simon Peyton-Jones, The implementation of functional programming languages, Prentice-Hall International, Englewood Ciffs, NJ, 1987.

[PJWa90] Phil Wadler and Simon Peyton Jones, A static semantics for Haskell, Dept. Computing Science, University of Glasgow, Glasgow, Scotland, 1990.

[Rem89] Didier Rémy, Type-checking records and variants in a natural extension of ML. In *Proceedings of the 16'th Annual Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[Rey74] J. C. Reynolds, Towards a theory of type structure, SLNCS 19, 1974, pp. 408-4025.

[Rey80] J. C. Reynolds, Using Category theory to design implicit conversions and generic operators. In *Semantics-directed compiler generation*, LNCS 94, 1980.

[Rey85] J. C. Reynolds, Three approaches to type structure, Springer-Verlag LNCS 185, 1985, pp. 97-138.

[Rob65*] J. A. Robinson, A machine orientated logic based on the resolution principal. JACM 12, 1, pp. 23-41, 1965.

[Sta88*] Ryan Stansifer, Type inference with sub-types. In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Str67*] Christopher Strachey, *Fundamental concepts in programming languages.* Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Str86] Bjarne Stroustrup, *The C++ Programming language.* Addison Wesley, 1986.

[Tof88] Mads Tofte, *Type inference for polymorphic references*, LFCS, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1988.

[Tur76*] D. A. Turner, SASL language manual, Technical report, University of St. Andrews, 1976.

[Tur85] D. A. Turner, Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pp. 1-16, Springer-Verlag LNCS 201, September 1985.

[VoSm91] Dennis Volpano and Geaffrey Smith, On the Complexity of ML typability with overloading, Dept. of Computer Science, Cornell University, Ithaca, New York14853, USA, 1991.

[WaBl89] Phil Wadler and Stephen Blott, How to make *ad-hoc* polymorphism less ad hoc. In *Proceedings of the 16'th Annual Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[Wan87] Mitch Wand, Complete type inference for simple objects. *In proceedings of the 2nd IEEE symposium on logic in computer science*, pp. 37-44, 1987.

[Wan89] Mitch Wand, Complete type inference for simple objects. In *Proceedings of the fourth annual symposium on Logic in Computer Science*, June 1989. See also *errata* in LICS 1990.

[Wir71*] N. Wirth, *The programming language Pascal*, Acta Informatica, 1, 1, 1971, pp. 35-63.