Records, variants and qualified types

Benedict R. Gaster

Technical report NOTTCS-TR-98-3

Thesis submitted to the University of Nottingham for the degree of
Doctor of Philosophy
July 1998

Contents

List of Figures

Abstract

Records and variants provide flexible ways to construct datatypes, but the restrictions imposed by type systems can prevent them from being used in flexible ways. For example, typed languages often prohibit basic operations such as adding fields to a record, and may not allow access to individual components unless the type, and hence the run-time representation of the record, is fixed at compile-time. These limitations are often the result of concerns about efficiency, or of the inability to express accurately the types of key operations.

This dissertation studies type systems that remedy these problems, supporting extensible records and variants, with a full complement of polymorphic operations on each. The systems are based on the theory of qualified types, from which they inherit a simple compilation method and effective type inference for the implicitly typed versions.

Qualified types is a general theory for constrained polymorphism, whose semantics (implicit) has previously been defined in terms of a translation into the second order polymorphic λ-calculus, supported by a statement of coherence. This dissertation describes two alternative (explicit) semantics for qualified types. The first describes the meaning of an overloaded expression via specialization, while the second interprets expressions in polynomial categories.

Acknowledgements

Many people have provided help during my time at the University of Nottingham, in particular with the development of this dissertation. First and foremost I must thank my supervisor Mark P. Jones, who far surpassed his position as supervisor and mentor. Mark's interest in programming language research, without restricting his focus to a single point within the field, has provided me with a unique opportunity, for which I will be forever grateful. Thanks are also due to all the members of the Languages and Programming Group at the University of Nottingham; our lunch time discussions will be sorely missed. Particular thanks must go to Graham Hutton, who introduced me to category theory—a subject I shall never be without—and to Colin Taylor, who started in the same office, as myself, on the same day. Thanks are also due to Paul Blampied, Tony Danniels, and Claus Reinke who have provided useful discussions on category theory and problems with understanding LaTeX. Particular thanks should go to my examiners, Simon Peyton Jones and Graham Hutton, who provided many useful comments and suggestions about this work, which helped to make it better as a result.

This work would not have been possible without the support of my family and friends. In particular, my mother and father have provided financial support, which has enabled me to travel to workshops and conferences around the world. My close friends—you know who you are—have provided a place to escape when it sometimes got too much!

This work was supported in part by an EPSRC studentship 9530 6293. Finally, the commutative diagrams were produced using Paul Taylor's macros [Tay90].

Chapter 1

Introduction

1.1 Types and type systems

In most programming languages, a system of *types* is used to distinguish between different types of values. Types are "checked" in some way, either statically (during program compilation) or dynamically (during program execution). In compile-time type checking, the system ensures that each sub-expression of a program defines an element of a specific type. This may be specified explicitly as part of the program text (function parameters in C [KR88], for example), or inferred implicitly from the way the program syntax is used.

The following properties are some of the advantages of checking type correctness statically:

- The static detection of errors. A program that tries to add an integer to a string, can be detected, and rejected, by a compile-time type checker before the program is executed. Thus compile-time type checking provides an analysis for reducing the number of possible errors that can occur during program execution.

- The validity of optimizations. Consider a data structure such as a record whose layout (and therefore selection function) depends on the type of fields it contains. If the type and, therefore, the size of the fields are known at compile-time, then pointer arithmetic can be used to compile a more efficient selection function.

- Documentation. It is generally considered good programming practice to document a function definition with its corresponding type. For example, it is often the case that each top level definition in a Haskell[PH97] program will be supplied with a suitable type. In the case of explicitly type languages (e.g., Java [AG96]) the programmer must supply a type for each field and method within a class definition.

For the work in this dissertation we will, in general, be concerned with programs that can be checked statically for type correctness. We shall pay particular attention to two different kinds of polymorphism, noted by Strachey [Str67], suited to programming. The first, *parametric* polymorphism, captures the fact that certain values behave independently of type—a function, for example, to reverse a list is unconcerned with the type of elements contained within that list. The second, or *constrained* polymorphism, captures the notion of overloading. For example, an equality operator (==) is often defined for more than one type. Furthermore, we shall often consider general algorithms, similar in spirit to those described by Damas and Milner [Mil78, DM82, Dam85], for inferring the types of implicitly typed expressions.

1.2 Qualified types

Qualified types, as described by Jones [Jon92b, Jon94b], is a general theory of constrained polymorphism, and comes with a natural generalization of Damas and Milner style type inference [Mil78, Dam85]. For example, a partial ordering operator $\sqsubseteq$ can be assigned the type

$$\sqsubseteq : \forall \alpha . \mathit{PartialOrd}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathit{Bool},$$

where the predicate $\mathit{PartialOrd}\ \alpha$ constrains the instantiation of α to types with a partial ordering.

To define the semantics of qualified types, Jones provides a translation for overloaded terms into a variant of the second order polymorphic lambda calculus [Gir72, Rey74], which is an explicitly typed lambda calculus with type abstraction. Unfortunately, the mathematical structures required to model expressions of the polymorphic lambda calculus are often more complicated than required to understand applications involving qualified types, and consequently are too strong a requirement. Furthermore, to prove correctness of specific applications, of qualified types, one must be able to show correctness of overloaded operators within the models of the polymorphic lambda calculus. This is often a very difficult task.

1.3 Records and variants

Products and sums play fundamental roles in the construction of datatypes—products describe grouping of data items, while sums express choices between alternatives. For example, we might represent a date as a product, with three integer components specifying the day, month, and year

$$\mathit{Date} \quad = \quad \mathit{Int} \times \mathit{Int} \times \mathit{Int}.$$

For a simple example of a sum, consider the type of input events to a window system, with one alternative indicating that a character has been entered on the keyboard, and another indicating a mouse click at a particular point on the screen

$$\mathit{Event} \quad = \quad \mathit{Char} + \mathit{Point}.$$

These definitions are adequate, but they are not particularly easy to work with in practice. For example, it is easy to confuse datatype components when they are accessed by their position within a product or sum, and programs written in this way can be hard to modify and extend.

To avoid these problems, many programming languages allow the components of products, and the alternatives of sums, to be identified using names drawn from some given set of *labels*. Labelled products are more commonly known as *records* or *structs*, while labelled sums are better known as *variants* or *unions*. For example, the *Date* and *Event* datatypes described above might be defined more attractively as

$$\begin{array}{lcl} \mathit{Date} & = & \mathit{Rec}\ \{\!|\mathit{day}{:}\mathit{Int},\ \mathit{month}{:}\mathit{Int},\ \mathit{year}{:}\mathit{Int}|\!\} \\ \mathit{Event} & = & \mathit{Var}\ \{\!|\mathit{key}{:}\mathit{Char},\ \mathit{mouse}{:}\mathit{Point}|\!\}. \end{array}$$

This notation captures a common feature in the construction of record and variant types, using *rows* of the form $\{\!|l_1{:}\tau_1, \ldots, l_n{:}\tau_n|\!\}$ to describe mappings that associate a type τ_i with each of the (distinct) labels l_i. Record types are obtained by preceding rows with the symbol *Rec*. Variant types are constructed using *Var*. For example, if $r = \{\!|l_1{:}\tau_1, \ldots, l_n{:}\tau_n|\!\}$ and $e_1,\ldots,e_n$ have types $\tau_1,\ldots,\tau_n$, respectively, then we can form a record $(l_1 = e_1, \ldots, l_n = e_n)$ of type $\mathit{Rec}\ r$, or distinct variants,

$\langle l_1 = e_1\rangle, \ldots, \langle l_n = e_n\rangle$, each of type *Var r*. Thus, $(day = 17, month = 1, year = 1942)$ and $\langle key = 'a'\rangle$ represent values of type *Date* and *Event*, respectively.

Unfortunately, practical languages are often less flexible in the operations that they provide to manipulate records and variants. For example, many languages—from C to Standard ML (SML) [MTH90, MTH97]—will only allow the programmer to select the l component, $r.l$, from a record r if the type of r is uniquely determined at compile-time[1]. These languages do not support *polymorphic* operations on records—such as a general selector function $(\_.l)$ that will extract a value from *any* record that has an l field. A further weakness in many of these languages is that they provide no real support for *extensibility*; there are no general operators for adding a field, removing a field, renaming a field, or replacing a field (possibly with a value of a different type) in a record value.

1.4 This dissertation

This dissertation addresses both the theoretical and practical issues of type systems for records and variants. A complete formal foundation for the different systems is developed, while also considering more pratical concerns, such as the problems of making the types and operations fit in to existing general purpose functional programming languages.

The major contributions of this dissertation are

Core type system: The type system described in Chapter 3 supports

- extensible records and variants;
- a full complement of polymorphic operations;
- effective type inference; and
- simple efficient implementation.

The system of records and variants, and its extensions, described in this dissertation combines many of the ideas that have been used in previous work into a

[1] In implementations using boxed representations for values, only the set of labels in r is needed; the actual component types are not required.

practical type system for implicitly typed languages like SML and Haskell. In particular, it supports polymorphism and extensibility, records and variants, type inference, and compilation. The type system is an application of qualified types, extended to deal with a general concept of rows. Positive information about the fields in a given row (i.e., which labels are used) is captured in the type language using row extension, while negative information (i.e., which labels are not used) is reflected by the use of predicates.

The most obvious benefit of this approach is that we can adapt results and properties from the general framework of qualified types—such as a type inference algorithm and a compilation method—without having to go back to first principles. The result is a considerable simplification of both the overall presentation and of specific proofs. Another important advantage of this approach is that it guarantees compatibility with other applications of qualified types. For example, our type system can be used—and indeed, has already been used in our prototype implementation—in conjunction with the type class mechanisms of Haskell.

Although these ideas are not new in themselves, having been studied previously by a wide selection of different authors, we are the first to propose a single system combining these features. For example, Rémy [Rém94a] describes a very expressive set of record and variant operations but does not provide a simple and efficient method for compilation, while Ohori [Oho95] provided a simple implementation but lacks support for extensibility.

Formal semantics: To provide a formal semantics for our system of records and variants, and more generally to provide a formal foundations for qualified types, this dissertation (Chapters 4 and 5) develops two alternative semantics for qualified types, supporting a statement of soundness.

Inspired by early work of Wadler and Blott [WB89], the first semantics translates away overloaded expressions, providing a platform for reasoning about qualified types with respect to any model of the simply typed lambda calculus. The second takes a more direct categorical approach, interpreting constrained and non-constrained types within distinct categories. Unlike the models for Jones' translation into the polymorphic lambda calculus, the alternative semantics defined in this dissertation do not depend on the difficult, sometimes counter-intuitive, mathematical structures that are needed to model type abstraction.

Although our proposed systems are based on the general theory of qualified types, it was not clear that the original semantics described by Jones [Jon94b] was enough (in fact it would have required extensions). To this end we have developed two alternative semantics for qualified types. These provide general frameworks for reasoning syntactically or semantically about applications of qualified types and are not restricted simply to records and variants. As an example, we used these results to describe a simple categorical semantics for Haskell style type classes.

Extensions: Chapter 7 goes further than previous approaches, introducing the following extensions to our original proposal

- row polymorphism;
- first-class labels; and
- casting.

To our knowledge it is the first time that these ideas have been proposed in the presence of type inference. In fact we are, to our knowledge, the first to propose first-class labels in any form, while row polymorphism was discussed in an explicitly typed setting by Pierce and Turner [PT94]. Chapter 7 also proposes support for a casting operator for extensible records, corresponding closely to functionality supported by a wide selection of object-oriented languages. Although similar operators have been studied by other authors (see Abadi and Cardelli [AC96], for example) we are, to our knowledge, the first to study such an operator in the presence of type inference.

Concrete proposal: A final, but important contribution of this dissertation is the proposal for extending the functional programming language Haskell with extensible records (Chapter refchapter-haskell). We believe, this proposal shows that, although our work is generally theoretically based, it also has practical and worthwhile applications.

1.5 Dissertation outline

The dissertation is separated into two parts; the first is concerned with theoretical issues, while the second considers practical aspects of the ideas developed in part one. To help the reader understand the structure of the dissertation Figure 1.1

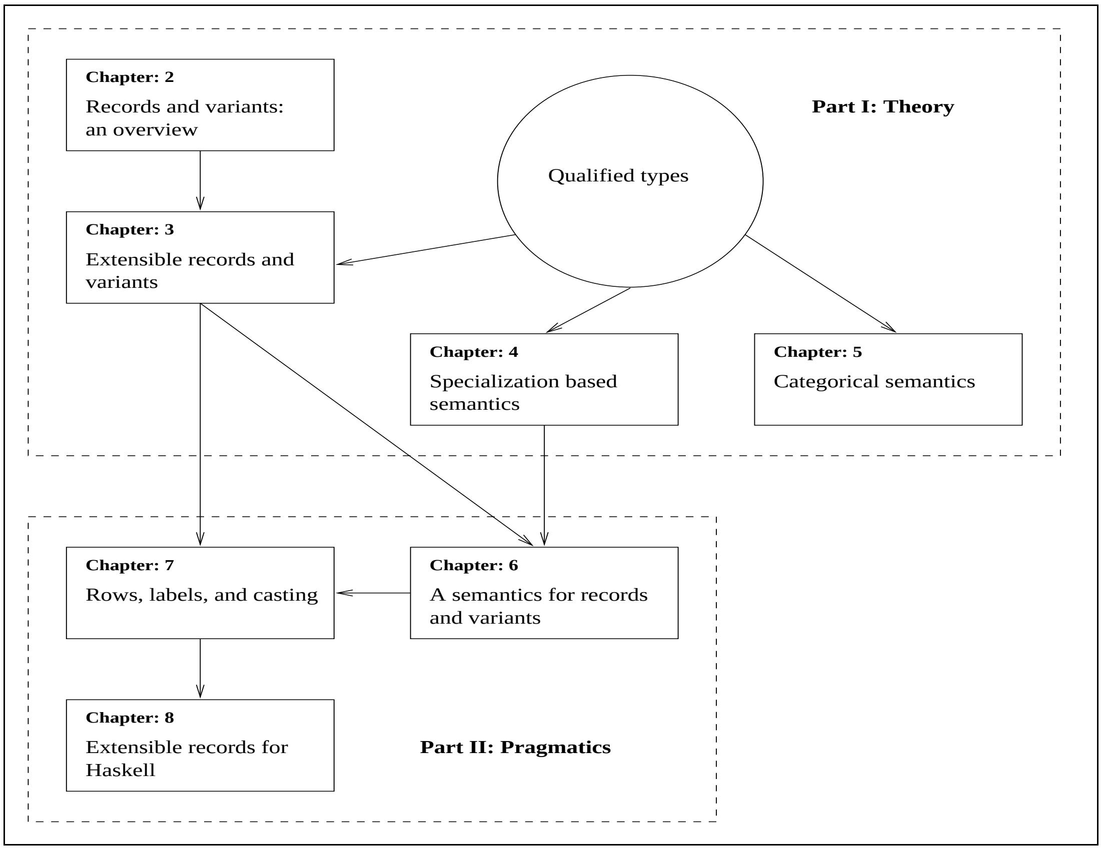


Figure 1.1: Dissertation outline.

provides a pictorial view of the chapters contained within the different parts, and the relationships between them. The bubble, denoted qualified types, is not a chapter, but a conceptual point indicating how the general theory of qualified types is incorporated into our work. Although there is no separate chapter on qualified types, Chapters 3, 4, and 5 describe alternative syntactic presentations of the general theory, while Chapters 4 and 5 also provide alternative formal foundations for qualified types.

We conclude this introduction by summarizing the remaining chapters.

- Chapter 2 focuses on datatypes, paying particular attention to their construction using sums (variants) and products (records). An informal overview of our proposal for extensible records and variants is given. A number of basic operations are considered, themselves natural generalizations of operations in category theory (see Maclane [Lan72], for example) and logic (see Hamiton [Ham88], for example).

- Chapter 3 provides a formal presentation of our type system for extensible records and variants, based on the general theory of qualified types. To encourage the practical use of our system, this chapter presents general algorithms for type inference and record and variant compilation.

- Chapter 4 provides a specialization based semantics for qualified types. A translation is described into the simply typed lambda calculus [Chu40], providing a platform for proving equational soundness for qualified types.

- Chapter 5 introduces an alternative semantics for qualified types, based on polynomial categories. A semantics for Haskell style type classes is outlined as an example application of the categorical semantics.

- Chapter 6 makes use of the specialization semantics for qualified types to provide a semantics, combined with a statement of soundness, for our system of record and variants. To further support our choice of types for the primitive record and variant operations, the semantics is used to show that other reasonable types for the basic operations have the same semantic meaning.

- Chapter 7 considers extensions to our original system of records and variants. In particular, a notion of row polymorphism is discussed, allowing the introduction of an extensible first-class case construct, first-class labels, and an operator to test for field membership.

- Chapter 8 presents an alternative proposal for records in Haskell, based upon ideas from previous chapters. An informal presentation of extensible records in Haskell is considered, paying particular attention to pragmatic issues (e.g., syntax), which must be considered in the case of a new language feature.

- Chapter 9 provides concluding remarks combined with a summary of contributions and considers a selection of ideas for further work.

- Appendix A provides detailed descriptions of results stated, but not proved in the main body of the report. We have tried to keep the dissertation as coherent as possible and often found that including a proof in the main text interrupted the flow. To this end, the reader will find that most proofs appear outside of the main text. However, in some cases, such as to introduce a new technique or notation, proofs are included in the main body.

- Appendix B introduces some important categorical concepts, which are used in Chapters 4 and 5. In particular, we describe the concepts of cartesian closure and polynomial extension.

Part I

Theory

Chapter 2

Records and variants: an overview

This chapter outlines, informally, the set of basic operations for records and variants, and considers details of implementation. The sections of this chapter are as follows. Section 2.1 provides a general overview, including examples, of the operations we expect over records and variants. Section 2.2 describes compilation and runtime related issues for records and variants.

2.1 Overview

Both record and variant types are defined in terms of rows, and these are constructed by extension, starting from the empty row, $\{\!|\,|\!\}$. For example, the *Date* and *Event* types of the previous chapter are expressed as

$$\begin{array}{lcl} \mathit{Date} & = & \mathit{Rec}\ \{\!|\mathit{day} : \mathit{Int} \,|\, \{\!|\mathit{month} : \mathit{Int} \,|\, \{\!|\mathit{year} : \mathit{Int} \,|\, \{\!|\,|\!\}|\!\}|\!\}|\!\} \\ \mathit{Event} & = & \mathit{Var}\ \{\!|\mathit{key} : \mathit{Char} \,|\, \{\!|\mathit{mouse} : \mathit{Point} \,|\, \{\!|\,|\!\}|\!\}|\!\}. \end{array}$$

It is convenient to introduce abbreviations for rows obtained in this way

$$\begin{array}{lcl} \{\!|l_1 {:} \tau_1,\ \ldots,\ l_n {:} \tau_n \,|\, r|\!\} & = & \{\!|l_1 {:} \tau_1 \,|\, \ldots \{\!|l_n {:} \tau_n \,|\, r|\!\} \ldots |\!\} \\ \{\!|l_1 {:} \tau_1,\ \ldots,\ l_n {:} \tau_n|\!\} & = & \{\!|l_1 {:} \tau_1,\ \ldots,\ l_n {:} \tau_n \,|\, \{\!|\,|\!\}|\!\}. \end{array}$$

Note, however, that we treat rows, and hence record or variant types, as equal if they include the same fields, regardless of the order in which those fields are listed.

Thus, although it is standard in Europe to display the day field before the month, while in America the reverse is true, the following records types are considered equal

$$Rec\ \{\!|day : Int, month : Int, year : Int|\!\} = \\ Rec\ \{\!|month : Int, day : Int, year : Int|\!\}.$$

2.1.1 Basic operations

Intuitively, a record of type $Rec\ \{\!|l : \alpha \,|\, r|\!\}$ is like a pair whose first component is a value of type α, and whose second component is a record of type $Rec\ r$. This motivates our choice of basic operations on records, which correspond directly to the two projections and the pairing constructor for products in category theory or logic. For example, the operation to extract a value of type α for a field l can be assigned the type

$$(\_.l) : Rec\ \{\!|l : \alpha \,|\, r|\!\} \to \alpha,$$

corresponding to the first projection. There is in fact a family of record selection operators, one for each possible choice of α and r. This notion is captured by universally quantifying over α and r, reflected in the following type for selection

$$(\_.l) : \forall\alpha.\forall r.Rec\ \{\!|l : \alpha \,|\, r|\!\} \to \alpha.$$

There is, however, one complication; we do not allow repeated uses of any label within a particular row, so the expression $\{\!|l : \alpha \mid r|\!\}$ is only valid if l does not appear in r. This is reflected by prefixing the type for record selection with a predicate $(r \backslash l)$, pronounced "r lacks l", that restricts instantiation of r to rows without an l field. Thus, the final type for record selection is

$$(\_.l) : \forall\alpha.\forall r.(r \backslash l) \Rightarrow Rec\ \{\!|l : \alpha \,|\, r|\!\} \to \alpha.$$

The types of the remaining primitive records operations, restriction and extension, are constructed in a similar fashion

- Restriction: to remove a field labelled l

$$(\_ - l) : \forall\alpha.\forall r.(r \backslash l) \Rightarrow Rec\ \{\!|l : \alpha \,|\, r|\!\} \to Rec\ r.$$

- Extension: to add a field l to an existing record

$$(l = \_ \,|\, \_) : \forall\alpha.\forall r.(r \backslash l) \Rightarrow \alpha \to Rec\ r \to Rec\ \{\!|l : \alpha \,|\, r|\!\}.$$

We can use these basic operations to implement a number of additional operators, including

- Update/replace: to update the value in a particular field, possibly with a value of a different type

$$\begin{array}{lcl} (l := \_ \,|\, \_) & : & \forall\alpha.\forall\beta.\forall r.(r\backslash l) \Rightarrow \alpha \to \mathit{Rec}\ \{\!|\, l : \beta \,|\, r \,|\!\} \to \mathit{Rec}\ \{\!|\, l : \alpha \,|\, r \,|\!\} \\ (l := x \,|\, r) & = & (l = x \,|\, r - l) \end{array}$$

- Renaming: to change the label attached to a particular field

$$\begin{array}{lcl} \_[l \leftarrow m] & : & \forall\alpha.\forall r.(r\backslash l, r\backslash m) \Rightarrow \mathit{Rec}\ \{\!|\, l : \alpha \,|\, r \,|\!\} \to \mathit{Rec}\ \{\!|\, m : \alpha \,|\, r \,|\!\} \\ r[l \leftarrow m] & = & (m = r.l \,|\, r - l) \end{array}$$

The empty record, (), plays an important role as the only proper value of type $\mathit{Rec}\ \{\!|\,|\!\}$. Again, it is convenient to introduce abbreviations for the construction of record values by repeated extension

$$\begin{array}{lcl} (l_1{=}e_1,\ \ldots,\ l_n{=}e_n \,|\, r) & = & (l_1{=}e_1 \,|\, \ldots (l_n{=}e_n \,|\, r) \ldots) \\ (l_1{=}e_1,\ \ldots,\ l_n{=}e_n) & = & (l_1{=}e_1,\ \ldots,\ l_n{=}e_n \,|\, ()). \end{array}$$

Intuitively, a variant of type $\mathit{Var}\{\!|\, l : \alpha \,|\, r \,|\!\}$ is either a value of type α tagged with the label l, or is a value of type $\mathit{Var}\ r$. We can specify the basic operations on variants in a similar way. Again, they correspond closely to the standard operations on sums in category theory or logic

- Injection: to tag a value with the label l

$$\langle l = \_ \rangle : \forall\alpha.\forall r.(r\backslash l) \Rightarrow \alpha \to \mathit{Var}\ \{\!|\, l : \alpha \,|\, r \,|\!\}.$$

- Embedding: to embed a value in a variant type that also allows for a new label, l

$$\langle l \,|\, \_ \rangle : \forall\alpha.\forall r.(r\backslash l) \Rightarrow \mathit{Var}\ r \to \mathit{Var}\ \{\!|\, l : \alpha \,|\, r \,|\!\}.$$

- Decomposition: to act on the value in a variant, according to whether or not it is labelled with l

$$\begin{array}{l} (l \in \_ ?\_ : \_) : \forall\alpha.\forall\beta.\forall r.(r\backslash l) \Rightarrow \mathit{Var}\ \{\!|\, l : \alpha \,|\, r \,|\!\} \to (\alpha \to \beta) \\ \hspace{18em} \to (\mathit{Var}\ r \to \beta) \to \beta. \end{array}$$

The empty variant, $\langle\rangle$, is the only proper value of type $Var\ \{\!|\,|\!\}$.

More sophisticated language constructs, for example, pattern matching facilities, or record update, are easily described using the operations listed here. As an example, consider a function that returns true if a mouse click has been performed, and false otherwise, which might implemented as

$$\begin{array}{lcl} mouseClick? & : & Var\,\{\!|mouse : Point, key : Char|\!\} \to Bool \\ mouseClick?\ \langle mouse = \_\rangle & = & True \\ mouseClick?\ \_ & = & False. \end{array}$$

Intuitively, an expression of the form $(mouseClick?\ E)$ is evaluated from top to bottom, first testing if the corresponding summand is labelled $mouse$ and resulting in the value $True$, otherwise the next equation is tried. This definition can easily be translated into one defined completely in terms of variant primitives

$$\begin{array}{lcl} mouseClick? & : & Var\,\{\!|mouse : Point, key : Char|\!\} \to Bool \\ mouseClick?\ v & = & mouse \in v?(\lambda x.True) : (\lambda y.False). \end{array}$$

More generally we can explain the semantics of pattern matching over variants using a translation of the form

$$\begin{array}{l} \lambda\langle l = p\rangle.e = \lambda v.l \in v?(\lambda x \to \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad p \to E \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \to \bot) : \bot, \end{array}$$

where $\bot$ is an undefined expression of the appropriate type—capturing the possibility that a pattern match may fail. Note, this translation differs slightly from that applied to the function $mouseClick?$. The problem is that it is not possible, in general, to calculate the complete set of labels for any given variant at compile time, and thus, determine if a pattern match is total. Consequently, the general translation must cater for the possibility of failure and is the reason for the introduction of the value $\bot$. However, if the compiler can determine that a pattern match will not fail then this case can be eliminted statically. Chapter 8 describes an analogous translation for record patterns.

In addition, we expect that practical implementations will use, but not display predicates implied by the context in which they appear. For example, all of the types given above for the record and variant primitives include a row $\{\!|l : \alpha \mid r|\!\}$ that is only valid if $r\backslash l$; so displaying this predicate is, in some sense, redundant. However, as we will see in the next section, this predicate plays a central role in the implementation of the basic operations.

2.2 Implementation details

Our next task is to explain how the data structures and operations described above can be implemented. We will focus on the treatment of records and, in particular, the implementation of extension, $(l = \_ \,|\, \_)$, which is one of the most frequently used basic operations. A naive approach would be to represent a record by an association list, pairing labels with values. This would allow simple implementations for each of the basic operations, with the type system providing a guarantee that the search for any given labelled field would not fail. A major disadvantage is that it does not allow constant time access to record components.

To avoid these problems, we will assume instead that a record value is represented by a contiguous block of memory that contains a value for each individual field. To extend a particular record r with a field l, we need to know the offset where the l field can be inserted in the block of memory representing r. Languages without polymorphic extension will usually only allow an expression of the form $(l = e \,|\, r)$ if the offset value, and hence the structure or even the full type of r, is known at compile-time. For example, the record (year = 1942, day = 17, month = 1) might be represented as the block of memory

17	1	1942

Notice that to inforce a canonical representation of records in memory we have assumed the ordering $day < month < year$ on labels. In general, we shall require that any given set of labels, L, is supplied with a total ordering, $<$.

However, it is not actually necessary to know the position of every field at compile-time; instead, we can treat unknown offsets as implicit parameters whose values will be supplied at run-time when the full types of the records concerned are known. This is essentially the compilation method that was used by Ohori [Oho95], and also suggested, independently, by Jones [Jon94b]. If we forget about typing issues for a moment and assume that records are implemented as vectors, then the extension operator, $(l = v \mid r)$, can be implemented by a function $\lambda i.pext\ i\ v\ r$, using the extra parameter i to supply the offset at which the value v is to be inserted into the record r. Here the function $pext$ is a primitive operation on arrays, which given an offset i, an expression E, and an array r inserts E into r at offset i. As an example, the expression

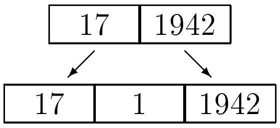
$$(\lambda r.(month = 1 \,|\, r))\ (day = 17, year = 1942),$$

can be implemented by compiling it to

$$(\lambda i.\lambda v.\lambda r.\mathit{pext}\ i\ v\ r)\ 1\ 1\ (17\ 1942).$$

Here the natural number 1 represents the offset into the vector (17 1) at which to insert the component 1. The resulting expression, *pext* 1 1 (17 1942), can be implemented by simple copying procedures, allowing the correct insertion of the value 1.

This process is captured diagrammatically by the following diagram:

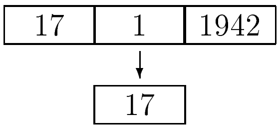


Note that all fields with labels considered less than the label being inserted, remain in the same position in the array, while all fields following the inserted field are shifted up one position. Record restriction can be implemented in a similar fashion, where, instead of larger field labels being shifted up one position, they are shifted down one.

As with the compilation of extension, record selection, $(\_.l)$, can be implemented by a function $\lambda i.\lambda r.r[i]$[1], using the extra parameter i to supply the offset of l in r. For example, the expression $(\mathit{day} = 17, \mathit{month} = 1, \mathit{year} = 1942).\mathit{day}$, can be implemented by compiling it to

$$(\lambda i.\lambda r.r[i])\ 0\ (17\ 1\ 1942).$$

which evaluates to 17, as expected. Again this can be pictured diagrammatically as



[1]As a notational convenience we adapt standard vector notation, by representing indexing through the concatenation of the expression $[i]$, where i is the corresponding index, to an arbitrary vector expression.

Of course, there are run-time overheads in calculating and passing offset values as extra parameters. However, an attractive feature of our system is that these costs are only incurred when the extra flexibility is required. This point can be highlighted by observing that, if the shape of a record or variant is completely determined at compile time (indicated by its type), then evidence for any given label will be constant. Consequently, we can use standard compiler optimizations to eliminate corresponding β-redexes. For example, consider selecting the field labelled *plus* from the expression $(id = 0, plus = plusInt)$, which has type $Rec\{\!|id : Int, plus : Int \to Int \to Int|\!\}$. The resulting expression can be implemented by compiling it to

$$(\lambda i.\lambda r.r[i])\ 1\ (0\ plusInt),$$

with the constant 1 being supplied to the abstraction λi to indicate the offset for *plus*. Thus, a compiler can simply perform two β-reductions to obtain the following inlined array selection

$$(0\ plusInt)[1].$$

Of course, we could further reduce this expression to *plusInt*, however, in general, this will not be possible, while it will always be the case that constant offset β-redexs can be eliminated at compile time. The idea of β-redex elimination, for overloaded values, is in fact a simple form of partial evaluation, applying not only to the compilation of records and variants but, in general, to any application of qualified types. Chapter 4 describes a general algorithm, called specialization, for overloading elimination, which provides the basis for defining a semantics for qualified types that provide a foundation for describing sound rules for constant elimination.

Each predicate $(r\backslash l)$ in the type of a function signals the need for an extra run-time parameter to specify the offset at which a field labelled l would be inserted into a record of type $Rec\ r$. Obviously, the same offset can also be used to locate or remove the l field from a record of type $Rec\ \{\!|l : \alpha \,|\, r|\!\}$, or treated as ordinal numbers to access and tag values in a variant. So, this one extra piece of information is all that we need to implement the basic operations.

Operations like record extension and restriction will, in general, be implemented by copying. Optimizations can be used to combine multiple extensions or restrictions of records, avoiding unnecessary allocation and initialization of intermediate values. For example, a compiler can generate code that will allocate and initialize the storage for a record $(x = 1, y = 2, z = 3)$ in a single step, rather than a sequence of three individual allocations and extensions as a naive interpretation might suggest.

The typechecker gathers and simplifies the predicates generated by each use of an operator on records or variants. For example, if *today* is a value of type *Date*, then an expression like *today.month* will generate a single constraint, $\{\!|\mathit{day} : \mathit{Int},\ \mathit{year} : \mathit{Int}|\!\}\backslash \mathit{month}$. Predicates like this, involving rows whose structure is known at compile-time, are easily discharged by calculating the appropriate offset value. Obviously, a compiler can use this information to produce efficient code by inlining and specializing the selector function, $(\_.\mathit{month})$.

Predicates that are not discharged within a section of code will, instead, be reflected in the type assigned to it. For example, there is nothing in the following definition to indicate the full type of d

$$\mathit{newYear}\ d \quad = \quad d.\mathit{day} = 1 \wedge d.\mathit{month} = 1,$$

so the inferred type will be

$$(r\backslash \mathit{day}, r\backslash \mathit{month}) \Rightarrow \mathit{Rec}\ \{\!|\mathit{day} : \mathit{Int},\ \mathit{month} : \mathit{Int} \,|\, r|\!\} \rightarrow \mathit{Bool}.$$

We would not expect this definition to have been accepted at all by a compiler for SML which requires the set of labels in a record to be uniquely determined by 'program context'. But the meaning of this phrase is defined only loosely by an informal note in the definition of SML [MTH90, MTH97]. Now, with the ideas used in this chapter, there is a way to make this precise: a definition is only acceptable in SML if the inferred type does not contain any predicates. For programs written with these restrictions, a language based on our type system should offer the same levels of performance as SML.

It is possible that our more general treatment of record operations could result in compiled programs that are littered with unwanted offset parameters. However, experience with our prototype implementations have so far provided no evidence that is in fact the case. As discussed above, simple compiler optimizations (e.g., constant folding) can be used to eliminate offsets known at compile time. Furthermore, there are simple steps that can be taken to avoid such problems. For example, a compiler might reject any definition with an inferred type containing predicates, unless an explicit type signature has been given to signal the programmer's acceptance. This is closely related to the *monomorphism restriction* in Haskell [PH97] and to the *value restriction* in SML [Wri95, Ler93, MTH97].

Chapter 3

Extensible records and variants

In the previous chapter, we introduced, informally, a system of extensible records and variants. This chapter presents a formal development of that system.

The sections of this chapter are as follows. Section 3.1 provides a formal presentation of our new type system for extensible records and variants. This is followed by discussions of type inference in Section 3.2 and of compilation in Section 3.3. Finally, Section 3.4 describes related work.

An earlier version of this chapter has previously been distributed in the form of a technical report [GJ96].

3.1 Formal presentation

This section provides a formal presentation of our type system, based on two particular ingredients

- The theory of qualified types [Jon94b], which provides a general framework for describing constrained polymorphism and overloading. In the current application, we use constraints to capture assumptions about the occurrences of labels within rows.
- A higher-order version of the Hindley-Milner type system [Hin69, Mil78, DM82], originally introduced in the study of *constructor classes* [Jon95c]. Among other things, this provides a simple way to introduce the new constructs for rows, records, and variants without the need for special, ad-hoc

syntax.

We split the presentation into sections on kinds (Section 3.1.1), types and constructors (Section 3.1.2), predicates (Section 3.1.3), and typing rules (Section 3.1.4).

3.1.1 Kinds

One of the most important aspects of the work described here is the use of a *kind* system to distinguish between different kinds of type constructor. Formally, the set of kinds is specified by the following grammar

$$\begin{array}{lcll} \kappa & ::= & * & \textit{the kind of all types} \\ & | & \textit{row} & \textit{the kind of all rows} \\ & | & \kappa_1 \to \kappa_2 & \textit{function kinds.} \end{array}$$

Intuitively, the kind $\kappa_1 \to \kappa_2$ represents constructors that take something of kind κ_1 and return something of kind κ_2. The *row* kind is new to the system presented here and was not part of the type system used in the development of constructor classes.

3.1.2 Types and constructors

For each kind κ, we have a collection of constructors C^κ (including variables α^κ) of kind κ

$$\begin{array}{lcll} C^\kappa & ::= & \chi^\kappa & \textit{constants} \\ & | & \alpha^\kappa & \textit{variables} \\ & | & C^{\kappa' \to \kappa}\, C^{\kappa'} & \textit{applications} \\ \tau & ::= & C^* & \textit{types} \end{array}$$

The usual collection of types, represented here by the symbol τ, is just the set of constructors of kind $*$. We assume given a countable set of labels, $\langle L; < \rangle$, including a total ordering, $<: L \times L \to \mathbb{B}$, on labels. For the purposes of this dissertation, we assume that the set of constant constructors includes at least the following, writing $\chi{:}\kappa$ to indicate the kind κ associated with each constant χ:

$$\begin{array}{lcll}
\to & : & * \to * \to * & \text{function space;} \\
\{\!|\,|\!\} & : & \mathit{row} & \text{empty row;} \\
\{\!|l{:}_\,|\,_|\!\} & : & * \to \mathit{row} \to \mathit{row} & \text{extension, for each } l \in L; \\
\mathit{Rec} & : & \mathit{row} \to * & \text{record construction; and} \\
\mathit{Var} & : & \mathit{row} \to * & \text{variant construction.}
\end{array}$$

For example:

- The result of applying the function space constructor $\to$ to two types τ and τ' is the type of functions from τ to τ', and is written as $\tau \to \tau'$ in more conventional notation. As usual the constructor $\to$ is assumed to associate to the right. That is, for example, the kind $* \to * \to *$ means $* \to (* \to *)$ rather than $(* \to *) \to *$.
- The result of applying the *Rec* constant to the empty row $\{\!|\,|\!\}$ of kind *row* is the type $\mathit{Rec}\ \{\!|\,|\!\}$ of kind $*$.
- The result of applying an extension constructor $\{\!|l{:}\_\,|\,\_|\!\}$ to a type τ and a row r is a row, usually written as $\{\!|l{:}\tau\,|\,r|\!\}$, obtained by extending r with a field labelled l of type τ. Note that we include an extension constructor for each different label l. To avoid problems later, we will also need to prohibit partial application of extension constructors.

The kind system is used to ensure that type expressions are well-formed. While it is sometimes convenient to annotate individual constructors with their kinds, there is no need in practice for a programmer to supply these annotations. Instead, they can be calculated automatically using a simple kind inference process [Jon95c].

We consider two rows to be equivalent if they include the same fields, regardless of the order in which they are listed. This is described formally by the equation

$$\{\!|l{:}\tau,\ l'{:}\tau'\,|\,r|\!\} = \{\!|l'{:}\tau',\ l{:}\tau\,|\,r|\!\},$$

and extends in the obvious way to an equality on arbitrary constructors.

For the purposes of later sections, we define a *membership* relation, $(l : \tau)\!\in\! r$, to describe when a particular field $(l : \tau)$ appears in a row r

$$(inVar) \quad (l : \tau)\!\in\!\{\!| l : \tau \,|\, r |\!\}$$

$$(inRow) \quad \frac{(l : \tau)\!\in\! r \quad l \neq l'}{(l : \tau)\!\in\!\{\!| l' : \tau' \,|\, r |\!\}}$$

and a *restriction* operation, $r - l$, that returns the row obtained from r by deleting the field labelled l

$$\begin{array}{lcl} \{\!| l : \tau \,|\, r |\!\} - l & = & r \\ \{\!| l' : \tau \,|\, r |\!\} - l & = & \{\!| l' : \tau \,|\, r - l |\!\}. \end{array}$$

It is easy to prove that these operations are well-defined with respect to the equality on constructors, and to confirm these intuitions we have the following lemma.

Lemma 3.1 *If* $(l : \tau)\!\in\! r$, *then* $r = \{\!| l : \tau \,|\, r - l |\!\}$.

A proof of this result is given in Section A.1.1 of Appendix A.

3.1.3 Predicates

The syntax for rows allows examples like $\{\!| l : \tau,\ l : \tau' |\!\}$ where a single label appears in more than one field. Such an example is not appropriate for our work with records or variants, in which we allow at most one field with any given label. Clearly, some additional mechanisms are needed to enable us to specify that a type of the form $Rec\ \{\!| l : \tau \mid r |\!\}$, for example, is only valid if the row r does not contain a field labelled with l.

One way to achieve this is to use a more sophisticated kind system, with sets of labels, L, as kinds instead of the single *row* kind. For example, rows with field labels $l_1, \ldots l_n$ can be represented by the kind $L = \{l_1, \ldots, l_n\}$. This is essentially the approach adopted by Ohori [Oho95]. Unfortunately, this becomes much more complicated if we try to extend it to deal with extensible rows. In particular, we would need to assign whole families of kinds, indexed by label sets, L, to some of the constructor constants introduced in the previous section

$$\begin{array}{lcl} \{\!| l{:}\_ \,|\, \_ |\!\} & :: & * \to L \to (L \cup \{l\}) \quad l \not\in L \\ Rec,\ Var & :: & L \to * \end{array}$$

The alternative that we adopt in this dissertation is based on the theory of qualified types [Jon94b], using *predicates* to capture any side conditions that are required to ensure that a given type expression is valid. In fact, only a single form of predicate is needed for this purpose

$$\pi \quad ::= \quad C^{row}\backslash l$$

Intuitively, the predicate $r\backslash l$ can be read as an assertion that the row r does not contain an l field. More precisely, we explain the meaning of predicates using the *entailment* relation defined in Figure 3.1.

$$P \cup \{\pi\} \Vdash \pi \qquad \frac{P \Vdash r\backslash l \quad l \neq l'}{P \Vdash \{\!|l' : \tau \,|\, r|\!\}\backslash l} \qquad P \Vdash \{\!||\!\}\backslash l$$

Figure 3.1: Predicate entailment for rows.

A derivation of $P \Vdash \pi$ from these rules can be understood as a proof that, if all of the predicates in the set P hold, then so does π. As an example of predicate entailment consider extending records $r : Rec\ \{\!|x : Int, y : Int|\!\}$ and $r' : Rec\ \{\!|x : Int, y : Int, colour : Colour|\!\}$ with a field $colour : Colour$. Upon application of the extension operation the predicates $\{\!|x : Int, y : Int|\!\}\backslash colour$ and $\{\!|colour : Colour, x : Int, y : Int|\!\}\backslash colour$ are introduced and must be provable under predicate entailment. The following derivation shows that it is straightforward to justify the extension with a field not already present:

$$\frac{\dfrac{\emptyset \Vdash \{\!||\!\}\backslash colour \quad y \neq colour}{\emptyset \Vdash \{\!|y : Int \,|\, \{\!||\!\}|\!\}\backslash colour} \qquad x \neq colour}{\emptyset \Vdash \{\!|x : Int \,|\, \{\!|y : Int \,|\, \{\!||\!\}|\!\}|\!\}\backslash colour.}$$

However, suppose we try to construct a derivation asserting

$$\emptyset \Vdash \{\!|colour : Colour, x : Int, y : Int|\!\}\backslash colour.$$

It is clear that although it is possible to give a proof for $\emptyset \Vdash \{\!|x : Int, y : Int|\!\}\backslash colour$ (the previous derivation, for example), the remaining hypothesis $colour \neq colour$ is, in general, not provable. The important point here is to observe that a predicate of the form $\{\!|l : \tau \,|\, r|\!\}\backslash l'$ is proven simply by proving a similar assertion for its subcomponent r.

It is easy to prove that the relation $\Vdash$ is well-defined with respect to equality of constructors.

3.1.4 Typing rules

Following Damas and Milner [DM82], we distinguish between the simple types, τ, described above, and type schemes, σ, described by the grammar

$$\begin{array}{lcllll} \sigma & ::= & \rho & \mid & \forall\alpha.\rho & \textit{type schemes} \\ \rho & ::= & \tau & \mid & \pi \Rightarrow \tau & \textit{qualified types.} \end{array}$$

Restrictions on the instantiation of universal quantifiers, and hence on polymorphism, are described by encoding the required constraints as a set of predicates, P, in a qualified type of the form $P \Rightarrow \tau$. The set of free type variables in an object X is written as $TV(X)$.

The term language is just core-ML, an implicitly typed λ-calculus, extended with constants and a **let** construct, and described by the following grammar

$$E \;::=\; x \mid c \mid E\,F \mid \lambda x.E \mid \textbf{let}\; x = E \;\textbf{in}\; F.$$

We assume that the set of constants c includes the operations and values required for manipulating records and variants as described in Section 2.1, and that each constant c is assigned a closed type scheme, σ_c.

The typing rules are presented in Figure 3.2. A judgement of the form $P \,|\, A \vdash E : \sigma$ represents an assertion that, if the predicates in P hold, then the term E has type σ, using assumptions in A to provide types for free variables. These are just the standard rules for qualified types [Jon94b], extending the rules of Damas and Milner [DM82] to account for the use of predicates. Note that uses of the symbols τ, ρ, and σ in these rules is particularly important in restricting their application to particular classes of types or type schemes.

3.2 Type Inference

This section provides a formal presentation of a type inference algorithm for our system. The most important feature is the introduction of *inserters* in Section 3.2.1 to account for non-trivial equalities between row expressions during unification.

$$(const) \qquad P \mid A \vdash c : \sigma_c$$

$$(var) \qquad \frac{(x : \sigma) \in A}{P \mid A \vdash x : \sigma}$$

$$(\to E) \qquad \frac{P \mid A \vdash E : \tau' \to \tau \quad P \mid A \vdash F : \tau'}{P \mid A \vdash EF : \tau}$$

$$(\to I) \qquad \frac{P \mid A_x, x : \tau' \vdash E : \tau}{P \mid A \vdash \lambda x.E : \tau' \to \tau}$$

$$(\Rightarrow E) \qquad \frac{P \mid A \vdash E : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid A \vdash E : \rho}$$

$$(\Rightarrow I) \qquad \frac{P \cup \{\pi\} \mid A \vdash E : \rho}{P \mid A \vdash E : \pi \Rightarrow \rho}$$

$$(\forall E) \qquad \frac{P \mid A \vdash E : \forall \alpha.\sigma}{P \mid A \vdash E : [\tau/\alpha]\sigma}$$

$$(\forall I) \qquad \frac{P \mid A \vdash E : \sigma \quad \alpha \notin TV(A) \cup TV(P)}{P \mid A \vdash E : \forall \alpha.\sigma}$$

$$(let) \qquad \frac{P \mid A \vdash E : \sigma \quad Q \mid A_x, x : \sigma \vdash F : \tau}{P, Q \mid A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$

Figure 3.2: Typing rules.

3.2.1 Unification and insertion

Unification is a standard tool in type inference, and is used, for example, to ensure that the formal and actual parameters of a function have the same type. Formally, a substitution[1] S is a *unifier* of constructors $C, C' \in C^\kappa$ if $SC = SC'$, and is a *most general unifier* of C and C' if every unifier of these two constructors can be written in the form RS, for some substitution R.

The rules in Figure 3.3 provide an algorithm for calculating unifiers, writing $C \overset{U}{\sim} C'$

[1]For the purposes of this dissertation, we restrict our attention to *kind-preserving substitutions*; that is, to substitutions that map variables $\alpha \in C^\kappa$ to constructors of the corresponding kind, κ.

$$\begin{array}{lll}
(id) & C \overset{id}{\sim} C & \\[2ex]
(bind) & \left.\begin{array}{c} \alpha \overset{[C/\alpha]}{\sim} C \\[1ex] C \overset{[C/\alpha]}{\sim} \alpha \end{array}\right\} & \alpha \notin TV(C) \\[2ex]
(apply) & \dfrac{C \overset{U}{\sim} C' \quad UD \overset{U'}{\sim} UD'}{CD \overset{U'U}{\sim} C'D'} & \\[2ex]
(row) & \dfrac{(l : \tau) \overset{I}{\in} r' \quad Ir \overset{U}{\sim} (Ir' - l)}{\{\!|\, l : \tau \,|\, r \,|\!\} \overset{UI}{\sim} r'} &
\end{array}$$

Figure 3.3: Kind-preserving unification.

for the assertion that U is a unifier of the constructors $C, C' \in C^{\kappa}$. The first three rules are standard [Rob65], and are even suitable for unifying two row expressions that list exactly the same components with exactly the same ordering in each. The fourth rule, (row), is needed to deal with the more general problems of row unification, taking account of differences in ordering the fields.

To understand how (row) works, consider the task of unifying two rows $\{\!|\, l : \tau \,|\, r \,|\!\}$ and $\{\!|\, l' : \tau' \,|\, r' \,|\!\}$, where l,l' are distinct labels, and r,r' are distinct row variables. Our goal is to find a substitution S such that

$$S\{\!|\, l : \tau \,|\, r \,|\!\} = \{\!|\, l : S\tau \,|\, Sr \,|\!\} = \{\!|\, l' : S\tau' \,|\, Sr' \,|\!\} = S\{\!|\, l' : \tau' \,|\, r' \,|\!\}$$

Clearly, the row on the left includes an $(l : S\tau)$ field, while the last row on the right includes an $(l' : S\tau')$ field. If these two types are to be equal, then we must choose the substitution S so that it will 'insert' the missing fields into the two rows r' and r, respectively. In this particular case, assuming that $r, r' \notin TV(\tau, \tau')$, which is the standard occurs check, then we can choose

$$S = [\{\!|\, l' : \tau' \,|\, r'' \,|\!\}/r, \{\!|\, l : \tau \,|\, r'' \,|\!\}/r']$$

where r'' is a new type variable.

More generally, we will say that a substitution S is an *inserter* of $(l : \tau)$ into $r \in C^{row}$ if $(l : S\tau) \in Sr$. S is a *most general inserter* of $(l : \tau)$ into r if every

$$(inVar) \qquad (l : \tau) \overset{[\{\!|l:\tau|r'|\!\}/r]}{\in} r, \quad r \notin TV(\tau),\ r' \text{ new}$$

$$(inTail) \qquad \frac{(l : \tau) \overset{I}{\in} r \quad l \neq l'}{(l : \tau) \overset{I}{\in} \{\!|l' : \tau \,|\, r|\!\}}$$

$$(inHead) \qquad \frac{\tau \overset{U}{\sim} \tau'}{(l : \tau) \overset{U}{\in} \{\!|l : \tau' \,|\, r|\!\}}$$

Figure 3.4: Kind-preserving insertion.

such inserter can be written in the form RS, for some substitution R. The rules in Figure 3.4 define an algorithm for calculating inserters, writing $(l : \tau) \overset{I}{\in} r$ for the assertion that I is an inserter of $(l : \tau)$ into $r \in C^{row}$. An expression of the form r *new* states that the variable r is new and thus, has not been used previously. Note that the last rule here, (*inHead*), makes use of the unification algorithm in Figure 3.3, so the two algorithms are mutually recursive. The important properties of the two algorithms—both soundness and completeness—are captured in the following result

Theorem 3.2 *The unification (insertion) algorithm defined by the rules in Figure 3.3 (Figure 3.4) calculates most-general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.*

A proof of this result is given in Section A.1.2 of Appendix A.

3.2.2 A type inference algorithm

Given the unification algorithm described in the previous section, we can use the type inference algorithm for qualified types [Jon94b] as a type inference algorithm for the type system presented in this chapter. For completeness, we include a definition of the algorithm using the rules in Figure 3.5. Following Rémy [Rém94a], these rules can be understood as an attribute grammar; in each typing judgement $P \,|\, TA \overset{W}{\vdash} E : \tau$, the type assignment A and the term E are inherited attributes, while the predicate assignment P, type τ, and substitution T are synthesized.

$$(var)^{\mathrm{W}} \qquad \frac{(x : \forall\alpha_i.P \Rightarrow \tau) \in A \quad \beta_i \text{ new}}{[\beta_i/\alpha_i]P \,|\, A \overset{\mathrm{W}}{\vdash} x : [\beta_i/\alpha_i]\tau}$$

$$(\to E)^{\mathrm{W}} \qquad \frac{\begin{array}{c} P \,|\, TA \overset{\mathrm{W}}{\vdash} E : \tau \qquad Q \,|\, T'TA \overset{\mathrm{W}}{\vdash} F : \tau' \\ T'\tau \overset{U}{\sim} \tau' \to \alpha \qquad \alpha \text{ new} \end{array}}{U(T'P \cup Q) \,|\, UT'TA \overset{\mathrm{W}}{\vdash} EF : U\alpha}$$

$$(\to I)^{\mathrm{W}} \qquad \frac{P \,|\, T(A_x, x : \alpha) \overset{\mathrm{W}}{\vdash} E : \tau \quad \alpha \text{ new}}{P \,|\, TA \overset{\mathrm{W}}{\vdash} \lambda x.E : T\alpha \to \tau}$$

$$(let)^{\mathrm{W}} \qquad \frac{\begin{array}{c} P \,|\, TA \overset{\mathrm{W}}{\vdash} E : \tau \qquad \sigma = Gen(TA, P \Rightarrow \tau) \\ P' \,|\, T'(TA_x, x : \sigma) \overset{\mathrm{W}}{\vdash} F : \tau' \end{array}}{P' \,|\, T'TA \overset{\mathrm{W}}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

Figure 3.5: Type inference algorithm W.

The $(let)^{\mathrm{W}}$ rule uses an auxiliary function to calculate the generalization of a qualified type ρ with respect to a type assignment A. This is specified by the following definition.

Definition 3.3 *The generalization of a qualified type ρ with respect to the type assignment A is written $Gen(A, \rho)$ and defined by*

$$Gen(A, \rho) = \forall\alpha_i.\rho, \quad \textit{where } \{\alpha_i\} = FTV(\rho)\backslash FTV(A).$$

The type inference algorithm is both sound and complete with respect to the original typing rules.

Theorem 3.4 *The algorithm described by the rules in Figure 3.5 can be used to calculate a principal type for a given term E under assumptions A. The algorithm fails precisely when there is no typing for E under A.*

A proof of this result is given by Jones [Jon94b] and we do not reproduce the details here.

3.3 Compilation

Previously, we have described informally how programs involving operations on records and variants can be compiled and executed using a language that adds extra parameters to supply appropriate offsets. This section shows how this process can be formalized, including the calculation of offset values.

3.3.1 Compilation by translation

In the general treatment of qualified types [Jon94b], programs are compiled by translating them into a language that adds extra parameters to supply *evidence* for predicates appearing in the types of the values concerned. The whole process can be described by extending the typing rules to use judgements of the form

$$P \,|\, A \vdash E \leadsto E' : \sigma,$$

which include both the original source term E and a possible *translation*, E'. A further change here is the switch from predicate sets to *predicate assignments*; the symbol P used above represents a set of pairs $(v : \pi)$ in which no variable v appears twice. Each variable v corresponds to an extra parameter that will be added during compilation; v can be used whenever evidence for the corresponding predicate π is required in E'.

$$P \cup \{v : \pi\} \Vdash v : \pi$$

$$\frac{P \Vdash e : (r \backslash l)}{P \Vdash m : (\{\!|l' : \tau \,|\, r|\!\} \backslash l)} \qquad m = \begin{cases} e, & l < l' \\ e+1, & l' < l \end{cases}$$

$$P \Vdash 0 : (\{\!|\,|\!\} \backslash l)$$

Figure 3.6: Predicate entailment .

In the current setting, predicates are expressions of the form $(r \backslash l)$ whose evidence is the offset in r at which a field labelled l would be inserted. The calculation of evidence is described by the rules in Figure 3.6, which are direct extensions of the earlier rules for predicate entailment that were given in Figure 3.1. Intuitively, a derivation of $P \Vdash e : \pi$ tells us that we can use e as evidence for the predicate π in any environment where the assumptions in P are valid. The second rule is the most interesting and tells us how to find the position at which a label l should be inserted in a row $\{\!|l' : \tau \,|\, r|\!\}$

- If l comes before l' in the total ordering, $<$, on labels, then the required offset will be the same as the offset e of l in r.
- If l' comes before l, then we need to use an offset of $e + 1$ to account for the insertion of l'.

In general, these rules calculate offsets that are either a fixed natural number, or a fixed offset from one of the variables in P. For simplicity, we have assumed a boxed representation in which all record components occupy the same amount of storage. It is easy to allow for varying component sizes by replacing $e + 1$ in the calculation above with $e + \mathit{size}(\tau)$.

For illustration of the translation process we restrict ourselves to describing the two rules that account for the use and introduction of offset parameters, the complete set of rules for translation are described in Figure 3.7. The first of these is a variation on function application

$$\frac{P \,|\, A \vdash E \leadsto E' : \pi \Rightarrow \rho \quad P \Vdash e : \pi}{P \,|\, A \vdash E \leadsto E'\, e : \rho.}$$

This tells us that we need to supply suitable evidence e in the translation of any program whose type is qualified by a predicate π. The second rule is analogous to function abstraction, and allows us to move constraints from the predicate assignment P into the inferred type

$$\frac{P \cup \{v : \pi\} \,|\, A \vdash E \leadsto E' : \rho}{P \,|\, A \vdash E \leadsto \lambda v.E' : \pi \Rightarrow \rho}$$

These two rules are direct extensions of $(\Rightarrow E)$ and $(\Rightarrow I)$ in Figure 3.2 and, combined with simple extensions of the other rules there, can be used to construct a translation for any term in the source calculus.

Following the approach towards type inference described in Section 3.2.2 we can use the generalized type inference algorithm for qualified types [Jon94b]. This

$$(var)\quad \frac{(x:\sigma)\in A}{P\,|\,A\vdash x\leadsto x:\sigma}$$

$$(\to E)\quad \frac{P\,|\,A\vdash E\leadsto E':\tau'\to\tau \qquad P\,|\,A\vdash F\leadsto F':\tau'}{P\,|\,A\vdash EF\leadsto E'F':\tau}$$

$$(\to I)\quad \frac{P\,|\,A_x, x:\tau'\vdash E\leadsto E':\tau}{P\,|\,A\vdash \lambda x.E\leadsto \lambda x.E':\tau'\to\tau}$$

$$(\Rightarrow E)\quad \frac{P\,|\,A\vdash E\leadsto E':\pi\Rightarrow\rho \qquad P\Vdash e:\pi}{P\,|\,A\vdash E\leadsto E'e:\rho}$$

$$(\Rightarrow I)\quad \frac{P, v:\pi, P'\,|\,A\vdash E\leadsto E':\rho}{P, P'\,|\,A\vdash E\leadsto \lambda v.E':\pi\Rightarrow\rho}$$

$$(\forall E)\quad \frac{P\,|\,A\vdash E\leadsto E':\forall t.\sigma}{P\,|\,A\vdash E\leadsto E':[\tau/t]\sigma}$$

$$(\forall I)\quad \frac{P\,|\,A\vdash E\leadsto E':\sigma \qquad t\notin TV(A)\wedge t\notin TV(P)}{P\,|\,A\vdash E\leadsto E':\forall t.\sigma}$$

$$(let)\quad \frac{P\,|\,A\vdash E\leadsto E':\sigma \qquad Q\,|\,A_x, x:\sigma\vdash F\leadsto F':\tau}{P, Q\,|\,A\vdash (\mathbf{let}\ x=E\ \mathbf{in}\ F)\leadsto(\mathbf{let}\ x=E'\ \mathbf{in}\ F'):\tau}$$

Figure 3.7: Typing rules for evidence insertion.

algorithm describes an effective procedure to infer the most general type, if one exists, for an expression E, which also produces, in some formal sense, a most general translation E'. The complete set of rules are given in Figure 3.8.

3.4 Related work

In this chapter we have described a type system capturing extensible records and variants, a full complement of polymorphic operations, which supports effective type inference and simple efficient implementation. There have been many previous attempts to design type systems for records and variants that support polymorphism and extensibility. Such work is important, not just in its own right, but also in its application to the study of object-oriented or database program-

$$(var)^{\mathrm{W}} \quad \frac{(x : \forall \alpha_i . P \Rightarrow \tau) \in A \qquad \beta_i \text{ and } v \text{ new}}{v : [\beta_i/\alpha_i]P \,|\, A \overset{\mathrm{W}}{\vdash} x \rightsquigarrow xv : [\beta_i/\alpha_i]\tau}$$

$$(\rightarrow E)^{\mathrm{W}} \quad \frac{P \,|\, TA \overset{\mathrm{W}}{\vdash} E \rightsquigarrow E' : \tau \qquad Q \,|\, T'TA \overset{\mathrm{W}}{\vdash} F \rightsquigarrow F' : \tau' \qquad T'\tau \overset{U}{\sim} \tau' \to \alpha}{U(T'P, Q) \,|\, UT'TA \overset{\mathrm{W}}{\vdash} EF \rightsquigarrow E'F' : U\alpha}$$
$$\text{where } \alpha \text{ is a new variable}$$

$$(\rightarrow I)^{\mathrm{W}} \quad \frac{P \,|\, T(A_x, x : \alpha) \overset{\mathrm{W}}{\vdash} E \rightsquigarrow E' : \tau \qquad \alpha \text{ new}}{P \,|\, TA \overset{\mathrm{W}}{\vdash} \lambda x.E \rightsquigarrow \lambda x.E' : T\alpha \to \tau}$$

$$(let)^{\mathrm{W}} \quad \frac{v : P \,|\, TA \overset{\mathrm{W}}{\vdash} E \rightsquigarrow E' : \tau \qquad P' \,|\, T'(TA_x, x : \sigma) \overset{\mathrm{W}}{\vdash} F \rightsquigarrow F' : \tau'}{P' \,|\, T'TA \overset{\mathrm{W}}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) \rightsquigarrow (\mathbf{let}\ x = \lambda v.E'\ \mathbf{in}\ F') : \tau'}$$
$$\text{where } \sigma = Gen(TA, P \Rightarrow \tau)$$

Figure 3.8: Type inference algorithm with translation.

ming languages where these facilities seem particularly useful. In this section we summarize the key features of some of these earlier systems.

To help further the relationships between different systems of records and variants Figure 3.9 charts a table listing the features supported by a wide selection of authors. Of course, properties supported or not supported by the different systems may be argued by the different authors. For example, many of the systems listed do not have a tick in the implementation row, however, it is clear that a naive implementation exists for any of the record and variant systems—simply use an association list representation, consisting of (label, value) pairs, and implement the primitive operations as necessary. This kind of implementation, of course, is not what we had in mind when ticking (or not ticking) this row. To provide an indication of the kind of properties each row in Figure 3.9 is trying to indicate, the following list provides a short description of our interpretation.

Records: It may come as a surprise to the reader that we have included a row confirming that each system supports some form of records. However, this row serves two purposes; the first is to highlight that records have often been studied alone (not in the presence of variants), and secondly to allow us to select systems

	[Car84]	[Wan87]	[Jat88]	[Rem94a]	[Wan91]	[CM91]	[HP90]	[HP91]	[Oho95]		This thesis
Records	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
Variants	✓			✓					✓		✓
Polymorphic Operations		✓	✓	✓	✓	✓	✓	✓	✓		✓
Extensibility		✓	✓	✓	✓	✓	✓	✓			✓
Most General Unifers			✓	✓					✓		✓
Implementation									✓		✓
Explicitly Typed	✓					✓	✓	✓			✓
Checked Operations			✓			✓	✓	✓	✓		✓
Unchecked Operations	✓	✓		✓	✓						
Concatenation		✓			✓			✓			

Figure 3.9: Type systems for records and variants

that all aim to provide a set of basic operations over records. By this we mean that the basic system supports some notion of records and primitive operations over them, in particular, it does not imply additional features, such as, polymorphic operations or extensiblility—these each have their own row in Figure 3.9. In general, we will use the record row to mean simply that the system provides some form of labelled products.

Variants: A quick glance at Figure 3.9 shows that although all of the systems support some form of records, only a small subset provide similar features for variants. In many cases this seems not to be because the systems can not be extended to provide the additional functionality but rather the authors have not considered it worthwhile or maybe even a trivial addition—noting that variants are dual to records. However, it may be the case that in the underlying theory records and variants are dual, but when considering them within a practical programming language there are important differences. For example, there is the question of syntax, what syntatic lexicons should be used to indicate the use of records and variants, how does the new syntax interact with other features of the language? There are other, possibly not so clear, questions that might need to be considered. For example, Ohori [Oho95] and Gaster [Gas96] have both observed that the use of extensible variants may often have to be annotated with (some) extra type information in practical applications. We believe that this is enough to justify the requirment that variants be described explicitly rather than an author just noting that they are (in theory) just the dual to records.

Polymorphic Operations: Although many of the systems listed in Figure 3.9 provide support for polymorphic operations over records and (or) variants some do not. Of course, one may quote many systems that do not include polymorphic operations over records or variants, but are not included in Figure 3.9—ranging from C to Standard ML. So why include the choices of Figure 3.9, while, admitting others? The answer to this question is not simple and, of course, is subjective. We have tried to include systems that can be seen as having a direct relationship to our field of study. By this we mean systems that are either trying to express similar functionality (e.g., extensibility) or might be considered as fundemental to the ideas developed in later systems. This latter point may be best illustrated by the development of Wand's [Wan87, Wan88] original proposal for extensible records, which was inspired by the early work of Cardelli [Car84] where he gave a semantics for multiple inheritance in terms of record and variant subtyping.

Extensibility: The notion of adding or removing a field from a record, for example. It is important to note that supporting polymorphic operations over records and variants is not enough, on its own, to provide for the additional functionality that is captured through extensibility. An excellent example of this can be seen in Ohori's [Oho95] system, which supports polymorphic operations but lacks the additional expressiveness that extensibility gives. Of course, Ohori's systems has other benefits such as a simple and efficient implementation, while not requiring type annotations.

Most General Unifiers: This dissertation is, in the most part, concerned with type systems for records and variants that support automatic type inference. Following Hindley [Hin69] and Milner [Mil78], we have reduced the problem of type inference to a first-order unification problem and consequently require the existance of an algorithm that given types τ and τ' produces a most general unifier S such that $S\tau = S\tau'$ or else fails. Figure 3.9 indicates which of the different systems has a known first-order unification alogrithm for solving equalities between different expressions in the specified language of types. It is important to note that many of the systems described are explicitly typed (see below) and thus, it probably was not important to the authors whether type inference was possible. This is not the case for all the systems considered in Figure 3.9.

Implementation: As described briefly above it fairly straightforward to describe a naive implementation for records and variants using assocation lists and then implementating each of the required primitives using standard specifications for similar functions over lists. However, in general, this approach does not lead to the efficient implementation of many of the primitives for records and variants. For example, consider the selection of a field l from a record of type

$$Rec\ \{\!|l_1 : \tau_1, \ldots, l : \tau, \ldots, l_n : \tau_n|\!\}.$$

Assuming records are represented as lists of label and value pairs then record selection must be implemented be simply iterating over the list comparing the stored label with the label being selected and return the associated value on matching of to labels—the type system then simply guarantees that this search will never fail. The problem with this approach is that the complexity of $(\_.l)$ is linear in the size of the record (i.e., it is possible that record selection will have to iterate over the complete list to find the required field).

With the above discussion in mind the implementation row included in Figure 3.9 has only been ticked for the case where the author has explicitly described an implementation and furthermore the implementation of record selection is known to be constant time. This is a hard property to satisfy and is not without its own problems. In particular, Rémy [Rém92a] has shown that if one is to consider records as certain forms of hash tables then many of the primitive operations over records can be implemented efficiently. However, there are two problems with Rémy's approach in that some type information must remain at run-time, allowing the calculation and matching of labels. Furthermore, it is not possible, in general, to provide a constant time implementation of record selection, for example. In the special case when the set of possible labels for all records is known at compile-time it may be possible to in-line record selection and expand the hashing functions to determine the index of a given value before execution of the program at run-time.

Explicitly Typed: Many interesting systems that provide support for extensible records and variants are set in an explicitly typed setting, see Cardelli [Car84], Cardelli and Mitchell [CM91], and Harper and Pierce [HP90, HP91], for example. An explicitly typed calculus requires, in general, that terms be annotated with types. Consequently to check that a term is well-formed one need only check that each sub-term is well-formed with respect to a specified inference system. An important consequence of explicit typing is that impredicative polymorphism, as in the polymorphic λ-calculus [Gir72, Rey74], can be supported without loss of decidability [Wel94]. An interesting point to note with reference to Figure 3.9 is the support of an explicitly typed version of the system presented in this chapter. Although we have not given an explicit definition of such a system it can be derived from Jones' [Jon92b] original paper on qualified types.

Checked Operations: In this chapter the primitive operations provided to manipulate records and variants have been checked. For example, the extension operator for a label l can only be applied to a record r if the record r does not contain an l field already. Of course, there are applications when it may be useful to extend a record with a field that might already be present, such operations are often referred to as unchecked. Both Wand [Wan87, Wan88] and Rémy [Rém94a] support unchecked operations, which enables them to capture the semantics of multiple inheritance, as proposed by Cardelli [Car84].

It is important to note that although the type system for extensible records and variants described in this chapter does not support unchecked operations, as indicated in Figure 3.9, it does not mean it cannot provide such functionality. In particular, an unchecked casting operation is described for our system in Chapter 7.

Concatenation: Inspired by the early work of Cardelli [Car84], Wand [Wan87, Wan88] described a system of extensible records, which supported a record concatenation operation for modelling inheritance. There are two different kinds of concatenation operation that may be provided as primitive in any particular system. The first, symmetric concatenation (see Harper and Pierce [HP91], for example) merges two records, which have distinct set of labels, resulting in a new record containing the fields of both records. As the two sets of labels must be disjoint it is clear that symmetric record concatenation must be a checked operation. The second, asymmetric record concatenation is the same as symmetric except that the two sets of labels need not be disjoint. In other words, asymmetric is unchecked record concatenation. To avoid an ambiguous semantics—if a label l appears in both records then we must select one field over the other—most systems that support asymmetric concatenation pick the right field over the corresponding left.

More recently Rémy [Rém94b] has shown that if a particular type system supports checked record extension then symmetric record concatenation can be obtained for free via a simple encoding. Of course, the same encoding provides asymmetric record concatenation if the underlying extension is unchecked. So it is clear that all systems supporting record extension can also support some form of record concatenation, and thus, can be inferred from the extensibility row in Figure 3.9. Consequently the row labelled concatenation in Figure 3.9 is representative of systems that provide record concatenation as primitive rather than building on top of extension.

3.4.1 Subtyping

Subtyping is one of the most widely used techniques for building type systems for records and variants [Car84, CM91, Car92, PT94]. We can define a subtyping relation by specifying that a row r_1 is a subrow of r_2, written $r_1 \leq r_2$, if r_1 contains all the fields of r_2, and possibly more. The intuition here is that, for example, a record of type $Rec\ r_1$ could be used in any context where a value of type $Rec\ r_2$ is expected, and conversely, that a variant of type $Var\ r_2$ can be substituted in any context where a value of type $Var\ r_1$ is required. In particular, the selection operator $(\_.l)$ can be treated as a function of type

$$\forall \alpha . \forall r \leq \{\!|l{:}\alpha|\!\}. Rec\ r \rightarrow \alpha.$$

This operation is implemented, at least conceptually, by coercing from $Rec\ r$ to a known type—the singleton $Rec\ \{\!|l{:}\alpha|\!\}$—and then extracting the required field. One weakness of this approach is that information about the *other* fields in the record is lost, so it is harder to describe operations like record extension. For example, observing that bounded quantification is not by itself sufficient, Cardelli and Mitchell [CM91] used an overriding operator on types to overcome this problem.

3.4.2 Row extension

Motivated by studies of object-oriented programming, Wand [Wan87] introduced the concept of *row variables* to allow incremental construction of record and variant types. For example, a record of type $Rec\ \{\!|l : \tau \,|\, r|\!\}$ has all of the fields of a record of type $Rec\ r$, together with a field l of type τ. Wand did not discuss compilation, but his approach supports both polymorphism and extensibility. For example, the selection operator $(\_.l)$ has type

$$\forall\alpha.\forall r.Rec\ \{\!|l : \alpha \,|\, r|\!\} \to \alpha.$$

However, the operations and types in Wand's system are *unchecked*; for example, extending a row with an l field may either add a completely new field, or replace an existing field labelled with l. As a result, some programs do not have principal types [Wan88].

3.4.3 Flags

Rémy has developed a flexible treatment for extensible records and variants in a natural extension of ML [Rém94a]. A key feature of his system is the use of *flags* to encode both positive and negative information—that is, to indicate which fields must be present, and which must be absent. Again, a concept of row variables is used to deal with other fields whose presence or absence is not significant in a particular situation. For example, the selection operator has type

$$\forall\alpha.\forall r.Rec\ \{\!|l : pre(\alpha) \,|\, r|\!\} \to \alpha,$$

where $pre(\alpha)$ is a flag indicating the presence of a field of type α, and r is a row variable representing the rest of the record. This allows the system to prevent access to undefined components, while being expressive enough to support some unchecked operations. On the other hand, the type system does not lead to a simple and efficient implementation. By this we mean that it is not possible to give an implementation that, in general, does not require some form of run-time tagging. For example, Rémy [Rém92a] describes an implementation for his early system of extensible records and variants based on hash tables. The problem is that hashing is based on the notion of simple key that on application of a hashing function describes where to look up the required value in some abstract datatype for tables. Consequently, it is not possible, in general, to avoid two values being stored at the same index and thus some additional search must be preformed in these cases. This has the affect of requiring some form of tagging, which in effect is simply run-time type information.

3.4.4 Predicates

Harper and Pierce [HP90, HP91] studied type systems for extensible records using predicates on types to capture information about presence or absence of fields, and to restrict attention to checked operations. For example, writing $r_1 \# r_2$ for the assertion that the rows r_1 and r_2 have disjoint sets of labels, the selection operator $(\_.l)$ has type

$$\forall \alpha . \forall r . (r \# \{\!| l : \alpha |\!\}) \Rightarrow \mathit{Rec}\ (r \parallel \{\!| l : \alpha |\!\}) \rightarrow \alpha,$$

where $r_1 \parallel r_2$ is the row obtained by merging r_1 and r_2, and is only defined if $r_1 \# r_2$.

Harper and Pierce's work does not deal with variants, type inference, or compilation, and does not provide an operational interpretation of predicates. However, their approach to record typing was one of the motivating examples in Jones' work on qualified types [Jon94b] where a general framework for type inference and compilation was developed. As a special case, Jones outlined a type system for extensible records, but some of the important details were either omitted or unresolved. For example, he did not address the problems of unifying record types. Indeed, his full system lacks most general unifiers—the result of including record restriction in the type language[2]. One of the achievements of the present work is to refine and extend that work to a practical system [Gas97a].

3.4.5 Kinds

Ohori [Oho95] described a type system that extends SML with polymorphic operations on both records and variants. Significantly, Ohori also presented a simple and effective compilation method: input programs are translated into a target language that adds extra parameters to specify field offsets. In fact, the end result is much the same as that suggested by Jones' work on qualified types, even though the two approaches were developed independently. But Ohori's work differs substantially from other systems in its use of a kind system; this allows variables ranging

[2] A counter-example, which can be used to show the loss of most general unifiers in Jones' system, is given in Chapter 7 Section 7.3.2.

over record types to be annotated with a specification of the fields that they are expected to contain. For example, the selection operator operator $(\_.l)$ has type

$$\forall\alpha.\forall r^{\{\!|l:\alpha|\!\}}.Rec\ r \rightarrow \alpha.$$

The main limitation of Ohori's type system is its lack of support for extensibility.

3.4.6 Constraints

Sulzmann [Sul97], has subsequently described a type system, as an application of $HM(X)$ [SOW97], based upon the early presentations of Ohori [Oho95], Gaster and Jones [GJ96], and the work described in this dissertation. The basic idea is that operations over records can be constrained by predicates of the form $(r\ has\ l : \tau)$, which asserts that the record r contains at least the field l of type τ. As is the case with qualified types [Jon94b] each application of $HM(X)$ includes an entailment relation for proving statements about predicates. The required instance of this relation for predicates that constrain records is similar in spirit to the entailment relation described in this chapter. As an example, the record selection operator, $(\_.l)$, can be assigned the following type in $HM(X)$

$$(\_.l) : \forall r.\forall\alpha.(r\ has\ l : \alpha) \Rightarrow r \rightarrow \alpha.$$

One shortcoming of Sulzmann's work seems to be the lack of a general compilation method. It is clear that the methods of compilation described by Ohori [Oho95] and in this chapter can be applied to the type assigned to record selection. However, it is not clear that this is the case for the complete set of record operations described by Sulzmann, in particular record extension, or more broadly for other applications of $HM(X)$.

Chapter 4

Specialization based semantics

As discussed in the introduction, this chapter and the following are not concerned directly with records and variants. Instead these two chapters study the more general, problem of semantic foundations for qualified types.

This chapter describes a semantics for qualified types nased upon the notion of specialization, which describes the process of overloading elimination.

The sections of this chapter are as follows. Section 4.1 gives an overview of the specialization based semantics for qualified types. Section 4.2 introduces a system of qualified types (OML). Section 4.3 presents two alternative (but equivalent) definitions of core-ML (Polymorphic ML (PML) and Monomorphic ML (MML)), while Section 4.4 introduces an algorithm (Spec) for translating overloaded terms into core-ML. Section 4.5 discusses a semantics for the simply typed λ-calculus ($T\Lambda$) and its relationship to core-ML. This is followed by a formal semantics for qualified types in Section 4.6. Finally, Section 4.7 concludes with a discussion on related work.

An earlier version of this chapter has previously been distributed in the form of a technical report [Gas97b].

4.1 Overview

Polymorphism arises in a variety of forms in computer science. In particular, Strachey observed two kinds of polymorphism that are particularly suited to programming [Str67]. The first, *parametric* polymorphism, captures the fact that certain values behave independently of type—a function, for example, to reverse a list is unconcerned with the type of elements contained within that list. The programming languages C++ (templates), Haskell, Pizza (parameterized classes), and SML, for example, all provide support for parametric polymorphism. The second, *ad-hoc* or *constrained* polymorphism, captures the notion of overloading—a single symbol may have many interpretations. For example, the addition operator (+) is often defined on integer and floating point numbers. A parametric function is well-defined at all instances, while an ad-hoc function is only defined for a subset of instances. Overloading has been incorporated into a wide selection of programming languages including C, C++, Haskell, Java, Pizza, and SML, for example. Unfortunately, each of these languages have different implementations for overloading, depending on alternative extensions to the type system.

Motivated by the early work of Kaes [SK88] and of Wadler and Blott [WB89], the theory of *qualified types*, as described by Jones [Jon92b, Jon94b], provides a general theory for ad-hoc polymorphism and type inference. For example, the (==) operator of SML and Haskell can be assigned the type

$$(==) : \forall \alpha . Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool,$$

where the predicate $Eq\ \alpha$ constrains the instantiation of α to types with an equality operator.

To describe a semantics for qualified types Jones gives a translation for overloaded terms into a variant of Girard and Reynolds' second order polymorphic λ-calculus [Gir72, Rey74]. Inspired by the analogy between propositions and types [How80, Coq90], the translation inserts 'evidence' abstractions and applications on the introduction and elimination of implicit overloading. We have already seen one example of evidence in Chapter 3, where integer offsets were provided as proofs for lacks predicates. As another example, assuming $eqInt : Int \times Int \rightarrow Bool$, consider the expression

$$(\lambda x . x == x)\ 10,$$

which is translated to

$$(\lambda v . \lambda x . v\ (x, x))\ eqInt\ 10.$$

In general, there may be many different translations for any given term. To avoid depending on any translation, Jones provided a coherence result, which, under reasonable assumptions, stated that different translations are in a precise sense equivalent.

Although a reasonable solution the semantics of qualified types depends on the construction of a suitable model for the polymorphic λ-calculus. Such models require structures suitable for modelling explicit impredicative polymorphism, which, in light of the fact that qualified types supports only predicative polymorphism, seems a strong requirement. In particular, constructing a semantics for qualified types based upon Jones' translation rules out any chance of a simple set-based model [Rey84]. We strongly believe that such a model can provide a simple platform for understanding and reasoning about qualified types, using a language familiar to a wide range of computer scientists. As such, a key aim of this chapter is to present a semantics for qualified types, suitable for a simple set-based model.

In this chapter we develop a semantics for qualified types as an extension of Ohori's [Oho89a] work on core-ML. Ohori considered the interpretation of a polymorphic type to be the set of monomorphic instances for that type. For example, the type

$$\forall\alpha.\alpha \to \alpha \to \mathit{Bool},$$

is interpreted as the set

$$\{\tau \to \tau \to \mathit{Bool} | \tau \in \mathit{Type}\}.$$

Ohori's interpretation of polymorphic types extends naturally to constrained types by observing that the set of monomorphic instances for a given polymorphic type can be generated with respect to some predicate. For example, the constrained type

$$\forall\alpha.\mathit{Eq}\ \alpha \Rightarrow \alpha \to \alpha \to \mathit{Bool},$$

can be interpreted as

$$\{\tau \to \tau \to \mathit{Bool} | \tau \in \mathit{Type}, \tau \in \mathit{Eq}\},$$

where $\tau \in \mathit{Eq}$ asserts the existence of equality over τ.

Inspired by the work of Wadler and Blott [WB89, Blo92], we interpret overloaded terms as sugar for more verbose core-ML terms. However, unlike the translation of Wadler and Blott we introduce a notion of specialization, translating away any nested polymorphism introduced through overloading. Thus, unlike the work of Jones [Jon94b], we avoid translation into the polymorphic λ-calculus and instead have any model of the simply typed λ-calculus at our disposal. In particular, one can construct a simple set-based model for qualified types.

To our knowledge, we are the first to present a direct semantics, supported by a proof of soundness, for qualified types. The following chapter extends this further, providing a complete categorical treatment of constrained first-order polymorphism.

$$
\begin{tikzcd}
OML \arrow[r, shift left, "\mathrm{Spec}"] & PML \arrow[l, shift left, "\supset"] \arrow[r, Leftrightarrow] & MML \arrow[r, shift left, "\mathrm{TT}"] & T\Lambda \arrow[l, shift left, "\supset"]
\end{tikzcd}
$$

Figure 4.1: Roadmap.

Figure 4.1 provides a simple ‘roadmap’ highlighting the relationships between the different systems introduced in this chapter.

4.2 A system of qualified types

As discussed in the introduction to this chapter, qualified types provides a general framework for type systems with constrained types. In general, such a system requires the notion of a predicate, introduced to capture restrictions on the way a term can be used. Section 4.2.1 expands this notion by introducing the idea of an entailment relation over predicates, for which the entailment relation of Chapter 3 is an instance. Section 4.2.2 introduces the notion of evidence, providing a language of expressions for predicates. Section 4.2.3 outlines a variation on the general theory of qualified types, denoted as *OML*.

4.2.1 Predicates

The general theory of qualified types is motivated by the requirement to include predicates in the type of a term, enforcing restrictions on the way it may be used. The exact form of individual predicates is not significant but, in practical applications, they are often written using expressions of the form $p\ \tau_1 \cdots \tau_n$ where p is a predicate symbol corresponding to an n-place relation between types. One can read the predicate $p\ \tau_1 \cdots \tau_n$ as assertion that the types denoted by the type expressions $\tau_1, \cdots, \tau_n$ are in the relation p. In general, we impose only one condition on the set of predicates: it must be closed under substitutions mapping type variables to type expressions. That is, for any substitution S and predicate π of the form $p\ \tau_1 \cdots \tau_n$, the expression

$$S\pi = p\ (S\tau_1) \cdots (S\tau_n),$$

must also be a predicate.

There are various examples of predicates which apply to a wide variety of applications. Of course, the predicate $r \backslash l$ introduced in chapter 3 is one such application. Many other applications have also been studied, including the Haskell class hierarchy [PH97], subtyping [Jon94b] and a general set of operations over lattices [Jon92a].

Predicates have a number of formal properties specified by an entailment relation, written $\Vdash$. This relation is read such that if P and Q are sets of predicates, then $P \Vdash Q$, asserts that the predicates Q can be proved from the hypothesis P using the axioms and rules of the predicate system. In general, we require that entailment is monotonic (i.e, if $P \supseteq Q$, then $P \Vdash Q$), transitive (i.e., if $R \Vdash P$ and $P \Vdash Q$, then $R \Vdash Q$), and closed under substitutions (i.e., if $P \Vdash Q$ and S is a substitution, then $SP \Vdash SQ$). These, are typically, taken as part of the definition of predicate entailment in specific applications.

4.2.2 Evidence

The previous section introduced the notion of a predicate, describing an entailment relation for asserting properties about predicates. However, the term language for qualified types, to be described in the following section, requires *evidence* for a predicate. In a logical setting evidence e provides proof for predicate π, and is analogous to the Curry-Howard isomorphism [How80], where terms and types of the simply typed λ-calculus correspond to proofs and propositions in constructuve propositional logic.

Unlike the definition of predicates, we make no assumptions about the form of evidence expressions. We let e range over sequences of evidence expressions, while v and w correspond to sequences of evidence variables. As before, predicates may have a number of formal properties specified by an entailment relation, written $\Vdash$. This relation is read such that if $v : P$ and $w : Q$ are sets of predicates, then $v : P \Vdash w : Q$, asserts that the predicates $w : Q$ can be proved from the hypothesis $v : P$ and the axioms of the predicate system. An important point to note is that the predicate sets P and Q must correspond to sequences of predicates. Again, we require only that, in general, entailment is monotonic, transitive, and closed under substitutions. It will be convenient in later sections, and in describing a categorical semantics for qualified types, to reformulate entailment in terms of an inference system. A set of inference rules for predicate entailment are given in Figure 4.2.

$$
\begin{array}{ll}
(id) & v : P \Vdash v : P \\
(term) & v : P \Vdash \emptyset \\
(fst) & v : P, w : Q \Vdash v : P \\
(snd) & v : P, w : Q \Vdash w : Q \\
(univ) & \dfrac{v : P \Vdash e : Q \quad v : P \Vdash e' : R}{v : P \Vdash e : Q, e' : R} \\
(trans) & \dfrac{v : P \Vdash e : Q \quad v' : Q \Vdash e' : R}{v : P \Vdash [e/v']e' : R} \\
(close) & \dfrac{v : P \Vdash e : Q}{Sv : SP \Vdash e : SQ} \\
(evars) & \dfrac{v : P \Vdash e : Q}{EV(e) \subseteq v}
\end{array}
$$

Figure 4.2: Predicate entailment with evidence.

4.2.3 *OML*

Following Damas and Milner [DM82], we distinguish between the simple types, τ, and type schemes, σ, described by the grammar below

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \iota \mid \tau \to \tau & \textit{monotypes} \\
\rho & ::= & P \Rightarrow \tau & \textit{qualified types} \\
\sigma & ::= & \forall\{\alpha_i\}.\rho & \textit{type schemes}
\end{array}
$$

The symbols α and ι range over sets of type variables and base types, respectively. Restrictions on the instantiation of universal quantifiers, and hence on polymorphism, are described by encoding the required constraints as a sequence of predicates, P, in a qualified type of the form $P \Rightarrow \tau$.

The term language of *OML* is just core-ML, an implicitly typed λ-calculus, extended with constants, evidence abstraction and application

$$
\begin{array}{rcll}
E & ::= & x & \textit{variables} \\
& | & c & \textit{constants} \\
& | & EF & \textit{application} \\
& | & \lambda x.E & \textit{abstraction} \\
& | & \mathbf{let}\ x = E\ \mathbf{in}\ F & \textit{local bindings} \\
& | & \lambda v.E & \textit{evidence abstraction} \\
& | & Ee & \textit{evidence application.}
\end{array}
$$

Unlike other presentations of qualified types, we make evidence abstraction and application explicit. The advantage of making evidence explicit is that it allows more insight into how qualified types really work and helps in the development of a semantics for *OML*.

The typing rules are presented in Figure 4.3. A judgement of the form $P \mid C, A \vdash E : \sigma$ represents an assertion that, if the predicates in P hold, then the term E has type σ, using assumptions in C and A to provide types for the free overloaded and non-overloaded variables, respectively. These are just the standard rules for qualified types [Jon94b], extending the rules of Damas and Milner [DM82] to account for the use of predicates.

An equation in *OML*, denoted $P|C, A \vdash E = F : \tau$, is a pair of terms E and F with $P \mid C, A \vdash E : \tau$ and $P \mid C, A \vdash F : \tau$. An equational theory for *OML*, written $\mathcal{T}_{OML}$, includes the standard rules for equational reasoning (i.e., reflexivity, symmetry, transitivity, and congruence) and for core-ML (i.e., (α), (β), (η), and a rule for **let** equivalence). In addition it should contain rules for reasoning about evidence expressions (i.e., (α_e), (β_e) and (η_e)). Figure 4.4 contains some of the rules for proving equalities between *OML* terms.

A theorem is provable, denoted $\mathcal{T}_{OML} \vdash P|C, A \vdash E = F : \tau$, if it is derivable from the standard rules and the equations of $\mathcal{T}_{OML}$.

4.3 Core-ML

In this section we define two inference systems for core-ML. The first, Polymorphic ML (*PML*), defined in Section 4.3.1, is a variation on the inference system given by Damas and Milner [DM82]. Section 4.3.2 gives a reformulation of this system,

$$(const)\quad \frac{(x:\sigma)\in C}{P\,|\,C,A\vdash x:\sigma}$$

$$(var)\quad \frac{(x:\sigma)\in A}{P\,|\,C,A\vdash x:\sigma}$$

$$(\to E)\quad \frac{P\,|\,C,A\vdash E:\tau'\to\tau \quad P\,|\,C,A\vdash F:\tau'}{P\,|\,C,A\vdash EF:\tau}$$

$$(\to I)\quad \frac{P\,|\,C,A_x,x:\tau'\vdash E:\tau}{P\,|\,C,A\vdash \lambda x.E:\tau'\to\tau}$$

$$(\Rightarrow E)\quad \frac{P\,|\,C,A\vdash E:\pi\Rightarrow\rho \quad P\Vdash e:\pi}{P\,|\,C,A\vdash Ee:\rho}$$

$$(\Rightarrow I)\quad \frac{P,v:\pi\,|\,A\vdash C,E:\rho}{P\,|\,C,A\vdash \lambda v.E:\pi\Rightarrow\rho}$$

$$(\forall E)\quad \frac{P\,|\,C,A\vdash E:\forall\alpha.\sigma}{P\,|\,C,A\vdash E:[\tau/\alpha]\sigma}$$

$$(\forall I)\quad \frac{P\,|\,C,A\vdash E:\sigma \quad \alpha\not\in TV(A)\cup TV(P)}{P\,|\,C,A\vdash E:\forall\alpha.\sigma}$$

$$(let)\quad \frac{P\,|\,C,A\vdash E:\sigma \quad Q\,|\,C,x:\sigma,A_x\vdash F:\tau}{P,Q\,|\,C,A\vdash (\mathbf{let}\ x=E\ \mathbf{in}\ F):\tau}$$

Figure 4.3: Typing rules for *OML*.

which we call Monomorphic ML (*MML*), such that it uses only monomorphic types, and thus can be translated directly into the simply typed λ-calculus. Following Mairson [Mai92], we show that a typing judgement in *PML* is valid if and only if an equivalent typing judgement is valid in *MML*.

4.3.1 *PML*

The types of *PML* are just the types of *OML* that contain no predicates, and thus the terms of *PML* are simply the terms of *OML* without evidence abstraction or

$$
\begin{array}{ll}
(\beta) & P|C, A \vdash (\lambda x.E)\ F = [F/x]E : \tau \\
(\beta_e) & P|C, A \vdash (\lambda v.E)\ e = [e/v]E : \tau \\
(\beta\text{-}let) & P|C, A \vdash \textbf{let}\ x = E\ \textbf{in}\ F = [E/x]F : \tau \\
(\eta) & \dfrac{x \notin FV(E)}{P|C, A \vdash (\lambda x.Ex) = E : \tau} \\
(\eta_e) & \dfrac{v \notin EV(E)}{P|C, A \vdash (\lambda v.Ev) = E : \tau}
\end{array}
$$

Figure 4.4: *OML* equality.

application. The set of well-typed *PML* terms is defined by the *OML* inference system, without the rules for typing evidence abstraction and application (i.e., ($\Rightarrow I$) and ($\Rightarrow E$)), and the rule for overloaded constant introduction (i.e., (*const*)). Although the well-typed terms of *PML* are those identifiable in the original system proposed by Damas and Milner [DM82], this is not the case for the types. One problem here is that the types of *PML* contain empty predicates sets, which are not present in Damas and Milner's definition. However, in practice this does not cause any problems due to the fact that we can simply eliminate the empty predicate set (or, dually, insert the empty predicate set) to get from one system of types to the other.

PML equations and theories, denoted $A \vdash E =_{PML} F : \tau$ and $\mathcal{T}_{PML}$, respectively, are simply the equations and theories of *OML* without the rules for evidence.

A formal definition of *PML* appears in the lefthand column of Figure 4.5.

4.3.2 *MML*

Following the definition of *PML* in the previous section, we define a monomorphic version of core-ML as follows: *MML* terms are the terms of *PML* and the types are the monotypes of *PML*. Well-typed terms are formed from the rules of *PML*, noting that, as support is provided for monotypes only, the inference system of *MML* does not contain rules for universal quantification introduction and elimination. However, due to this restriction, it is impossible, in *MML*, to type let-polymorphism without altering the inference rule for let-bindings. As in the

PML	*MML*
$\tau \ ::= \ \alpha \mid \iota \mid \tau \to \tau$	$\tau \ ::= \ \alpha \mid \iota \mid \tau \to \tau$
$\sigma \ ::= \ \forall\{\alpha_i\}.\tau$	
$E \ ::= \ x$	$E \ ::= \ x$
$\mid \ EE$	$\mid \ EE$
$\mid \ \lambda x.E$	$\mid \ \lambda x.E$
$\mid \ \mathbf{let}\ x = E\ \mathbf{in}\ E$	$\mid \ \mathbf{let}\ x = E\ \mathbf{in}\ E$
$\dfrac{(x : \tau) \in A}{A \vdash x : \tau}$	$\dfrac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$
$\dfrac{A \vdash E : \tau' \to \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau}$	$\dfrac{\Gamma \vdash E : \tau' \to \tau \quad \Gamma \vdash F : \tau'}{\Gamma \vdash EF : \tau}$
$\dfrac{A_x, x : \tau' \vdash E : \tau}{A \vdash \lambda x : \tau'.E : \tau' \to \tau}$	$\dfrac{\Gamma_x, x : \tau' \vdash E : \tau}{\Gamma \vdash \lambda x : \tau'.E : \tau' \to \tau}$
$\dfrac{A \vdash E : \sigma \quad A, x : \sigma \vdash F : \tau}{A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$	$\dfrac{\Gamma \vdash E : \tau' \quad \Gamma \vdash [E/x]F : \tau}{\Gamma \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$
$\dfrac{\Gamma \vdash E : \forall\alpha.\sigma}{\Gamma \vdash E : [\tau/\alpha]\sigma}$	
$\dfrac{\Gamma \vdash E : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash E : \forall\alpha.\sigma}$	

Figure 4.5: *PML* and *MML*.

work of Ohori [Oho89a] and Mairson [Mai92], we replace the inference rule for **let** with the following monomorphic version

$$\frac{\Gamma \vdash E : \tau' \quad \Gamma \vdash [E/x]F : \tau}{\Gamma \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau.}$$

A formal definition of *MML* appears in the righthand column of Figure 4.5, to facilitate ease of comparsion of *MML* with *PML*.

Equational theories for *MML* are defined in the same way to those for *PML* and we leave the details to the reader.

The following result shows that, although *MML* provides only support for monotypes, the alternative rule for let-bindings is enough to regain the expressness of *PML*. This provides support for our interpretation of let-polymorphism in the simply typed λ-calculus. Following Mairson [Mai92], we define what it means for the systems *PML* and *MML* to be related and then state the required theorem.

Definition 4.1 *Let* $\Gamma' = \{x_1 : \tau_1, \cdots, x_m : \tau_m\}$ *be any context of monotype bindings, and* $\Gamma = \{y_1 : \sigma_1, \cdots, y_n : \sigma_n\}$ *be any context of polymorphic bindings. Let* $F = \{F_1, \cdots, F_n\}$ *be a set of terms where* σ_i *is the principal type of* F_i*; in particular, we require that for* $1 \leq i \leq n$*,*

$$\textit{if } \Gamma', \{y_1 : \sigma_1, \cdots, y_{i-1} : \sigma_{i-1}\} \vdash^{PML} F_i : \sigma_i,$$

$$\textit{then } \Gamma' \vdash^{MML} [F_1/y_1] \cdots [F_{i-1}/y_{i-1}]F_i : \overline{\sigma_i},$$

where $\overline{\sigma_i}$ *is* σ_i *with all the quantifiers removed.*

We now state the theorem of equivalence for *PML* and *MML*.

Theorem 4.2 (Mairson [Mai92]) *Let* E *be any PML or MML term,* τ *a monomorphic type, and* Γ', Γ, *and* $F = \{F_1, \ldots, F_n\}$ *be defined as in Definition 4.1; then*

$$\Gamma, \Gamma' \vdash^{PML} E : \tau \iff \Gamma' \vdash^{MML} [F_1/y_1] \cdots [F_n/y_n]E : \tau.$$

A proof of this result is given by Mairson [Mai92] and we do not reproduce the details here.

4.4 Specialization from *OML* to *PML*

In this section we describe an algorithm to translate *OML* terms into *PML*, providing a stepping stone to our semantic definition for *OML*. Following Jones [Jon94a], specialization is described using expressions of the form $x\ e \rightsquigarrow x'$ for variables x and x' (new) and evidence e. For any given program, several monomorphic versions of a polymorphic function may be required, thus, we work with finite sets of expressions of the form $x\ e \rightsquigarrow x'$ called specialization sets. Any specialization set S must satisfy the constraint

$$(x\ e \rightsquigarrow x'), (y\ e' \rightsquigarrow x') \in S \Rightarrow x = y \wedge e = e',$$

ensuring that S is well-formed. This constraint is exactly the restriction required to ensure that any specialization set S can be interpreted as a substitution where each $(x\ e \rightsquigarrow x') \in S$ represents the substitution of the expression $x\ e$ for the variable x'. For example, applying the specialization set $\{x\ e \rightsquigarrow x'\}$ to the expression $(\lambda x.\lambda y.x)\ x$ gives $(\lambda x.\lambda y.x)\ x\ e$.

We introduce some special notation for working with specialization sets

- If V is a set of variables, then S_V is the set

$$S_V \quad = \quad \{(x\ e \rightsquigarrow x') \in S \mid x \notin V\}.$$

  As a special case, we write S_x as an abbreviation for $S_{\{x\}}$.

- The relation *extends* defines the specialization sets that can be obtained from a given set S, but with different specializations for variables bound in a given B

$$S' \ \mathit{extends}\ (B, S) \Longleftrightarrow \exists S''. \mathit{Vars}\ S'' \subseteq \mathit{dom}\ B \wedge S' = S_{(\mathit{dom}\ B)} \cup S''.$$

  We require that *extends* is restricted such that, for any set of bindings B and specialization set S, if S' *extends* (B, S) and S'' *extends* (B, S) then $S' = S''$.

The specialization algorithm is defined by the rules in Figure 4.6. A judgement of the form $S \vdash E \rightsquigarrow E'$ asserts that, under the specialization set S, E' represents E without overloaded let-bindings and evidence redexes. We write $S \vdash A \rightsquigarrow A'$ if, for each $(f\ e \rightsquigarrow f') \in S$, $P \mid C, A' \vdash f' : \sigma$, for some *PML* type σ, bindings C, and predicates P.

The judgement of the form $S, S' \vdash B \rightsquigarrow B'$ used in the anticedent of the rule (*let*) describes the process of specializing a group of bindings B with respect to a pair of specialization sets S and S' to obtain a set of bindings B' that are evidence free.

$(var\text{-}\lambda)$	$\dfrac{x \notin S}{S \vdash x \rightsquigarrow x}$
$(var\text{-}let)$	$\dfrac{(x\ e \rightsquigarrow x') \in S \quad e \Longrightarrow d}{S \vdash x\ e \rightsquigarrow x'}$
(β_{evi})	$\dfrac{S \vdash [e/v]E \rightsquigarrow E'}{S \vdash (\lambda v.E)e \rightsquigarrow E'}$
$(abs\text{-}evi)$	$\dfrac{S_v \vdash E \rightsquigarrow E'}{S \vdash \lambda v.E \rightsquigarrow \lambda v.E'}$
(app)	$\dfrac{S \vdash E \rightsquigarrow E' \quad S \vdash N \rightsquigarrow N'}{S \vdash E\ N \rightsquigarrow E'\ N'}$
(abs)	$\dfrac{S_x \vdash E \rightsquigarrow E'}{S \vdash \lambda x.E \rightsquigarrow \lambda x.E'}$
(let)	$\dfrac{S, S' \vdash B \rightsquigarrow B' \quad S' \vdash E \rightsquigarrow E' \quad S'\ extends(B, S)}{S \vdash let\ B\ in\ E \rightsquigarrow let\ B'\ in\ E'}$

Figure 4.6: Specialization algorithm for *OML*.

This process is defined by

$$S, S' \vdash B \rightsquigarrow B' \Longleftrightarrow B' = \{x' = N' \mid \begin{array}{l} (x = \lambda v.N) \in B \\ \wedge (x\ e \rightsquigarrow x') \in S' \\ \wedge S \vdash [e/v]N \rightsquigarrow N'\}. \end{array}$$

Note that we now work with sets of bindings, in **let** expressions, which differs from the syntax of Section 4.2.3. However, this does not introduce any new complications as the syntax used throughout this section is a simple generalisation of the syntax defined in Section 4.2.3.

The judgement $e \Longrightarrow d$ used in the hypothesis of the $(var\text{-}let)$rules implies compile-time evaluation of the evidence expressions e to the evidence constants d. Jones [Jon94a] required this rule to ensure that this calculation can always be carried out without risk of non-termination, allowing the specialization algorithm to be part of pratical compiler—at least one that would not fail to terminate due to specialization. A more fundemental consequence of restricting evidence expressions, such that they can be evaluated at compile time, is the failure to capture polymorphic recursion in the presence of specialization. To see that this is the case consider the following Haskell definition:

$$\begin{array}{lcl} \mathit{foo} & : & \mathit{Eq\ alpha} \Rightarrow \alpha \rightarrow \mathit{Bool} \\ \mathit{foox} & = & x == x \ \&\& \ \mathit{foo}\ [x] \end{array}$$

This can not be typed under the standard Damas and Milner type system since the function *foo* is used at two different types within the its own body. However, Haskell uses the type definition of *foo* to provide a polymorphic type for *foo* within its own definition. As a consequence the set of evidence constants that are required to evaluate *foo True* is infinite and the specialization algorithm will not terminate with this program.

As an example, of specialization consider the expression

$$\begin{array}{l} \mathbf{let}\ \mathit{eq}\ x\ y = x == y \\ \mathbf{in}\ (\mathit{eq}\ \mathit{True}\ \mathit{False},\ \mathit{eq}\ 'a'\ 'a'), \end{array}$$

where $(==) : Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$ and the overloaded function *eq* is used at two distinct instances *Bool* and *Char*. Applying specialization results in an expression of the form

$$\begin{array}{ll} \mathbf{let} & \mathit{eqBool}\ x\ y = \mathit{primEqBool}\ x\ y \\ & \mathit{eqChar}\ x\ y = \mathit{primEqChar}\ x\ y \\ \mathbf{in} & (\mathit{eqBool}\ \mathit{True}\ \mathit{False}, \\ & \mathit{eqChar}\ 'a'\ 'a'), \end{array}$$

where $\mathit{primEqBool} : \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$ and $\mathit{primEqChar} : \mathit{Char} \rightarrow \mathit{Char} \rightarrow \mathit{Bool}$ represent primitive equality operators over booleans and characters, respectively.

The following proposition, extending an earlier result of Jones [Jon94a], establishes the correctness of the specialization algorithm.

Proposition 4.3 *If* $P \mid C, A \vdash E : \tau$, $S \vdash E \leadsto E'$, *and* $S \vdash A \leadsto A'$, *then* $P \mid C, A' \vdash E' : \tau$ *and* $P|C, A \vdash E = SE' : \tau$.

A proof of this result is given in Section A.2.1 of Appendix A.

We conclude this section with a coherence result for specialization, which shows that, if an expression E specializes to two different evidence free expressions, then the resulting expressions are equal. This will play a fundamental role in the proof of soundness for qualified types.

Proposition 4.4 *If* $P \mid C, A \vdash E : \tau$, $S \vdash A \rightsquigarrow A'$, $S \vdash E \rightsquigarrow F$, *and* $S \vdash E \rightsquigarrow F'$, *then* $P|C, A' \vdash F = F' : \tau$.

A proof of this result is given in Section A.2.2 of Appendix A.

4.5 Simply typed λ-calculus

This section provides a formal presentation of Church's simply typed λ-calculus [Chu40], denoted $T\Lambda$, and its relationship with the calculi defined in previous sections. We split the presentation into two parts: Section 4.5.1 gives a static and denotational semantics for $T\Lambda$, and Section 4.5.2 outlines the relationship between *OML*, *PML*, *MML*, and $T\Lambda$.

4.5.1 $T\Lambda$

The types of $T\Lambda$ are just the ground types of *MML* extended with product types[1], given by the following grammar

$$\tau \quad ::= \quad \iota \mid \tau \to \tau \mid \tau \times \tau \mid ().$$

We write $Type_{T\Lambda}$ for this set, dropping the $T\Lambda$ when it follows from context.

The term language of $T\Lambda$ is just an explicitly typed λ-calculus extended with constants and products, as described by the following grammar

$$E \quad ::= \quad x \mid c \mid EE \mid \pi_i\ E \mid (E, E) \mid () \mid \lambda x : \tau.E.$$

The typing rules are presented in Figure 4.7. A judgement of the form $A \vdash E : \tau$ represents an assertion that the term E has type τ, using the assumptions in A to provide types for free variables.

[1] We include products here as they are required in the semantic definition of $T\Lambda$. It is, of course, possible to include products in the definitions of *OML*, *PML*, and *MML*, but, for simplicity we have avoided this step.

$$(var)^{T\Lambda} \quad \frac{(x : \tau) \in A}{A \vdash x : \tau}$$

$$(\rightarrow E)^{T\Lambda} \quad \frac{A \vdash E : \tau' \rightarrow \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau}$$

$$(\rightarrow I)^{T\Lambda} \quad \frac{A_x, x : \tau' \vdash E : \tau}{A \vdash \lambda x : \tau'.E : \tau' \rightarrow \tau}$$

$$(E\times)^{T\Lambda} \quad \frac{A \vdash E : \tau \times \tau'}{A \vdash \pi_1\ E : \tau}$$

$$(\times E)^{T\Lambda} \quad \frac{A \vdash E : \tau \times \tau'}{A \vdash \pi_2\ E : \tau}$$

$$(\times I)^{T\Lambda} \quad \frac{A \vdash E : \tau \quad A \vdash F : \tau'}{A \vdash (E, F) : \tau \times \tau'}$$

$$(unit)^{T\Lambda} \quad A \vdash () : ()$$

Figure 4.7: Typing rules for $T\Lambda$.

An equation in $T\Lambda$, written $A \vdash E =_{T\Lambda} F : \tau$, is a four tuple with a typing environment A, a type τ, and a pair of terms E and F such that, $A \overset{T\Lambda}{\vdash} E : \tau$ and $A \overset{T\Lambda}{\vdash} F : \tau$. An equational theory for $T\Lambda$, written $\mathcal{T}_{T\Lambda}$, is defined analogously to those for *OML*, *PML*, and *MML* and is combined with the standard rules (α), (β), (η), and (ξ) of the simply typed λ-calculus, and the standard rules for equational reasoning (i.e., reflexivity, symmetry, transitivity, and congruence). We say a theorem is provable, written $\mathcal{T}^{T\Lambda} \vdash_{T\Lambda} A \vdash E =_{T\Lambda} F : \tau$, if it is derivable from the standard rules and the equations of $\mathcal{T}_{T\Lambda}$.

Following Crole [Cro93], we define models of $T\Lambda$ in terms of cartesian closed categories.

Definition 4.5 *A categorical model for $T\Lambda$ consists of*

- *A cartesian closed category $\mathbb{C}$, as defined in Definition B.3 of Appendix B.*
- *A structure, $\mathcal{M}$, in $\mathbb{C}$ over $T\Lambda$, defined by*

– *for every ground type* ι *a non-empty object* $\mathcal{M}[\![\iota]\!]$ *of* $\mathbb{C}$,
– *for every constant function symbol* $c : \tau$, *a global element* $\mathcal{M}[\![c]\!] : \mathbb{1} \to \mathcal{M}[\![\tau]\!]$, *where*

$$\begin{array}{lcl} \mathcal{M}[\![\tau \to \tau']\!] & = & [\mathcal{M}[\![\tau]\!] \to \mathcal{M}[\![\tau']\!]] \\ \mathcal{M}[\![\tau \times \tau']\!] & = & \mathcal{M}[\![\tau]\!] \times \mathcal{M}[\![\tau']\!] \\ \mathcal{M}[\![()]\!] & = & \mathbb{1}. \end{array}$$

As a notational convenience we may write $[\![\_]\!]$, *when in fact we really mean* $\mathcal{M}[\![\_]\!]$. *The meaning of a context* A, *is defined inductively by*

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \mathbb{1} \\ [\![A, x : \tau]\!] & = & [\![A]\!] \times [\![\tau]\!]. \end{array}$$

For every judgement $A \vdash^{T\Lambda} M : \tau$ *we specify an arrow* $[\![A \vdash^{T\Lambda} M : \tau]\!] : [\![A]\!] \to [\![\tau]\!]$ *in* $\mathbb{C}$. *The semantics of* $T\Lambda$ *terms are specified inductively over the typing rules of Figure 4.7 as follows*[2]

$$\begin{array}{lcl} [\![A, x : \tau \vdash x : \tau]\!] & = & \pi_2 \\ [\![A, y : \tau \vdash x : \tau]\!] & = & [\![A \vdash x : \tau]\!] \circ \pi_1 \\ [\![A \vdash c : \tau]\!] & = & [\![c]\!] \circ !_{[\![A]\!]} \\ [\![A \vdash \lambda x : \tau'.E : \tau' \to \tau]\!] & = & \mathit{curry}([\![A_x, x : \tau' \vdash E : \tau]\!]) \\ [\![A \vdash EF : \tau]\!] & = & \mathit{eval} \circ \langle [\![A \vdash E : \tau' \to \tau]\!], [\![A \vdash F : \tau]\!] \rangle \\ [\![A \vdash (E, F) : \tau \times \tau']\!] & = & \langle [\![A \vdash E : \tau]\!], [\![A \vdash F : \tau']\!] \rangle \\ [\![A \vdash \pi_1\ E : \tau]\!] & = & \pi_1 \circ [\![A \vdash E : \tau \times \tau']\!] \\ [\![A \vdash \pi_2\ E : \tau']\!] & = & \pi_2 \circ [\![A \vdash E : \tau \times \tau']\!] \\ [\![A \vdash () : ()]\!] & = & !_{[\![A]\!]}. \end{array}$$

An equation $A \vdash E =_{T\Lambda} F : \tau$ is valid in $\mathcal{M}$, denoted $\mathcal{M} \models_{T\Lambda} A \vdash E =_{T\Lambda} F : \tau$, if $[\![A \vdash^{T\Lambda} E : \tau]\!]$ and $[\![A \vdash^{T\Lambda} F : \tau]\!]$ are equal arrows in $\mathbb{C}$. We say that the structure $\mathcal{M}$ is a model for $T\Lambda$, if every equation $A \vdash E =_{T\Lambda} F : \tau$ is valid in $\mathcal{M}$. We have the following soundness and completeness of equational theories for $T\Lambda$

Theorem 4.6 (Lambek [Lam80]) *Let* $\mathbb{C}$ *be a cartesian closed category,* $\mathcal{T}_{T\Lambda}$ *an equational theory, and* $\mathcal{M}$ *a model of* $\mathcal{T}_{T\Lambda}$ *in* $\mathbb{C}$. *Then*

$$A \vdash E =_{T\Lambda} F : \tau \iff \mathcal{M} \models_{T\Lambda} A \vdash E =_{T\Lambda} F : \tau.$$

[2]We use the symbol $!_A$ to denote the unique arrow $! : A \to \mathbb{1}$.

A proof of this result is given by Lambek and Scott [LS86] and we do not reproduce the details here.

4.5.2 Relationship between *MML* and $T\Lambda$

In this section, we define a translation, TT (a mnemonic for TypedTerm), from *MML* derivations to $T\Lambda$ terms. The rules of Figure 4.8 describe this translation process for all possible *MML* derivations. A judgement of the form $A \vdash E \rightsquigarrow E' : \tau$ asserts that the function TT is applied to a derivation of the form $A \vdash^{MML} E : \tau$ and results in the $T\Lambda$ term $A \vdash^{T\Lambda} E' : \tau$. We may sometimes write $TT(\Delta)$ to mean the result of applying the rules in Figure 4.8 to a typing derivation Δ. As an example, of rules in Figure 4.8 consider the *MML* judgement $A \vdash^{MML} \lambda x.\lambda y.x : \tau \to \tau' \to \tau$, which by application of the rules $(\to I)^{TL}$, $(\to I)^{TL}$, and $(var)^{TL}$results in the $T\Lambda$ expression $\lambda x : \tau.\lambda y : \tau'.x$.

$$(var)^{TL} \quad \frac{(x : \tau) \in A}{A \vdash x \rightsquigarrow x : \tau}$$

$$(\to E)^{TL} \quad \frac{A \vdash E \rightsquigarrow E' : \tau' \to \tau \quad A \vdash F \rightsquigarrow F' : \tau'}{A \vdash EF \rightsquigarrow E'F' : \tau}$$

$$(\to I)^{TL} \quad \frac{A_x, x : \tau' \vdash E \rightsquigarrow E' : \tau}{A \vdash \lambda x.E \rightsquigarrow \lambda x : \tau'.E' : \tau' \to \tau}$$

$$(let)^{TL} \quad \frac{A \vdash E \rightsquigarrow E' : \tau' \quad A \vdash [E/x]F \rightsquigarrow F' : \tau}{A \vdash (\textbf{let } x = E \textbf{ in } F) \rightsquigarrow F' : \tau}$$

Figure 4.8: Translation from *MML* to $T\Lambda$.

The following result, due to Ohori [Oho89a], asserts that, although there may be many derivations for a given *MML* typing judgement, they all have the same semantic meaning.

Proposition 4.7 (Ohori [Oho89a]) *If Δ_1 and Δ_2 are derivations of the same typing judgement $A \vdash^{MML} E : \tau$, then*

$$A \vdash TT(\Delta_1) =_{T\Lambda} TT(\Delta_2) : \tau.$$

A proof of this result is given by Ohori [Oho89b] and we do not reproduce the details here.

Define the type erasure of a $T\Lambda$ expression E, denoted by E^*, as follows

$$\begin{array}{lcl} x^* & = & x \\ (EE')^* & = & E^* E'^* \\ (\lambda x : \tau . E^*) & = & \lambda x . E^* . \end{array}$$

The following two propositions capture the relationship between MML and $T\Lambda$, showing that if an expression is well-typed in one of the systems, then translation of the same term is well-typed in the other system.

Proposition 4.8 (Ohori [Oho89a]) *If* $A \vdash^{T\Lambda} E : \tau$ *then there is a derivation* Δ *of*

$$A \vdash^{MML} E^* : \tau,$$

where

$$A \vdash TT(\Delta) =_{T\Lambda} E : \tau .$$

A proof of this result is given by Ohori [Oho89b] and we do not reproduce the details here.

Proposition 4.9 (Ohori [Oho89a]) *If* Δ *is a typing derivation of* $A \vdash^{MML} E : \tau$*, then*

$$A \vdash^{T\Lambda} TT(\Delta) : \tau$$

and

$$A \vdash TT(\Delta)^* =_{MML} \mathit{LetExpand}(E) : \tau ,$$

where $\mathit{LetExpand}(E)$ *is the same as the expression* E*, except that all let bindings have been expanded (see Ohori for details [Oho89a]).*

A proof of this result is given by Ohori [Oho89b] and we do not reproduce the details here.

Straightforward corollaries of these two results (in combination with Theorem 4.2 and Proposition 4.3) highlight the relationship between *OML*, *PML*, and $T\Lambda$, showing that if a term is well-typed in one system than a translated version is well-typed in the other systems.

Corollary 4.10 *Given* $\Gamma = \{y_1 : \sigma_1, \cdots, y_n : \sigma_n\}$ *and* Γ' *such that for each* σ_i *there exists an*

$$\Gamma', \{y_1 : \sigma_1, \cdots, y_{i-1} : \sigma_{i-1}\} \overset{PML}{\vdash} F_i : \sigma_i.$$

Then if $\Gamma, \Gamma' \overset{PML}{\vdash} E : \tau$ *there exists a derivation* Δ *of*

$$\Gamma' \overset{MML}{\vdash} [F_1/y_1] \cdots [F_n/y_n]E : \tau$$

such that $\Gamma' \overset{T\Lambda}{\vdash} TT(\Delta) : \tau$.

Proof: It follows that

$$\Gamma' \overset{MML}{\vdash} [F_1/y_1] \cdots [F_{i-1}/y_{i-1}]F_i : \overline{\sigma_i},$$

and then by application of Theorem 4.2 we have

$$\Gamma' \overset{MML}{\vdash} [F_1/y_1] \cdots [F_n/y_n]E : \tau.$$

If Δ is a proof of this judgement, then it follows by application of Proposition 4.9 that

$$\Gamma' \overset{T\Lambda}{\vdash} TT(\Delta) : \tau,$$

as required.

(*This completes the proof.* $\square$)

Corollary 4.11 *Given* $\Gamma = \{y_1 : \sigma_1, \cdots, y_n : \sigma_n\}$, *and* Γ' *such that for each* σ_i *there exists an*

$$\Gamma', \{y_1 : \sigma_1, \cdots, y_{i-1} : \sigma_{i-1}\} \overset{PML}{\vdash} F_i : \sigma_i.$$

Then if $\emptyset \,|\, C, \Gamma, \Gamma' \vdash E : \tau$, $S \vdash A \rightsquigarrow A$, *and* $S \vdash E \rightsquigarrow E'$ *there exists a derivation* Δ *of*

$$\Gamma' \overset{MML}{\vdash} [F_1/y_1] \cdots [F_n/y_n]E' : \tau$$

such that $\Gamma' \overset{T\Lambda}{\vdash} TT(\Delta) : \tau$.

Proof: Observe that by Proposition 4.3 we have

$$\emptyset \,|\, C, A \vdash E' : \tau,$$

and by the definition of specialization the expression E' does not contain any free overloaded variables. Thus, the typing context C can be removed, giving

$$\emptyset \,|\, A \vdash E' : \tau,$$

which by definition is a *PML* typing judgement

$$A \vdash^{PML} E' : \tau.$$

Thus the required result follows by Corollary 4.10.

(*This completes the proof.* $\Box$)

We conclude this section by defining the semantics of *MML* terms relative to any model of $T\Lambda$.

Definition 4.12 (Semantics of *MML* terms) *The semantics of MML terms relative to a model $\mathcal{M}$ of $T\Lambda$ is defined as*

$$\mathcal{M}[\![A \vdash^{MML} E : \tau]\!]^{MML} = \mathcal{M}[\![A \vdash^{T\Lambda} TT(\Delta) : \tau]\!],$$

for some derivation Δ of $A \vdash^{MML} E : \tau$.

By Proposition 4.7 and Theorem 4.6, this definition does not depend on the choice of Δ.

4.6 A semantics for *PML* and *OML*

We are now in a position to give a semantics for the systems *PML* and *OML*. Section 4.6.1 lays out a semantics for these systems in terms of the developments of previous sections, particularly with respect to the semantics of *MML*.

4.6.1 Formal semantics

Although it is possible, in theory, to give a single definition describing the meaning of both *PML* and *OML*, it would be difficult to read and it would hide some intuitive understanding. Thus we begin this section by defining the meaning of monomorphic and polymorphic *PML* terms with respect to any model of *MML* terms, concluding with soundness of equational theories. We then extend this definition, describing the semantics of *OML* terms with respect to any model for *PML*, and conclude with a result about the soundness of *OML* equational theories.

Definition 4.13 (Semantics of monomorphic *PML* **terms)** *The semantics of the monomorphic PML term* $\Gamma, \Gamma' \vdash^{PML} E : \tau$*, where* $\Gamma = y_1 : \sigma_1, \cdots, y_n : \sigma_n$*, with relation to any model,* $\mathcal{M}$*, of MML is defined as*

$$\mathcal{M}[\![\Gamma, \Gamma' \vdash^{PML} E : \tau]\!]^{PML} = \mathcal{M}[\![\Gamma' \vdash^{MML} [N_j/y_j]E : \tau]\!]^{MML}.$$

As an example, a function that returns its second argument might be specified as

$$\emptyset \vdash^{PML} \lambda x.\lambda y.y : \tau \to \tau' \to \tau'.$$

Applying the above definition we have

$$curry(curry(\pi_2)) : \mathbb{1} \to [[\![\tau]\!] \to [[\![\tau']\!] \to [\![\tau']\!]]].$$

We now extend this semantics to polymorphic *PML* terms.

Definition 4.14 (Semantics of polymorphic *PML* **terms)** *The semantics*

$$\mathcal{M}[\![A \vdash^{PML} E : \forall\{\alpha_i\}.\tau]\!]^{PML}$$

of a PML term $A \vdash^{PML} E : \forall\{\alpha_i\}.\tau$*, relative to any model* $\mathcal{M}$ *is an arrow*

$$\mathcal{M}[\![\Gamma']\!] \to \Pi\nu_i \in Type.\mathcal{M}^{[\![[\nu_i/\alpha_i]\tau]\!]},$$

defined as follows

$$\begin{aligned}\mathcal{M}[\![A \vdash^{PML} E : \forall\{\alpha_i\}.\tau]\!]^{PML}\eta = \\ \{([\nu_i/\alpha_i]\tau, \mathcal{M}[\![A \vdash^{PML} E : [\nu_i/\alpha_i]\tau]\!]^{PML}\eta) | \nu_i \in Type\}.\end{aligned}$$

As an example, consider again a function that returns it second argument, but this time is polymorphic in its two arguments:

$$\emptyset \vdash^{PML} \lambda x.\lambda y.y : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta.$$

Applying the above definition we have, for each $\tau, \tau' \in \mathit{Type}$,

$$\mathit{curry}(\mathit{curry}(\pi_2)) : \mathbb{1} \to [\llbracket\tau\rrbracket \to [\llbracket\tau'\rrbracket \to \llbracket\tau'\rrbracket]],$$

as an element of the set

$$\llbracket\emptyset \vdash^{PML} \lambda x.\lambda y.y : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta\rrbracket.$$

We now state the first result of this section, asserting that the equational theories of *PML* are sound with respect to the semantics.

Theorem 4.15 (Soundness of *PML* theories) *Let $\mathcal{M}$ be any model and $\mathcal{T}_{PML}$ an equational theory; then $A \vdash E =_{PML} F : \sigma \Rightarrow \mathcal{M} \models_{PML} A \vdash E =_{PML} F : \sigma$.*

A proof of this result is given in Section A.2.3 of Appendix A.

Finally, we are in a position to define a semantics for *OML* in terms of specialization and *PML*.

Definition 4.16 *(Semantics of OML terms)*

$$\begin{aligned}&\mathcal{M}\llbracket w : P \mid C, A \vdash E : \forall\alpha_i.Q \Rightarrow \tau\rrbracket^{OML}\eta = \\ &\qquad \{([\nu_i/\alpha_i]\tau, \mathcal{M}\llbracket A \vdash^{PML} \mathit{Spec}([\nu_i/\tau], E) : [\nu_i/\alpha_i]\tau\rrbracket^{PML}\eta) \mid \nu_i \in \mathit{Type}, \\ &\qquad\qquad \Vdash P, [\nu_i/\alpha_i]Q\},\end{aligned}$$

where $\mathit{Spec}([\nu_i/\tau], E)$ is defined as

$$\begin{aligned}\mathit{Spec}([\nu_i/\tau], E) \quad &= \quad E' \\ \textbf{where}\quad & \\ \Vdash e : P, e' : [\nu_i/\alpha_i]Q & \\ S \vdash A \leadsto A & \\ S \vdash ([e/w]E)e' \leadsto E'. &\end{aligned}$$

By Proposition 4.4 and Theorem 4.15, this definition does not depend on the choice of S in *Spec*.

As an example, consider the following implementation for overloaded $(==) : \forall\alpha.\mathit{Eq}\ \alpha \Rightarrow \alpha \to \alpha \to \mathit{Bool}$,

$$P \mid \{(==) : \sigma\} \vdash \lambda v.\lambda x.\lambda y.(==)\ v\ x\ y : \sigma,$$

where $\sigma = \forall\alpha.Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$. Applying the definition of semantics for *OML*, and assuming the only ground implementations for equality are $\emptyset \vdash^{T\Lambda} eqInt : Int \rightarrow Int \rightarrow Bool$ and $\emptyset \vdash^{T\Lambda} eqBool : Bool \rightarrow Bool \rightarrow Bool$, we have

$$\begin{aligned}
&[\![P \,|\, (==) : \sigma \vdash \lambda v.\lambda x.\lambda y.(==)\ v\ x\ y : \sigma]\!] = \\
&\qquad \{\ (Int \rightarrow Int \rightarrow Bool, \\
&\qquad\quad \emptyset \vdash^{T\Lambda} \lambda x : Int.\lambda y : Int.eqInt\ x\ y : Int \rightarrow Int \rightarrow Bool), \\
&\qquad\quad (Bool \rightarrow Bool \rightarrow Bool, \\
&\qquad\quad \emptyset \vdash^{T\Lambda} \lambda x : Bool.\lambda y : Bool.eqBool\ x\ y : Bool \rightarrow Bool \rightarrow Bool)\}.
\end{aligned}$$

Expanding further gives the set

$$\begin{aligned}
&\{(Int \rightarrow Int \rightarrow Bool, curry(curry(eval(eval(eqInt, \pi_2 \circ \pi_1), \pi_2)))), \\
&(Bool \rightarrow Bool \rightarrow Bool, curry(curry(eval(eval(eqBool, \pi_2 \circ \pi_1), \pi_2))))\},
\end{aligned}$$

as expected.

We conclude this section with the main result of this chapter—soundness of *OML* equational theories.

Theorem 4.17 (Soundness of *OML* theories) *Let $\mathcal{M}$ be any model and $\mathcal{T}_{OML}$ an equational theory, then*

$$P|C, A \vdash E = F : \sigma \Rightarrow \mathcal{M} \models_{OML} P|C, A \vdash E = F : \sigma.$$

A proof of this result is given in Section A.2.4 of Appendix A.

4.7 Related work

Of course, there have been many other attempts to provide a denotational semantics for implicitly typed languages, and we summarize the key points of some of these approaches.

4.7.1 Milner and Damas core-ML semantics

Milner [Mil78], and later Damas [Dam85], described a semantics for implicitly typed core-ML by presenting a denotational semantics for the untyped core-ML terms. Introducing a special value *wrong* for runtime errors, they showed that any well-typed core-ML term would not "go wrong". Although adequate, interpreting well-typed terms with respect to the set of untyped terms introduces extra, unnecessary, constraints on models for core-ML. For example, any model of the untyped calculus must guarantee the existence of fix points, ruling out the possibility of a simple set-based model [Sco80]. However, it is well-known that type inference for core-ML restricts the set of untyped terms such that fix points (Turing's Θ functional [Tur37], for example) are not expressible, and thus one might reasonably expect a definition of core-ML models to reflect this.

4.7.2 Explicit core-ML

Harper and Mitchell [HM93] considered the implicitly typed terms of core-ML as a shorthand for terms of an explicitly typed language called *XML*—a predicative version of the polymorphic λ-calculus. They defined a translation from the typing derivations of a Damas and Milner style inference system into typing derivations of *XML*. For example, the expression

$$\lambda x.x : \forall\alpha.\alpha \to \alpha$$

becomes

$$\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha.$$

Harper and Mitchell described a number of semantic models, including one based on sets, for *XML*, which can be considered models of core-ML via their translation. To guarantee well-formedness of translation, any model of *XML* must satisfy a coherence condition. Unfortunately, this condition is not always satisfied by the set-based model.

4.7.3 Polymorphic types as sets

Motivated by the weakness of previous approaches Ohori [Oho89a] proposed interpreting the terms of core-ML as terms of the simply typed λ-calculus [Chu40], describing a translation from the derivations of a Damas and Milner style inference system[3] into simply typed expressions. Again, to guarantee the well-formedness of translation, Ohori required a coherence condition. However, unlike the coherence

[3]Technically, Ohori described an alternative typing rule for let-polymorphism. We described a variation of this rule and its relationship to Damas and Milner's rule in Section 4.3.

constraint of Harper and Mitchell this is purely syntactic and guaranteed to hold in all models. A key aspect of Ohori's approach is the interpretation of a polymorphic type as the set of its monomorphic instances. For example, the type $\forall\alpha.\alpha \to \alpha$ is represented by the set

$$\{\tau \to \tau | \tau \in \mathit{Type}\}.$$

Although Ohori's presentation avoided the specifics of models, it is well-known that any cartesian closed category is a model for the simply typed λ-calculus [Lam80], and thus a model of core-ML. In particular the category of sets is cartesian closed, providing a set-based model for core-ML.

Of course, the semantics for *OML* described in this chapter builds upon the early work of Ohori.

4.7.4 Type classes

Wadler and Blott [WB89, Blo92] describe a system for ad-hoc polymorphism, capturing relations such as equality and ordering through constrained types. For example, overloaded equality can be specified as

$$\mathit{over}(==) : \forall\alpha.\alpha \to \alpha \to \mathit{Bool}.$$

Here the over construct can be thought of as introducing a new Haskell style type class. In this example, it would introduce a new predicate symbol *Eq* constraining the instances of α, in the type of $==$, to be ones supporting an equality operation. Semantically, overloaded terms are considered sugar for more verbose core-ML terms, justified by a formal translation. However, unlike the more general theory of qualified types, an overloaded expression cannot be assigned a type with more than one type variable. For example, consider an operation *strict* for inserting explicit strictness annotations

$$\mathit{over}(\mathit{strict}) : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \beta.$$

Here the type variable α is constrained, while β is universal for all instances of *strict*. This leads to the observation that Wadler and Blott's translation may, if applied to arbitrary expressions, introduce non-overloaded terms possessing types with nested quantifiers (i.e., goes beyond the Damas and Milner system).

4.7.5 A second look at overloading

Odersky, Wadler, and Wehr [OWW95] describe a system for ad-hoc polymorphism, capturing a restricted form of Wadler and Blott style type classes and a notion of bounded polymorphism with application to polymorphic records. For example, overloaded record selection can be assigned the type

$$(\_.l) : \forall r.\forall \alpha.(r \leq \{l : \alpha\}) \Rightarrow r \rightarrow \alpha,$$

where the predicate $r \leq \{l : \alpha\}$ constrains r to be a record containing at least the field l.

Motivated by the observation that an overloaded expression of Wadler and Blott style type classes, or qualified types, cannot be assigned a meaning independent of its type, Odersky et al. restrict the type of an overloaded symbol to have the form $\alpha \rightarrow \tau$, where α is constrained. Restricting overloaded expressions in this way allows the semantics of overloaded terms to be understood within an untyped framework, similar in style to that of Damas and Milner. Unfortunately, although there are a variety of operations whose types satisfy the required restriction, there are many interesting ones that do not. This includes the operation $\langle l = \_\rangle$ (given in Chapter 2), which has type

$$\langle l = \_\rangle : \forall \alpha.\forall r.(r \backslash l) \Rightarrow \alpha \rightarrow \mathit{Var}\ \{\!|\, l : \alpha \,|\, r \,|\!\},$$

and tags a value with the label l in a variant $\mathit{Var}\ \{\!|\, l : \alpha \,|\, r \,|\!\}$.

Chapter 5

Categorical semantics

In the previous chapter, we developed, a semantics for qualified types based upon specialization. This chapter develops an alternative categorical semantics for qualified types using the notion of polynomial categories. One important difference of the categorical semantics described in this chapter, over the specialization based approach, is its ability to model Haskell's polymorphic recursion.

The sections of this chapter are as follows. Section 5.1 gives an overview of the categorical semantics for qualified types. Section 5.2 introduces an alternative presentation for qualified types. Section 5.3 describes a categorical semantics for qualified types, while Section 5.4 presents an application of our semantics to Haskell's class system. Finally, Section 5.5 considers related work, with particular attention to categorical semantics for first-order polymorphism and constrained types.

A description of the categorical structures (e.g., polynomial categories) used in this chapter can be found in Appendix B.

5.1 Overview

Inspired by the early work of Phoa and Fourman [Pho92, PF92], we interpret first-order polymorphism through the notion of an indeterminate object. The idea of a categorical predicate system is used to provided an interpretation of the abstract notion of predicate. Constrained types are then understood by providing a concrete mapping from the categorical notion of predicates into the underlying category used to interpret expressions. The interpretation places no unexpected constraints on the definition of a predicate system, unlike specialization which required evidence, and thus, predicates to be expressed in $T\Lambda$. This allows the design and specification of individual predicate systems to be considered independently of the more general framework for the underlying language.

As an example, consider a function which duplicates its argument

$$\lambda x.(x, x) : \forall \alpha. Eq\ \alpha \Rightarrow \alpha \to \alpha \times \alpha,$$

and is constrained to arguments supporting an equality operation—asserted by the predicate $Eq\ \alpha$. Applying the translation rules of Figure 3.7 we need only consider the semantics for the expression

$$\lambda v.\lambda x.(x, x) : \forall \alpha. Eq\ \alpha \Rightarrow \alpha \to \alpha \times \alpha,$$

where syntactically λv, and λx represent abstraction over evidence and values, respectively. However, semantically these binding operators are interpreted by the same constructs and it is only the types of the arguments (i.e., predicates and normal expressions) that are considered distinct.

The semantics of an overloaded expression can then be understood in a cartesian closed category $\mathbb{C}[X]$ providing there is a mapping, $\mathcal{E}$, from the semantic meaning of predicates into the category $\mathbb{C}[X]$. In general, predicates are interpreted in a cartesian category and thus the mapping $\mathcal{E}$ is a structure preserving functor between the two categories.

The semantics for our duplicating function can then be understood as the arrow[1]

$$[\![\lambda v.\lambda x.(x, x)]\!] = \ulcorner curry(\langle \pi_2, \pi_2 \rangle) \urcorner : \mathbb{1} \to [\mathcal{E}(Eq\ X) \to [X \to X \times X]].$$

In his dissertation [Jon94b], Jones discusses a categorical treatment for a monomorphic system of qualified types but he does not consider polymorphism. To our knowledge, we are the first to propose a categorical semantics for qualified types—providing a categorical interpretation both for predicate entailment and for first-order polymorphism. This provides a general framework for reasoning about predicates independently of the types that they constrain. However, we retain the more operational view that predicates (can) represent tuples of functions in the same underling category. This is the approach adopted by the languages Gofer [Jon95a] and Haskell [PH97], for example.

[1]Following Lambek and Scott [LS86], we use $\ulcorner f \urcorner : \mathbb{1} \to [A \to B]$ to represent the arrow, referred to as the 'name' of $f : A \to B$, implied by the isomorphism $hom(A, B) \cong hom(\mathbb{1}, [A \to B])$.

5.2 A syntax directed *OML*

In this section we give an alternative syntax-directed presentation of *OML*. This alternative presentation provides a natural formalization for specifying the categorical semantics by induction over the structure of a type derivation.

The alternative typing rules are presented in Figure 5.1. A judgement of the form $T; P \mid C, A \vdash E : \sigma$ represents an assertion that, if the predicates in P hold, then the term E has type σ, using assumptions in C and A to provide types for polymorphic and monomorphic variables, respectively. Intuitively one may think of variables assigned polymorphic types (i.e., variables mentioned in C), as being **let**-bound, while monomorphic variables (i.e., variables mentioned in A) are bound via lambda abstraction.

The set T denotes the set of type variables (bound and unbound) in a given derivation—T may also be used to denote the set of type variables in a monotype type τ. We define $\varphi(\_) : TVar \to \mathbb{N}$ as a bijection from type variables into the naturals, giving a mapping from type variables to indeterminates. The function φ extends to a function over $\mathbb{P}(TVar) \to \mathbb{P}(\mathbb{N})$, providing functionality to calculate the set of indeterminates to be adjoined to a category $\mathbb{C}$. The rules for predicate entailment described in Figure 4.2 must also be extended to include a corresponding set of type variables T. These extended rules are given in the left hand coloum of Figure 5.3.

The rule (let) of Figure 5.1, requires a mention as it is the only place in which polymorphism may be introduced. This occurs through the generalization of a qualified type with respect to the environment in which it was deduced. This process is described using the function $Gen()$ given in Definition 3.3.

5.2.1 Operational semantics for *OML*

Following Tofte [Tof88], we define a big-step operational semantics (often referred to as a natural semantics—see Gunter [Gun92], for example) and show that a well-typed *OML* expression will not 'go wrong'.

A judgement of the form $E \Downarrow V$ represents an assertion that the term E reduces to the value V in one or more steps. The semantic domain V, of values, is described by the following grammar

$$V, U \ ::= \ \lambda x.E \mid \lambda v.E \mid (V, V) \mid ().$$

$$(const) \qquad T; P \,|\, C, A \vdash c : \sigma_c$$

$$(varP) \qquad T; P \,|\, C, x : \forall\alpha.\rho, A \vdash x : [\tau/\alpha]\rho$$

$$(varM) \qquad T; P \,|\, C, A, x : \tau \vdash x : \tau$$

$$(E\times) \qquad \frac{T; P \,|\, C, A \vdash E : \tau \times \tau'}{T; P \,|\, C, A \vdash \mathit{fst}\ E : \tau}$$

$$(\times E) \qquad \frac{T; P \,|\, C, A \vdash E : \tau \times \tau'}{T; P \,|\, C, A \vdash \mathit{snd}\ E : \tau'}$$

$$(\times I) \qquad \frac{T; P \,|\, C, A \vdash E : \tau \quad T; P \,|\, C, A \vdash F : \tau'}{T; P \,|\, C, A \vdash (E, F) : \tau \times \tau'}$$

$$(unit) \qquad T; P \,|\, A \vdash () : ()$$

$$(\rightarrow E) \qquad \frac{T; P \,|\, C, A \vdash E : \tau' \rightarrow \tau \qquad T; P \,|\, C, A \vdash F : \tau'}{T; P \,|\, C, A \vdash EF : \tau}$$

$$(\rightarrow I) \qquad \frac{T; P \,|\, C, A_x, x : \tau' \vdash E : \tau}{T; P \,|\, C, A \vdash \lambda x.E : \tau' \rightarrow \tau}$$

$$(\Rightarrow E) \qquad \frac{T; P \,|\, C, A \vdash E : \pi \Rightarrow \rho \qquad T; P \Vdash e : \pi}{T; P \,|\, C, A \vdash Ee : \rho}$$

$$(\Rightarrow I) \qquad \frac{T; P, v : \pi, P' \,|\, C, A \vdash E : \rho}{T; P, P' \,|\, C, A \vdash \lambda v.E : \pi \Rightarrow \rho}$$

$$(let) \qquad \frac{T; P \,|\, C, A \vdash E : \tau \qquad T; Q \,|\, C, A_x, x : \sigma \vdash F : \tau}{T; P, Q \,|\, C, A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$
$$\text{where } \sigma = \mathit{Gen}(C, A, P \Rightarrow \tau)$$

Figure 5.1: Syntax directed typing rules for *OML*.

$$\frac{E \Downarrow \lambda x.E' \quad [F/x]E' \Downarrow V}{EF \Downarrow V}$$

$$\frac{E \Downarrow \lambda v.E' \quad [e/v]E' \Downarrow V}{Ee \Downarrow V}$$

$$\lambda x.E \Downarrow \lambda x.E$$

$$\lambda v.E \Downarrow \lambda v.E$$

$$\frac{E \Downarrow (V, U)}{\mathit{fst}\ E \Downarrow V}$$

$$\frac{E \Downarrow (V, U)}{\mathit{snd}\ E \Downarrow U}$$

$$\frac{E \Downarrow V \quad F \Downarrow U}{(E, F) \Downarrow (V, U)}$$

$$() \Downarrow ()$$

$$\frac{[E/x]F \Downarrow V}{\mathbf{let}\ x = E\ \mathbf{in}\ F \Downarrow V}$$

Figure 5.2: Natural semantics for *OML*.

The reduction rules for expressions are given in Figure 5.2.

Following Wright and Felleisen [WF94], we capture the notion that reduction preserves type, and thus, will not 'go wrong', by proving subject reduction for *OML*.

Proposition 5.1 (Subject reduction (syntactic type soundness)) *If* $T; P \mid C, A \vdash E : \rho$ *and* $E \Downarrow V$, *then* $T; P \mid C, A \vdash V : \rho$.

A proof of this result is given in Section A.3.1 of Appendix A.

Intuitively, subject reduction tells us that an implementation of qualified types need not perform run-time type checking, as a terminating reduction sequence produces a well-typed value. Somes benfits of not preforming run-time type checking might include, but not limited to: faster execution, as run-time checks need not inserted and thus are executed, requiring extra computation time; and smaller compiled programs as type information can be discarded and run-time checks need not be inserted.

5.3 A categorical semantics

This section presents a categorical semantics for *OML*. To ease the technical presentation we break the definition into sections: Section 5.3.1 describes a categorical interpretation of monomorphic types, while Section 5.3.2 introduces the notion of a categorical predicate system. Finally, Section 5.3.3 presents the complete categorical semantics for *OML*.

5.3.1 Categorical treatment of types

Following Lambek [Lam80], we interpret the meaning of simple types as objects in a cartesian closed category, built by induction from ground types and type variables. As discussed in the introduction, type variables are interpreted as indeterminate objects in some cartesian closed category $\mathbb{C}[X]$.

Definition 5.2 (Semantics of monotypes) *Given a monotype τ, a set $T = TV(\tau)$, and a cartesian closed category $\mathbb{C}[\varphi(T)]$, then a structure $\mathcal{M}$ is specified by giving an object $\mathcal{M}[\![\sigma_c]\!]$ in $\mathbb{C}[\varphi(T)]$ for each constant type σ_c and an interpretation of τ by induction, through the following clauses*

$$\begin{array}{lcl} \mathcal{M}[\![\alpha_i]\!] & = & \varphi(\alpha_i) \\ \mathcal{M}[\![\tau \to \tau']\!] & = & [\mathcal{M}[\![\tau]\!] \to \mathcal{M}[\![\tau']\!]] \\ \mathcal{M}[\![()]\!] & = & \mathbb{1} \\ \mathcal{M}[\![\tau \times \tau']\!] & = & \mathcal{M}[\![\tau]\!] \times \mathcal{M}[\![\tau']\!]. \end{array}$$

As a notational convenience we may write $[\![]\!]$, when in fact we really mean $\mathcal{M}[\![]\!]$.

We conclude this section with a categorical interpretation of type substitutions.

Definition 5.3 (Semantics of substitutions) *Given monotypes $\tau_1, \cdots, \tau_n$, a model $\mathcal{M}$ over $\mathbb{C}[\varphi(T)]$, such that $\bigcup_{i=1}^{n} TV(\tau_i) \subseteq T$, and a substitution $[\tau_i/\alpha_i]$, we define the semantics of $[\tau_i/\alpha_i]$ to be the substitution functor $\mathcal{M}[\![[\tau_i/\alpha_i]]\!]_{\vec{D}} : \mathbb{C}[\varphi(T)] \to \mathbb{C}[\varphi(T)]$ where $\vec{D} = \mathcal{M}[\![\tau_1]\!], \cdots, \mathcal{M}[\![\tau_n]\!]$, and $D_i = X_i$ otherwise, which uniquely extends the identity functor on $\mathbb{C}$.*

To provide some intuition behind the categorical definition for substitutions, and to motivate the following distribution lemma, consider the type $\alpha \to \alpha$ and the substitution $[Nat/\alpha]$, where $\mathcal{M}[\![Nat]\!]$ is an object representing the type of natural numbers, for example. The substitution functor for $[Nat/\alpha]$ is the unique functor $[\![[Nat/\alpha]]\!] : \mathbb{C}[X] \to \mathbb{C}$ such that $[\![[Nat/\alpha]]\!]([\![\alpha]\!]) = [\![Nat]\!]$ and acts as the identity everywhere else. This gives $[\![[Nat/\alpha]]\!][\![\alpha \to \alpha]\!] = [\![[Nat/\alpha](\alpha \to \alpha)]\!]$, as expected. This is a general property captured by the following substitution lemma.

Lemma 5.4 *If τ and ν are monotypes over $\mathbb{C}[\varphi(TV(\tau) \cup TV(\nu))]$ is a model for monotypes, and $[\nu/\alpha]$ a substitution, then*

$$\mathcal{M}[\![[\nu/\alpha]\tau]\!] = \mathcal{M}[\![[\nu/\alpha]]\!]\mathcal{M}[\![\tau]\!].$$

A proof of this result is given in Section A.3.2 of Appendix A.

5.3.2 Categorical predicate systems

We now consider the semantics for predicate entailment, describing the notion of a categorical predicate system in the following definition.

Definition 5.5 (Semantics of predicates) *A categorical model of predicate entailment, with respect to a set of type variables T, consists of*

- *A cartesian category $\mathbb{C}[\varphi(T)]$.*
- *A structure, $\mathcal{P}$, in $\mathbb{C}[\varphi(T)]$ over entailment, defined by*
  - *for every predicate symbol π, an object $\mathcal{P}[\![\pi]\!]$ in $\mathbb{C}[\varphi(T)]$.*

As a notational convenience we may write $[\![\_]\!]$, when in fact we really mean $\mathcal{P}[\![\_]\!]$. The meaning of a context P is defined inductively by

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \mathbb{1} \\ [\![P, v : \pi]\!] & = & [\![P]\!] \times [\![\pi]\!]. \end{array}$$

For every judgement $T; P \Vdash Q$, we specify an arrow $[\![T; P \Vdash Q]\!] : [\![P]\!] \to [\![Q]\!]$ in $\mathbb{C}[\varphi(T)]$. The semantics of $\Vdash$ terms are specified inductively over the rules for entailment. Figure 5.3 gives the complete set of entailment rules in the left hand column with the corresponding categorical interpretation on the right.

(id)	$T; v : P \Vdash v : P$	$[\![P]\!] \xrightarrow{id} [\![P]\!]$
$(term)$	$T; v : P \Vdash \emptyset$	$[\![P]\!] \xrightarrow{!} \mathbb{1}$
(fst)	$T; v : P, w : Q \Vdash v : P$	$[\![P, Q]\!] \xrightarrow{\pi_1} [\![P]\!]$
(snd)	$T; v : P, w : Q \Vdash w : Q$	$[\![P, Q]\!] \xrightarrow{\pi_2} [\![Q]\!]$
$(univ)$	$\dfrac{T; v : P \Vdash e : Q \qquad T; v : P \Vdash e' : R}{T; v : P \Vdash e : Q, e' : R}$	$\dfrac{[\![P]\!] \xrightarrow{e} [\![Q]\!] \quad [\![P]\!] \xrightarrow{e'} [\![R]\!]}{[\![P]\!] \xrightarrow{<e,e'>} [\![Q, R]\!]}$
$(trans)$	$\dfrac{T; v : P \Vdash e : Q \qquad T; v' : Q \Vdash e' : R}{T; v : P \Vdash [e/v']e' : R}$	$\dfrac{[\![P]\!] \xrightarrow{e} [\![Q]\!] \quad [\![Q]\!] \xrightarrow{e'} [\![R]\!]}{[\![P]\!] \xrightarrow{e} [\![Q]\!] \xrightarrow{e'} [\![R]\!]}$

Figure 5.3: Categorical interpretation of predicate entailment.

The following definition extends the interpretation of monotypes given in Section 5.3.1 to qualified monotypes.

Definition 5.6 (Semantics of qualified types) *Given a qualified type $P \Rightarrow \tau$, a set $T = TV(P \Rightarrow \tau)$, a predicate system $\mathcal{P}$ over a cartesian category $Pred[\varphi(T)]$ for P, and a model $\mathcal{M}$ over a cartesian closed category $\mathbb{C}[\varphi(T)]$ for τ, then $\mathcal{M}[\![P \Rightarrow \tau]\!]$ is defined by specifying*

- *A cartesian functor $\mathcal{E} : Pred[\varphi(T)] \to \mathbb{C}[\varphi(T)]$ mapping predicates to evidence. For any entailment $P \xrightarrow{e} Q$, the arrow $\mathcal{E}P \xrightarrow{\mathcal{E}e} \mathcal{E}Q$ should be uniquely determined by P and Q alone to guarantee 'uniqueness of evidence'.*
- *$\mathcal{M}[\![]\!]$ is extended to predicates and constrained types by defining $\mathcal{M}[\![\pi]\!] = \mathcal{E}(\mathcal{P}[\![\pi]\!])$ and $\mathcal{M}[\![\pi \Rightarrow \rho]\!] = [\mathcal{M}[\![\pi]\!] \to \mathcal{M}[\![\rho]\!]]$.*

We conclude this section with a statement of soundness for predicates.

Lemma 5.7 *(Predicate soundness) Given a judgement $T; v : P \Vdash w : Q$, and a predicate system $\mathcal{P}$, then*

$$\mathcal{P}[\![T; v : P \Vdash w : Q]\!] : \mathcal{P}[\![P]\!] \longrightarrow \mathcal{P}[\![Q]\!].$$

A proof of this result is given in Section A.3.3 of Appendix A.

5.3.3 Categorical *OML*

Before defining a semantics for *OML* terms, we first give an interpretation for type schemes and type contexts. The following definition formalizes the interpretation of a type scheme, σ, at a given instance.

Definition 5.8 (Semantics of type schemes) *Given a type scheme $\forall\alpha.\rho$, a monotype ν, and a model $\mathcal{M}$ over a cartesian closed category, $\mathbb{C}[\varphi(TV(\nu))]$, for ρ, then $\mathcal{M}[\![\forall\alpha.\rho]\!]$ with respect to a substitution $[\nu/\alpha]$ is specified by*

$$\mathcal{M}[\![\forall\alpha.\rho]\!]^{[\nu/\alpha]} = \mathcal{M}[\![[\nu/\alpha]]\!]\mathcal{M}[\![\rho]\!].$$

Suppose that $T; P \mid C, A \vdash E : \rho$, and that $x_1, \cdots, x_n$ are the occurrences of $(x : \forall\alpha.\rho) \in C$ in the term E. It follows that, in the unique derivation of $T; P \mid C, A \vdash E : \rho$, each x_i is introduced via an application of rule $(varP)$, such as

$$\frac{(x_i : \forall\alpha_j.Q \Rightarrow \nu) \in C}{T; P \mid C, A \vdash x_i : [\tau_j/\alpha_j]Q \Rightarrow \nu.}$$

Each x_i can be understood by providing an expression of type $[\tau_j/\alpha_j]Q \Rightarrow \nu$. Thus a polymorphic context, C, assigning types to free variables (used at different instances) in the expression E, can be interpreted as a product of all instances of x_i in E.

Definition 5.9 (Semantics of polymorphic contexts) *Given $T; P \mid C, A \vdash E : \rho$ is derivable, where $C = x_1 : \sigma_1, \cdots, x_n : \sigma_n$, $\mathcal{M}$ is a model over a cartesian* *closed category* $\mathbb{C}[\varphi(T)]$ *for* ρ, *and suppose*[2] $x_i^{[\nu/\alpha]_1}, \cdots, x_i^{[\nu/\alpha]_{r_j}}$ *are the occurrences of* x_i *in* E, *for* $1 \leq i \leq n$. *Then the meaning* $\mathcal{M}[\![C]\!]_E$ *of the context* C *with respect to the expression* E *is given by*

$$\mathcal{M}[\![C]\!]_E = \prod_{l=1}^{r_1} [\![\sigma_1]\!]^{[\nu/\alpha]_l} \times \cdots \times \prod_{l=1}^{r_n} [\![\sigma_n]\!]^{[\nu/\alpha]_l} \in C[\varphi(T)].$$

Note that

$$[\![C]\!]_{EF} \cong [\![C]\!]_{\mathbf{let}\ x=E\ \mathbf{in}\ F} \cong [\![C]\!]_E \times [\![C]\!]_F.$$

Finally, we are in position to describe a categorical semantics for *OML*, captured by the following definition.

Definition 5.10 (Semantics of *OML***)** *A categorical model of OML with respect to a set of type variables,* T, *consists of*

- *A predicate system* $\mathcal{P}$, *specified by Definition 5.5;*
- *a model* $\mathcal{M}$, *specified by Definition 5.6; and*
- *for every constant,* $c : \sigma_c$, *a global element*

$$\mathcal{M}[\![c]\!] : \mathbb{1} \to \mathcal{M}[\![\sigma_c]\!].$$

As a notational convenience we may write $[\![\_]\!]$, *when in fact we really mean* $\mathcal{M}[\![\_]\!]$. *The meaning of a context* A *is defined inductively by*

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \mathbb{1} \\ [\![A, x : \tau]\!] & = & [\![A]\!] \times [\![\tau]\!]. \end{array}$$

For every judgement $T; P \mid C, A \vdash E : \rho$ *we specify an arrow* $\mathcal{M}[\![T; P \mid C, A \vdash E : \rho]\!] : \mathcal{E}(\mathcal{P}[\![P]\!]) \times \mathcal{M}[\![C]\!]_E \times \mathcal{M}[\![A]\!] \to \mathcal{M}[\![\rho]\!]$ *in* $\mathbb{C}[\varphi(T)]$. *The semantics of OML terms are specified inductively over the typing rules. Figures 5.4 and 5.5 give the complete set of rules for the categorical semantics.*

[2]For ease of reading we have abused notion slightly by avoiding subscripts on ν in a substitution $[\nu/\alpha]_j$. Formally, each ν should be annotated ν_j, highlighting the fact that each occurrence of x_i may be used at a different instantiation.

$$(\to I)\quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times ([\![A]\!] \times [\![\tau]\!]) \xrightarrow{E} [\![\tau']\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{curry(E \circ s)} [[\![\tau]\!] \Rightarrow [\![\tau']\!]]}$$

$$where\ s : (A \times B \times C) \times D \cong A \times B \times (C \times D)$$

$$(\Rightarrow E)\quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\mathcal{E}[\![\pi]\!] \Rightarrow [\![\rho]\!]] \quad [\![P]\!] \xrightarrow{e} [\![\pi]\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{\langle E, \mathcal{E} e \circ \pi_1 \rangle} [\mathcal{E}[\![\pi]\!] \Rightarrow [\![\rho]\!]] \times \mathcal{E}[\![\pi]\!] \xrightarrow{eval} [\![\rho]\!]}$$

$$(\Rightarrow I)\quad \frac{(\mathcal{E}[\![P]\!] \times \mathcal{E}[\![\pi]\!]) \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\![\rho]\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{curry(E \circ r)} [\mathcal{E}[\![\pi]\!] \Rightarrow [\![\rho]\!]]}$$

$$where\ r : (A \times B) \times C \times D \cong (A \times C \times D) \times B$$

$$(let)\quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\![\tau']\!] \quad \mathcal{E}[\![Q]\!] \times [\![C]\!]_F \times I \times [\![A]\!] \xrightarrow{F} [\![\tau]\!]}{\begin{array}{r}(\mathcal{E}[\![P]\!] \times \mathcal{E}[\![Q]\!]) \times [\![C]\!]_{\mathbf{let}\ x=E\ \mathbf{in}\ F} \times [\![A]\!] \xrightarrow{\langle [\nu/\alpha]_1 E, \cdots, [\nu/\alpha]_r E \rangle \times id \circ a} \\ I \times (\mathcal{E}[\![Q]\!] \times [\![C]\!]_F \times [\![A]\!]) \xrightarrow{F \circ b} [\![\tau]\!]\end{array}}$$

$$where\ I = \textstyle\prod_{l=1}^{r_1} [\![Gen(C, A, P \Rightarrow \tau)]\!]^{[\nu/\alpha]_l}$$

$$a : (A \times B) \times (C \times D) \times E \longrightarrow (A \times C \times E) \times (B \times D \times A)$$

$$b : A \times (B \times C \times D) \longrightarrow B \times C \times A \times D$$

Figure 5.4: Categorical semantics for *OML*—Part 1.

The following lemma shows that, if a typing assertion is provable, then it also holds semantically. We may interpret this to mean that well-typed *OML* expressions do not contain type errors.

Lemma 5.11 (Type soundness) *If* $T; P \,|\, C, A \vdash E : \rho$ *then*

$$\mathcal{M}[\![T; P \,|\, C, A \vdash E : \rho]\!] : \mathcal{E}(\mathcal{P}[\![P]\!]) \times \mathcal{M}[\![C]\!]_E \times \mathcal{M}[\![A]\!] \longrightarrow \mathcal{M}[\![\rho]\!].$$

A proof of this result is given in Section A.3.4 of Appendix A.

$$(unit) \quad \mathcal{E}[\![P]\!] \times \mathbb{1} \times [\![A]\!] \xrightarrow{!} [\![()]\!]$$

$$(E\times) \quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\![\tau \times \tau']\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_{fst\ E} \times [\![A]\!] \xrightarrow{\pi_1 \circ E} [\![\tau]\!]}$$

$$(\times E) \quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\![\tau \times \tau']\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_{snd\ E} \times [\![A]\!] \xrightarrow{\pi_2 \circ E} [\![\tau']\!]}$$

$$(\times I) \quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [\![\tau]\!] \quad \mathcal{E}[\![P]\!] \times [\![C]\!]_F \times [\![A]\!] \xrightarrow{F} [\![\tau']\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![C]\!]_F \times [\![A]\!] \xrightarrow{\langle E \circ \pi_1, F \circ \pi_2 \rangle \circ p} [\![\tau \times \tau']\!]}$$
$$\textit{where } p : A \times B \times C \times D \longrightarrow (A \times B \times D) \times (A \times C \times D)$$

$$(varP)_1 \quad \mathcal{E}[\![P]\!] \times \mathbb{1} \times [\![[\tau/\alpha]\rho]\!] \times [\![A]\!] \xrightarrow{\pi_1 \circ \pi_2 \circ \pi_2} [\![[\tau/\alpha]\rho]\!]$$

$$(varP)_2 \quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_x \times [\![A]\!] \xrightarrow{x} [\![\rho]\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_{C,x} \times \mathbb{1} \times [\![A]\!] \xrightarrow{x \circ \pi_2 \circ \eta} [\![\rho]\!]}$$
$$\textit{where } \eta : A \times B \times C \times D \cong C \times A \times B \times D$$

$$(varM)_1 \quad \mathcal{E}[\![P]\!] \times \mathbb{1} \times [\![A]\!] \times [\![\tau]\!] \xrightarrow{\pi_2 \circ \pi_2 \circ \pi_2} [\![\tau]\!]$$

$$(varM)_2 \quad \frac{\mathcal{E}[\![P]\!] \times \mathbb{1} \times [\![A]\!] \xrightarrow{x} [\![\tau]\!]}{\mathcal{E}[\![P]\!] \times \mathbb{1} \times [\![A]\!] \times [\![\tau']\!] \xrightarrow{x \circ \pi_1 \circ \pi_2 \circ \pi_2} [\![\tau]\!]}$$

$$(\to E) \quad \frac{\mathcal{E}[\![P]\!] \times [\![C]\!]_E \times [\![A]\!] \xrightarrow{E} [[\![\tau]\!] \to [\![\tau']\!]] \quad \mathcal{E}[\![P]\!] \times [\![C]\!]_F \times [\![A]\!] \xrightarrow{F} [\![\tau]\!]}{\mathcal{E}[\![P]\!] \times [\![C]\!]_{EF} \times [\![A]\!] \xrightarrow{\langle E,F \rangle \circ \delta} [[\![\tau]\!] \to [\![\tau']\!]] \times [\![\tau]\!] \xrightarrow{eval} [\![\tau']\!]}$$
$$\textit{where } \delta : A \times (B \times C) \times D \longrightarrow (A \times B \times D) \times (A \times C \times D)$$

Figure 5.5: Categorical semantics for *OML*—Part 2.

Furthering the connection between the natural semantics of Section 5.2.1, and the categorical semantics described in this Section, the following theorem captures the usual soundness property for reduction with respect to a denotational semantics: If a typing assertion is provable and its corresponding expression reduces to some value, then this value and the original expression are equal denotationally.

Theorem 5.12 (Soundness of *OML* reduction) *If* $T; P \mid C, A \vdash E : \rho$ *and* $E \Downarrow V$*, then*

$$\mathcal{M}[\![T; P \mid C, A \vdash E : \rho]\!] = \mathcal{M}[\![T; P \mid C, A \vdash V : \rho]\!].$$

A proof of this result is given in Section A.3.5 of Appendix A.

This theorem is closely related to Theorem 4.17 (soundness of *OML* equational theories). Intuitively, we might hope to induce the equations of Figure 4.4 from the natural semantics described in Figure 5.2. We do not consider this point further, but note that it may be an interesting area for future work.

5.4 Towards a semantics for Haskell type classes

In this section we give an example of our semantics in relation to the programming language Haskell [PH97]. We consider a subset of the Haskell class system, highlighting a possible approach to defining a denotational semantics for Haskell. For simplicity we consider a specific instance of ad-hoc polymorphism, overloaded equality, and a fixed entailment system. However, expanding this to a system where entailment is extended statically by the user, as in Haskell, should not require too much extra work.

In Haskell, operations for testing equality are introduced through instances of the following class

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{class } \mathit{Eq}\ \alpha \textbf{ where}} \\ \quad (==), (/=) & :: & \alpha \to \alpha \to \mathit{Bool} \\ \quad (/=) & = & \lambda x.\lambda y.\mathit{not}\ (x == y). \end{array}$$

This definition introduces a new predicate symbol Eq.

An instance of the type class Eq, at a given type τ, provides at least an implementation for equality, at type τ. However, if this latter implementation is not provided, then the default $\lambda x.\lambda y.not\ (x == y)$ is used. For example, given a primitive operation $eqInt : Integer \rightarrow Integer \rightarrow Bool$, determining equality over integers, an instance of Eq at type $Integer$ is given as

$$\begin{array}{l} \textbf{instance } Eq\ Integer \textbf{ where} \\ \quad (==) \;\; = \;\; eqInt. \end{array}$$

Most Haskell implementations construct a *dictionary* as evidence for equality at type $Integer$. This includes implementations for each of the operations supported by the class Eq. Following Wadler and Blott [WB89], we represent dictionaries as Haskell data types. For example, the dictionary for the Eq class could be defined as

$$\textbf{data } EqDict\ \alpha \;\; = \;\; EqDict\ (\alpha \rightarrow \alpha \rightarrow Bool)\ (\alpha \rightarrow \alpha \rightarrow Bool).$$

A Haskell implementation specifies suitable implementations for $(==) : EqDict\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$ and $(/=) : EqDict\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$. The following definitions suffice for the current presentation

$$\begin{array}{rcl} (==)\ (EqDict\ eq\ \_) & = & eq \\ (/=)\ (EqDict\ \_\ neq) & = & neq, \end{array}$$

which simply extract the appropriate implementations.

To provide a semantics for our system we begin by specifying objects for predicates of the form $Eq\ \tau$. For simplicity, we consider equality at instances α and $Integer$, i.e., $\emptyset \Vdash eqInt : Eq\ Integer$, and $v : Eq\ \alpha \Vdash v : Eq\ \alpha$. We now specify a cartesian closed category $\mathbb{C}$, and thus a predicate system $\mathcal{P}$, for our entailment relation as follows[3]

- Objects are chain complete partial orders (CPOs), including at least the (lifted) CPOs $\mathcal{P}[\![Integer]\!] = \mathbb{Z}_\perp$, $\mathcal{P}[\![Bool]\!] = \mathbb{B}_\perp$ and closed with respect to the standard constructions of products and exponentials. Arrows are total continuous functions between CPOs.

- We specify an arrow

$$\langle eqInt, not \circ eqInt \rangle : \mathbb{1} \xrightarrow{eqint} [\mathbb{Z} \rightarrow [\mathbb{Z} \rightarrow \mathbb{B}]] \times [\mathbb{Z} \rightarrow [\mathbb{Z} \rightarrow \mathbb{B}]],$$

  which represents the implementation for the equality class at type $Integer$.

[3] We assume a function $not : Bool \rightarrow Bool$ to negate a boolean value, and the arrows $\mathcal{P}[\![\emptyset; \emptyset \mid \emptyset \vdash eqInt : Integer \rightarrow Integer \rightarrow Bool]\!]$ and $\mathcal{P}[\![\emptyset; \emptyset \mid \emptyset \vdash not : Bool \rightarrow Bool]\!]$ are labelled $eqInt$ and not, respectively.

It follows by Proposition B.7 that the cartesian closed polynomial category, $\mathbb{C}[\varphi(\alpha)]$, is closed over the rules of Figure 5.3, and thus, is a predicate system. Given the cartesian closed category of complete partial orders and total continuous functions, denoted *CPO*, and a structure $\mathcal{M}$ as specified in Definition 5.10, the predicate functor $\mathcal{E}$ is then defined simply as the identity inclusion—the predicate system $\mathcal{P}$ for equality is a full subcategory of *CPO*.

Our semantics interprets dictionaries of type *EqDict* α by the functor

$$EqDict(A) = [A \to [A \to \mathbb{B}]] \times [A \to [A \to \mathbb{B}]],$$

and, as such, the functions $(==)$ and $(/=)$ can be interpreted as

$$\begin{array}{rcl}
\mathcal{M}[\![\emptyset; \emptyset \,|\, \emptyset \vdash ==: \forall\alpha.Eq\ \alpha \Rightarrow \alpha \to \alpha \to Bool]\!] & : & [X \to [X \to \mathbb{B}]] \times [X \to [X \to \mathbb{B}]] \\
 & & \quad \to [X \to [X \to \mathbb{B}]] \\
\mathcal{M}[\![\emptyset; \emptyset \,|\, \emptyset \vdash ==: \forall\alpha.Eq\ \alpha \Rightarrow \alpha \to \alpha \to Bool]\!] & = & \pi_1 \\
\mathcal{M}[\![\emptyset; \emptyset \,|\, \emptyset \vdash /= : Eq\ \alpha \Rightarrow \alpha \to \alpha \to Bool]\!] & : & [X \to [X \to \mathbb{B}]] \times [X \to [X \to \mathbb{B}]] \\
 & & \quad \to [X \to [X \to \mathbb{B}]] \\
\mathcal{M}[\![\emptyset; \emptyset \,|\, \emptyset \vdash /= : \forall\alpha.Eq\ \alpha \Rightarrow \alpha \to \alpha \to Bool]\!] & = & \pi_2.
\end{array}$$

As an example consider the expression

$$\lambda x.(==)\ (EqDict\ eqInt\ (not \circ eqInt))\ x\ 10,$$

which has type $Integer \to Bool$ in any type and predicate context. This expression is assigned the meaning

$$\begin{array}{rl}
 & [\![T; P \,|\, C, A \vdash \lambda x.(==)\ (EqDict\ eqInt\ (not \circ eqInt))\ x\ 10 : Integer \to Bool]\!] \\
= & curry([\![T; P \,|\, C, A, x : Integer \vdash (==)\ (EqDict\ eqInt\ (not \circ eqInt))\ x\ 10 : Bool]\!]) \\
= & curry(eval \circ \langle\ [\![T; P \mid C, A, x : Integer \vdash\ (==)\ (EqDict\ eqInt\ (not \circ eqInt))\ x : \\
 & \qquad\qquad\qquad\qquad\qquad\qquad Integer \to Bool]\!], \\
 & \qquad\qquad [\![T; P \,|\, C, A, x : Integer \vdash 10 : Integer]\!]\rangle) \\
= & curry(eval \circ \langle eval \circ \langle eqInt, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle),
\end{array}$$

where $\overline{10}$ is a global element for the integer 10 and we have omitted some diagonal maps for ease of reading.

Assume that we are given the following definition

$$curry(curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle)),$$

for the arrow $eqInt$. The symbol $==$ represents primitive (uncurried) equality on integers, and can be defined in the standard way. Now applying the result of the above calculation to the global element $\overline{5}$ (representing the integer 5), gives the following equality

$$\begin{array}{ll}
 & eval \circ \langle curry(eval \circ \langle eval \circ \langle eqInt, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle), \overline{5}\rangle \\
= & eval \circ curry(eval \circ \langle eval \circ \langle eqInt, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle) \times id \circ \langle id, \overline{5}\rangle \\
= & eval \circ \langle eval \circ \langle eqInt, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle) \circ \langle id, \overline{5}\rangle \\
= & eval \circ \langle eval \circ \langle curry(curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle)), \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle \circ \langle id, \overline{5}\rangle \\
= & eval \circ \langle eval \circ curry(curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle)) \times id \circ \langle id, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle \circ \langle id, \overline{5}\rangle \\
= & eval \circ \langle curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle) \circ \langle id, \pi_2 \circ \pi_2 \circ \pi_2\rangle, \overline{10}\rangle \circ \langle id, \overline{5}\rangle \\
= & eval \circ \langle curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle), \overline{10}\rangle \circ \langle id, \pi_2 \circ \pi_2 \circ \pi_2\rangle \circ \langle id, \overline{5}\rangle \\
= & eval \circ curry(== \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle) \times id, \circ\langle id, \overline{10}\rangle \circ \langle id, \pi_2 \circ \pi_2 \circ \pi_2\rangle \circ \langle id, \overline{5}\rangle \\
= & == \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle \circ \langle id, \overline{10}\rangle \circ \langle id, \pi_2 \circ \pi_2 \circ \pi_2\rangle \circ \langle id, \overline{5}\rangle \\
= & == \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle \circ \langle id, \overline{10}\rangle \circ \langle\langle id, \overline{5}\rangle, \overline{5}\rangle \\
= & == \circ\langle \pi_2 \circ \pi_1, \pi_2\rangle \circ \langle\langle\langle id, \overline{5}\rangle, \overline{5}\rangle, \overline{10}\rangle \\
= & == \circ\langle \overline{5}, \overline{10}\rangle \\
= & F.
\end{array}$$

5.5 Related work

There have been other attempts to provide categorical semantics for implicitly typed languages and constrained polymorphism, and we summarize the key points of some of these approaches here.

5.5.1 A categorical model for core-ML

As discussed in the introduction to this chapter, the categorical semantics of Section 5.3.3 is based upon the early work of Phoa.

Phoa [Pho92] described a semantics for implicitly typed core-ML. Interpreting generic type variables—Milner's definition of type variables universally quantified at the outermost level [Mil78]—to be some (yet) undetermined object in a cartesian closed category $\mathbb{C}[X]$ obtained by adjoining an 'indeterminate object' X to the cartesian closed category $\mathbb{C}$. It then makes sense to interpret expressions whose types contain polymorphic (i.e., generic) variables in the category $\mathbb{C}[X]$ rather than the underlying category $\mathbb{C}$. For example, consider again the duplicating function which, unlike the definition at the beginning of the chapter, places no constraints on its argument.

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

In Phoa's interpretation this function is understood as the arrow

$$[\![\lambda x.(x, x)]\!] = \ulcorner \langle id_X, id_X \rangle \urcorner : \mathbb{1} \to [X \to X \times X],$$

in $\mathbb{C}[X]$. Thus given that the expression $\lambda x.(x, x)$ is at some point applied to an expression of type τ, corresponding to an object C in the underlying category $\mathbb{C}$, then the instantiated expression is understood by applying the substitution $X \mapsto C$ to $\ulcorner \langle id_X, id_X \rangle \urcorner : \mathbb{1} \to [X \to X \times X] \in \mathbb{C}[X]$, resulting in $\ulcorner \langle id_C, id_C \rangle \urcorner : \mathbb{1} \to [C \to C \times C]$, and thus, an arrow for $\lambda x.(x, x)$ in $\mathbb{C}$.

5.5.2 A categorical semantics for F^ω

Seely [See87] describes a categorical semantics for Girard's Higher-Order Polymorphic λ-calculus (F^ω), that, following Lambek and Scott [LS86], interprets typed λ-calculus via cartesian closed categories. However, unlike the structures required to model the simply typed λ-calculus, a structure capturing the properties of F^ω must be able to model higher-order function types, $\forall \alpha.\sigma$, where α may itself be instantiated to a higher-order function type. To permit such higher-order types, the categorical structure must also be "complete" in some sense (i.e., must allow the formation of "arbitrary" products). Of course, this is impossible for an arbitrary cartesian closed category—see Reynolds [Rey84] for a proof in the category *Set* and Reynolds and Plotkin [RP90] for a categorical proof.

To overcome this dilemma Seely interprets the categorical structure in an appropriate "universe", a so called "PL category", which must be an internal complete cartesian closed category in some other category than the category *Set*—the solution is to introduce a class of cartesian closed categories indexed by an appropriate base category, which mediates the well-formedness of types and thus terms within each fibre (i.e., a particular cartesian closed category). Unfortunately, although Seely's interpretation can be used to construct models for implicitly typed λ-calculi (e.g., PML), the set of provable equalities for these calculi is only a subset of equalities provable algebraically in F^ω. As a consequence, modelling implicitly typed λ-calculi within Seely's framework can often introduce unnecessary complications. For example, it is often simpler to choose the category *Set* as a model for PML, providing a model theoretic framework known to a wide selection of computer scientists.

5.5.3 A categorical semantics for type classes

Hilken and Rydeheard [HR92], propose a categorical semantics of Haskell style type classes, that, following Seely [See87], interprets a language with terms, types, and kinds with respect to indexed categories. The relationship between types and type classes is described through Lawvere's [Law70] comprehension schema in an indexed category, which defines subsets through predicates as $x \in \{y : A|\phi\} \Leftrightarrow x \in A \wedge \phi(x)$. To illustrate this interpretation, consider the Haskell class introducing a predicate *PartialOrd*

$$\begin{array}{l} \textbf{class}\ \mathit{PartialOrd}\ \alpha\ \textbf{where} \\ \quad (\sqsubseteq) \ : \ \alpha \to \alpha \to \mathit{Bool}, \end{array}$$

asserting that, for any type included in the unary-relation *PartialOrd*, there is an operation $(\sqsubseteq)$ of the appropriate type. Intuitively, the class *PartialOrd* is interpreted by a comprehension $\{K|\phi\}$. Here, K is the kind of the type variable α, which in this case is the kind of all types, while the predicate ϕ restricts types in K to those of the form $\alpha \to \alpha \to \mathit{Bool}$.

To account for the notion of subclass—the class *Eq* can be considered a subclass of *PartialOrd* by defining $(==)$ in terms of $(\sqsubseteq)$, for example—Hilken and Rydeheard introduce rules for class introduction and elimination. Separating *form* and *formation* introduces the problem that a type derivation is no longer determined by an expressions structure. Hilken and Rydeheard provide a coherence result restoring the connection between sequents and their derivations. Unfortunately, the implicit polymorphism of core-ML introduces further coherence constraints that interact with the coherence results of Hilken and Rydeheard. Furthermore, Hilken and Rydeheard's semantics is based on a single application of qualified types, Haskell style type classes, and although it seems possible that their work may generalize to qualified types we have not considered this in detail.

Part II

Pragmatics

Chapter 6

A semantics for records and variants

Previous chapters of this dissertation have outlined a type system for extensible records and variants based on qualified types, and two possible definitions for the semantics of qualified types. However, hitherto we have not addressed the question of a semantics for qualified types including records and variants. In this chapter we give, as an application of the work in Chapter 4, a semantics for *OML* extended with records and variants.

The record and variant system of Chapter 3 used row extension and lacks predicates to assert that a specified row does or does not contain a given field. Early work by Harper and Pierce [HP90] and Jones [Jon94b] included extra, so called *has* predicates to assert the existence of fields in a record. However, it was later noted by Harper and Pierce [HP91] that the combination of extension and lacks predicates was enough to capture all types expressible using has predicates, although they did not formalize this remark. In this chapter we give an effective procedure, that given a type, possibly containing has predicates, results in a type without any has predicates. Applying the semantics developed in this chapter for *OML* extended with records and variants we show that this translation preserves meaning.

We split the presentation of a semantics for *OML* plus records and variants into sections. Section 6.1 describes a semantics for record and variant types. Section 6.2 considers the semantics of predicate entailment for lacks predicates. Section 6.3 gives a semantics to each of the record and variant primitives, concluding with a statement of soundness. Section 6.4 concludes this chapter answering, in the positive, the question 'are lacks predicates enough?'

6.1 A semantics for record and variant types

The semantic definition given in Chapter 4 (Section 4.6.1) requires that we provide an interpretation for monomorphic record and variant types in $T\Lambda$. Although one might simply extend $T\Lambda$ with records and variants, we consider records to be binary products and variants to be coproducts. One technical problem with this interpretation is that rows are considered equal up to reordering of fields and, as a consequence, a naive semantics may distinguish syntactically equivalent types. For example, consider the record type $\mathit{Rec}\{\!|l : \tau, l' : \tau'|\!\}$, which semantically can be interpreted as the types

$$\tau \times \tau'$$

and

$$\tau' \times \tau.$$

Of course, these types are isomorphic and so the choice of representation is not important, assuming that we are consistent. To provide a canonical representation we define the semantics of record and variant types in terms of products and coproducts, with respect to the total-ordering on labels. For example, if $l' < l$, then the record type $\mathit{Rec}\ \{\!|l : \mathit{Bool}, l' : \mathit{Int}|\!\}$ is represented as $\mathit{Int} \times \mathit{Bool}$. Of course, we could have chosen to represent the type $\mathit{Rec}\ \{\!|l : \mathit{Bool}, l' : \mathit{Int}|\!\}$ as $\mathit{Bool} \times \mathit{Int}$, however, for a canonical representation we must choose one and, consequently, we have selected the former interpretation.

Instead of describing a semantic interpretation for the kinds *row* and $*$ that we used to enforce the well-formedness of record and variant types in Chapter 3, we consider record and variants types to be expressed directly in the syntax of monotypes

$$\begin{array}{lcl} \tau & ::= & \ldots \mid \mathit{Rec}\ \mathit{row} \mid \mathit{Var}\ \mathit{row} \mid \ldots \\ \mathit{row} & ::= & \{\!||\!\} \mid \{\!|l : \tau \,|\, \mathit{row}|\!\} \end{array}$$

The main motivation for this choice is the lack of support of higher-order polymorphism in the semantics of Chapters 4 and 5. However, as discussed in Chapter 9 we see no real technical difficulty in extending either of the semantic definitions to support higher-order polymorphism, but we feel that it complicates the presentation unnecessarily for the purposes of this discussion.

We can now extend the interpretation of monotypes described in Definition 4.5 to provide a semantics for monomorphic record and variant types[1].

$$
\begin{array}{lcll}
[\![Rec\ \{\!|\,|\!\}]\!] & = & () & \\
[\![Var\ \{\!|\,|\!\}]\!] & = & 0 & \\
[\![Rec\ \{\!| l : \tau \,|\, r |\!\}]\!] & = & [\![Rec\ r]\!] \times \tau & \forall l' \in labs(r). l' < l \\
[\![Var\ \{\!| l : \tau \,|\, r |\!\}]\!] & = & [\![Var\ r]\!] + \tau & \forall l' \in labs(r). l' < l
\end{array}
$$

To provide an interpretation for variants in terms of coproducts, the monotypes of $T\Lambda$ must be extended to include coproducts. The cartesian closed category, $\mathbb{C}$, of Definition 4.5 becomes a bicartesian closed category with sums interpreted in the obvious way. All of the previous results, including Theorem 4.6, extend naturally to bicartesian closed categories.

6.2 A semantics for lacks predicates

In Chapter 3 we considered evidence for predicates of the form $(r\backslash l)$ to be an integer for the label l. In this chapter, we consider evidence for predicates to be projections (for decomposing a sum or product) and injections (for building sums and products). Thus, evidence for a predicate of the form $(r\backslash l)$, is a four tuple, (p, e, i, d), whose elements are as follows

- p: select the corresponding component of the product used to represent the record r with a component labelled l.
- e: insert the corresponding element into a record r, labelled l.
- i: inject a value into a coproduct with the corresponding label l.
- d: deconstruct a coproduct value with corresponding label l.

Note that the interpretation of lacks predicates differs from that given in Chapter 3. We return to this point once we have established soundness for *OML* plus records and variants.

For the interpretation of lacks predicates to be justified we must be able to express n-ary products/coproducts in an arbitrary (bi)cartesian closed category. However,

[1]We use the symbol 0 to denote the type of the initial object.

this is straightforward if one considers the existence of tuples in any cartesian closed category, asserted by the following lemma.

Lemma 6.1 (N-ary products) *If $\mathbb{C}$ is a cartesian closed category then for any objects $A_1, \cdots, A_n$ there exists an object $(A_1, \cdots, A_n)$, unique up to isomorphism, and arrows $\pi_i^n : (A_1, \cdots, A_n) \to A_i$, $\langle f, \cdots, f_n \rangle : A \to (A_1, \cdots, A_n)$, and $insert_i^n : (A_1, \cdots, A_n) \to [A_i \to (A_1, \times, A_i, \cdots, A_{n+1})]$ such that $\pi_i^n(A_1, \cdots, A_n) = A_i$ and $\pi_i \circ \langle f_1, \cdots, f_i, \cdots, f_n \rangle = f_i$.*

Proof: The proof of this lemma is standard and reproduced here to help with the general presentation.

Firstly define the object $(A_1, \cdots, A_n)$ by induction on n as

$$\begin{array}{lcl} () & = & \mathbb{1} \\ (A) & = & \mathbb{1} \times A \\ (A_1, \cdots, A_n) & = & (A_1, \cdots, A_{n-1}) \times A_n. \end{array}$$

The generalized projections are specified as

$$\begin{array}{lcll} \pi_n^n & = & \pi_2 & \\ \pi_i^n & = & \pi_i^{n-1} \circ \pi_1 & \quad i < n. \end{array}$$

By definition, it is clear that an object $(A_1, \cdots, A_n)$ is unique up to isomorphism. The split arrow can be defined as

$$\begin{array}{lcl} \langle\rangle & = & \mathbb{1} \\ \langle f \rangle & = & \langle !, f \rangle \\ \langle f_1, \cdots, f_n \rangle & = & \langle \langle f_1, \cdots, f_{n-1} \rangle, f_n \rangle. \end{array}$$

It is clear that the required equalities for projections and splits follow from analogous properties for categorical products. Now define the arrow

$$insert_i'^n : (A_1, \cdots, A_n) \times A_i \to (A_1, A_i, \cdots, A_{n+1})$$

as

$$\begin{array}{lcl} insert_n'^n & = & \langle \pi_1, id \rangle \\ insert_i'^n & = & \langle insert_i'^{n-1} \circ \langle \pi_1 \circ \pi_1, \pi_2 \rangle, \pi_2 \circ \pi_1 \rangle \end{array}$$

and thus, $insert_i^n = curry(insert_i'^n)$, as required.

(*This completes the proof.* $\Box$)

Similarly we can generalize categorical coproducts to n-ary coproducts. We do not work out the details of this fact here. Instead, we note that statement of such a lemma can be derived from Lemma 6.1 using $\mathbb{C}^{op}$ in place of $\mathbb{C}$, where $\mathbb{C}$ is the cartesian closed category of Lemma 6.1. We label the dualized projections and splits inj_i and $join_i$, respectively.

Finally, we are in a position to restate the rules for predicate entailment with evidence. The calculation of evidence is described by the rules in Figure 6.1.

$$P \Vdash (\pi_1, ins_1, in_1, join_1) : (\{\!|\,|\!\}\backslash l)$$

$$\frac{P \Vdash (p_i, p_i', s_i, s_i') : (r\backslash l)}{P \Vdash (\pi_m, ins_m, inj_m, join_m) : (\{\!|l' : \tau' \,|\, r|\!\}\backslash l)} \qquad m = \begin{cases} i, & l < l' \\ i+1, & l' < l \end{cases}$$

Figure 6.1: Predicate entailment for rows with evidence.

To conclude this section we observe that, to be able to apply Theorem 4.17 (soundness of *OML* theories) to *OML* plus our record and variants, we must be able to express evidence constructed by the rules in Figure 6.1 as $T\Lambda$ expressions. However, this introduces no new problems, as all of the projections, injections etc, described in Lemma 6.1 and its dual can be expressed in $T\Lambda$. In fact, they are defined as in the proof of Lemma 6.1.

6.3 A semantics for record and variant operations

For each primitive record and variant operation of Chapter 3 a corresponding operation must be added to *OML*, providing both a type and a suitable implementation. Of course, the types for these operations are just the types assigned to the original operations and all that is required here are the implementations. As an example implementation, record selection might be specified as

$$(\lambda v.\lambda r.\pi_1\ v\ r : \forall\alpha.\forall r.(r\backslash l) \Rightarrow Rec\{\!|l : \alpha \,|\, r|\!\} \rightarrow \alpha) \in C,$$

where the lambda bound variable v is evidence for the predicate $(r\backslash l)$. By Definition 4.16 we have

$$[\![\emptyset \vdash^{T\Lambda} (\lambda r : (Int, Bool).\pi_2^2\ r) : (Int, Bool) \rightarrow Bool]\!],$$

as an element of the set

$$[\![\emptyset\,|\,\emptyset \vdash (\lambda v.\lambda r.\pi_1\ v\ r) : \forall\alpha.\forall r.(r\backslash l) \Rightarrow Rec\{\!|l : \alpha\,|\,r|\!\} \rightarrow \alpha]\!],$$

at type $(Int, Bool) \rightarrow Bool$. By η-equivalence, we observe that record selection is simply an appropriate projection function, constructed with respect to a given record type. Figure 6.2 describes suitable implementations for the complete set of record and variant primitives given in Chapter 3.

$$
\begin{array}{lcl}
pRecSel & : & \forall\alpha.\forall r.(r\backslash l) \Rightarrow Rec\ \{\!|l : \alpha\,|\,r|\!\} \rightarrow \alpha \\
pRecSel & = & \pi_1 \\
\\
pRecExt & : & \forall\alpha.\forall r.(r\backslash l) \Rightarrow \alpha \rightarrow Rec\ r \rightarrow Rec\ \{\!|l : \alpha\,|\,r|\!\} \\
pRecExt & = & \pi_2 \\
\\
pVarInj & : & \forall\alpha.\forall r.(r\backslash l) \Rightarrow \alpha \rightarrow Var\{\!|l : \alpha\,|\,r|\!\} \\
pVarIn & = & \pi_3 \\
\\
pVarDe & : & \forall\alpha.\forall r.(r\backslash l) \Rightarrow Var\ \{\!|l : \alpha\,|\,r|\!\} \rightarrow (\alpha \rightarrow \beta) \rightarrow (Var\ r \rightarrow \beta) \rightarrow \beta \\
pVarDe & = & \pi_4
\end{array}
$$

Figure 6.2: Record and variant implementations.

Finally, to conclude this section, we now give a statement of soundness for *OML* with extensible records and variants as a corollary to Theorem 4.17. We shall write $P|C, A \vdash E =_{r+v} F : \sigma$ and $\mathcal{M} \models_{OML} P|C, A \vdash E =_{r+v} F : \sigma$ for syntactic and semantic equations, respectively.

Corollary 6.2 (Soundness of *OML* + extensible records and variants) *Let $\mathcal{M}$ be any model and $\mathcal{T}_{OML}$ an equational theory, then*

$$P|C, A \vdash E =_{r+v} F : \sigma \Rightarrow \mathcal{M} \models_{OML} P|C, A \vdash E =_{r+v} F : \sigma.$$

Proof: The proof follows directly by application of Theorem 4.17 and the fact that each of the record and variant operations are expressed as expressions in $T\Lambda$. Furthermore, the equalities for reasoning about records and variants can be derived directly from the provable equations for the corresponding equalities for products and coproducts in an arbitrary bicartesian closed category. It follows that these rules are sound with respect to our semantics.

(*This completes the proof.* □)

Although we have established the soundness of *OML* plus records and variants, we must still justify the implementation of records and variants described in Chapter 3. Fortunately, this is straightforward as the rules for predicate entailment given in Figure 3.6 are in one-to-one correspondence with those given in Figure 6.1. Furthermore, it is clear that the record and variant primitives given in Chapter 3 are suitable implementations, for the operations on n-ary products and coproducts described in this chapter.

6.4 Lacks predicates are enough

Chapter 3 introduced predicates of the form $(r \backslash l)$ to assert that the row r does not contain a field l, and used row extension to extend a row with a new label. Thus lacks predicates represent the assertion of 'negative' information, while extension captures 'positive' information about the fields appearing in a given row. However, it is also possible to dualize this notion, introducing predicates of the form[2]

$$r\ \mathit{has}\ (l : \tau)$$

asserting the existence of a label l in r, and using row restriction instead of row extension, to remove a label from a given row[3]. Thus has predicates convey positive information, while negative assertions are transfered into row expressions. As an example, consider the type of Jones' [Jon94b] record restriction, which is stated using only positive predicates and row restriction

$$(\_ - l) : \forall \alpha. \forall r. (r\ \mathit{has}\ (l : \alpha)) \Rightarrow \mathit{Rec}\ r \rightarrow \mathit{Rec}\ (r - l).$$

Intuitively, at least, we expect the different types for record restriction to represent the same semantic value. But why choose one over the other or, more importantly, is it possible to write down a type using positive predicates that does not have a corresponding type using only lacks predicates? If this question is answered in the

[2]The type τ is required to preserve type soundness.

[3]As discussed in Chapter 3, the introduction of row restriction into the type language leads to the loss of most general unifiers. However, this is not relevant in the context of the current discussion.

positive, then there are quite reasonable expressions over record and variants that the type system of Chapter 3 will reject. Why then did we choose to use, only lacks predicates? This question has been asked before in the work of Harper and Pierce [HP91], who initially used both positive and negative constraints [HP90], but later refined use to predicates capturing only negative information. In answer to this question Harper and Pierce conjectured that lacks predicates were enough: Any type constrained by has predicates could be translated into a type constrained only by lack predicates, while preserving semantic meaning. In the remainder of this section we describe a formal procedure for performing just such a translation. Building upon the semantics of this and previous chapters, we show that the semantics of the original and translated types coincide. Thus we give a proof of Harper and Pierce's conjecture, justifying the use of only negative constraints. As a side effect, the translation makes it possible to translate a type containing row restriction and corresponding positive constraints (which can lead to the failure of most general unifiers) into one containing row extension and negative constraints—regaining most general unifiers.

The remaining sections are as follows. Section 6.4.1 extends the predicates and monotypes of Chapter 3 to include has predicates and row restriction and describes a simple semantics. Section 6.4.2 presents a translation from types, which may contain has predicates, into types containing only lacks predicates. Finally, Section 6.5 applies the translation procedure to Jones' [Jon94b] record restriction, resulting in a type for record restriction that is the same type as assigned to record restriction in this dissertation.

6.4.1 Types and semantics

To keep the presentation simple, we consider record types only, making the observation that variant types can be catered for in a similar fashion. Following Chapter 3, we distinguish between types, τ, type schemes, σ, and qualified types, ρ, as described by the following grammar

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \mathit{Rec}\ \mathit{row} \mid \tau \to \tau & \mathit{monotypes} \\
\mathit{row} & ::= & r \mid \{\!|\,|\!\} \mid \{\!|\, l : \tau \,|\, \mathit{row} \,|\!\} \mid \mathit{row} - l & \mathit{rows} \\
\rho & ::= & P \Rightarrow \tau & \mathit{qualified\ types} \\
\sigma & ::= & \forall \alpha . \rho \mid \rho & \mathit{type\ schemes}
\end{array}
$$

As before, the symbols α and r range over countable sets of disjoint variables. P represents sequences of predicates, described by the following grammar

$$\begin{array}{rcll} P & ::= & \emptyset & \textit{empty sequence} \\ & | & (r\ \mathit{has}\ l : \tau), P & \textit{positive constraints} \\ & | & r \backslash l, P & \textit{negative constraints.} \end{array}$$

To enforce well-formedness of types, we require that any row of the form

$$\{\!| l : \alpha \,|\, r |\!\} \quad \text{and} \quad (r - l),$$

is constrained by corresponding predicates

$$r \backslash l \quad \text{and} \quad r\ \mathit{has}\ l : \alpha,$$

respectively. Furthermore, we require that for any predicate of the form $(r \backslash l)$ or $(r\ \mathit{has}\ l : \alpha)$ the row r is always a variable. This does not introduce any new problems as the rules for predicate entailment can be used to simplify (normalize) any, well-formed, predicate to one of these forms. These restrictions are enough to ensure that $\{\!| l : \_ | \_ |\!\}$ and $\_ - l$ are total functions.

Following Section 6.1, we describe the semantics of monotypes by giving a translation into $T\Lambda$, as follows

$$\begin{array}{lcll} [\![\tau \rightarrow \tau]\!] & = & [\![\tau]\!] \rightarrow [\![\tau]\!] & \\ [\![\mathit{Rec}\ \{\!|\,|\!\}]\!] & = & () & \\ [\![\mathit{Rec}\ \{\!| l : \tau \,|\, r |\!\}]\!] & = & [\![\mathit{Rec}\ r]\!] \times [\![\tau]\!] & \forall l' \in \mathit{labs}(r). l' < l \\ [\![\mathit{Rec}\ (\{\!| l' : \tau \,|\, r |\!\} - l)]\!] & = & [\![\mathit{Rec}\ (r - l)]\!] \times [\![\tau]\!] & \forall l'' \in \mathit{labs}(r). l'' < l \wedge l \neq l' \\ [\![\mathit{Rec}\ (\{\!| l : \tau \,|\, r |\!\} - l)]\!] & = & [\![\mathit{Rec}\ r]\!] & \end{array}$$

As before, the semantics of a type scheme, σ, is determined by the semantics of its monomorphic instances, as follows

$$[\![\forall \alpha . P \Rightarrow \tau]\!] \quad = \quad \{[\![[\nu/\alpha]\tau]\!] \mid \nu \in \mathit{Type}, \Vdash [\nu/\alpha]P\}.$$

6.4.2 Translation from *has* to *lacks*

The rules in Figure 6.3 define a translation relation that, given a type scheme σ (possibly containing both positive, and negative predicates), results in a type

$$(scheme)\quad \frac{Q = hasP(P) \quad Q' = lacksP(P) \quad (Q \Rightarrow \tau) \leadsto (P' \Rightarrow \tau')}{(\forall\alpha.P \Rightarrow \tau) \leadsto (\forall\alpha.Q' \cup P' \Rightarrow \tau')}$$

$$(empty)\quad \frac{}{\{\} \Rightarrow \tau \leadsto \{\} \Rightarrow \tau}$$

$$(\text{arrow})\quad \frac{P \Rightarrow \tau \leadsto P' \Rightarrow \tau'' \quad P \Rightarrow \tau' \leadsto P'' \Rightarrow \tau'''}{P \Rightarrow \tau \rightarrow \tau' \leadsto P' \cup P'' \Rightarrow \tau'' \rightarrow \tau'''}$$

$$(minus)\quad \frac{}{\begin{array}{c}(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ ((r - l_1) \cdots - l_n) \leadsto \\ (r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r\end{array}}$$

$$(noHas)\quad \frac{r\ has\ \_ \not\in P}{P \Rightarrow Rec\ r \leadsto \{\} \Rightarrow Rec\ r}$$

$$(ext)\quad \frac{r\ has\ \_ \not\in P}{\begin{array}{l}(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ r \leadsto \\ \qquad (r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\{\!|l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r|\!\}\end{array}}$$

$$(noHE)\quad \frac{r\ has\ \_ \not\in P}{P \Rightarrow Rec\{\!|l_1 : \tau_1 \,|\, r|\!\} \leadsto \{\} \Rightarrow Rec\{\!|l_1 : \tau_1 \,|\, r|\!\}}$$

$$(hasExt)\quad \frac{r\ has\ \_ \not\in P}{\begin{array}{l}(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ \{\!|l'_1 : \tau'_1, \cdots, l'_k : \tau'_k \,|\, r|\!\} \leadsto \\ \quad (r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\{\!|l'_1 : \tau'_1, \cdots, l'_k : \tau'_k, l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r|\!\}\end{array}}$$

$$(empR)\quad \frac{}{P \Rightarrow Rec\{\!||\!\} \leadsto \{\} \Rightarrow Rec\{\!||\!\}}$$

$$(var)\quad \frac{}{P \Rightarrow \alpha \leadsto \{\} \Rightarrow \alpha}$$

Figure 6.3: Translation.

scheme σ' (containing only negative predicates). A judgement of the form $\sigma \leadsto \sigma'$ asserts that the type scheme σ has translation σ'. The (*scheme*) rule uses two auxiliary functions to calculate predicates sets containing only has or lacks predicates, defined as follows

$$\begin{array}{lcl} hasP(P) & = & \{(r\ has\ l : \tau) \mid (r\ has\ l : \tau) \in P, l \in L, \tau \in Type, r \in row\} \\ lacksP(P) & = & \{(r \backslash l) \mid (r \backslash l) \in P, l \in L, r \in row\}. \end{array}$$

Furthermore, a number of the rules in Figure 6.3 can only be applied on the assertation that a predicate set P does not contain any specific has predicates (see the (*ext*) rule, for example). The following definition captures this notion

$$r\ has\ \_ \notin P \quad = \quad \{(r\ has\ l : \tau) \mid (r\ has\ l : \tau) \in P, l \in L, \tau \in Type\} = \{\}.$$

Before answering Harper and Pierce's [HP91] original conjecture—lacks predicates are enough— we define the auxiliary function, *nHas*, which, given a type scheme, σ, calculates the number of predicates of the form $(r\ has\ l : \tau)$ in σ, as follows

$$\begin{array}{lcl} nHas\ (\forall \alpha.P. \Rightarrow \tau) & = & nHas'\ P \\ \quad \textbf{where} & & \\ \quad nHas'\ \emptyset & = & 0 \\ \quad nHas'\ (\_has\_, P) & = & 1 + nHas'\ P \\ \quad nHas'\ (\_\backslash\_, P) & = & nHas'\ P \end{array}$$

The following proposition confirms that, for any type scheme, σ, there exists a unique type scheme, σ', such that the semantics of σ and σ' coincide, and σ' contains only negative information.

Proposition 6.3 (Lacks predicates are enough) *If σ is a type scheme, then there exists a type scheme, σ', such that $\sigma \leadsto \sigma'$, $nHas(\sigma') = 0$, and $[\![\sigma]\!] = [\![\sigma']\!]$. Furthermore, if there exists a type scheme, σ'', such that $\sigma \leadsto \sigma''$, then $\sigma' = \sigma''$.*

A proof of this result is given in Section A.4.1 of Appendix A.

6.5 Record restriction—an example

In this section, we give an example of the translation process. For this we consider Jones'[Jon94b] type for record restriction

$$(\_ - l) : \forall \alpha.\forall r.(r\ has\ (l : \alpha)) \Rightarrow Rec\ r \rightarrow Rec\ (r - l).$$

The following derivation shows that, modulo the translation rules of Figure 6.3, Jones' type for record restriction corresponds to the type assigned to record restriction in Chapter 3.

$$\dfrac{\dfrac{\begin{array}{c}(r\ \mathit{has}\ l:\alpha) \Rightarrow \mathit{Rec}\ r \leadsto (r\backslash l) \Rightarrow \mathit{Rec}\{\!|\, l:\alpha \,|\, r\,|\!\}\\ (r\ \mathit{has}\ l:\alpha) \Rightarrow \mathit{Rec}\ (r-l) \leadsto (r\backslash l) \Rightarrow \mathit{Rec}\ r\end{array}}{(r\ \mathit{has}\ l:\alpha) \Rightarrow \mathit{Rec}\ r \to \mathit{Rec}\ (r-l) \leadsto (r\backslash l) \Rightarrow \mathit{Rec}\{\!|\, l:\alpha \,|\, r\,|\!\} \to \mathit{Rec}\ r}}{\forall\alpha.\forall r.(r\ \mathit{has}\ l:\alpha) \Rightarrow \mathit{Rec}\ r \to \mathit{Rec}\ (r-l) \leadsto \forall\alpha.\forall r.(r\backslash l) \Rightarrow \mathit{Rec}\{\!|\, l:\alpha \,|\, r\,|\!\} \to \mathit{Rec}\ r}$$

The derivation is proved by application of the translation rules, from top to bottom, (*ext*), (*minus*), (*arrow*), and (*scheme*).

Jones also assigns the following type to record selection

$$(\_.l) : \forall\alpha.\forall r.(r\ \mathit{has}\ (l:\alpha)) \Rightarrow \mathit{Rec}\ r \to \alpha,$$

which, under our translation, becomes

$$(\_.l) : \forall\alpha.\forall r.(r\backslash l) \Rightarrow \mathit{Rec}\{\!|\, l:\alpha \,|\, r\,|\!\} \to \alpha,$$

and is of course the record selection of Chapter 3.

The translation procedure, given in Figure 6.3, combined with Proposition 6.3 is enough to justify our restriction to lacks predicates and row extension.

One question we have not answered here is whether the relation $\leadsto$ is in fact a bijection. If this is the case then has predicates would also be enough, and thus, in a system where most general unifiers are not important (for example, Harper and Pierce's [HP91] system is explicitly typed, and consequently, does require type inference) one could use positive predicates and row restriction rather than negative predicates and row extension.

Chapter 7

Rows, labels and casting

Previous chapters of this dissertation have been concerned with the theoretical basis for a calculus of extensible records and variants and with a selection of semantic models for this system and for more general systems of qualified types. This chapter considers a number of extensions to our original calculi, which may be useful when adapting our proposal to realistic programming languages.

The chapter is split into three sections. Section 7.1, inspired by category theory and logic, studies the importance of rows, introducing a collection of generalized operations for constructing and deconstructing records and variants. Section 7.2 considers labels as first class values. Finally, Section 7.3 considers encoding objects as records, paying particular attention to an operation that is known as 'casting' in a wide selection of strongly typed object-oriented programming languages.

7.1 Row polymorphism

Working with a general notion of rows has provided us with an elegant way to deal with the common structure in record and variant types. However, we have not seen any examples in the previous sections where it was essential to consider rows separately from records and variants; we could have just defined completely independent sets of record and variant types.

However, there are some applications in which the ability to separate rows from records and variants offers significant benefits. To illustrate this, consider again the basic operations that were discussed in Chapter 3. For example, if we generalize the rules for decomposing a sum to deal with n-ary sums, then we obtain the following rule

$$\frac{A_1 \rightarrow C \quad \ldots \quad A_n \rightarrow C}{A_1 + \ldots + A_n \rightarrow C}.$$

In terms of records and variants, this rule provides a method for decomposing a variant—represented by the sum $A_1 + \ldots + A_n$ in the conclusion—using a record of functions—represented by the hypotheses $A_i \rightarrow C$. This suggests a general operation for variant elimination

$$\mathit{sumElim} : \forall\alpha.\forall r.\mathit{Rec}\ (\mathit{to}\ \alpha\ r) \rightarrow \mathit{Var}\ r \rightarrow \alpha.$$

The $\mathit{to}\ \alpha\ r$ construct used here is defined as follows

$$\begin{array}{lcl} \mathit{to}\ \tau\ \{\!|\,|\!\} & = & \{\!|\,|\!\} \\ \mathit{to}\ \tau\ \{\!|\,l : \tau' \,|\, r\,|\!\} & = & \{\!|\,l : \tau' \rightarrow \tau \,|\, \mathit{to}\ \tau\ r\,|\!\}. \end{array}$$

The function expression $\mathit{to}\ \alpha\ r$ behaves like a particular kind of *map* operation on rows, replacing each component type τ' in r with a type of the form $\tau' \rightarrow \alpha$. For example, the expression $\mathit{to}\ \mathit{Char}\ \{\!|\,l : \mathit{Bool}, l' : \mathit{Int}\,|\!\}$ evaluates to the row $\{\!|\,l : \mathit{Bool} \rightarrow \mathit{Char}, l' : \mathit{Int} \rightarrow \mathit{Char}\,|\!\}$. Similar operations have been suggested in earlier work by Rémy [Rém92b] and by Hofmann and Pierce [HP95].

For an example of where such an operation may prove useful, consider the type of integer lists that can be obtained as a fixpoint of the following functor [MH95]

$$\mathit{data}\ L\ l \quad = \quad L\ (\mathit{Var}\ \{\!|\,\mathit{nil} : \mathit{Rec}\ \{\!|\,|\!\}, \mathit{cons} : \mathit{Rec}\ \{\!|\,\mathit{tl} : l, \mathit{hd} : \mathit{Int}\,|\!\}\,|\!\}).$$

The sum of a list of integers can be calculated using a general catamorphism

$$\mathit{cata}\ (\lambda(L\ v).\mathit{sumElim}\ (\mathit{nil} = \lambda().0, \quad \mathit{cons} = \lambda r.(r.\mathit{hd}) + (r.\mathit{tl}))\ v).$$

From this example, it is clear that *sumElim* is an alternative to the **case** construct of languages like Haskell and SML. However, unlike these languages, it is not a built-in part of the syntax; instead, it allows us to treat **case** constructs as first-class, extensible values.

The *sumElim* operation that we have considered here is just one of four operators that correspond to the standard ways of constructing or deconstructing sums or products shown in Figure 7.1. The full set of operations are specified by the

$$\frac{A_1 \to C \quad \dots \quad A_n \to C}{A_1 + \dots + A_n \to C} \qquad \frac{A \to C_1 + \dots + A \to C_n}{A \to C_1 + \dots + C_n}$$

$$\frac{A_1 \to C + \dots + A_n \to C}{A_1 \times \dots \times A_n \to C} \qquad \frac{A \to C_1 \quad \dots \quad A \to C_n}{A \to C_1 \times \dots \times C_n}$$

Figure 7.1: Rules for product and sums.

following type signatures

$$\begin{array}{lcl} \mathit{sumElim} & : & \forall \alpha.\forall r.\mathit{Rec}\ (\mathit{to}\ \alpha\ r) \to \mathit{Var}\ r \to \alpha \\ \mathit{sumIntro} & : & \forall \alpha.\forall r.\mathit{Var}\ (\mathit{from}\ \alpha\ r) \to \alpha \to \mathit{Var}\ r \\ \mathit{prodElim} & : & \forall \alpha.\forall r.\mathit{Var}\ (\mathit{to}\ \alpha\ r) \to \mathit{Rec}\ r \to \alpha \\ \mathit{prodIntro} & : & \forall \alpha.\forall r.\mathit{Rec}\ (\mathit{from}\ \alpha\ r) \to \alpha \to \mathit{Rec}\ r. \end{array}$$

The $\mathit{from}\ \alpha\ r$ construct used in the types of *sumIntro* and *prodIntro* is as an obvious dual to $\mathit{to}\ \alpha\ r$. An expression of the form $\mathit{from}\ \alpha\ r$ replaces each component of type τ in r with a component of type $\alpha \to \tau$. It is defined by the following definition

$$\begin{array}{lcl} \mathit{from}\ \tau\ \{\!|\,|\!\} & = & \{\!|\,|\!\} \\ \mathit{from}\ \tau\ \{\!| l : \tau' \,|\, r |\!\} & = & \{\!| l : \tau \to \tau' \,|\, \mathit{from}\ \tau\ r |\!\}. \end{array}$$

Given our earlier representations for records and variants, it is easy to implement each of these operations as built-in primitives. For example, a suitable implementation for *sumElim* might be

$$\mathit{sumElim}\ r\ \langle l = x \rangle = (r.l)\ x,$$

It is easy to see that this implementation is sound because $\mathit{sumElim's}$ type guarantees that, whatever summand l represents, the corresponding function will be present in the record r. Figure 7.2, gives implementations for each of the generalized operators. From a practical perspective it is straightforward to implement each of these operations, with the possible exception of *prodIntro*, which requires additional information about the length of a given record.

One technical difficulty that we face with this approach is in extending the treatment of unification to deal with uses of the *from* and *to* constructs. This turns out to be straightforward, except for potential complications caused by the presence of empty rows. For example, in unifying two rows $\mathit{to}\ \tau\ r$ and $\mathit{to}\ \tau'\ r'$, we cannot

$$
\begin{array}{lcl}
\mathit{sumElim}\ r\ \langle l = x\rangle & = & (r.l)\ x \\
\mathit{sumIntro}\ \langle l = f\rangle\ x & = & \langle l = f\ x\rangle \\
\mathit{prodElim}\ \langle l = f\rangle\ (l = x \,|\, \_) & = & f\ x \\
\mathit{prodIntro}\ (l_1 = f_1, \ldots, l_n = f_n)\ x & = & (l_1 = f_1\ x, \ldots, l_n = f_n\ x)
\end{array}
$$

Figure 7.2: Implementations for generalized operators.

simply unify τ with τ'; if r, and hence r', is empty, then there may not be any direct relationship between τ and τ'. More precisely, to obtain most general unifiers for *from* and *to* constructs, we need to restrict ourselves to work with non-empty rows.

7.1.1 Non-empty rows

Of course, rows are still built by extension but instead of starting with the empty row, the singleton row constructor is used in its place:

$$\{\!|\, l : \_ \,|\!\} : * \to \mathit{row}.$$

The following singleton record constructor is included as an additional primitive:

$$(l = \_) : \forall\alpha.\alpha \to \mathit{Rec}\ \{\!|\, l : \alpha \,|\!\}.$$

With the restriction to non-empty rows we must check that the types of the original primitives (e.g., record selection, extension, and restriction) are still correct. The point here is that a row of the form $\{\!|\, l : \tau \mid r \,|\!\}$ now has at least two elements—due to the restriction to non-empty rows, the row variable r must include at least one field. As a consequence the record selection operator of Chapter 3 cannot be applied to a singleton record. However, there is a straightforward solution to this problem: Assign an alternative type to record selection—one that can be unified with both singleton and extended records. In fact, we have already seen this type for record selection in Chapter 6 and is the type assigned by Jones [Jon94b] using *has* predicates, instead of lacks predicates, to constrain r to records containing an l field:

$$(\_.l) : \forall\alpha.\forall r.(r\ \mathit{has}\ l : \alpha) \Rightarrow \mathit{Rec}\ r \to \alpha.$$

The row variable r unifies with both singleton row and extended row types.

With the inclusion of *has* predicates we must extend the definition of predicate entailment, given in Figure 3.6, to include rules for determining evidence for both *has* and *lacks*. Furthermore, we must restrict entailment such that it is defined over non-empty rows, only. The complete set of rules for predicate entailment supporting non-empty rows and including *has* predicates is given in Figure 7.3.

$$P \cup \{v : \pi\} \Vdash v : \pi$$

$$\frac{P \Vdash e : (r \backslash l)}{P \Vdash m : (\{\!|l' : \tau \mid r|\!\} \backslash l)} \qquad m = \begin{cases} e, & l < l' \\ e + 1, & l' < l \end{cases}$$

$$P \Vdash m : (\{\!|l' : \tau|\!\} \backslash l) \qquad m = \begin{cases} 0, & l < l' \\ 1, & l' < l \end{cases}$$

$$\frac{P \Vdash e : (r \backslash l)}{P \Vdash m : (\{\!|l' : \tau \mid r|\!\}\ \mathit{has}\ (l : \tau))} \qquad m = \begin{cases} e, & l < l' \\ e + 1, & l' < l \end{cases}$$

$$P \Vdash 0 : (\{\!|l : \tau|\!\}\ \mathit{has}\ (l : \tau))$$

Figure 7.3: Predicate entailment for rows with evidence.

It is important to note that the algorithm given in Chapter 6, to translate types containing *has* predicates into types containing only *lacks* predicates, is not valid in the current setting. It is clear that this is the case, by the argument given above for changing the type of record selection. To see this note that the original type for record selection restricted field selection to records with at least two components, while the alternative (including *has* predicates) is defined for all records containing the field l. However, this is not a limitation of the translation procedure, rather a consequence of restricting row expressions to be non-empty. From a logical perspective, this is like throwing away the terminal object in a cartesian closed category, and so it is perhaps not surprising that certain equalities are lost.

Fortunately, the types of the other operations (e.g., extension, injection) remain unchanged. However, with the introduction of *has* predicates we must mention problems of type ambiguity and satisfiability. Consider the following type

$$\forall\alpha.\forall r.(\mathit{Num}\ \alpha, r\ \mathit{has}\ l : \alpha) \Rightarrow \mathit{Rec}\ r \rightarrow \mathit{Rec}\ r,$$

which, under the standard notion of ambiguity[1], would be rejected. It is possible to weaken the notion of ambiguous type to allow for this and other similar examples.

We conclude this section with a small but important point concerning the notion of satisfiability for constrained types. Consider the following type

$$\forall\alpha.\forall r.(r\ \mathit{has}\ l : \alpha, r\ \mathit{has}\ l : \beta) \Rightarrow \mathit{Rec}\ r \rightarrow (\alpha, \beta),$$

which to make sense semantically must enforce $\alpha = \beta$. Fortunately, this equality can be induced by the notion of principal satisfiability proposed by Jones [Jon95b]. An implementation can make use of this fact to reduce the above type to the corresponding type

$$\forall\alpha.\forall r.(r\ \mathit{has}\ l : \alpha) \Rightarrow \mathit{Rec}\ r \rightarrow (\alpha, \alpha).$$

7.1.2 Unification of non-empty rows

To conclude the discussion of row polymorphism we consider the problem of unification for our extended type language, which now includes types of the form $\mathit{to}\ \tau\ r$ and $\mathit{from}\ \tau\ r$.

Formally, the set of type constructors is extended with the constants[2]

$$\begin{array}{lcl} \mathit{to} & : & * \rightarrow \mathit{row} \rightarrow \mathit{row} \\ \mathit{from} & : & * \rightarrow \mathit{row} \rightarrow \mathit{row}. \end{array}$$

As a consequence the set of rules for calculating unifiers (inserters), given in Figure 3.3 (Figure 3.4), must be extended to include cases for types of the form $\mathit{to}\ \tau\ r$ and $\mathit{from}\ \tau\ r$. These extensions are straightforward, except for expressions of the form

$$\mathit{to}\ \tau\ r {\sim} \mathit{from}\ \tau'\ r'.$$

It is clear that if the types $\mathit{to}\ \tau\ r$ and $\mathit{from}\ \tau'\ r'$ are to unify then the set of labels in r and r' must be equal. Furthermore, each type assigned to a label l in r must

[1] Jones classed any type that contained a reference to a variable in the set of predicates but not in the constrained type itself as ambiguous.

[2] We disallow partial application of these constructors.

be unifiable with τ', and each type assigned to a label l' in r' must be unifiable to τ.

To capture this observation formally we introduce a new kind, *labels*, denoting sets of labels, and two type constructors for building label sets

$$\begin{array}{lcl} \{\!|l|\!\} & : & \mathit{labels} \\ \{\!|l \mid \_|\!\} & : & \mathit{labels} \rightarrow \mathit{labels}. \end{array}$$

Note that sets of labels correspond to rows without types for the labels and that duplicate labels are not allowed. Inserters are defined as for rows but without regard for the type of a given label.

The following constructor, *array*, is used to construct rows in which each label is assigned the same type (supplied as the first argument to *array*)

$$\begin{array}{lcl} \mathit{array} & : & * \rightarrow \mathit{labels} \rightarrow \mathit{row} \\ \mathit{array}\ \tau\ \{\!|l|\!\} & = & \{\!|l : \tau|\!\} \\ \mathit{array}\ \tau\ \{\!|l \mid r|\!\} & = & \{\!|l : \tau \mid \mathit{array}\ \tau\ r|\!\}. \end{array}$$

As an example, consider a type to represent two-dimensional points, which might be expressed as $\mathit{Rec}\ \{\!|x : \mathit{Int}, y : \mathit{Int}|\!\}$, using the notation of Chapter 3. Using the constructor *array* this type can be denoted as $\mathit{array}\ \mathit{Int}\ \{x, y\}$.

The rules for unification (inserters), given in Figure 3.3 (Figure 3.4), are extended to include the additional rules in Figure 7.4 (Figure 7.5)[3]. The important properties of unification and insertion—both soundness and completeness—captured in Theorem 3.2, extended to include the restriction to nonempty rows and the additional rules for unification.

Theorem 7.1 *The unification (insertion) algorithm defined by the rules in Figure 3.3 (Figure 3.4), extended to include the additional rules given in Figure 7.4 (Figure 7.5), calculates most-general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.*

A proof of this result is given in Section A.5.1 of Appendix A.

Having established an algorithm for calculating most-general unifiers for constructors including non-empty rows, *to* and *from* we can again use the type inference and compilation algorithms for qualified types.

[3] Technically the rules for unification (insertion) given in Figures 3.3 and 3.4 must be changed to unify (insert) non-empty rows. However, this is straightforward.

$$\frac{\tau \overset{U}{\sim} \tau' \quad Ur \overset{U'}{\sim} Ur'}{\mathit{array}\ \tau\ r \overset{U'U}{\sim} \mathit{array}\ \tau\ r}$$

$$\frac{\tau \overset{U}{\sim} \alpha \to \tau' \quad Ur \overset{U'}{\sim} U(\mathit{array}\ \alpha\ \mathit{fs}) \quad \alpha, \mathit{fs}\ \text{new}}{\mathit{array}\ \tau\ \mathit{fs} \overset{U'U}{\sim} \mathit{to}\ \tau'\ r}$$

$$\frac{\tau \overset{U}{\sim} \tau' \to \alpha \quad Ur \overset{U'}{\sim} U(\mathit{array}\ \alpha\ \mathit{fs}) \quad \alpha, \mathit{fs}\ \text{new}}{\mathit{array}\ \tau\ \mathit{fs} \overset{U'U}{\sim} \mathit{from}\ \tau'\ r}$$

$$\frac{r \overset{U}{\sim} \mathit{array}\ \tau'\ \mathit{fs} \quad Ur' \overset{U'}{\sim} \mathit{array}\ (U\tau)\ (U\mathit{fs}) \quad \mathit{fs}\ \mathit{new}}{\mathit{to}\ \tau\ r \overset{U'U}{\sim} \mathit{from}\ \tau'\ r'}$$

$$\frac{\tau \overset{U}{\sim} \tau' \quad (l) \overset{I}{\in} U\mathit{fs}}{\mathit{array}\ \tau\ \mathit{fs} \overset{IU}{\sim} \{\!|l : \tau'|\!\}}$$

$$\frac{\tau \overset{U}{\sim} \tau' \quad (l) \overset{I}{\in} U\mathit{fs} \quad \mathit{array}\ (IU\tau)\ (IU\mathit{fs} - l) \overset{U'}{\sim} IUr}{\mathit{array}\ \tau\ \mathit{fs} \overset{U'IU}{\sim} \{\!|l : \tau \,|\, r|\!\}}$$

$$\frac{\tau' \overset{U}{\sim} \alpha \to \tau \quad Ur \overset{U'}{\sim} \{\!|l : U\alpha \,|\, r''|\!\} \quad \mathit{to}\ (U'U\tau)\ (U'Ur'')) \overset{U''}{\sim} U'Ur' \quad \alpha, r''\ \text{new}}{\mathit{to}\ \tau\ r \overset{U''U'U}{\sim} \{\!|l : \tau' \,|\, r'|\!\}}$$

$$\frac{\tau \overset{U}{\sim} \tau' \quad Ur \overset{U'}{\sim} Ur'}{\mathit{to}\ \tau\ r \overset{U'U}{\sim} \mathit{to}\ \tau'\ r'}$$

$$\frac{\tau' \overset{U}{\sim} \tau \to \alpha \quad Ur \overset{U'}{\sim} \{\!|l : U\alpha \,|\, r''|\!\} \quad \mathit{from}\ (U'U\tau)\ (U'Ur'')) \overset{U''}{\sim} U'Ur' \quad \alpha, r''\ \text{new}}{\mathit{from}\ \tau\ r \overset{U''U'U}{\sim} \{\!|l : \tau' \,|\, r'|\!\}}$$

$$\frac{\tau \overset{U}{\sim} \tau' \quad Ur \overset{U'}{\sim} Ur'}{\mathit{from}\ \tau\ r \overset{U'U}{\sim} \mathit{from}\ \tau'\ r'.}$$

Figure 7.4: Additional rules for unification.

$$\frac{\tau \overset{U}{\sim} \tau' \quad (l) \overset{I}{\in} U fs}{(l : \tau) \overset{IU}{\in} \mathit{array}\ \tau\ fs}$$

$$\frac{\tau \overset{U}{\sim} \alpha \to \tau' \quad (l : U\alpha) \overset{I}{\in} Ur \quad \alpha \text{ new}}{(l : \tau) \overset{IU}{\in} \mathit{to}\ \tau'\ r}$$

$$\frac{\tau \overset{U}{\sim} \tau' \to \alpha \quad (l : U\alpha) \overset{I}{\in} Ur \quad \alpha \text{ new}}{(l : \tau) \overset{IU}{\in} \mathit{from}\ \tau'\ r}$$

Figure 7.5: Additional rules for insertion.

7.2 Labels

In previous chapters of the dissertation, we have considered the labels used to refer to fields as part of the basic syntax of our language. As a result, we had to describe selection from a record using a family of functions

$$(\_.l) : \forall\alpha.\forall r.(r\backslash l) \Rightarrow \mathit{Rec}\ \{\!|\, l : \alpha \,|\, r \,|\!\} \to \alpha,$$

with one function for each choice of label l. A more attractive approach is to allow primitive operations on records and variants to be parameterized over labels. We can extend the type system described in previous chapters to accomplish this, treating selection, for example, as a single function of type

$$(\_.\_) : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Rec}\ \{\!|\, l : \alpha \,|\, r \,|\!\} \to \mathit{Label}\ l \to \alpha.$$

This requires another extension of the kind system in Section 3.1.1 with a new kind lab, and also a new type constant Label of kind $\mathit{lab} \to *$. Intuitively, each type of the form $\mathit{Label}\ l$ contains a unique label value, often referred to as singleton types in type theory. The l parameter is important because it establishes a connection between types and label values; a nullary Label type would not have provided any way for us to express the necessary typing constraints. We can also dispense with the family of extension constructors defined in Section 3.1.2, replacing them with a single constructor constant[4]

$$\{\!|\_ : \_ \,|\, \_|\!\} : \mathit{lab} \to * \to \mathit{row} \to \mathit{row}.$$

[4]Again we do not allow partial application.

Finally, we need to generalize the lacks predicate from Section 3.1.3 to a two place relation $r\backslash l$ which takes both a row r and a label l of kind *lab*. This can be defined in the same way as before, and has the same interpretation as an offset value in the underlying implementation.

We can generalize the other basic operations on records and variants in a similar way. For example, the expression $\lambda x.\lambda y.(y = 2, x = 3)$, which involves two uses of extension, will be assigned the type

$$\{\!|x : \mathit{Int}|\!\}\backslash y \Rightarrow \mathit{Label}\ x \to \mathit{Label}\ y \to \mathit{Rec}\ \{\!|x : \mathit{Int}, y : \mathit{Int}|\!\}.$$

The generalized types for each of the specific record and variants of Chapter 3 are

- Selection: extract a value

$$(\_.\_) : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Rec}\ \{\!|l : \alpha \,|\, r|\!\} \to \mathit{Label}\ l \to \alpha.$$

- Restriction: remove a field

$$(\_ - \_) : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Rec}\ \{\!|l : \alpha \,|\, r|\!\} \to \mathit{Label}\ l \to \mathit{Rec}\ r.$$

- Extension: to add a field l to an existing record

$$(\_ = \_|\_) : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Label}\ l \to \alpha \to \mathit{Rec}\ r \to \mathit{Rec}\ \{\!|l : \alpha \,|\, r|\!\}.$$

- Injection: to tag a value

$$\langle\_ = \_\rangle : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Label}\ l \to \alpha \to \mathit{Var}\ \{\!|l : \alpha \,|\, r|\!\}.$$

- Embedding: to embed a value in a variant type

$$\langle\_|\_\rangle : \forall\alpha.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Label}\ l \to \mathit{Var}\ r \to \mathit{Var}\ \{\!|l : \alpha \,|\, r|\!\}.$$

- Decomposition: to act on the value in a variant

$$\begin{aligned}(\_ \in \_ ? \_ : \_) : \forall\alpha.\forall\beta.\forall r.\forall l.(r\backslash l) \Rightarrow\ & \mathit{Label}\ l \\ & \to \mathit{Var}\ \{\!|l : \alpha \,|\, r|\!\} \\ & \quad \to (\alpha \to \beta) \\ & \qquad \to (\mathit{Var}\ r \to \beta) \\ & \qquad\quad \to \beta.\end{aligned}$$

As before, we can define additional operations naturally in terms of these primitives. For example, record update can be defined as

$$\begin{array}{lcl} (\_ := \_ \,|\, \_) & : & \forall\alpha.\forall\beta.\forall r.\forall l.(r\backslash l) \Rightarrow \mathit{Label}\ l \to \alpha \to \mathit{Rec}\ \{\!|l : \beta \,|\, r|\!\} \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Rec}\ \{\!|l : \alpha \,|\, r|\!\} \\ (l := x \,|\, r) & = & (l = x \,|\, r - l), \end{array}$$

as expected. The important point here is that is a single definition for record update, unlike that of Chapter 3 where there was an implementation for each label l.

To our knowledge, this is the first work—in either implicitly or explicitly typed record and variant calculi—to consider a type system in which labels can be treated as first-class values. This increases the expressiveness of our system, and we already have some interesting applications for these new features, some of which are discussed in the sequel. We have implemented a variant of the functional programming language Haskell (in Java) that includes extensible records and variants with first-class labels. Each of the applications, of first-class labels, described in the sequel have been implemented in this prototype implementation and seem to work well in practice.

For the remainder of this section we consider two different applications of first class labels. The first example, described in Section 7.2.1 is concerned with describing types for operations of the Java virtual machine [LY96], and is based on work by Jones [Jon97]. The second example, described in Section 7.2.2, is concerned with the problem of array bounds checking for a simply typed λ-calculus extended with integer arrays. The key motivation for this example is that it provides evidence that the introduction of first class labels does not break type soundness, but it also highlights some of the limits of expressiveness of first-class labels.

7.2.1 Type checking Java byte code

The programming language Java [AG96] has been designed specifically with portability in mind. Java source code is compiled into a bytecode format that can be executed on a *Java virtual machine* (JVM) [LY96]. The designers of Java paid a lot of attention to security. For example, Lindholm and Yellin [LY96] describe a process of bytecode verification—a bytecode *verifier* identifies and rejects badly behaved programs. Although Lindholm and Yellin describe bytecode verification through the use of natural language, Jones [Jon97] argues that this process can be understood more formally as a kind of type inference problem.

The key idea in Jones' presentation is to view bytecode files as a special kind of concrete syntax for programs with the same dynamic semantics, but typed in a functional language. The importance of this work in the current setting is that the functional language is fairly conventional. However, Jones has found that, if one is to provide types for the complete set of Java bytecode operations, then both extensible records (used to model function frames) and first-class labels (capturing load frame to stack operations) can be useful in describing the correct types. In this section we give an overview of Jones' approach to bytecode verification, paying particular attention to the application of extensible records and first-class labels. We conclude this section describing a successor function for labels that allows us to assign a type to the operation of loading two-word value of type *Long*.

A Java bytecode program can be thought of as the execution of a sequence of commands. Each command has access to the machine's state, which can be represented by a set of local arguments (the command's frame) and a machine stack. Modelling the machine's state by a tuple containing the local frame and machine stack, Jones represents commands by functions that describe transitions from one machine state to another. For example, the addition of two integers stored on the top of the machines stack can be specified by the following definition[5]

$$\begin{array}{lcl} iadd & : & \forall f.\forall s.(f, Push\ Int\ (Push\ Int\ s)) \to (f, Push\ Int\ s) \\ iadd & = & \lambda(f, Push\ x\ (Push\ y\ s)).(f, Push\ (x+y)\ s). \end{array}$$

A sequence of commands is then specified by combining the different instructions together using forward function composition[6]. For example, the following expression adds three integers together

$$iadd; iadd,$$

and has type $(f, Push\ Int\ (Push\ Int\ (Push\ Int\ s))) \to (f, Push\ Int\ s)$.

The reader may have observed that the operator ; represents composition in the Kleisli category constructed with respect to the original category of computation and the identity monad [Lan72, Mog91, Wad90]. Furthermore, to capture the possible effects that a command may produce (e.g., exceptions) during execution, Jones transforms the identity monad to support the required features. This work on monads is treated in detail by Jones [Jon97].

[5]We assume the existence of a stack data type—defined in Haskell as $newtypePush\ a\ s = Push\ a\ s$.

[6]We denote forward function composition by $(\_;\_) : \forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$.

Hitherto we have not considered any Java bytecodes which make use of the local frame. One such operation is the load variable command, that simply extracts a value stored in the frame and pushes it onto the stack. This command is an excellent application for extensible records and first-class labels. It can be defined as

$$\begin{array}{lcl} load & : & \forall\alpha.\forall s.\forall f.\forall l.\ f\backslash l \Rightarrow \\ & & \qquad\qquad\qquad Label\ l \rightarrow \\ & & \qquad\qquad\qquad\ (Rec\ \{\!|l : \alpha\,|\,f|\!\}, s) \rightarrow \\ & & \qquad\qquad\qquad\ \ (Rec\ \{\!|l : \alpha\,|\,f|\!\}, Push\ \alpha\ s) \\ load\ l & = & \lambda(f, s).(f, Push\ (f.l)\ s). \end{array}$$

Here the *load* operator is now parametrized by a label, representive of the frame memory cell to be loaded, and is defined simply to extract the corresponding value from the frame and push on to the stack.

Finally, to conclude this section, we consider the command *lload*, which, due to the Java bytecode specification, has an unexpected type and implementation. The problem is that although *Long* is a defined type, values of this type cannot be stored in frames or on the stack. Instead, the low and high words of a corresponding long are stored in related variables. It is possible to calculate the name of the high byte by incrementing the name of the low byte. However, using first-class labels we can assign a name to the low byte, but how can a label be incremented? The answer to this question lies in our original definition for labels, specified in Chapter 3. The only requirement we placed on this set was that it must be countable and closed under a specified total-ordering. One such set that satisfies this specification is the set, $\mathbb{N}$, of natural numbers. Choosing the set $\mathbb{N}$ for our set of labels implies that the set of labels is closed under Peano's axioms and, as such, supports a successor function. To capture the notion of incrementing a label we introduce the following type constructor

$$Succ : lab \rightarrow lab,$$

which can be seen as the successor function satisfying Peano's second axiom (see MacLane and Birkhoff [LB93], for example). Furthermore, we add an additional term constant used to increment a given label

$$succ : Label\ l \rightarrow Label\ (Succ\ l).$$

Finally, using the successor operation on labels we can define *lload* as

$$
\begin{array}{lcl}
lload & : & \forall s.\forall r.\forall l.\ (r\backslash l, r\backslash(Succ\ l)) \Rightarrow \\
 & & \qquad\qquad\quad Label\ l \rightarrow \\
 & & \qquad\qquad\qquad (Rec\ \{\!|l : Word, (Succ\ l) : Word \,|\, r|\!\}, s) \rightarrow \\
 & & \qquad\qquad\qquad\quad (Rec\ \{\!|l : Word, (Succ\ l) : Word \,|\, r|\!\}, \\
 & & \qquad\qquad\qquad\quad Push\ Word\ (Push\ Word\ s)) \\
lload & = & \lambda l.\lambda(f, s).(f, Push\ (f.l)\ (Push\ (f.(succ\ l))\ s)).
\end{array}
$$

Notice the use of the label successor function to gain correct access to the high word.

Extending the record and variant type system of Chapter 3 to include first-class labels supporting successor operation introduces no further technical problems. For example, unification and type inference are unchanged, while an advanced compilation process may remove references to first class labels, if it so chooses.

7.2.2 Simple array bounds checking

Static array bounds checking is concerned with determining, at compile time, whether a expression tries to select or update an array component outside the bounds of the array. Thus, as is the case with static type checking, this process can reduce the need for passing and checking bounds information at runtime. For example, the expression $arr[9] = 20$ is safe on the assumption that the array *arr* contains at least ten components[7], otherwise evaluation may, depending on the language definition, cause an unexpected exception. In this section, we study a simple calculus supporting functional arrays and we describe how a translation procedure can be specified from expressions of this language into our record calculus with first class labels. The soundness of this translation provides us with a simple guarantee that any, well typed, translated expression will not 'go wrong', thus, guaranteeing that no outer of bounds error will occur. However, as will become clear in the sequel it is not the aim of this section to provide a solution to the problem of array bounds checking, but rather to highlight some of the shortcomings of first class labels.

The types for the array language are those of the simply typed λ-calculus extended with types for functional arrays, given by the following grammar

[7]We assume that the first element of an array is indexed at offset 0.

$$\nu \quad ::= \quad \tau \to \tau \mid Ix \mid Array_n \mid Elem$$

Here Ix corresponds to the type of natural numbers, whose elements, $n \in Ix$, are used to access components of the array. $Elem$ is the type of values that can be stored in an array. Finally, the type $Array_n$ represents arrays of size $n \in \mathbb{N}$, whose elements are of type $Elem$.

The term language, $Array$, is an implicitly typed λ-calculus extended with the following constants

- Array definition: to define an array of length n, whose components are set to some default value of type $Elem$

$$new_n : Elem \to Array_n.$$

- Index constants

$$n : Ix.$$

- Index addition: to add two array indices

$$+ : Ix \to Ix \to Ix.$$

- Access: to select a given component from an array

$$\_[\_] : Array_n \to Ix \to Elem.$$

- Update: to update a given array component

$$\_[\_] = \_ : Array_n \to Ix \to Elem \to Array_n.$$

We use the symbol M to range over expressions of type $Array$.

To enable the following translation procedure to capture the notion of index addition, we must extend our term operations over labels to support label addition. The following the addition constant is added to the term language

$$\$+ : Label\ x \to Label\ y \to Label\ (x + y),$$

observing the need for a new type constructor

$$+ : lab \to lab \to lab.$$

Due to the static nature of labels, it may not be necessary to extend the unification procedure to support addition over labels. However, we leave further investigation of this subject to future work, noting simply that, if an extended unification procedure is required, then the work of Kennedy [Ken96] on type inference in the presence of dimension types may be a good place to begin.

We now define a translation from expressions of *Array* into expressions of our record and variant calculus extended with first class labels[8]

$$\begin{array}{lcl}
[\![i]\!] & = & \$i \\
[\![x]\!] & = & x \\
[\![\lambda x.M]\!] & = & \lambda x.[\![M]\!] \\
[\![M\ M']\!] & = & [\![M]\!][\![M']\!] \\
[\![e]\!] & = & () \\
[\![new_n]\!] & = & (\$0 = (), \ldots, \$(n-1) = ()) \\
[\![M[i]]\!] & = & [\![M]\!].[\![i]\!] \\
[\![M[i] = M']\!] & = & ([\![i]\!] := [\![M']\!] \,|\, [\![M]\!]) \\
[\![+]\!] & = & \$+ .
\end{array}$$

The following propostion shows that any well-typed array program that has a well-typed translation into *OML* plus records and first-class labels, supporting addition, does not produce an array outer bound error.

Proposition 7.2 (Array indexing) *If* $\Gamma \vdash^{Array} M : \nu$ *and* $P \,|\, \Gamma' \vdash [\![M]\!] : \tau$, *then there exists no subexpression indexing an array, such that, the index is greater than the size of the array itself under the standard interpretation of* λ*-calculus extended with functional arrays.*

Proof: It is clear that by application of Corollary 6.2 that evaluation of the term $P \,|\, \Gamma' \vdash [\![M]\!] : \tau$ will not select a record component that is not present. Thus the required result follows by observation that array selection, update, and indexes all map to the appropriate record and label operations.

(*This completes the proof.* $\Box$)

[8]The expression $\$n$ denotes a value (i.e., label) of type *Label* n, where $n \in \mathbb{N}$.

Although this proposition provides us with a method for automatically checking, at compile time, that a given program will not try to access an array component outside of the array's bounds, this algorithm is not complete. A consequence of this observation is that it is possible find reasonable programs, that are rejected.

Unfortunately, many interesting, well-typed, programs are rejected under the above translation. For example, consider the expression[9]

$$\begin{aligned}\mathit{fix}\ (\lambda f.\lambda arr.\lambda i.\mathbf{if}\ i == 3\ &\mathbf{then}\ arr\\ &\mathbf{else}\ f\ (arr[i] = e)\ (i+1))\ 1\ (new_{10}),\end{aligned}$$

which simply updates the first two elements of a ten element array and returns this as its result. Using the reduction rules for λ-calculus and functional arrays one can quickly reduce this expression to normal form, without a runtime out of bounds error. However, applying the above translation to this expression produces an output expression that is not typeable. It seems that any array expression depending on recursion, will not be well-typed under the resulting translation. Thus the number of 'interesting' examples for which Proposition 7.2 applies seems to be rather small. However, it is this limitation that leaves us observing the addition of first class labels supporting arithmetic is sound. The point here is that, if it was possible to capture iteration through first-class labels, then we would be able to construct an expression that simply discarded a given field from a record and then recursively called itself with the label incremented by one. Thus at some point the newly generated label would represent a 'not present' field, leading to a failure in reduction.

7.3 Casting

Although records have been studied in the context of programming languages from the early days of Cobol, polymorphism and extensibility are more recent developments. Inspired by the need for type systems for Smalltalk like languages, Cardelli [Car84] and Wand [Wan87] proposed encoding objects in terms of record calculi with extensibility. In this section we study a selection of operations on records found in many strongly typed object-oriented programming languages, including Java and C++.

[9] To simplify the example we assume that the language *Array* is extended with a functional fixpoint operator and with conditionals.

To motivate this operation we begin by laying out the implementation of a simple hierarchy, pictured in Figure 7.6, by example. We use records to represent objects. For example, the type definition

$$\textbf{type}\ \mathit{Academics}\ r \quad = \quad \mathit{Rec}\ \{\!|\mathit{name} : \mathit{String}, \mathit{sex} : \mathit{String} \,|\, r|\!\},$$

captures the class *Academics* in Figure 7.6,

Observe that the type *Academics* is parametrised by a row variable r, representing the possible, yet unspecified, additional methods that a derived class may contain. An example academic 'Bart Simpson' can be defined using extension as follows

$$\begin{array}{l}(\ \mathit{name} = \text{“}\texttt{Bart Simpson}\text{”},\\ \ \ \mathit{sex} = \text{“}\texttt{Male}\text{”}) : \mathit{Rec}\ \{\!|\mathit{name} : \mathit{String}, \mathit{sex} : \mathit{String}|\!\}.\end{array}$$

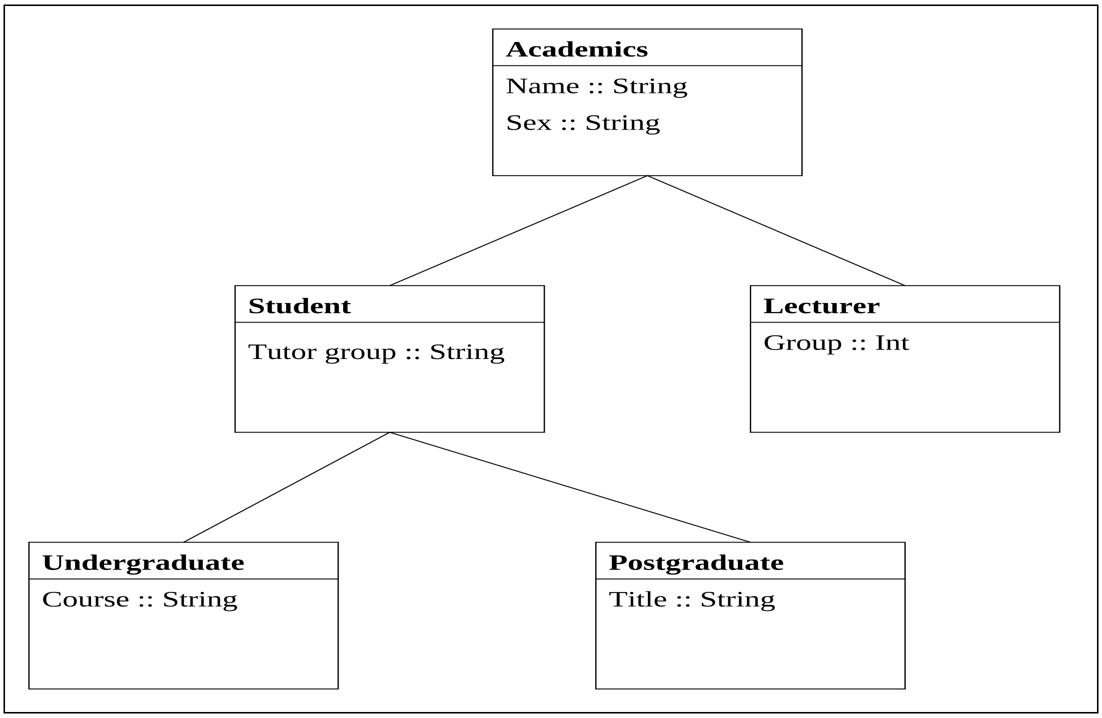


Figure 7.6: Academic hierarchy.

The following type definition captures the *Student* class of Figure 7.6, which must express that an object *Student* will contain all the fields of an *Academic* plus some additional ones of its own.

$$\textbf{type}\ \mathit{Student}\ r \quad = \quad \mathit{Academic}\ \{\!|\mathit{tutorGroup} : \mathit{String} \,|\, r|\!\}.$$

Capturing definitions such as

$$\begin{aligned}(\ & name = \text{“}\texttt{Bart Simpson}\text{”}, sex = \text{“}\texttt{Male}\text{”},\\ & tutorGroup = \text{“}\texttt{Math-A-1}\text{”}) : Student\ r.\end{aligned}$$

As was the case with the polymorphic functions of Chapter 3, functions defined over *Academic* objects work equally well over *Student* objects. For example, a simple function to print an information header for any given academic, might be defined as

$$\begin{array}{lcl} header & : & \forall r.(r \setminus name,\ r \setminus sex) \Rightarrow Academics\ r \rightarrow String \\ header\ obj & = & \text{“}\texttt{Name: }\text{”} +\!\!+\ obj.name +\!\!+ \text{“}\texttt{\\n}\text{”} +\!\!+ \\ & & \text{“}\texttt{Sex : }\text{”} +\!\!+\ obj.sex. \end{array}$$

Thus the expressions

$$header\ (name = \text{“}\texttt{Bart Simpson}\text{”}, sex = \text{“}\texttt{Male}\text{”})$$

and

$$header\ (name = \text{“}\texttt{Bart Simpson}\text{”}, sex = \text{“}\texttt{Male}\text{”}, tutorGroup = \text{“}\texttt{Math-A-1}\text{”})$$

both reduce to the string

$$\begin{array}{l} Name : Bart\ Simpson \\ Sex : Male. \end{array}$$

This kind of polymorphism, often referred to as inheritance subtyping in the object-oriented literature [Bud91], is just that introduced by the row extension operators of Chapter 3 and fits naturally with the development of those ideas. However, another important notion, used extensively in the object-oriented paradigm, is the idea of casting or inheritance supertyping, which can be seen as the dual to inheritance subtyping and is not captured by any of the operators presented so far.

Casting provides the ability to widen the number of known fields for a given object, with respect to a known hierarchy. For example, above we considered ‘Bart Simpson’ to be both an academic and a student, thus, it is reasonable that an object representing *Bart* may first be considered simply as an academic and later, if required, be considered as a student. More generally, some object, A, extended by some other object, B, can be considered in place of an object B if at some point it was in fact an object B. For example, students can be considered as both academics and students, but a lecturer is not a student. An important difference here between inheritance subtyping and inheritance supertyping is that the latter does not, in general, preserve totality. To see this consider the following extended header function for students

$$\begin{array}{lcl}
\mathit{bigHeader} & : & \forall r.\ (r \setminus \mathit{name},\ r \setminus \mathit{sex},\ r \setminus \mathit{tutorGroup}) \Rightarrow \\
 & & \qquad\qquad\qquad\qquad \mathit{Student}\ r \to \mathit{String} \\
\mathit{bigHeader}\ \mathit{obj} & = & \text{“}\texttt{Tutor group: }\text{”} +\!\!+\ \mathit{obj} \,.\, \mathit{tutorGroup} +\!\!+ \text{“}\texttt{\textbackslash n}\text{”} +\!\!+ \\
 & & \text{“}\texttt{Name}\qquad\ \ \texttt{: }\text{”} +\!\!+\ \mathit{obj} \,.\, \mathit{name} +\!\!+ \text{“}\texttt{\textbackslash n}\text{”} +\!\!+ \\
 & & \text{“}\texttt{Sex}\qquad\quad\ \texttt{: }\text{”} +\!\!+\ \mathit{obj} \,.\, \mathit{sex}.
\end{array}$$

which produces an error when applied to the object

$$(\mathit{name} = \text{“}\texttt{Bart Simpson}\text{”}, \mathit{sex} = \text{“}\texttt{Male}\text{”}).$$

For the rest of this section, we consider a number of possible operations for type casting with respect to the record and variant system of Chapter 3.

7.3.1 The ? predicate

Before considering any specific casting operations this section revisits row predicates and the implementation required to capture the notion of possible field membership. The point here is that casting asks the question whether a given label is absent or present. The lacks predicate of Chapter 3 captures the assertion that a given row does not contain a certain label and provides a positional offset for the insertion of the corresponding label. Consider a new predicate $r?l$, which, represents the assertion that the label l may be present in the row r. Evidence for this new predicate is represented by a boolean and integer tuple, where evidence (True, i) asserts that the field l is present at offset i, while (False, i) asserts that the field l is absent but would be inserted at offset i.

The calculation of evidence for predicates of the form $r?l$ is described by the rules in Figure 7.7. The second rule is the most interesting and tells us how to find the position at which a label l can be found in a row $\{\!|l : \tau \mid r|\!\}$ by calculating the position at which it can be inserted into the row r. This calculation is made by application of the rules in Figure 3.6 for lacks predicates and provides the assertion that the label l does not appear more than once in the row $\{\!|l : \tau \mid r|\!\}$, which preserves the early invariant that a given label cannot be duplicated. A

$$P \cup \{v : \pi\} \Vdash v : \pi$$

$$\frac{P \Vdash i : (r\backslash l)}{P \Vdash (\mathit{True}, i) : (\{\!|l : \tau \,|\, r|\!\}?l)}$$

$$\frac{P \Vdash (b, i) : (r?l)}{P \Vdash (b, m) : (\{\!|l' : \tau \,|\, r|\!\}?l)} \qquad m = \begin{cases} i, & l < l' \\ i + 1, & l' < l \end{cases}$$

$$P \Vdash (\mathit{False}, 0) : (\{\!||\!\}?l)$$

Figure 7.7: Predicate entailment for ? predicate.

consequence of this invariant seems to be that, although it is possible to describe an unchecked casting operation, it does not seem possible to capture the unchecked merge operation of Wand [Wan91] in our current setting.

7.3.2 Casting operators

Casting is closely related to variant elimination: It is possible that a variant of type $\mathit{Var}\ \{\!|l : \tau \mid r|\!\}$ may be in the summand labelled l, otherwise it must be in $\mathit{Var}\ r$. Motivated by this observation, the following possible casting operation is similar in spirit to variant elimination, but constrains the row variable r using the predicate $r?l$, instead of the lacks predicate used for variant operations:

$$\begin{aligned} \mathit{cast}_l : \forall r.\forall\alpha.\forall\beta.r?(l : \alpha) \Rightarrow\ & (\mathit{Rec}\{\!|l : \alpha \,|\, r - l|\!\} \to \beta) \to \\ & (\mathit{Rec}\ (r - l) \to \beta) \to \\ & \mathit{Rec}\ r \to \beta. \end{aligned}$$

However, there is a problem with this operation, it leads to the loss of most general unifiers. In fact it is the same problem that we highlighted for Jones' [Jon94b] system of extensible records and is introduced through the use of the row restriction operator, $\_ - l$. The problem is how can we find a most general unifier, U, such that

$$(r - l) \overset{U}{\sim}_{row} (r' - l)?$$

The possible values for the row variable r arise naturally from two different perspectives. To see this consider the following row expression

$$\{\!|x : \alpha, y : \beta|\!\},$$

which can be bound to r and as such, the expression $\{\!|x : \alpha, y : \beta|\!\} - l$ is equal to $\{\!|x : \alpha, y : \beta|\!\}$. However, if we extend our original row with a new field l, of any type and bind this to r, we also have that $\{\!|x : \alpha, y : \beta, l : \gamma|\!\} - l$ is equal to $\{\!|x : \alpha, y : \beta|\!\}$. A similar argument can be applied to the row variable r' leading to the observation that there are, in general, a number of possible solutions to the equation $(r - l) \overset{U}{\sim}_{row} (r' - l)$, all of which are in some sense most general.

A similar, but simpler casting operator, providing just the functionality required to express the header functions discussed earlier, while preserving principal types and thus effective type inference, can be captured through the *Maybe* monad [Mog91]

$$(\_?l) : \forall r.\forall \alpha.r?(l : \alpha) \Rightarrow \mathit{Rec}\ r \rightarrow \mathit{Maybe}\ \alpha.$$

Here $r?l$ tests the record r for the existence of a field l, returning the value stored at l if present (in the *Just* projection), and returning *Nothing* if the field l is found not to exist.

Finally, to conclude this section, we return to the hierarchy of Figure 7.6. The function *bigHeader* can now be rewritten to check for the existence of a *tutorGroup* field and to display an appropriate message if found. The remaining functionality is expressed by calling the original header function, defined for all academics

$$\begin{array}{lcl}
\mathit{bigHeader} & : & \forall r.\ (r \setminus \mathit{name},\ r \setminus \mathit{sex},\ r?\mathit{tutorGroup}) \Rightarrow \\
 & & \qquad\qquad\qquad \mathit{Academic}\ r \rightarrow \mathit{String} \\
\mathit{bigHeader}\ \mathit{obj} & = & \textbf{case}\ \mathit{obj}?\mathit{tutorGroup}\ \textbf{of} \\
 & & \qquad \begin{array}{lcl} \mathit{Just}\ d & \rightarrow & \text{“}\texttt{Tutor group: }\text{”} +\!\!+\ d\ +\!\!+ \\ & & \text{“}\texttt{\char`\\n}\text{”} +\!\!+\ \mathit{header}\ \mathit{obj} \\ _ & \rightarrow & \mathit{header}\ \mathit{obj}. \end{array}
\end{array}$$

Evaluating the expression

$$bigHeader\ (name = \text{“}\texttt{Bart Simpson}\text{”}, sex = \text{“}\texttt{Male}\text{”})$$

results in the string

$$\begin{array}{l} Name : Bart\ Simpson \\ Sex : Male, \end{array}$$

as expected.

Chapter 8

Extensible records for Haskell

Previous chapters have described, among over things, a system of extensible records and variants supporting a number of different features. However, the basic type system and its extensions were described in the setting of a simply typed λ-calculus and not within a practical functional programming language. In this chapter we consider the paragmatics of extending Haskell to support extensible records and variants.

The sections of this chapter are as follows. Section 8.1 gives an overview of extensible records in Haskell. Section 8.2 provides an informal overview of our proposal for extensible records in Haskell. A number of basic record operations are considered, which are natural generalizations of operators that are already present in Haskell. Although there are no real surprises in this section, it does provide us with the chance to describe further applications of extensible records. Section 8.3 comments briefly on the integration of lacks predicates and record primitives with the implicit dictionary implementation of Haskell type classes. Section 8.4 considers a number of pragmatic issues concerning the integration of extensible records into Haskell. In particular, any serious proposal extending Haskell with new primitive datatypes must consider the general framework of deriving instances for standard classes (e.g., equality), and must address questions of syntax, and pattern matching.

An earlier version of this chapter has previously been published at the ACM Haskell Workshop 1997 [Gas97a].

8.1 Overview

In functional languages like Haskell [PH97] and SML[1] [MTH90, MTH97], products provide support for defining datatypes, allowing a selection of data items to be grouped together. For example, a datatype representing a point in the plane, might be represented by the following Haskell definition

$$\textbf{data}\ \mathit{Point} \quad = \quad \mathit{MkPoint}\ \mathit{Int}\ \mathit{Int}.$$

As discussed in Chapter 2, this definition is not particularly easy to work with in practice. For example, it is easy to confuse fields when they are accessed by position within a product.

To avoid these problems, the programming languages Haskell and SML allow components of products to be identified using names drawn from some set of *labels*. Haskell 1.4 provides support for labelled products by allowing a datatype declaration to include field labels for components of the datatype. For example, the *Point* type described above might be defined more attractively as

$$\textbf{data}\ \mathit{Point} \quad = \quad \mathit{MkPoint}\ \{x :: \mathit{Int},\ y :: \mathit{Int}\}.$$

SML supports a more general notion of record type, which considers labelled products as part of the core type language. In this setting, our *Point* example can be reformulated as[2]

$$\textbf{type}\ \mathit{Point} \quad = \quad \mathit{Rec}\ \{x :: \mathit{Int},\ y :: \mathit{Int}\}.$$

Although SML does not require the programmer to define a type synonym for *Point*, in practice, this does provide a useful way of documenting one's intentions.

Both Haskell and SML provide mechanisms allowing field names to be used in the construction and selection of record components, without concern for the overall structure of the datatype. For example, Haskell ensures that, for each new label, a function working as a selector for that component is introduced at the top-level. Unfortunately, this has the undesirable side effect of forcing any two datatypes defined within the same scope to use mutually exclusive field names. For example,

[1]This chapter uses references to SML to help support the point that much of our work is language independent.

[2]To emphasize the notion of record types, we choose to incorporate a record constructor, *Rec*, where the actual SML definition is **type** $\mathit{Point} = \{x : \mathit{Int},\ y : \mathit{Int}\}$.

a datatype of circles including components to specify its centre point and radius, might be defined as

$$\textbf{data}\ \mathit{Circle} \quad = \quad \mathit{MkCircle}\ \{x :: \mathit{Int},\ y :: \mathit{Int},\ r :: \mathit{Int}\}.$$

However, this definition is not valid if it appears in the same scope as the *Point* shape described above, because it contains fields overlapping with those of a point. Moreover, datatypes defined in separate modules sharing common field names may only be used in the same namespace with careful use of qualified names. SML avoids imposing similar restrictions on record fields by requiring that the type of a record r is uniquely determined at compile-time. In effect, each different record type that includes an l field comes with its own method for extracting the value of that field. By requiring that the record type can be determined during type checking, the overloading that results from using the same notation for each of these operations is easily resolved.

An unfortunate consequence of the restrictions imposed on record types by both the Haskell and SML type systems is that operations provided for manipulating records are less flexible than those described in Chapters 3 and 7. For example, consider an operation to extract the centre point of a given shape. We might reasonably expect that polymorphism would provide the ability to define a single definition for all shapes

$$\mathit{centre}\ \mathit{shape} \quad = \quad (\mathit{shape}\,.\,x,\ \mathit{shape}\,.\,y).$$

However, although both Haskell and SML provide support for polymorphic definitions, no support is provided for row polymorphism, which is required to capture the notion that a record may contain more than just the fields x and y. It is the requirement that record types be completely determined at compile-time—enforced by the application of constructors in Haskell, and by user specified type annotations in SML—that limit operations over records to monomorphic type. A further weakness of the Haskell and SML record systems is the lack of support for *extensibility*; there are no general operators for adding and removing fields in a record value, for example. The following definition, which is not legal in either Haskell or SML, shows how extensibility might be applied to allow an additional colour field to be incorporated into arbitrary shape values

$$\mathit{colour}\ c\ \mathit{shape} \quad = \quad (\mathit{colour} = c \,|\, \mathit{shape}).$$

This chapter presents an alternative proposal for records in Haskell[3], by combining ideas of Chapters 3 and 7 to develop a practical type system. In particular, it supports extensible records, with a full complement of polymorphic operations. For example, the point and circle shapes described above can be reformulated as

$$\begin{array}{lcl}\textbf{type}\ \mathit{Shape}\ r & = & \mathit{Rec}\ \{\!|x :: \mathit{Int},\ y :: \mathit{Int}\,|\,r|\!\} \\ \textbf{type}\ \mathit{Circle} & = & \mathit{Shape}\ \{\!|\mathit{rad} :: \mathit{Int}|\!\}.\end{array}$$

Here, the row extension operator of Chapter 3 provides us with the ability to define *Circle* as an extension of the type *Shape*, which includes at least the fields x and y; i.e., a value of type *Shape* is at least a value of type *Point* introduced above.

Of course, the combination of extensibility, and the ability to define polymorphic operations over records provides us with the functionality to define, and type correctly, the definition of *centre* described above

$$\begin{array}{lcl}\mathit{centre} & :: & (r\backslash x, r\backslash y) \Rightarrow \mathit{Rec}\ \{\!|x :: \mathit{Int},\ y :: \mathit{Int}\ \ |\ r|\!\} \to (\mathit{Int},\ \mathit{Int}) \\ \mathit{centre\ shape} & = & (\mathit{shape}\,.\,x,\ \mathit{shape}\,.\,y).\end{array}$$

This chapter provides an informal presentation of extensible records for Haskell and refrains from a more in depth discussion of related record calculi. In particular, we do not consider proposals for extending SML with similar record operations, as in the work of Rémy [Rém92b, Rém94a] and Ohori [Oho95]. Previous chapters of this dissertation have discussed these and other proposals in detail.

8.2 Basic record operations

In this section we review some of the ideas introduced in Chapter 3 within the context of Haskell. In particular, we show, by example, how record extensibility and polymorphism can be used to avoid the limitations of Haskell style records, discussed in the introduction to this chapter. An important aim of this section, and of this chapter, is to provide some more concrete examples of extensible records in practice. Note that, although we may be somewhat flexible with our choice of

[3]The record system discussed in this chapter would be equally suitable for an extension of SML. However, as we have seen in previous chapters our type system is based on the notion of qualified types, which is the core type system of Haskell, and as such, Haskell seems an obvious choice.

syntax, every example has been tested and experimented with in Haskell implementations. In particular, we have implemented a prototype implementation of extensible records, called TREX (Typed Row Extensions), in Hugs 1.4[4] and also a prototype Java implementation of Mondrian, a Haskell variant [Mei97]. This latter implementation includes the complete set of record and variant operations introduced in Chapter 3, and also supports the first-class labels of Chapter 7.

As a concrete example of the proposed operations for Haskell, and again highlighting the usefulness of extensibility, consider a hierarchy of algebraic structures. Monoids (structures with a set and a binary operation that is associative and has a unit) form the base of the hierarchy, and group and ring structures are defined as extensions of monoids and groups, respectively. A group supports all the operations of a monoid plus an inverse, and a ring supports all operations of a group plus some of its own. Given an appropriate implementation of this hierarchy, a user might reasonably expect to define operations, requiring only the functionality of monoids, over all algebraic structures. Figure 8.1 provides an implementation of this hierarchy in terms of extensible records, accompanied by sample implementations for the ring of integers with the standard operations. The parameter r of the types *Monoid*, *Group*, and *Ring* ranges over rows, capturing the notion that a group, for example, may support some additional operations.

The standard list function *sum*, for computing the sum of a list, can now be recast in terms of any monoid

$$\begin{array}{lcl} \mathit{sum} & :: & (r\backslash \mathit{plus}, r\backslash \mathit{id}) \Rightarrow \mathit{Monoid}\ \alpha\ r \rightarrow [\alpha] \rightarrow \alpha \\ \mathit{sum}\ \mathit{mon} & = & \mathit{foldr}\ (\mathit{mon.plus})\ (\mathit{mon.id}) \end{array}$$

Here, r ranges over rows containing zero or more fields, which in the case when the function *sum* is applied to *iGroup*, r is bound to the single field *negate*. Thus extensibility captures a form of sub-typing that is also present, although in a slightly different form, in the Haskell class system. However, we believe that this notion of sub-typing is present in a number of different programming situations, many of which are more suited to extensibility than to obscure encodings using the class mechanism.

As discussed briefly in Chapter 7, extensibility provides a simple form of inheritance, more commonly found in object-oriented languages [Wan87, Car84, Bud91]. Hughes and Sparud [HS95] have shown that the Haskell class system provides an

[4]Hugs is an implementation of Haskell 1.4, developed jointly by the Languages and Programming Group at the University of Nottingham and the Haskell Group at Yale University.

$$
\begin{array}{lcl}
\textbf{type}\ Monoid\ v\ r & = & Rec\ \{\!|\ plus :: v \to v \to v, \\
 & & \qquad\quad id :: v \mid r |\!\} \\[1ex]
\textbf{type}\ Group\ v\ r & = & Monoid\ v\ \{\!| inv :: v \to v \,|\, r |\!\} \\[1ex]
\textbf{type}\ Ring\ v\ r & = & Group\ v\ \{\!|\ mult :: v \to v \to v, \\
 & & \qquad\qquad one :: v \mid r |\!\} \\[1ex]
iMonoid & :: & Monoid\ Integer\ \{\!||\!\} \\
iMonoid & = & (plus = (+), id = 0) \\[2ex]
iGroup & :: & Group\ Integer\ \{\!||\!\} \\
iGroup & = & (inv = negate \,|\, iMonoid) \\[2ex]
iRing & :: & Ring\ Integer\ \{\!||\!\} \\
iRing & = & (mult = (*), one = 1 \,|\, iGroup)
\end{array}
$$

Figure 8.1: Example algebraic hierarchy.

alternative form of inheritance, which can be utilized to encode object-oriented features. It remains to be seen whether records with extensibility will provide a practical platform for incorporating object-oriented features into Haskell.

8.3 Compilation issues

One of Haskell's most interesting design decissions was the inclusion of type classes, which is implemented in almost all Haskell implementations, through implicit dictionary parameters, similar to the implementation of lacks predicates described in Chapter 3. Early versions of the Haskell type system and the dictionary implementation was based on the work of Wadler and Blot [WB89] and Kaes [SK88]. However, since Haskell 1.3, the corresponding type system was generalized to an application of qualified types and constructor classes [Jon94b, Jon95c].

As discussed throughout this dissertation, our proposed type system for extensible records and variants, and its extensions, is based on a system of qualified types.

Furthermore, we showed in Chapter 6 that the soundness of our type system could be justified by application of a theorem of soundness for qualified types. As a consequence, the implementation for extensible records described in Section 3.3 integrates naturally with the dictionary passing implementation of Haskell 1.4. In fact, assuming the implementation of predicate entailment is extended to calculate offsets for lacks predicates, storing them in appropriate dictionary structures for lacks predicates, an implementation then simply supplies the primitive record operations, which are analogous to dictionary selector functions for type classes[5].

8.4 Pragmatic issues

In previous sections, we have highlighted a number of shortcomings with the current solution for records in Haskell, and we have proposed a system of polymorphic extensible records, naturally extending the Haskell type system. However, hitherto, we have avoided the more pragmatic issues, which often arise with proposed extensions to non-trivial languages, such as Haskell. In this section, we consider three particularly important practical concerns for extensible records in Haskell. Section 8.4.1 considers the question of pattern matching over records, while Section 8.4.2 highlights the difficulty in selecting a suitable syntax. Finally, Section 8.4.3 presents extensions of the Haskell class mechanism to allow for derived instances of equality and text operations over records.

8.4.1 Pattern matching

As with other datatypes in Haskell, pattern matching is often a natural way to extract the components of a record. For example, considering again the algebraic hierarchy of Section 8.2, pattern matching provides an alternative definition for the function *sum*

$$\begin{array}{lcl} sum & :: & (r\backslash plus, r\backslash id) \Rightarrow Monoid\ \alpha\ r \rightarrow [\alpha] \rightarrow \alpha \\ sum\ (plus = p, id = i \mid r) & = & foldr\ p\ i. \end{array}$$

Intuitively, an expression of the form $sum\ e$ is evaluated from left to right, first evaluating the pattern bindings for p and i, and then binding all other components

[5]As a consequence of soundness a compiler may pass offsets around unboxed, adjusting the primitive record operations as needed.

to the pattern r. More generally, we can explain the semantics of pattern matching over records using a translation of the form

$$\begin{array}{lcl} \backslash(l = p \mid r) \to e & = & \backslash x \to \textbf{case } x\,.\,l \textbf{ of} \\ & & \qquad\qquad p \;\;\to\;\; \textbf{case } x - l \textbf{ of} \\ & & \qquad\qquad\qquad\qquad r \;\;\to\;\; e. \end{array}$$

8.4.2 Issues of Syntax

Unquestionably, choosing an appropriate syntax plays an important role in the success or failure of programming language design—it has a direct effect on the way that programmers use and express themselves within the language. For example, Haskell allows pattern matching on the left hand side of function bindings, which in turn provides a convenient mechanism for describing inductive definitions. If, however, pattern matching was supported only within the case construct, then inductive definitions may not seem as attractive.

Chapter 3 and Section 8.2 introduced a syntax for record types and operations, overlaping with the syntax of Haskell. For example, record selection was written using the symbol (.), which is already used for function composition in Haskell. The fact that we use the symbols $\{\!|$ and $|\!\}$, which are not defined by Haskell's lexical syntax, complicates matters even further.

Any discussion of syntax for extensible records in Haskell must consider whether records, as described by the Haskell 1.4 report, are to be retained. If not, the syntax for record types can be simply that of Section 2.1, replacing the symbols $\{\!|$ and $|\!\}$ with $\{$ and $\}$, respectively. As the system proposed in this chapter includes the record operations of Haskell, we believe that it is not unreasonable to consider replacing one by the other.

We now turn our attention to the question of syntax for record values. Our main concern is that the syntax of Chapter 3 and Section 8.2 introduces ambiguities when considered with respect to Haskell's syntax—for example, when parsing the symbol (.) should it be interpreted as function composition or record selection? Unfortunately, although record extension integrates smoothly, this is not the case for either record selection or restriction. In practice we have found only a few applications for record restriction, and in such cases pattern matching over records has proven adequate. In contrast, record selection appears in all but the most trivial of record applications, and although pattern matching provides an alternative, we believe that this operation must be supported using a convenient notation.

Record selection operators in SML [MTH90, MTH97] are written by prefixing the appropriate label with the symbol #. Thus selection of the field l of a record r is denoted by the expression $\#l\ r$. However, having experimented with this notation, we found that important program details were often hard to visualize. Furthermore, $(\_.l)$ is used for field selection in a wide variety of programming languages, including C, C++, and Java. With this in mind, we strongly believe that $(\_.l)$ is the correct notation for record selection. This leaves us with the important question of what to do about function composition, which is denoted by the symbol (.) in current versions of Haskell. In fact, we propose that function composition be represented by the symbol #, even though we felt it to be inappropriate for record selection. Moreover, because function composition is, in fact, an instance of the Haskell class *Functor*, at type $((\rightarrow)\alpha)$, it could be predefined as part of the *Functor* class

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\ \quad (\#)\ \ ::\ \ (\alpha \rightarrow \beta) \rightarrow (f\ \alpha \rightarrow f\ \beta) \end{array}$$

Function composition is then defined simply as an instance of this class

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ \alpha)\ \textbf{where} \\ \quad (f\#g)\ x\ \ =\ \ f\ (g\ x). \end{array}$$

Fortunately, associativity for function composition is preserved. To see this, recall that the following equation is satisfied by any functor [Lan72]

$$(f\ \#\ g)\ \#\ h = f\ \#\ (g\ \#\ h),$$

for which f, g and h are of the appropriate types. Instantiating the different uses of (#) to function composition, of the appropriate types, gives the equality[6]

$$(f\ .\ g)\ .\ h = f\ .\ (g\ .\ h),$$

which is precisely the required associativity law.

Figure 8.2 contains our proposed extensions for the Haskell grammar, which appears in appendix B of the Haskell report [PH97].

With a eye to future, we might consider tuples of type $(\tau_1, \cdots, \tau_n)$ to be shorthand for records of type $\mathit{Rec}\ \{\!|1 : \tau_1, \cdots, n : \tau_n|\!\}$, where individual components are labelled by numbers. A similar relationship between tuples and records has been adopted by SML [MTH90, MTH97], and seems to provide a number of practical benefits, not least, a general mechanism for selecting arbitrary components of tuples.

[6]We return briefly to denoting function composition as (.), in order to help the discussion.

$$
\begin{array}{lcll}
\mathit{label} & ::= & \mathit{varid} & \textit{(field names)} \\
\\
\mathit{rowvar} & ::= & \mathit{tyvar} & \textit{(row variables)} \\
\\
\mathit{row} & ::= & \{\mathit{label}_1 :: \mathit{type}_1, \cdots, \mathit{label}_n :: \mathit{type}_n\} & (n \geq 0) \\
 & | & \{\mathit{label}_1 :: \mathit{type}_1, \cdots, \mathit{label}_n :: \mathit{type}_n \mid \mathit{row}\} & (n \geq 1) \\
 & | & \mathit{rowvar} & \textit{(row variables)} \\
\\
\mathit{atype} & ::= & \cdots & \\
 & | & \mathit{Rec\ row} & \textit{(record type)} \\
\\
\mathit{fexp} & ::= & [\mathit{fexp}]\,\mathit{dexp} & \textit{(function application)} \\
\\
\mathit{dexp} & ::= & \mathit{dexp.label} & \textit{(record selection)} \\
 & | & \mathit{aexp} & \\
\\
\mathit{aexp} & ::= & \cdots & \\
 & | & (\mathit{label}_1 = \mathit{exp}_1, \cdots, \mathit{label}_n = \mathit{exp}_n) & (n \geq 0) \\
 & | & (\mathit{label}_1 = \mathit{exp}_1, \cdots, \mathit{label}_n = \mathit{exp}_n \,|\, \mathit{exp}) & (n \geq 1) \\
\\
\mathit{apat} & ::= & \cdots & \\
 & | & (\mathit{label}_1 = \mathit{pat}_1, \cdots, \mathit{label}_n = \mathit{pat}_n) & (n \geq 0) \\
 & | & (\mathit{label}_1 = \mathit{pat}_1, \cdots, \mathit{label}_n = \mathit{pat}_n \,|\, \mathit{pat}) & (n \geq 1)
\end{array}
$$

Figure 8.2: Proposed syntax for extensible records in Haskell.

8.4.3 Records and the Haskell class system

Often, the design and implementation of a new datatype requires more than just specifying the datatype definition itself. For example, one must ask questions such as: is equality defined over elements of the new datatype? Are elements of this type printable? And so on. Although many of these questions will be related to specific applications, there is a class of operations that arise for almost all datatypes (e.g., equality).

To ease the programming burden, Haskell provides a number of predefined type classes for operations such as equality and printing, for which instances can be derived automatically by the compiler. This section considers how an implementation might automatically derive instances of the *Eq* and *Show* classes over records.

An obvious first attempt at defining equality over records might involve having the compiler generate instance declarations of the form

$$\begin{array}{l} \textbf{instance}\ (Eq\ (Rec\ r),\ Eq\ \alpha) \Rightarrow (Rec\ \{\!|l :: \alpha \,|\, r|\!\})\ \textbf{where} \\ \quad r == r' \quad = \quad (r.l == r'.l)\ \&\&\ (r - l == r' - l), \end{array}$$

for each record extension with a field l. However, this does not give a well-defined notion of equality. To see this, consider the expression

$$(x = 10, y = \bot) \quad == \quad (x = 20, y = 30),$$

where $\bot$ is a diverging term of type *Int*. Evaluating this expression from left to right results in the boolean value *False*. However, we consider rows equal modulo reordering of fields, thus applying commutativity and evaluating from left to right gives $\bot$ for the above equality. So, if we are not careful when deriving equality over records, then it is possible that different implementations may produce differing results. The problem lies in the fact that rows, and thus records, are considered equal modulo reordering of fields.

The clue to resolving this problem lies in Section 3.3, where the well-formedness of record compilation was guaranteed by considering a total ordering on labels. Intuitively, equality over records is well-defined if corresponding pairs of fields are compared in an order determined by their labels in the record type that we are concerned with. Operationally, one can think of record equality as: given any two records of the same type, construct an ordered list of pairs, in which the first element of each pair is the string for a particular label and the second is a (delayed) boolean test of equality for the values associated with a given label. The following class definition captures this notion of equality over records

$$\begin{array}{l} \textbf{class}\ EqRecRow\ r\ \textbf{where} \\ \quad eqRecRow \quad :: \quad Rec\ r \to Rec\ r \to [(String,\ Bool)]. \end{array}$$

To ensure that the definition of equality over records is well-defined, an implementation must guarantee that instances of this class for row extension are only be generated internally.

The instance for the empty row can be predefined, in a suitable library, as

$$\begin{array}{l} \textbf{instance}\ EqRecRow\ \{\!|\,|\!\}\ \textbf{where} \\ \quad eqRecRow\ \_\ \_ \quad = \quad [\,] \end{array}$$

Now, providing that suitable implementations are constructed on each application of record extension, we can safely define a single, general, instance for equality over records

$$\begin{array}{l} \textbf{instance}\ \mathit{EqRecRow}\ r \Rightarrow \mathit{Eq}\ (\mathit{Rec}\ r)\ \textbf{where} \\ \quad x == y \ \ = \ \ \mathit{all}\ (\mathit{map}\ \mathit{snd}\ (\mathit{eqRecRow}\ x\ y)) \end{array}$$

Derivable instances may be defined similarly for the classes *Ord* and *Show*. For example, Figure 8.3 contains an implementation for showing record values[7]. As with the function *eqRecRow* described above, the function *showRecRow* generates an ordered list of pairs for a given record. The second component of each pair represents the showable value associated with a given label l.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{ShowRecRow}\ r \Rightarrow \mathit{Show}\ (\mathit{Rec}\ r)\ \textbf{where}} \\
\quad \mathit{showsPrec}\ d & = & \mathit{showFields} \# \mathit{showRecRow} \\
\\
\mathit{showFields} & :: & [(\mathit{String},\ \mathit{ShowS})] \to \mathit{ShowS} \\
\mathit{showFields}\ [\,] & = & \mathit{showString}\ \text{“}\mathtt{()}\text{”} \\
\mathit{showFields}\ \mathit{xs} & = & \mathit{showChar}\ '('\\
 & \# & \mathit{foldr1}\ \mathit{comma}\ (\mathit{map}\ \mathit{fld}\ \mathit{xs}) \\
 & \# & \mathit{showChar}\ ')' \\
\\
\mathit{comma}\ a\ b & = & a \# \mathit{showString}\ \text{“}\mathtt{,\ }\text{”} \# b \\
\mathit{fld}\ (s,\ v) & = & \mathit{showString}\ s \# \mathit{showChar}\ '=' \# v \\
\\
\multicolumn{3}{l}{\textbf{class}\ \mathit{ShowRecRow}\ r\ \textbf{where}} \\
\qquad \mathit{showRecRow} & :: & \mathit{Rec}\ r \to [(\mathit{String},\ \mathit{ShowS})] \\
\\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{ShowRecRow}\ \{\}\ \textbf{where}} \\
\quad \mathit{showRecRow}\ _ & = & [\,]
\end{array}$$

Figure 8.3: Functions to “show” record values.

[7]Following the discussion of Section 8.4.2 the composition of functions is represented by $(\#) :: (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$.

Chapter 9

Conclusion and future work

We have described a flexible, polymorphic type system for extensible records and variants with an effective type inference algorithm and compilation method. Prototype implementations have been written in several languages, and an implementation of records (TREX) has been added to Hugs, an implementation of Haskell 1.4. Our experience to date shows that these implementations work well in practice. To provide evidence for the correctness of these systems, we developed two alternative semantics for qualified types. These provide a foundation for soundness of our type system and for others based on the theory of qualified types—including the functional programming language Haskell.

There are many potential areas for further work, and the remaining sections of this chapter summarize some possibly interesting avenues of study.

9.1 Extensible data types

Functional programming languages, like Haskell and SML, are often claimed to allow a concise and modular programming style. For example, pattern matching allows function definitions to be specified by induction over the constructors of a data type. To see this, consider a data type, *List*, of integer cons lists and a function, *sum*, to calculate the sum of a list

$$
\begin{array}{lcl}
\mathit{data}\ \mathit{List} & = & \mathit{Null} \mid \mathit{Cons}\ \mathit{Int}\ \mathit{List} \\
\\
\mathit{sum}\ \mathit{Null} & = & 0 \\
\mathit{sum}\ (\mathit{Cons}\ i\ \mathit{xs}) & = & i + \mathit{sum}\ \mathit{xs}.
\end{array}
$$

If, at some later point, it is decided that the list data type must also support a snoc constructor, then the programmer must go back to the original definition of *List*, appending a new summand for snoc elements and additionally provide an extra clause for *sum* at the point of definition. This, of course, is fairly straightforward for the example in question, but what if the funtion *sum* appears somewhere in millions of lines of source code with the data type definition for lists defined in some other module or section of code? An even worse possibility is that the implementation for lists is provided by a third party, who has chosen not to make the source code available to the programmer, in question. At this point, nothing can be done—the programmer may even have to reimplement the definitions for *List* and *sum*.

The problem is that, in general, most programming languages do not provide direct support for extending data types other than inserting extra constructors at the point of definition. However, this is just the functionality that row polymorphism provides. For example, the algebraic hierarchy example of Section 8.2 used row polymorphism to capture unimportant fields in a monoid. Although we used records in that early example, it applies equally well to variants, and combining the two provides us with an alternative for data type definitions, which can support a form of extensibility.

We do not take this idea any further here but note that there are many unanswered questions. For example, it is not clear how seperate compilation will be affected by the introduction of extensible data types. Furthermore, what are the implications of user specified type signatures in the presence of extensible data types? This is an important question as most programmers use such signatures to constrain more general types and as a form of documentation. This former point is particularly important in the context of Haskell where ambiguous overloading can often be resolved by the insertion of type annotations.

9.2 Unchecked operations

Wand's [Wan87] original extension operator for extensible records was unchecked, meaning that it would extend a record with a value for the label l even if the label l was initially present. The type system of Chapter 3 would result in a type error if extension was applied at label l to a record containing a label l. Thus record extension is checked and fails to capture some programs that are well type in Wand's system.

The problem is that information about the presence of a label is asserted by row extension or by lacks predicates, both of which are incapable of answering the question is a field present. To solve this problem, Section 7.3 introduced a new predicate $(\_?l)$ that delayed questions about field membership from compile-time to run-time. However, we also noted that by itself it was not enough to express unchecked record extension. Combined with row restriction the predicate $(\_?l)$ is enough to express Wand's record extension because. Unfortunately, this has the undesirable effect of leading to the loss of most general unifiers, and thus, a practical type inference algorithm.

One possible solution to this problem might be through the introduction of a new row operator, corresponding to maybe predicates in the same way that row extension relates to lacks predicates. Thus we might introduce the row operator

$$\{\!|\, l : \_?\_ \,|\!\} : * \to \mathit{row} \to \mathit{row},$$

which in turn would allow record extension to be assigned the following type

$$\forall r.\forall \alpha.r?l \Rightarrow \mathit{Rec}\ \{\!|\, l : \alpha ? r \,|\!\} \to \alpha \to \mathit{Rec}\ \{\!|\, l : \alpha ? r \,|\!\}.$$

However, it is unclear how a type $\mathit{Rec}\ \{\!|\, l : \alpha ? r \,|\!\}$ should unify with $\mathit{Rec}\ \{\!|\, l : \alpha \,|\, r \,|\!\}$ or with $\mathit{Rec}\ r$, if at all. The problem is that although it is unknown if a field l is present in the input record it will exist in the resulting record. Rémy [Rém94a] captures this fact by having predicates inside row expressions themselves and allowing predicate variables to range over the different possibilities. It is not clear if a similar approach could be taken here, but it seems reasonable and is an important area for future work.

9.3 Parametricity for qualified types

During the early development of the polymorphic λ-calculus both Girard [Gir72] and Reynolds [Rey74] observed that polymorphic functions acted independently of type. This notion was expanded by Reynolds in his work on parametricity [Rey83], leading to a complete categorical treatment of these properties developed by Reynolds and others [MR92, MS93]. Wadler [Wad89] showed that these results could be extended to types including predicates, such as equality, under the assumption that the evidence constructed for these predicates preserved some logical properties of parametricity.

Furthering the work of Wadler an interesting, and important, area for future work is to consider extending the notion of a logical relation [Sta85, MM85] over predicate entailment, to develop a general parametricity result for qualified types.

Working towards this aim we might develop a definition of logical relation and a proof of Reynold's [Rey83] abstraction theorem for the type system PML (described in Chapter 4).

If $\mathcal{M}$ and $\mathcal{M}'$ are models for $T\Lambda$, a logical relation $\mathcal{R}= \{R^\tau\}$ over $\mathcal{M}$ and $\mathcal{M}'$ is a family of relations such that[1]:

- $\forall \tau \in \mathit{Type}.R^\tau \subseteq \mathcal{M}^\tau \times \mathcal{M}'^\tau$.
- $R^{\tau\times\tau'}((x,x'),(y,y')) \iff R^\tau(x,y) \wedge R^{\tau'}(x',y')$. In the special case when $a : R^\tau$ and $b : R^{\tau'}$ are functions, then $a \times b : R^{\tau\times\tau'}$ is a function, defined by $a \times b\ (x,y) = (a\ x, b\ y)$.
- $R^{\tau\to\tau'}(f,g) \iff \forall x \in \mathcal{M}^\tau, y \in \mathcal{M}'^\tau.R^\tau(x,y) \Rightarrow R^{\tau'}(f\ x, g\ y)$. Thus functions f and g are related if and only if they map related arguments to related results.

  In general it is not the case that if $a : R^\tau$ and $b : R^{\tau'}$ are functions then $R^{\tau\to\tau'}$ is also a function, however in this special case $R^{\tau\to\tau'}(f,g)$ is equivalent to $g \circ a = b \circ f$.
- If c is a constant of type τ, then $R^\tau(\mathcal{M}^\tau(c), \mathcal{M}'^\tau(c))$.
- $R^{\forall\alpha.\sigma}(f,g) \iff \Pi\nu_i \in \mathit{Type}.R^{[\nu_i/\alpha_i]\tau}(f,g)$.

[1]Following Mitchell [Mit96] we write $R(x,y)$ to represent the assertion that x and y are related by the relation R.

The following result extends Reynold's abstraction theorem for Church's simply typed λ-calculus to PML, and is true for all models constructed with respect to the semantics of Chapter 4.

Theorem 9.1 *If* $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}'$ *is a logical relation,* $A \vdash^{PML} E : \sigma$*, and* $R(\eta, \eta')$*, then*

$$\mathcal{R}(\mathcal{M}[\![A \vdash^{PML} E : \sigma]\!]\eta, \mathcal{M}'[\![A \vdash^{PML} E : \sigma]\!]\eta').$$

It is clear that we can extend the above definition of a logical relation to qualified types by the following additions

- $\forall \pi \in Pred. R^{\pi} \subseteq \mathcal{M}^{\pi} \times \mathcal{M}'^{\pi}$.
- $R^{\pi \Rightarrow \rho}(f, g) \Longleftrightarrow \forall e \in \mathcal{M}^{\pi}, e' \in \mathcal{M}'^{\pi}. R^{\pi}(e, e') \Rightarrow R^{\rho}(f\ \ e, g\ \ e)$.

Theorem 9.1 can now be extended to *OML* as

Conjecture 9.2 *If* $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}'$ *is a logical relation,* $P \,|\, A \vdash E : \sigma$*, and* $R(\eta, \eta')$*, then*

$$\mathcal{R}(\mathcal{M}[\![P \,|\, A \vdash E : \sigma]\!]\eta, \mathcal{M}'[\![P \,|\, A \vdash E : \sigma]\!]\eta').$$

This work is a preliminary outline of parametricity for qualified types, providing a promising direction for future study.

Alternatively, work by Mitchell and Scedrov [MS93] looks at the categorical generalization of logical relations and may extend naturally to the categorical semantics for qualified types described in Chapter 5.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison Wesley, 1996.

[Blo92] Stephen Blott. *An Approach to Overloading with Polymorphism*. PhD thesis, University of Glasgow, June 1992. Technical report FP-192-1.

[Bud91] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1991.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented*

Programming: Types, Semantics, and Language Design (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[Coq90] Thierry Coquand. On the analogy between propositions and types. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 399–417. Addison-Wesley, 1990.

[Cro93] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[Dam85] Luis M. M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Technical report CST-33-85.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Gas96] Benedict R. Gaster. A polymorphic type system for extensible records and variants. Transfer dissertation: extended version of [GJ96], 1996.

[Gas97a] Benedict R. Gaster. Polymorphic extensible records for Haskell. In *ACM Haskell Workshop*, June 1997.

[Gas97b] Benedict R. Gaster. A semantics for qualified types. Technical Report NOTTCS-TR-97-5, Computer Science, University of Nottingham, September 1997.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GJ96] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Computer Science, University of Nottingham, November 1996.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[Ham88] A. G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1988.

[Hin69] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HM93] Robert Harper and John Mitchell. On the type structure of Standard ML. *ACM Transaction on Programming Languages and Systems*, 1993.

[How80] W. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[HP90] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, School of Computer Science, Carnegie Mellon University, Feburary 1990.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–636, 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HR92] Barney P. Hilken and David E. Rydeheard. Towards a categorical semantics of type classes. *Fundamenta Informaticae*, XVI:127–147, 1992.

[HS95] J. Hughes and J. Sparud. Haskell++: An object-oriented extension of Haskell. In *Proceedings of Haskell Workshop, La Jolla, California*, YALE Research Report DCS/RR-1075, 1995.

[Jon92a] Mark P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(4):475–504, October 1992.

[Jon92b] Mark P. Jones. A theory of qualified types. In *European symposium on programming, ESOP '92*, Rennes, France, February 1992. Springer Verlag LNCS 582.

[Jon94a] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[Jon94b] Mark P. Jones. *Qualified Types Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Jon95a] Mark P. Jones. The Gofer distribution. University of Nottingham, 1995.

[Jon95b] Mark P. Jones. Simplifying and improving qualified types. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995. At extended version, with proofs, appears as Research report YALE/DCS/RR-1040, Yale University, New Haven, Connecticut USA, June 1994.

[Jon95c] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995. An earlier version appeared in Proceedings Functional Programming and Computer Architecture 1993.

[Jon97] Mark P. Jones. Type safe machine language. In *Glasgow Workshop*, September 1997.

[Ken96] A. J. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, Laboratoire d'Informatique, Ecole Polytechnique, September 1996.

[KR88] Brian W. Kernigham and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, second edition, 1988.

[Lam80] Joachim Lambek. From λ-calculus to cartesian closed categories. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 375–402. Academic Press, London, 1980.

[Lan72] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, 1972.

[Law70] F.W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In *Proceedings of the American Mathematical Society Symposium on Pure Mathematics XVII*, pages 1–14, 1970.

[LB93] Saunders Mac Lane and Garrett Birkhoff. *Algebra*. Chelsea Publishing Company, 3rd edition, 1993.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. In *Principles of Programming Languages*, pages 220–231. ACM press, 1993.

[LS86] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics 7. Cambridge University Press, 1986.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. The Java Series. Addison Wesley, 1996.

[Mai92] Harry Mairson. Quantifier elimination and parametric polymorphism in programming languages. *Journal of Functional Programming*, 2(2):213–226, April 1992.

[Mei97] Erik Meijer. The design and implementation of Mondrian. In *ACM Haskell Workshop*, June 1997.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. Proc. 7th International Conference on Functional Programming and Computer Architecture, June 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Mit96] John Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, 1996.

[MM85] J.C. Mitchell and A.R. Meyer. Second-order logical relations. In *Logics of Programs*, pages 225–236, Berlin, June 1985. Springer-Verlag LNCS 193.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[MR92] QingMing Ma and John C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40, Berlin, 1992. Springer-Verlag.

[MS93] J.C. Mitchell and A. Scedrov. Notes on sconing and relators. In E. Boerger et al., editor, *Computer Science Logic '92, Selected Papers*, pages 352–378. Springer LNCS 702, 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[MTH97] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML (REVISED)*. The MIT Press, 1997.

[Oho89a] Atsushi Ohori. A simple semantics for ML polymorphism. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, September 1989.

[Oho89b] Atsushi Ohori. *A Study of Semantics, Type and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995. A preliminary version appears in Proceedings of ACM Symposium on Principles of Programming Languages, 1992, under the title: A compilation method for ML-style polymorphic record calculi.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 135–146. ACM SIGPLAN, June 1995.

[PF92] Wesley Phoa and Michael Fourman. A proposed categorical semantics for pure ML. University of Edinburgh technical report ECS-LFCS-92-213, Laboratory for the Foundations of Computer Science, 1992.

[PH97] John Peterson and Kevin Hammond. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.4). Technical report, April 1997. Available from http://www.haskell.org.

[Pho92] Wesley Phoa. A simple categorical semantics for ML polymorphism. University of Edinburgh technical report, Laboratory for the Foundations of Computer Science, 1992.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title: Object-Oriented Programming Without Recursive Types.

[Rém92a] Didier Rémy. Efficient representation of extensible records. In *Proceedings of the 1992 workshop on ML and its Applications*, page 12, San Francisco, USA, June 1992.

[Rém92b] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM press.

[Rém94a] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Type, Semantics, and Language Design*, Foundations of Computing Series. MIT Press, 1994. Early version appeared in Sixteenth Annual Symposium on Principles of Programming Languages. Austin, Texas, January 1989.

[Rém94b] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, Foundations of Computing Series. MIT Press, 1994.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing and Machinery*, 12(1):23–41, January 1965.

[RP90] John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 127–152. Addison-Wesley, 1990.

[Sco80] D.S. Scott. Relating theories of the lambda calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450. Academic Press, 1980.

[See87] R. A. G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(4):969–988, December 1987.

[SK88] Stefan Kaes. Parametric polymorphism. In *European symposium on programming*, Lecture Notes in Computer science 300, Nancy, France, 1988. Springer Verlag.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. Technical Report YALEU/DCS/RR-1129, Yale University, 1997.

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[Sul97] Martin Sulzmann. Designing record systems. Technical Report YALEU/DCS/RR-1128, Yale University, 1997.

[Tay90] Paul Taylor. Commutative diagrams in TEX. 1990.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Computer Science Department, Edinburgh University, 1988. CST-52-88.

[Tur37] Alan Turing. The ρ-function in λ-K-conversion. *Journal of Symbolic Logic*, 2:164, 1937.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings Conference on Functional Programming and Computer Architecture*. Springer, 1989.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings ACM Conference on Lisp and Functional Programming*, June 1990.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, (93):1–15, 1991. Preliminary version appeared in Proceedings of the 4th IEEE Symposium on Logic in Computer Science, 1989, 92–97.

[WB89] Philip Wadler and Stephen Blott. How to make ad hoc polymorphism less ad hoc. In *Proceedings 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[Wel94] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *(Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15th November 1994.

[Wri95] Andrew Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

Appendix A

Proofs

This appendix provides proofs for the main results of this dissertation, including the important lemmas. We only consider results whose proofs were not given within the main body of the text.

A.1 Proofs for Chapter 3

A.1.1 Lemma 3.1

Lemma 3.1 *if* $(l : \tau) \in r$*, then* $r = \{\!| l : \tau \,|\, r - l |\!\}$.

Proof: The proof is by induction on the structure of the derivation of $(l : \tau) \in r$. The proof for the case when the last rule of the derivation is (*inRow*) is straightforward. The remaining case is:

Case (inTail): . We have a derivation of the form

$$\frac{(l : \tau) \in r \quad l \neq l'}{(l : \tau) \in \{\!| l' : \tau' \,|\, r |\!\}.}$$

By induction $(l : \tau) \in r$ and $\{\!| l : \tau \,|\, r - l |\!\} = r$. Hence, as $l \neq l'$, we can extend r with $l' : \tau'$, as required.

(*This completes the proof.* $\Box$)

A.1.2 Theorem 3.2

Theorem 3.2 *The unification (insertion) algorithm defined by the rules in Figure 3.3 (Figure 3.4) calculates most-general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.*

Proof: For each case we first prove that U is a unifier (inserter), and then show that it is the most general one. The proof is by induction on the structure of the derivation. The proof for the cases when the last rule in the derivation is $(inVar)$, (id), or $(bind)$ are straightforward. The remaining cases are:

Case (inTail): Calculate as follows

$$\begin{array}{ll} & I\{\!|l' : \tau' \,|\, r|\!\} \\ \Leftrightarrow & \qquad \{\text{definition of substitution}\} \\ & \{\!|l' : I\tau' \,|\, Ir|\!\} \\ \Leftrightarrow & \qquad \{\text{induction, and } l \neq l'\} \\ & (l : I\tau) \in \{\!|l' : I\tau' \,|\, Ir|\!\} \\ \Leftrightarrow & \qquad \{\text{definition of substitution}\} \\ & (l : I\tau) \in I\{\!|l' : \tau' \,|\, r|\!\}. \end{array}$$

For the same case, we show that, if a substitution S also inserts $(l : \tau)$ into $r \in C^{row}$, then it is of the form $S = RI$ for some kind-preserving substitution R. Thus the required result follows by the following calculation.

$$\begin{array}{ll} & S\{\!|l' : \tau' \,|\, r|\!\} \\ \Rightarrow & \qquad \{(inVar)\} \\ & (l : \tau) \in S\{\!|l' : \tau' \,|\, r|\!\} \\ \Leftrightarrow & \qquad \{\text{definition of substitution}\} \\ & (l : \tau) \in \{\!|l' : S\tau' \,|\, Sr|\!\} \\ \Rightarrow & \qquad \{\text{induction}\} \\ & S = RI. \end{array}$$

Case (inHead): Calculate as follows

$$
\begin{array}{ll}
 & I\{\!|l : \tau' \,|\, r|\!\} \\
\Leftrightarrow & \qquad\qquad \{\text{definition of substitution}\} \\
 & \{\!|l : I\tau' \,|\, Ir|\!\} \\
\Leftrightarrow & \qquad\qquad \{\text{induction}\} \\
 & \{\!|l : I\tau \,|\, Ir|\!\} \\
\Leftrightarrow & \qquad\qquad \{\text{definition of substitution}\} \\
 & I\{\!|l : \tau \,|\, r|\!\}.
\end{array}
$$

For the same case, we show that, if a substitution S also inserts $(l : \tau)$ into $r \in C^{row}$, then it is of the form $S = RI$ for some kind-preserving substitution R. Thus the required result follows by the following calculation.

$$
\begin{array}{ll}
 & S\{\!|l : \tau' \,|\, r|\!\} \\
\Leftrightarrow & \qquad\qquad \{\text{definition of substitution}\} \\
 & \{\!|l : S\tau' \,|\, Sr|\!\} \\
\Rightarrow & \qquad\qquad \{\text{induction}\} \\
 & S = RI.
\end{array}
$$

Case (apply): Calculate as follows

$$
\begin{array}{ll}
 & U'\,UCD \\
\Leftrightarrow & \qquad\qquad \{\text{definition of substitution}\} \\
 & (U'\,UC)(U'\,UD) \\
\Leftrightarrow & \qquad\qquad \{\text{induction}\} \\
 & (U'\,UC')(U'\,UD') \\
\Leftrightarrow & \qquad\qquad \{\text{definition of substitution}\}
\end{array}
$$

$$U'UC'D'.$$

For the same case, we show that, if a substitution S also unifies constructors $C, C' \in C^{\kappa}$, then it is of the form $S = S'U'U$. Thus we calculate as follows

$$\begin{array}{ll} & U'UCD = U'UC'D' \\ \Leftrightarrow & \quad \{\text{definition of substitution}\} \\ & (U'UC)(U'UD) = (U'UC')(U'UD') \\ \Rightarrow & \quad \{CD = C'D' \Rightarrow C = C' \wedge D = D'\} \\ & U'UC = U'UC' \wedge U'UD = U'UD'. \end{array}$$

Hence, by induction $S = S'U$ as $C \overset{U}{\sim}_k C'$, and $S' = S''U'$ as $UD \overset{U'}{\sim}_k UD'$. Thus $S = S''U'U$, as required.

Case (row): Calculate as follows

$$\begin{array}{ll} & U'U\{\!|l : \tau \,|\, r|\!\} \\ \Leftrightarrow & \quad \{\text{definition of substitution}\} \\ & \{\!|l : U'U\tau \,|\, U'Ur|\!\} \\ \Leftrightarrow & \quad \{\text{induction}\} \\ & \{\!|l : U'U\tau \,|\, U'Ur' - l|\!\} \\ \Leftrightarrow & \quad \begin{array}{l}\{\text{Lemma 3.1} \\ \text{and definition of substitution}\}\end{array} \\ & U'Ur'. \end{array}$$

For the same case, we show that, if a substitution S unifies constructors $C, C' \in C^{\kappa}$, then it is of the form $S = S'U'U$. Then calculate as follows

$$S\{\!|l : \tau \,|\, r|\!\} = Sr'$$

$$
\begin{array}{ll}
\Leftrightarrow & \qquad \{\text{definition of substitution}\} \\
 & \{\!|\, l : S\tau \mid Sr \,|\!\} = Sr' \\
\Leftrightarrow & \qquad \{\text{induction and lemma 3.1}\} \\
 & \{\!|\, l : S\tau \mid Sr \,|\!\} = \{\!|\, l : S\tau \mid Sr' - l \,|\!\} \\
\Rightarrow & \qquad \{(\mathit{equal})\} \\
 & Sr = Sr' - l.
\end{array}
$$

Now, by induction $(l : S\tau) \in Sr'$, thus $S = S'U$ and therefore $S'Ur = S'Ur' - l$ and $S' = S''U'$. Hence, $S = S''U'U$ as required.

(*This completes the proof.* $\Box$)

A.2 Proofs for Chapter 4

A.2.1 Proposition 4.3

Proposition 4.3 *If* $P \mid C, A \vdash E : \tau$, $S \vdash E \leadsto E'$, *and* $S \vdash A \leadsto A'$, *then* $P \mid C, A' \vdash E' : \tau$ *and* $P|C, A \vdash E = SE' : \tau$.

Proof: The proof follows directly by Lemmas A.7 and A.8.

(*This completes the proof.* $\Box$)

A.2.2 Proposition 4.4

Proposition 4.4 *If* $P \mid C, A \vdash E : \tau$, $S \vdash A \leadsto A'$, $S \vdash E \leadsto F$, *and* $S \vdash E \leadsto F'$, *then* $P|C, A' \vdash F = F' : \tau$.

Proof: It follows, from Proposition 4.3, that $P \mid C, A' \vdash F : \tau$ and $P \mid C, A' \vdash F' : \tau$, thus, we need only show that $P|C, A' \vdash F = F' : \tau$. The proof proceeds by induction on the structure of E. The only non-trivial case is when E is a let-binding.

Case $E = \textbf{let}\ B\ \textbf{in}\ E'$: By assumption we know $P \mid C, A' \vdash \textbf{let}\ B\ \textbf{in}\ E' : \tau$, $S \vdash \textbf{let}\ B\ \textbf{in}\ E' \leadsto F$, and $S \vdash \textbf{let}\ B\ \textbf{in}\ E' \leadsto F'$, and by Definition 4.6 we have the derivations:

$$\frac{S, S' \vdash B \leadsto B' \quad S' \vdash E' \leadsto E'' \quad S'\ \mathit{extends}\ (B, S)}{S \vdash \textbf{let}\ B\ \textbf{in}\ E' \leadsto \textbf{let}\ B'\ \textbf{in}\ E'',}$$

and

$$\frac{S, S'' \vdash B \leadsto B'' \quad S' \vdash E' \leadsto E''' \quad S' \ extends\ (B, S)}{S \vdash \mathbf{let}\ B\ \mathbf{in}\ E' \leadsto \mathbf{let}\ B'\ \mathbf{in}\ E'''.}$$

By definition of the relation *extends* we know that $S' = S''$, and thus, by induction, $P|C, A'' \vdash E'' = E''' : \tau$, where A'' is A' extended with typings for the bindings B. Finally, we show that $B' = B''$ by the following calculation:

$$\begin{array}{ll}
 & B' \\
= & \quad \{\text{definition of } extends\} \\
 & \{x' = N' | x = \lambda v.N) \in B \wedge (x\ e \leadsto x') \in S' \wedge S \vdash [e/v]N \leadsto N'\} \\
= & \quad \{**\} \\
 & \{y' = N' | x = \lambda v.N) \in B \wedge (x\ e \leadsto y') \in S' \wedge S \vdash [e/v]N \leadsto N'\} \\
= & \quad \{\text{induction}\} \\
 & \{y' = N'' | x = \lambda v.N) \in B \wedge (x\ e \leadsto y') \in S' \wedge S \vdash [e/v]N \leadsto N'\} \\
= & \quad \{\text{definition of } extends\} \\
 & B''
\end{array}$$

The step labelled $**$ is justified by the fact that $(x\ e \leadsto x') \in S$ and $(x\ e \leadsto y') \in S$ implies $x' = y'$. The required result follows by application of congruence for let bindings. (*This completes the proof.* □)

A.2.3 Theorem 4.15

Theorem 4.15 (Soundness of PML theories) *If $\mathcal{C}$ is a cartesian closed category, $\mathcal{T}_{PML}$ an equational theory, $\mathcal{M}$ a model of $\mathcal{T}_{PML}$ in $\mathcal{C}$, and $\Gamma, \Gamma' \vdash E =_{PML} F : \tau$, then, $\mathcal{M} \models_{PML} \Gamma, \Gamma' \vdash E =_{PML} F : \tau$.*

Proof: The proof proceeds by induction on the height of the derivation $\Gamma, \Gamma' \vdash E =_{PML} F : \tau$. The standard rules for equational reasoning follow from the corresponding properties for equality of $\mathcal{C}$. The case when the last rule of the derivation is (β-*let*) follows directly as translation from *MML* to $T\Lambda$, given in Figure 4.8, simply induces the equality for let-bindings (rule $(let)^{TL}$). The remaining cases are:

Case η: We have a derivation of the form:

$$\frac{\Gamma, \Gamma' \vdash^{PML} \lambda x.Ex : \tau' \to \tau \quad x \notin Fv(E)}{\Gamma, \Gamma' \vdash \lambda x.Ex =_{PML} E : \tau}$$

By definition of the semantic function for PML we calculate as follows:

$$\begin{array}{ll}
 & \mathcal{M}\llbracket\Gamma, \Gamma' \vdash^{PML} \lambda x.Ex : \tau' \to \tau\rrbracket\eta \\
= & \qquad \{\text{definition of semantic function}\} \\
 & \{(\tau' \to \tau, \mathcal{M}\llbracket\Gamma' \vdash^{m} [N_j/\gamma_j](\lambda x.Ex) : \tau' \to \tau\rrbracket\eta)\} \\
= & \qquad \{\text{definition of substitution, and } \forall j.\gamma_j \neq x\} \\
 & \{(\tau' \to \tau, \mathcal{M}\llbracket\Gamma' \vdash^{m} \lambda x.([N_j/\gamma_j]E)x : \tau' \to \tau\rrbracket\eta)\} \\
= & \qquad \{\text{soundness of equational theories for MML}\} \\
 & \{(\tau' \to \tau, \mathcal{M}\llbracket\Gamma' \vdash^{m} [N_j/\gamma_j]E : \tau' \to \tau\rrbracket\eta)\} \\
= & \qquad \{\text{definition of semantic function}\} \\
 & \mathcal{M}\llbracket\Gamma, \Gamma' \vdash^{PML} E : \tau' \to \tau\rrbracket\eta,
\end{array}$$

as required.

Case β: We have a derivation of the form:

$$\frac{\Gamma, \Gamma', x : \tau' \vdash^{PML} E : \tau \quad \Gamma, \Gamma' \vdash^{PML} F : \tau'}{\Gamma, \Gamma' \vdash (\lambda x.E)F =_{PML} [F/x]E : \tau}$$

By definition of the semantic function for PML we calculate as follows:

$$\begin{array}{ll}
 & \mathcal{M}\llbracket\Gamma, \Gamma' \vdash^{PML} (\lambda x.E)F : \tau\rrbracket\eta \\
= & \qquad \{\text{definition of semantic function}\} \\
 & \{(\tau, \mathcal{M}\llbracket\Gamma' \vdash^{m} [N_j/\gamma_j]((\lambda x.E)F) : \tau\rrbracket\eta)\} \\
= & \qquad \{\text{definition of substitution, and } \forall j.\gamma_j \neq x\} \\
 & \{(\tau, \mathcal{M}\llbracket\Gamma' \vdash^{m} (\lambda x.[N_j/\gamma_j]E)([N_j/\gamma_j]F) : \tau\rrbracket\eta)\} \\
= & \qquad \{\text{soundness of equational theories for MML}\}
\end{array}$$

$$
\begin{array}{cl}
 & \{(\tau, \mathcal{M}[\![\Gamma' \models^{m} [[N_j/\gamma_j]F/x]([N_j/\gamma_j]E) : \tau]\!]\eta)\} \\
= & \quad \{\forall j.\gamma_j \neq x \wedge x \not\in Fv(N_j) \text{ so we can apply Lemma A.2}\} \\
 & \{(\tau, \mathcal{M}[\![\Gamma' \models^{m} [N_j/\gamma_j][F/x]E : \tau]\!]\eta)\} \\
= & \quad \{\text{definition of semantic function}\} \\
 & \mathcal{M}[\![\Gamma, \Gamma' \models^{PML} [F/x]E : \tau' \to \tau]\!]\eta,
\end{array}
$$

as required.

(*This completes the proof.* $\square$)

A.2.4 Theorem 4.17

Theorem 4.17 (Soundness of *OML* **theories)** *Let* $\mathcal{M}$ *be any model and* $\mathcal{T}_{PML}$ *an equational theory, then* $P|C, A \vdash E = F : \sigma \Rightarrow \mathcal{M} \models_{OML} P|C, A \vdash E = F : \sigma$.

Proof: The proof proceeds by induction on the height of the derivation of equality. The standard rules and those for β and η follow from Theorem 4.15. The remaining cases are β-evi and η-evi, we prove the case for η-evi, noting that the proof for β-evi is similar, although a little longer.

Case η-evi: We have a derivation of the form:

$$\frac{P, v : \pi, P' \,|\, C, A \vdash E : \rho \quad P \Vdash e : \pi \quad v \not\in Fv(E)}{P|C, A \vdash (\lambda v.Ev)e = [e/v]E : \rho}$$

Applying the definition of the semantic function to the left-hand side gives:

$$
\begin{array}{l}
\mathcal{M}[\![w : P \,|\, C, A \vdash \lambda v.Ev : Q \Rightarrow \tau]\!]^{OML}\eta = \\
\qquad\qquad \{(\tau, \mathcal{M}[\![A \models^{PML} \mathit{Spec}(id, \lambda v.Ev) : \tau]\!]^{PML}\eta) | \Vdash P, Q\}.
\end{array}
$$

Now unfolding $\mathit{Spec}(id, \lambda v.Ev)$ we have:

$$
\begin{array}{lll}
\mathit{Spec}(id, \lambda v.Ev) & = & M \\
\quad \textbf{where} & & \\
\qquad \Vdash e' : P, e'' : Q & & \\
\qquad S \vdash A \leadsto A & & \\
\qquad S \vdash ([e'/w]\lambda v.Ev)e'' \leadsto M, & &
\end{array}
$$

where $w \not\in Fv(e'')$, $v \not\in Fv(e') \cup Fv(e'')$, and $v \neq w$. Thus by the definition of substitution we have:

$$S \vdash (\lambda v.([e'/w]E)v)e'' \rightsquigarrow M.$$

Now we continue the proof by evaluating the right-hand side:

$$\mathcal{M}[\![w : P \,|\, C, A \vdash E : Q \Rightarrow \tau]\!]^{OML}\eta = \{(\tau, \mathcal{M}[\![A \vdash^{PML} \mathit{Spec}(id, E) : \tau]\!]^{PML}\eta) | \Vdash P, Q\}.$$

Now unfolding $\mathit{Spec}(id, E)$ we have:

$$\begin{array}{l} \mathit{Spec}(id, E) \;\; = \;\; N \\ \quad \textbf{where} \\ \qquad \Vdash e' : P, e'' : Q \\ \qquad S \vdash A \rightsquigarrow A \\ \qquad S \vdash ([e'/w]E)e'' \rightsquigarrow N. \end{array}$$

Now by Theorem 4.15 we need only show that the specialised terms M and N are equal. To see that this is the case, observe that by the definition of specialisation (Figure 4.6) we have a derivation of the form:

$$\frac{S \vdash [e''/v](([e'/w]E)v) \rightsquigarrow M}{S \vdash (\lambda v.([e'/w]E)v)e'' \rightsquigarrow M}$$

So by the definition of substitution it follows that:

$$S \vdash ([e'/w]E)e'' \rightsquigarrow M,$$

as $v \not\in Fv(e') \cup Fv(e'') \cup Fv(E)$. But by Proposition 4.4 it must be the case that $M = N$, as required.

(*This completes the proof.* $\Box$)

A.3 Proofs for Chapter 5

A.3.1 Proposition 5.1

Proposition 5.1 (Subject reduction (syntactic type soundness)) *If* $T; P \,|\, C, A \vdash E : \rho$ *and* $E \Downarrow V$, *then* $T; P \,|\, C, A \vdash V : \rho$.

Proof: The proof proceeds on the structure of the derivation $T; P \mid C, A \vdash E : \rho$. The cases when the last rule of the derivation is $(varP)$ or $(varM)$ are not applicable as reduction is defined for closed terms, only. The case when the last rule of the derivation is either $(\rightarrow I)$, $(\Rightarrow I)$, $(unit)$, or $(\times I)$ are straightforward. The remaining cases are:

Case (E×): We have a derivation of the form

$$\frac{T; P \mid C, A \vdash E : \tau \times \tau'}{T; P \mid C, A \vdash \mathit{fst}\ E : \tau}$$

There is one rule for reduction which applies

$$\frac{E \Downarrow (V, U)}{\mathit{fst}\ E \Downarrow V}$$

By the inductive hypothesis

$$T; P \mid C, A \vdash (V, U) : \tau \times \tau'.$$

Thus we have a derivation of the form

$$\frac{T; P \mid C, A \vdash V : \tau \quad T; P \mid C, A \vdash U : \tau'}{T; P \mid C, A \vdash (V, U) : \tau \times \tau'}$$

which implies $T; P \mid C, A \vdash V : \tau$, as required.

The case when the last rule in the derivation is $(\times E)$ follows by a similar argument.

Case (→E): We have a derivation of the form

$$\frac{T; P \mid C, A \vdash E : \tau' \rightarrow \tau \qquad T; P \mid C, A \vdash F : \tau'}{T; P \mid C, A \vdash EF : \tau}$$

There is one rule for reduction which applies

$$\frac{E \Downarrow \lambda x.E' \quad [F/x]E' \Downarrow V}{EF \Downarrow V}$$

By the inductive hypothesis

$$T; P \mid C, A \vdash \lambda x.E' : \tau' \to \tau,$$

and the fact that substitution preserves type, Lemma A.2, we have $T; P \mid C, A \vdash [F/x]E' : \tau$ and, by the inductive hypothesis, it follows that $[F/x]E' \Downarrow V$ and $T; P \mid C, A \vdash V : \tau$, as required.

The case when the last rule in the derivation is $(\Rightarrow E)$ follows by a similar argument.

Case (let): We have a derivation of the form

$$\frac{[E/x]F \Downarrow V}{\mathbf{let}\ x = E\ \mathbf{in}\ F \Downarrow V,}$$

which by the inductive hypothesis and the fact that substitution preserves type gives $T; P \mid C, A \vdash V : \tau$, as required. (*This completes the proof.* $\Box$)

A.3.2 Lemma 5.4

Lemma 5.4 *If τ and ν are monotypes, such that, $T = TV(\tau) \cup TV(\nu)$, $\mathcal{M}$ over $\mathbb{C}[\varphi(T)]$ is a model for monotypes, and $[\nu/\alpha]$ a substitution, then*

$$\mathcal{M}[\![[\nu/\alpha]\tau]\!] = \mathcal{M}[\![[\nu/\alpha]]\!]\mathcal{M}[\![\tau]\!].$$

Proof: The proof proceeds by induction on the structure of τ.

Case $\tau \equiv \iota$: In this case $[\nu/\alpha]\iota = \iota$, by definition of substitution, thus the required result follows by reflexivity of equality in the category $\mathbb{C}$ and $[\![[\nu/\alpha]]\!]([\![\iota]\!]) = [\![\iota]\!]$.

Case $\tau \equiv \beta$: There are two cases to consider

- The case when $\alpha = \beta$, and
- the case when $\alpha \neq \beta$.

We consider each in turn. In the case when $\alpha = \beta$, we have $[\![[\nu/\alpha]\alpha]\!] = [\![\nu]\!] = [\![[\nu/\alpha]]\!](X_i) = [\![[\nu/\alpha]]\!][\![\alpha]\!]$, as expected. In the case when $\alpha \neq \beta$ we have $[\![[\nu/\alpha]\beta]\!] = [\![\beta]\!] = X_k = [\![[\nu/\alpha]]\!](X_k) = [\![[\nu/\alpha]]\!]([\![\beta]\!])$, where $k \neq i$, as required.

Case $\tau \equiv ()$: In this case $[\nu/\alpha]() = ()$, by definition of substitution, thus the required result follows by reflexivity of equality in the category $\mathbb{C}$ and $[\![[\nu/\alpha]]\!]([\![()]\!]) = \mathbb{1} = [\![()]\!]$.

Case $\tau \equiv \tau \to \tau'$: By the inductive hypothesis we have $[\![[\nu/\alpha]\tau]\!] = [\![[\nu/\alpha]]\!]([\![\tau]\!])$ and $[\![[\nu/\alpha]\tau']\!] = [\![[\nu/\alpha]]\!]([\![\tau']\!])$. We now calculate as follows:

$$
\begin{array}{cl}
 & [\![[\nu/\alpha]\tau \to \tau']\!] \\
= & \quad \{\text{definition of substitution}\} \\
 & [\![[\nu/\alpha]\tau \to [\nu/\alpha]\tau']\!] \\
= & \quad \{\text{definition of semantic function for types}\} \\
 & [[\![[\nu/\alpha]\tau]\!] \to [\![[\nu/\alpha]\tau']\!]] \\
= & \quad \{\text{inductive hypothesis}\} \\
 & [[\![[\nu/\alpha]]\!]([\![\tau]\!]) \to [\![[\nu/\alpha]]\!]([\![\tau']\!])] \\
= & \quad \{[\![[\nu/\alpha]]\!] \text{ is by definition a cartesian closed functor}\} \\
 & [\![[\nu/\alpha]]\!][[\![\tau]\!] \to [\![\tau']\!]] \\
= & \quad \{\text{definition of semantic function for types}\} \\
 & [\![[\nu/\alpha]]\!]([\![\tau \to \tau']\!]),
\end{array}
$$

as required.

Case $\tau \equiv \tau \times \tau'$: By the inductive hypothesis we have $[\![[\nu/\alpha]\tau]\!] = [\![[\nu/\alpha]]\!]([\![\tau]\!])$ and $[\![[\nu/\alpha]\tau']\!] = [\![[\nu/\alpha]]\!]([\![\tau']\!])$. We now calculate as follows:

$$
\begin{array}{cl}
 & [\![[\nu/\alpha]\tau \times \tau']\!] \\
= & \quad \{\text{definition of substitution}\} \\
 & [\![[nu/\alpha]\tau \times [\nu/\alpha]\tau']\!] \\
= & \quad \{\text{definition of semantic function for types}\} \\
 & [\![[\nu/\alpha]\tau]\!] \times [\![[\nu/\alpha]\tau']\!] \\
= & \quad \{\text{inductive hypothesis}\} \\
 & [\![[\nu/\alpha]]\!]([\![\tau]\!]) \times [\![[\nu/\alpha]]\!]([\![\tau']\!]) \\
= & \quad \{[\![[\nu/\alpha]]\!] \text{ is by definition a cartesian closed functor}\} \\
 & [\![[\nu/\alpha]]\!]([\![\tau]\!] \times [\![\tau']\!]) \\
= & \quad \{\text{definition of semantic function for types}\} \\
 & [\![[\nu/\alpha]]\!]([\![\tau \times \tau']\!]),
\end{array}
$$

as required.

(*This completes the proof.* $\Box$)

A.3.3 Lemma 5.7

Lemma 5.7 (Predicate soundness) *Given a judgement* $T; v : P \Vdash w : Q$*, and a predicate system* $\mathcal{P}$*, then* $\mathcal{P}[\![T; v : P \Vdash w : Q]\!] : \mathcal{P}[\![P]\!] \longrightarrow \mathcal{P}[\![Q]\!]$.

Proof: The proof proceeds by induction on the structure of the derivation $T; v : P \Vdash w : Q$.

Case (id): We have a derivation of the form

$$v : P \Vdash v : P$$

So assuming $[\![P]\!]$ it follows, by the definition of a category, that we can construct the arrow:

$$[\![P]\!] \xrightarrow{id} [\![P]\!],$$

as required.

Case (term): We have a derivation of the form

$$v : P \Vdash \emptyset.$$

Again, assuming $[\![P]\!]$, we can construct, by the universal property of terminals, the arrow

$$[\![P]\!] \xrightarrow{term} \mathbb{1},$$

as required.

Case (fst): We have a derivation of the form

$$v : P, w : Q \Vdash v : P.$$

Assuming $[\![P, Q]\!]$ we can construct, by the definition of products, the arrow

$$[\![P]\!] \times [\![Q]\!] \xrightarrow{\pi_1} [\![P]\!],$$

as required.

The proof for (snd) is very similar.

Case (univ): We have a derivation of the form

$$\frac{v : P \Vdash e : Q \qquad v : P \Vdash e' : R}{v : P \Vdash e : Q, e' : R}.$$

By induction we have

$$[\![v : P \Vdash e : Q]\!] : [\![P]\!] \xrightarrow{e} [\![Q]\!],$$

and

$$[\![v : P \Vdash e' : R]\!] : [\![Q]\!] \xrightarrow{e'} [\![R]\!].$$

By the universal property of products we can construct the arrow

$$[\![P]\!] \xrightarrow{\langle e, e' \rangle} [\![Q]\!] \times [\![R]\!],$$

which by the definition of the semantic function gives the required result.

Case (trans): We have a derivation of the form

$$\frac{v : P \Vdash e : Q \qquad v' : Q \Vdash e' : R}{v : P \Vdash [e/v']e' : R}$$

By induction we have

$$[\![v : P \Vdash e : Q]\!] : [\![P]\!] \xrightarrow{e} [\![Q]\!],$$

and

$$[\![v' : Q \Vdash e' : R]\!] : [\![Q]\!] \xrightarrow{e'} [\![R]\!].$$

By the definition of composition of arrows we construct the arrow

$$[\![P]\!] \xrightarrow{e} [\![Q]\!] \xrightarrow{e'} [\![R]\!],$$

which by the definition of the semantic function gives the required result.
Case (close): We have a derivation of the form

$$\frac{v : P \Vdash e : Q}{v : SP \Vdash e : SQ}$$

By induction we have

$$[\![v : P \Vdash e : Q]\!] : [\![P]\!] \xrightarrow{e} [\![Q]\!].$$

Assuming we have the substitution functor

$$S[\![P]\!] : Pred[\vec{X}] \to Pred[\vec{X}],$$

respecting S, then we can construct an arrow

$$S[\![P]\!] \xrightarrow{e} S[\![Q]\!],$$

giving

$$[\![v : SP \Vdash e : SQ]\!] : S[\![P]\!] \xrightarrow{e} S[\![Q]\!],$$

as required.

(*This completes the proof.* $\square$)

A.3.4 Lemma 5.11

Lemma 5.11 (Type soundness) *If* $T; P \mid C, A \vdash E : \rho$ *then*

$$\mathcal{M}[\![T; P \mid C, A \vdash E : \rho]\!] : \mathcal{E}(\mathcal{P}[\![P]\!]) \times C_{C,E} \times \mathcal{M}[\![A]\!] \longrightarrow \mathcal{M}[\![\rho]\!].$$

Proof: The proof is by induction on the structure of $T; P \mid C, A \vdash E : \rho$ and is similar to the proof of Lemma 5.7.

(*This completes the proof.* $\square$)

A.3.5 Theorem 5.12

Theorem 5.12 (Soundness of *OML* reduction) *If* $T; P \mid A \vdash E : \rho$ *and* $E \Downarrow V$*, then*

$$\mathcal{M}[\![T; P \mid A \vdash E : \rho]\!] = \mathcal{M}[\![T; P \mid A \vdash V : \rho]\!].$$

Proof: The proof proceeds by induction on the structure of the derivation $T; P \mid A \vdash E : \rho$. The cases when the last rule of the derivation is either $(varM)$ or $(varP)$ do not apply as E must be closed for $E \Downarrow V$ to be valid. Furthermore, the cases when the last rule of the derivation is $(\to I)$, $(\Rightarrow I)$, $(\times I)$, or $(unit)$ follow directly by Proposition 5.1 and reflexivity. The remaining cases are:

Case $(\to E)$: We have derivations of the form:

$$\frac{T; P \mid C, A \vdash E : \tau' \to \tau \qquad T; P \mid C, A \vdash F : \tau'}{T; P \mid C, A \vdash EF : \tau} \qquad \frac{E \Downarrow \lambda x.E' \quad [F/x]E' \Downarrow V}{EF \Downarrow V}$$

The required result follows by simple calculation

$$\begin{array}{ll}
 & [\![T; P \mid A \vdash EF : \tau]\!] \\
= & \quad \{\text{definition of semantic function}\} \\
 & eval \circ \langle [\![T; P \mid A \vdash E : \tau' \to \tau]\!], [\![T; P \mid A \vdash F : \tau']\!] \rangle \\
= & \quad \{\text{inductive hypothesis}\} \\
 & eval \circ \langle [\![T; P \mid A \vdash \lambda x.E' : \tau' \to \tau]\!], [\![T; P \mid A \vdash F : \tau']\!] \rangle \\
= & \quad \{\text{definition of semantic function}\} \\
 & eval \circ \langle curry([\![T; P \mid A, x : \tau' \vdash E' : \tau]\!] \circ s), [\![T; P \mid A \vdash F : \tau']\!] \rangle \\
= & \quad \{\text{properties of products}\} \\
 & eval \circ curry([\![T; P \mid A, x : \tau' \vdash E' : \tau]\!] \circ s) \times id \circ \langle id, [\![T; P \mid A \vdash F : \tau']\!] \rangle \\
= & \quad \{\text{properties of exponentials}\} \\
 & [\![T; P \mid A, x : \tau' \vdash E' : \tau]\!] \circ s \circ \langle id, [\![T; P \mid A \vdash F : \tau']\!] \rangle \\
= & \quad \{\text{substitution is composition}\} \\
 & [\![T; P \mid A \vdash [F/x]E' : \tau]\!] \\
= & \quad \{\text{inductive hypothesis}\} \\
 & [\![T; P \mid A \vdash V : \tau]\!],
\end{array}$$

as required.

The case when the last rule in the derivation is $(\Rightarrow E)$ follows by a similar proof.

Case $(E\times)$: We have derivations of the form

$$\frac{T; P \,|\, C, A \vdash E : \tau \times \tau'}{T; P \,|\, C, A \vdash \mathit{fst}\ E : \tau} \qquad \frac{E \Downarrow (V, U)}{\mathit{fst}\ E \Downarrow V}$$

The required result follows by simple calculation

$$\begin{array}{ll} & [\![T; P \,|\, A \vdash \mathit{fst}\ E : \tau]\!] \\ = & \qquad \{\text{definition of semantic function}\} \\ & \pi_1 \circ [\![T; P \,|\, A \vdash E : \tau \times \tau']\!] \\ = & \qquad \{\text{inductive hypothesis}\} \\ & \pi_1 \circ [\![T; P \,|\, A \vdash (V, U) : \tau']\!] \\ = & \qquad \{\text{definition of semantic function}\} \\ & \pi_1 \circ \langle [\![T; P \,|\, A \vdash V : \tau]\!], [\![T; P \,|\, A \vdash U : \tau']\!] \rangle \\ = & \qquad \{\text{properties of products}\} \\ & [\![T; P \,|\, A \vdash V : \tau]\!], \end{array}$$

as required. We have been a little liberal with the diagonal maps as these only effect the polymorphic contexts, which may differ for each component of the tuple, but as the expression *fst* E is closed the expressions V and U are also closed, and thus, we need not be concerned with the input of polymorphic contexts (up to isomorphism) and consequently can ignore the diagonal maps.

The case when the last rule in the derivation is $(\times E)$ follows by a similar proof.

Case (let): We have derivations of the form

$$\frac{T; P \,|\, C, A \vdash E : \tau \qquad T; Q \,|\, C, A_x, x : \sigma \vdash F : \tau \quad \sigma = \mathit{Gen}(C, A, P \Rightarrow \tau)}{T; P, Q \,|\, C, A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$

$$\frac{[x/E]F \Downarrow V}{\mathbf{let}\ x = E\ \mathbf{in}\ F \Downarrow V}$$

We now calculate as follows

$$\begin{array}{ll} & [\![T; P \,|\, A \vdash \mathbf{let}\ x = E\ \mathbf{in}\ F : \tau]\!] \\ = & \qquad \{\text{definition of semantic function}\} \end{array}$$

$$\begin{array}{ll}
 & [\![T; P \,|\, x : Gen(A,\tau), A \vdash F : \tau']\!] \circ \langle\, [\![T; P \,|\, A \vdash E : \tau]\!]^{[\nu/\alpha]_1}, \cdots, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad [\![T; P \,|\, A \vdash E : \tau]\!]^{[\nu/\alpha]_r} \rangle \\
= & \qquad \{\text{substitution is composition (1)}\} \\
 & [\![T; P \,|\, A \vdash [E/x]F : \tau]\!] \\
= & \qquad \{\text{inductive hypothesis}\} \\
 & [\![T; P \,|\, A \vdash V : \tau]\!],
\end{array}$$

as required. To justify step (1) consider the the definition of polymorphic contexts given in Definition 5.9, which interprets a type scheme as the product of its instances used in the body of the expression being interpreted. If $x_1, \cdots, x_r$ are the instances of x in the expression F we can rewrite $T; P \,|\, x : Gen(A,\tau), A \vdash F : \tau'$ as $T; P \,|\, x_1 : Gen(A,\tau)^{[\nu/\alpha]_1}, \cdots, x_r : Gen(A,\tau)^{[\nu/\alpha]_r} \vdash F' : \tau'$, where F' is F with each occurrence of x renamed to the appropriate x_i. Thus the required result now follows by the fact that substitution is interpreted as composition and then simply rewriting the multiple substitution $[E/x_1, \cdots, E/x_1]$ as the substitution $[E/x]$ and renaming $x_1, \cdots, x_r$ in F' back to x, giving F.

(*This completes the proof.* $\Box$)

A.4 Proofs for Chapter 6

A.4.1 Proposition 6.3

Proposition 6.3 (Lacks predicates are enough) *If σ is a type scheme, then there exists a type scheme, σ', such that $\sigma \rightsquigarrow \sigma'$, $nHas(\sigma') = 0$, and $[\![\sigma]\!] = [\![\sigma']\!]$. Furthermore, if there exists a type scheme, σ'', such that $\sigma \rightsquigarrow \sigma''$, then $[\![\sigma']\!] = [\![\sigma'']\!]$.*

Proof: It is clear that for any well-formed type scheme σ there exists a σ' such that $\sigma \rightsquigarrow \sigma'$ as the rules for translation are defined by induction over the structure of a type scheme. We need only show that $[\![\sigma]\!] = [\![\sigma']\!]$. The proof proceeds by induction on the structure of the derivation $\sigma \rightsquigarrow \sigma'$.

Case (scheme): We have a derivation of the form

$$\frac{Q = \{\_has\_ \in P\} \quad Q' = \{\_\backslash\_ \in P\} \quad Q \Rightarrow \tau \rightsquigarrow P' \Rightarrow \tau'}{\forall\alpha.P \Rightarrow \tau \rightsquigarrow \forall\alpha.Q' \cup P' \Rightarrow \tau'}$$

By induction we have $nHas(P' \Rightarrow \tau') = 0$ and $[\![Q \Rightarrow \tau]\!] = [\![P' \Rightarrow \tau']\!]$. Thus it follows that $nHas(\forall\alpha.Q' \cup P' \Rightarrow \tau') = 0$, by definition of the function $nHas$ and the operator $\cup$. Finally, it follows by the semantic definition for polymorphic types, that, $[\![\forall\alpha.P \Rightarrow \tau]\!] = [\![\forall\alpha.Q' \cup P' \Rightarrow \tau']\!]$.

Case (empty): We have a derivation of the form

$$\frac{}{\{\} \Rightarrow \tau \rightsquigarrow \{\} \Rightarrow \tau}$$

The required result follows directly as $nHas(\{\} \Rightarrow \tau) = 0$, and $[\![\{\} \Rightarrow \tau]\!] = [\![\{\} \Rightarrow \tau]\!]$, by reflexivity.

Case $\Rightarrow$: We have a derivation of the form

$$\frac{P \Rightarrow \tau \rightsquigarrow P' \Rightarrow \tau'' \quad P \Rightarrow \tau' \rightsquigarrow P'' \Rightarrow \tau'''}{P \Rightarrow \tau \to \tau' \rightsquigarrow P' \cup P'' \Rightarrow \tau'' \to \tau'''}$$

Now by induction $nHas(P' \Rightarrow \tau'') = 0$, $nHas(P'' \Rightarrow \tau''') = 0$, $[\![P \Rightarrow \tau]\!] = [\![P' \Rightarrow \tau'']\!]$, and $[\![P \Rightarrow \tau']\!] = [\![P'' \Rightarrow \tau''']\!]$. It follows, by definition of $\cup$, that, $nHas(P' \cup P'' \Rightarrow \tau'' \to \tau''') = 0$, and by the semantic definition of monotypes we have $[\![\tau]\!] = [\![\tau'']\!]$ and $[\![\tau']\!] = [\![\tau''']\!]$. Thus, by definition of equality, for function arrows, we have $[\![\tau \to \tau']\!] = [\![\tau''' \to \tau''']\!]$, as required.

Case (minus): We have a derivation of the form

$$\frac{}{\begin{array}{r}(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ ((r - l_1) \cdots - l_n) \rightsquigarrow \\ (r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r\end{array}}$$

It follows that $nHas((r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r) = 0$, and thus we need only show that

$$\begin{array}{r}[\![(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ ((r - l_1) \cdots - l_n]\!] = \\ [\![(r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r]\!].\end{array}$$

To see that this is the case pick an arbitrary value for the row r, which respects the predicates $(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n)$ (by definition of semantics, given above, the predicate set P does not constrain the row r, and thus does not play any role):

$$\{\!| l_1 : \tau_1, \cdots, l_n : \tau_n, l'_1 : \nu_1, \cdots, l'_k : \nu_k |\!\},$$

implying that $Rec\ ((r - l_1) \cdots - l_n)) = Rec\{\!| l'_1 : \nu_1, \cdots, l'_k : \nu_k |\!\}$. Semantically we have:

$$[\![(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n) \Rightarrow Rec\ ((r - l_1) \cdots - l_n]\!] = \nu_1 \times \cdots \times \nu_k,$$

when $r = \{\!| l_1 : \tau_1, \cdots, l_n : \tau_n, l'_1 : \nu_1, \cdots, l'_k : \nu_k |\!\}$, assuming $l'_1 < \cdots < l'_k$.

Similarly, picking the same value for r on the right hand side of the translation, leaving out the fields $l_1 : \tau_1, \cdots, l_n : \tau_n$ as asserted by the predicates $r \backslash l_1, \cdots, r \backslash l_n$, gives:

$$[\![(r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r]\!] = \nu_1 \times \cdots \times \nu_k,$$

when $r = \{\!| l'_1 : \nu_1, \cdots, l'_k : \nu_k |\!\}$, assuming $l'_1 < \cdots < l'_k$.

Now as we made no assumptions about the row r, the types $(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n) \Rightarrow Rec\ ((r - l_1) \cdots - l_n$ and $(r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\ r$ are semantically equal, whenever r is chosen, such that, it satisfies the appropriate predicates.

Case (noHas): We have a derivation of the form

$$\frac{r\ has\ \_ \not\in P}{P \Rightarrow Rec\ r \rightsquigarrow \{\} \Rightarrow Rec\ r}$$

The required result follows directly as $nHas(\{\} \Rightarrow Rec\ r) = 0$, by definition, and $[\![P \Rightarrow Rec\ r]\!] = [\![\{\} \Rightarrow Rec\ r]\!]$, as the predicates P do not constrain the row r.

Case (ext): We have a derivation of the form

$$\frac{}{\begin{array}{l}(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ r \rightsquigarrow \\ \qquad\qquad (r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\{\!| l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r |\!\}\end{array}}$$

It follows that $nHas((r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\{\!| l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r |\!\}) = 0$, and thus, we need only show that

$$\begin{array}{l}[\![(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n, P) \Rightarrow Rec\ r]\!] = \\ \qquad\qquad [\![(r \backslash l_1, \cdots, r \backslash l_n) \Rightarrow Rec\{\!| l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r |\!\}]\!].\end{array}$$

To see that this is the case pick an arbitrary value for the row r, which respects the predicates $(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n)$ (by definition of the semantics, given above, the predicate set P as does not constrain the row r, and thus does not play any role):

$$\{\!|l_1 : \tau_1, \cdots, l_n : \tau_n, l'_1 : \nu_1, \cdots, l'_k : \nu_k|\!\},$$

implying that $Rec\ r = Rec\{\!|l_1 : \tau_1, \cdots, l_n : \tau_n, l'_1 : \nu_1, \cdots, l'_k : \nu_k|\!\}$. Semantically we have:

$$[\![(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n) \Rightarrow Rec\ r]\!] = \tau_1 \times \cdots \tau_n \times \nu_1 \times \cdots \times \nu_k,$$

assuming $l_1 < \cdots < l_n < l'_1 < \cdots < l'_k$.

Similarly, picking the same value for r on the right hand side of the translation, leaving out the fields $l_1 : \tau_1, \cdots, l_n : \tau_n$ as asserted by the predicates $r\backslash l_1, \cdots, r\backslash l_n$, gives:

$$[\![(r\backslash l_1, \cdots, r\backslash l_n) \Rightarrow Rec\ \{\!|l_1 : \tau_1, \cdots, l_n : \tau_n \,|\, r|\!\}]\!] = \tau_1 \times \cdots \times \tau_n \times \nu_1 \times \cdots \times \nu_k,$$

assuming $l_1 < \cdots < l_n < l'_1 < \cdots < l'_k$.

Now as we made no assumptions about the row r, the types $(r\ has\ l_1 : \tau_1, \cdots, r\ has\ l_n : \tau_n) \Rightarrow Rec\ r$ and $(r\backslash l_1, \cdots, r\backslash l_n) \Rightarrow Rec\ \{\!|l_1 : \tau, \cdots, l_n : \tau_n \,|\, r|\!\}$ are semantically equal, whenever r is chosen, such that it satisfies the appropriate predicates.

Case (noHE): We have a derivation of the form

$$\frac{r\ has\ \_ \not\in P}{P \Rightarrow Rec\{\!|l_1 : \tau_1 \,|\, r|\!\} \leadsto P \Rightarrow Rec\{\!|l_1 : \tau_1 \,|\, r|\!\}}$$

The required result follows directly as $nHas(P \Rightarrow Rec\{\!|l_1 : \tau_1 \,|\, r|\!\}) = 0$, by definition. Of course, in general it will not be the case that $[\![P \Rightarrow Rec\ \{\!|l_1 : \tau_1 \,|\, r|\!\}]\!] = [\![\{\} \Rightarrow Rec\ \{\!|l_1 : \tau_1 \,|\, r|\!\}]\!]$. However, as the predicates P contain no lacks predicates, as specified by the rule (*scheme*), and by assumption no has predicates we note that P can be eliminated and $[\![\{\} \Rightarrow Rec\ \{\!|l_1 : \tau_1 \,|\, r|\!\}]\!] = [\![\{\} \Rightarrow Rec\ \{\!|l_1 : \tau_1 \,|\, r|\!\}]\!]$ is valid as the resulting predicates Q will preserve the requirement that $r\backslash l$.

Case (empR): We have a derivation of the form

$$\frac{}{P \Rightarrow Rec\{\!||\!\} \leadsto \{\} \Rightarrow Rec\{\!||\!\}}$$

It is clear that $nHas(\{\} \Rightarrow Rec\{\!|\,|\!\}) = 0$. Now to see that $[\![P \Rightarrow Rec\{\!|\,|\!\}]\!] = [\![\{\} \Rightarrow Rec\{\!|\,|\!\}]\!]$, we first note that as $\sigma = Q \Rightarrow \tau$ is well-formed the predicates in P constrain the type τ in a non-ambiguous way and as such it is safe to eliminate P in this case. Thus $[\![\{\} \Rightarrow Rec\{\!|\,|\!\}]\!] = [\![\{\} \Rightarrow Rec\{\!|\,|\!\}]\!]$, as required.

The case when the last rule in the derivation is (*var*) follows by a similar argument. (*This completes the proof.* □)

A.5 Proofs for Chapter 7

A.5.1 Theorem 7.1

Theorem 7.1 *The unification (insertion) algorithm defined by the rules in Figure 3.3 (Figure 3.4), extended to include the additional rules given in Figure 7.4 (Figure 7.5), calculates most-general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.*

Proof: For each case we first prove that U is a unifier (inserter), and then show that it is the most general one. The proof is by induction on the structure of the derivation. The proof for the cases when the last rule in the derivation is (*inVar*), (*inTail*), (*inHead*), (*id*), (*bind*) (*apply*), and (*row*) are similar to the proof of Theorem 3.2. The cases for the rules involving the insertion of a given label into a set of labels are just simplified versions of the corresponding proofs for inserters, given in Theorem 3.2. The cases that involve the constructor *array* are straightforward. The remaining cases are:

Case (inTo): Calculate as follows

$$\begin{array}{ll}
 & I(to\ \tau'\ r) \\
\Leftrightarrow & \qquad \{\text{definition of substitution}\} \\
 & to\ (I\tau')\ (Ir) \\
\Leftrightarrow & \qquad \{*\} \\
 & to\ I\tau'\ \{\!|\, l : I\alpha \,|\, Ir - l \,|\!\} \\
\Leftrightarrow & \qquad \{\text{unfold definition of } to\} \\
 & \{\!|\, l : I\alpha \to I\tau' \,|\, to\ I\tau'\ Ir - l \,|\!\}
\end{array}$$

$$
\begin{array}{ll}
\Leftrightarrow & \qquad \{(\mathit{inRow})\} \\
& (l : I\alpha \to I\tau') \in \{\!|\, l : I\alpha \to I\tau' \,|\, \mathit{to}\ I\tau'\ Ir - l \,|\!\} \\
\Leftrightarrow & \qquad \{\text{by induction } I\tau = I(\alpha \to \tau')\} \\
& (l : I\tau) \in \{\!|\, l : I\alpha \to I\tau' \,|\, \mathit{to}\ I\tau'\ Ir - l \,|\!\} \\
\Leftrightarrow & \qquad \{\text{fold definition of } \mathit{to}\} \\
& (l : I\tau) \in \mathit{to}\ I\tau'\ \{\!|\, I\alpha \,|\, Ir - l \,|\!\} \\
\Leftrightarrow & \qquad \{*\} \\
& (l : I\tau) \in \mathit{to}\ I\tau'\ Ir \\
\Leftrightarrow & \qquad \{\text{definition of substitution}\} \\
& (l : I\tau) \in I(\mathit{to}\ \tau'\ r),
\end{array}
$$

as required.

We can justify the step $*$ by induction and Lemma 3.1, which gives

$$(l : I\alpha) \in Ir \Rightarrow Ir = \{\!|\, l : I\alpha \,|\, Ir - l \,|\!\}.$$

For the same case, we must show that, if a substitution S inserts $(l : \tau)$ into $\mathit{to}\ \tau'\ r$, then it is of the form $S = RIU$. To see this is the case note that by the definition of substitution $S(\mathit{to}\ \tau'\ r) = \mathit{to}\ S\tau'\ Sr$. Hence by induction we have $S = S'U$ as $U\tau = U(\alpha \to \tau')$, and $S' = RI$ as $(l : U\alpha) \overset{I}{\in} Ur$, as required.

The case for when the last rule in the derivation is (*inFrom*)is proved by simply replacing $\mathit{to}\ \tau'\ r$ with $\mathit{from}\ \tau'\ r$ and $\alpha \to \tau'$ with $\tau' \to \alpha$.

Case (ToTo): Calculate as follows

$$
\begin{array}{ll}
& U(\mathit{to}\ \tau\ r) \\
\Leftrightarrow & \qquad \{\text{definition of substitution}\} \\
& \mathit{to}\ U\tau\ Ur \\
\Leftrightarrow & \qquad \{\text{induction}\} \\
& \mathit{to}\ U\tau'\ Ur' \\
\Leftrightarrow & \qquad \{\text{definition of substitution}\} \\
& U(\mathit{to}\ \tau'\ r')
\end{array}
$$

as required.

For the same case, we must show that, if a substitution S unifies *to* τ r into *to* τ' r', then it is of the form $S = RU'U$. To see this is the case note that by the definition of substitution $S(to\ \tau\ r) = to\ S\tau\ Sr$. Hence by induction we have $S = S'U$ as $\tau \overset{U}{\sim} \tau'$, and $S' = RU'$ as $r \overset{U'}{\sim} Ur'$, as required.

The case for when the last rule in the derivation is (*FromFrom*)is proved by simply replacing *to* τ r and *to* τ' r' with *from* τ r and *from* τ' r', respectively.

(*This completes the proof.* $\Box$)

A.6 Proofs for lemmas

To conclude this appendix we outline the proofs for a number of simple lemmas, which are required for the proof of the main results of Chapters 4 and 5.

Lemma A.2 *If* $P \mid C, A \vdash [[N/y]F/x]([N/y]E) : \sigma$, $y \neq x$, *and* $x \notin Fv(N)$. *Then,* $P \mid C, A \vdash [N/y]([F/x]E) : \sigma$.

Proof: The proof proceeds by induction on the structure of E.

Case $E = z$: There are three cases to consider:

- $z = x$,
- $z = y$, and
- $z \neq x \wedge z \neq y$.

We consider each in turn.

Case $z = x$: Substituting z for x we can calculate as follows

$$
\begin{array}{cl}
 & P \mid C, A \vdash [[N/y]F/z]([N/y]z) : \sigma \\
= & \qquad \{\text{definition of substitution}\} \\
 & P \mid C, A \vdash [[N/y]F/z]z : \sigma \\
= & \qquad \{\text{definition of substitution}\} \\
 & P \mid C, A \vdash [N/y]F : \sigma \\
= & \qquad \{\text{definition of substitution}\} \\
 & P \mid C, A \vdash [N/y]([F/z]z) : \sigma,
\end{array}
$$

as required.

Case $z = y$: Substituting z for y we can calculate as follows

$$\begin{array}{ll} & P \,|\, C, A \vdash [[N/z]F/x]([N/z]z) : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash [[N/z]F/x]N : \sigma \\ = & \quad \{\text{definition of substitution and } x \notin Fv(N)\} \\ & P \,|\, C, A \vdash N : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash [N/z]z : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash [N/z]([F/x]z) : \sigma \end{array}$$

as required.

Case $z \neq x \wedge z \neq y$: Substituting z for y we can calculate as follows

$$\begin{array}{ll} & P \,|\, C, A \vdash [[N/y]F/x]([N/y]z) : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash [[N/y]F/x]z) : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash z : \sigma \\ = & \quad \{\text{definition of substitution}\} \\ & P \,|\, C, A \vdash [N/y]([F/x]z) : \sigma \end{array}$$

as required.

Case $E = \lambda z.M$: Without loss of generality we can assume that $z \neq x$ and $z \neq y$. This is justified by the fact that we assume terms equal modulo renaming of bound variables.

$$P \,|\, C, A \vdash [[N/y]F/x]([N/y]\lambda z.M) : \sigma$$

$$
\begin{aligned}
= & \quad \{\text{definition of substitution and } y \neq z\} \\
& P \,|\, C, A \vdash [[N/y]F/x](\lambda z.[N/y]M) : \sigma \\
= & \quad \{\text{definition of substitution and } x \neq z\} \\
& P \,|\, C, A \vdash \lambda z.[[N/y]F/x]([N/y]M) : \sigma \\
= & \quad \{\text{induction hypothesis}\} \\
& P \,|\, C, A \vdash \lambda z.[N/y]([F/x]M) : \sigma \\
= & \quad \{\text{definition of substitution and } y \neq z\} \\
& P \,|\, C, A \vdash [N/y](\lambda z.[F/x]M) : \sigma \\
= & \quad \{\text{definition of substitution and } x \neq z\} \\
& P \,|\, C, A \vdash [N/y]([F/x]\lambda z.M) : \sigma
\end{aligned}
$$

as required.

Case $E = \lambda v.M$: The proof for this case is very similar to that of term abstraction.

Case $E = MM'$: We calculate as follows

$$
\begin{aligned}
& P \,|\, C, A \vdash [[N/y]F/x]([N/y]MM') : \sigma \\
= & \quad \{\text{definition of substitution}\} \\
& P \,|\, C, A \vdash [[N/y]F/x](([N/y]M)([N/y]M')) : \sigma \\
= & \quad \{\text{definition of substitution}\} \\
& P \,|\, C, A \vdash ([[N/y]F/x]([N/y]M))([[N/y]F/x]([N/y]M')) : \sigma \\
= & \quad \{\text{induction hypothesis}\} \\
& P \,|\, C, A \vdash ([N/y][F/x]M)([N/y][F/x]M') : \sigma \\
= & \quad \{\text{definition of substitution}\} \\
& P \,|\, C, A \vdash [N/y](([F/x]M)([F/x]M')) : \sigma \\
= & \quad \{\text{definition of substitution}\} \\
& P \,|\, C, A \vdash [N/y]([F/x]MM') : \sigma
\end{aligned}
$$

as required.

Case $E = Me$: The proof for this case is very similar to that of term application. (*This completes the proof.* $\Box$)

The following lemma is a standard structural result which is proven by induction on the number of occurrences of quantifiers in a given type. The interested reader will find a proof in Mairson's paper discussing the equivalence of the type systems PML and MML [Mai92].

Lemma A.3 *If* $\Gamma, \Gamma' \vdash^{PML} E : \forall\{\alpha_i\}.\tau$, *and* E *is not a variable, then there exists a proof* $\Gamma, \Gamma' \vdash^{PML} E : \tau$ *where the last rule used is either* $(\to E)^{PML}$, $(\to I)^{PML}$, *or* $(let)^{PML}$.

Lemma A.4 *If* $P \,|\, C, A, x : \sigma' \vdash E : \sigma$ *and* $x \notin Fv(E)$, *then* $P \,|\, C, A \vdash E : \sigma$.

Proof: Without loss of generality we can rename all variables bound in E so that they are unique with respect to x. By Lemma A.3 we can eliminate applications of the rules $(\forall I)$ and $(\forall E)$, thus allowing the proof to proceed by induction on the structure of $P \,|\, C, A, x : \sigma' \vdash E : \sigma$. The proofs for the cases where the last rule in the derivation is $(\to E)$, $(\Rightarrow E)$, or $(\Rightarrow I)$ are straightforward. The remaining cases are:

Case (const): We have a derivation of the form

$$\frac{(y : \sigma) \in C}{P \,|\, C, A, x : \sigma' \vdash y : \sigma} \quad (const).$$

By assumption $y \neq x$, thus, the desired result follows by application of $(const)$. The required result follows similarly in the case when the last rule of the derivation is (var).

Case $(\to I)$: We have a derivation of the form

$$\frac{P \,|\, C, A_y, x : \sigma', y : \tau' \vdash E : \tau}{P \,|\, C, A, x : \sigma' \vdash \lambda y.E : \tau' \to \tau} \quad (\to I).$$

By assumption $E = \lambda y.E'$, $\sigma = \tau' \to \tau$, and $y \neq x$. By Lemma A.5 we know that $P \mid C, A_y, y : \tau', x : \sigma' \vdash E : \tau$ and thus, by induction, $P \mid C, A_y, y : \tau' \vdash E : \tau$.

Hence, the required result follows by application of $(\to I)$.

Case (let): We have a derivation of the form:

$$\frac{P \,|\, C, A, x : \sigma' \vdash E : \sigma'' \quad Q \,|\, C, A_y, x : \sigma', y : \sigma'' \vdash F : \tau}{P, Q \,|\, C, A, x : \sigma' \vdash (\mathbf{let}\ y = E\ \mathbf{in}\ F) : \tau} \quad (let).$$

By assumption $y \neq x$ and by induction and Lemma A.5 $P \mid C, A \vdash E : \sigma''$ and $Q \,|\, C, A_y, y : \sigma'' \vdash F : \tau$. Hence, the required result follows by application of (let).

(*This completes the proof.* $\square$)

The proof of the following lemma is similar in style to the previous result and is again a standard result for qualified types. We leave the details to the reader.

Lemma A.5 *If* $P \,|\, C, A, x : \sigma', y : \sigma'' \vdash E : \sigma$, *then* $P \,|\, C, A, y : \sigma'', x : \sigma' \vdash E : \sigma$.

Lemma A.6 *If* $P \mid C, A, x : \tau \vdash E : \sigma$ *and* $P \mid C, A \vdash E' : \tau$, *then* $P \mid C, A \vdash [E'/x]E : \sigma$.

Proof: Without loss of generality we rename all variables bound in E so they are unique with respect to x. The proof then proceeds by induction on the structure of E.

Case $E = z$: There are two cases to consider

- $x = z$, and
- $x \neq z$.

We consider each in turn.

Case $x = z$: By assumption we have $P \mid C, A, x : \tau \vdash x : \tau$, and it follows, by the definition of substitution, that $[E'/x]x = E'$, and thus, by assumption $P \,|\, C, A \vdash E' : \tau$ as required.

Case $x \neq z$: It follows, by the definition of substitution, that the expression $[E'/x]z$ reduces to z. By assumption $z : \sigma \in C, A$ which by application of either (var) or $(const)$ gives $P \,|\, C, A \vdash z : \sigma$.

Case $E = \lambda y.F$: By Lemma A.3 it follows that $P \,|\, C, A, x : \tau \vdash \lambda y.F : \tau' \to \tau''$ for some τ' and τ''. Thus, by applying Lemma A.3, we have reduced the proof of the judgement $P \,|\, C, A, x : \tau \vdash \lambda y.F : \tau' \to \tau''$ to a syntax directed proof, and by rule $(\to I)$ we have a derivation of the form:

$$\frac{P \,|\, C, A, x : \tau, y : \tau' \vdash F : \tau''}{P \,|\, C, A, x : \tau \vdash \lambda y.F : \tau' \to \tau''}$$

By Lemma A.5 we have $P \mid C, A, y : \tau', x : \tau \vdash F : \tau''$, and by induction it follows that $P \,|\, C, A, y : \tau' \vdash [E'/x]F : \tau''$. Now by application of $(\to I)$ we have $P \,|\, C, A \vdash \lambda y.[E'/x]F : \tau' \to \tau''$, and by definition of substitution it follows that $P \,|\, C, A \vdash [E'/x](\lambda y.F) : \tau' \to \tau''$. Finally the required result follows by zero or more applications of the rules $(\forall I)$ and $(\forall E)$.

The case when $E = \lambda v.F$ is proved similarly.

Case $E = MN$: By Lemma A.3 it follows that $P \,|\, C, A, x : \tau \vdash MN : \tau'$ for some τ'. Thus, by applying Lemma A.3 we have reduced the proof of the judgement $P \,|\, C, A, x : \tau \vdash MN : \tau'$ to a syntax directed proof, and by rule $(\to E)$ we have a derivation of the form:

$$\frac{P \,|\, C, A, x : \tau \vdash M : \tau'' \to \tau' \quad P \,|\, C, A, x : \tau \vdash N : \tau''}{P \,|\, C, A, x : \tau \vdash MN : \tau'}$$

Now by induction we have $P \,|\, C, A, \vdash [E'/x]M : \tau'' \to \tau'$ and $P \,|\, C, A \vdash [E'/x]N : \tau''$. Thus, by rule $(\to E)$we have $P \,|\, C, A, x : \tau \vdash ([E'/x]M)([E'/x]N) : \tau'$ and by definition of substitution $P \mid C, A, x : \tau \vdash [E'/x]MN : \tau'$. Finally, the required result follows by zero or more applications of the rules $(\forall I)$ and $(\forall E)$.

The case when $E = Fe$ is proved similarly.

Case $E = \textbf{let } y = M \textbf{ in } N$: By Lemma A.3 it follows that $P \mid C, A, x : \tau \vdash \textbf{let } y = M \textbf{ in } N : \tau'$ for some τ'. Thus, by applying Lemma A.3, we have reduced the proof of the judgement $P \mid C, A, x : \tau \vdash \textbf{let } y = M \textbf{ in } N : \tau'$ to a syntax directed proof, and by rule (let) we have a derivation of the form:

$$\frac{P \,|\, C, A, x : \tau \vdash M : \sigma' \quad Q \,|\, C, A, x : \tau, y : \sigma' \vdash N : \tau'}{P, Q \,|\, C, A, x : \tau \vdash \textbf{let } y = M \textbf{ in } N : \tau'}$$

By induction we have $P \mid C, A \vdash [E'/x]M : \sigma'$, and by first applying Lemma A.5 and then induction we also have $Q \mid C, A, y : \sigma' \vdash [E'/x]N : \tau'$. Now by application of the rule (*let*) we have $P, Q \mid C, A \vdash \textbf{let } y = [E'/x]M \textbf{ in } [E'/x]N : \tau'$, which, by definition of substitution, gives $P, Q \mid C, A \vdash [E'/x](\textbf{let } y = M \textbf{ in } N) : \tau'$. Thus the required result follows by zero or more applications of the rules $(\forall I)$ and $(\forall E)$.

(*This completes the proof.* $\square$)

Lemma A.7 *If $P \mid C, A \vdash E : \sigma$ and $S \vdash E \rightsquigarrow E'$, then $P \mid C, A' \vdash E' : \sigma'$ and $\sigma' \leq \sigma$.*

Proof: The proof is by induction on the structure of $S \vdash E \rightsquigarrow E'$. The proofs for the cases where the last rule in the derivation is (*abs-evi*), (*app*), (*abs*), or (*abs-evi*) are straightforward. The remaining cases are:

Case (var-λ): We have a derivation of the form:

$$\frac{x \not\in S}{S \vdash x \rightsquigarrow x} \quad (\textit{var-}\lambda).$$

By assumption $P \mid C, A \vdash x : \sigma$, thus $\sigma \leq \sigma$ as required.

Case (var-let): We have a derivation of the form:

$$\frac{(x\ e \rightsquigarrow x') \in S \quad e \Longrightarrow d}{S \vdash x\ e \rightsquigarrow x'} \quad (\textit{var-let}).$$

By assumption $P \mid C, A \vdash xe : \tau$ where $\sigma = \tau$ from some τ, and also as $e \Longrightarrow d$ we have $P \mid C, A \vdash xd : \tau$. Now, as $xe = Sx'$ and $(xd \rightsquigarrow x') \in S$ and the fact substitutions preserve type, $P \mid C, A' \vdash x' : \tau$, assuming $(x' : \tau) \in A'$. The required result follows from the fact that $\tau \leq \tau$.

Case (β_{evi}): We have a derivation of the form:

$$\frac{S \vdash [e/v]E \rightsquigarrow E'}{S \vdash (\lambda v.E)e \rightsquigarrow E'} \quad (\beta_{evi}).$$

By assumption $P \mid C, A \vdash (\lambda v.E)e : \sigma$, and it follows by Proposition A.3 $P \mid C, A \vdash (\lambda v.E)e : \rho$ for some ρ. Now by $(\Rightarrow E)$, $P \mid C, A \vdash \lambda v.E : \pi \Rightarrow \rho$ and $P \Vdash e : \pi$. It follows, by Lemma A.6 that $P \mid C, A \vdash [e/v]E : \rho$, thus, by induction, $P \mid C, A' \vdash E' : \rho$ and $\rho \leq \sigma$ as required.

Case (let): We have a derivation of the form:

$$\frac{S, S' \vdash B \leadsto B' \quad S' \vdash E \leadsto E' \quad S' \ \mathit{extends}(B, S)}{S \vdash \mathit{let}\ B\ \mathit{in}\ E \leadsto \mathit{let}\ B'\ \mathit{in}\ E'} \quad (\mathit{let})$$

By assumption we have $Q, P \mid C, A \vdash \mathit{let}\ B\ \mathit{in}\ E : \tau$ and $\sigma = \tau$. By induction, it follows, that if $(x_i = E_i) \in B$ and $Q \mid C, A \vdash E_i : \sigma_i$, then $(x_i' = E_i') \in B$ and $Q \mid C, A' \vdash E_i' : \sigma_i'$, where $\sigma_i' \leq \sigma_i$. Also $P \mid C, A, x_i : \sigma_i \vdash E : \tau$, thus, by induction $P \mid C, A', x_i' : \sigma_i' \vdash E' : \tau$. Hence, the required result follows by application of (let).
(*This completes the proof.* $\square$)

Lemma A.8 *If $S \vdash E \leadsto E'$, then $E = SE'$.*

Proof: The proof is by induction on the structure of $S \vdash E \leadsto E'$ and is straightforward, except when the last rule in the derivation is (let). In that case we have a derivation of the form

$$\frac{S \vdash [e/v]E \leadsto E'}{S \vdash (\lambda v.E)e \leadsto E'} \quad (\beta_{evi}).$$

Let $B = \{x_i = \lambda \nu_i.F_i\}$, it follows from the hypothesis S' *extends* (B, S) that S' can be written in the form $[x_{i_j} \ \ e_j/x_j']S$. From the hypothesis $S, S' \vdash B \leadsto B'$, and the earlier definition, we know that $B' = \{x_j' = F_j'\}$ from some F_j' such that $S \vdash [e_j/\nu_{i_j}]F_{i_j} \leadsto F_j'$, and hence, by induction, $[e_j/\nu_{i_j}]F_{i_j} = SF_j'$. The required equality can now be established through the following reasoning

$$\begin{array}{ll}
& \textbf{let}\ B\ \textbf{in}\ E \\
= & \qquad \{\text{induction}\} \\
& [\lambda \nu_i.F_i/x_i]E \\
= & \\
& [\lambda \nu_i.F_i/x_i]S'E' \\
= &
\end{array}$$

$$
\begin{aligned}
& [\lambda\nu_i.F_i/x_{i_j}][x_{i_j}\ \ e_j/x'_j](SE') \\
= & \\
& [(\lambda\nu_i.F_i)\ \ e_j/x_{i_j}]SE' \\
= & \\
& [(\lambda\nu_i.F_i)\ \ e_j/x'_j]SE' \\
= & \\
& [[e_j/\nu_i]F_i/x'_j]SE' \\
= & \qquad \{\text{induction}\} \\
& [SF'_j/x'_j]SE' \\
= & \\
& S([F'_j/x'_j]E') \\
= & \\
& S(\textbf{let } B' \textbf{ in } E'),
\end{aligned}
$$

as required. (*This completes the proof.* $\Box$)

Appendix B

Introduction to polynomial categories

This appendix describes the categorical notions used throughout this dissertation. It has been included as an appendix with the intention of allowing the reader quick reference to categorical definitions and notation used throughout. These concepts and results have appeared elsewhere and are not due to the current author, we restate them now simply to allow the reader to become familiar with the ideas and our choice of notation. We assume the reader has some knowledge of category theory and its application to program language semantics. The interested reader will find Mac Lane's text [Lan72] an excellent reference, while Crole's text [Cro93] provides an introduction to the application of category theory to formal language semantics.

Definition B.1 *A category $\mathbb{C}$ is cartesian if the unique functor $!\mathbb{C} : \mathbb{C} \to \mathbb{1}$ has a right adjoint, and the diagonal functor $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$, defined by $A \mapsto \langle A, A\rangle$ and $f \mapsto \langle f, f\rangle$, also has a right adjoint. We denote this later functor by $\_ \times \_ : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, with the standard projections denoted by π_1 and π_2. In otherwords a cartesian category has finite products and a terminal object.*

Definition B.2 *If $\mathbb{C}$ and $\mathbb{D}$ are cartesian categories, a* **cartesian functor** *$F : \mathbb{C} \to \mathbb{D}$ is a functor that preserves finite products, i.e., $F(A \times B) \cong FA \times FB$ and $F\mathbb{1} \cong \mathbb{1}$.*

Definition B.3 *A cartesian category* $\mathbb{C}$ *is cartesian closed if the functor* $\_ \times A : \mathbb{C} \to \mathbb{C}$ *has a right adjoint. We denote this functor by* $[\_ \to \_] : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C}$*, with the adjoint transpose denoted as* **curry** *and evaluation as* **eval**.

Definition B.4 *If* $\mathbb{C}$ *and* $\mathbb{D}$ *are cartesian closed categories, a* **cartesian closed functor** $F : \mathbb{C} \to \mathbb{D}$ *is a cartesian functor that also preserves exponentials, i.e.,* $F([A \to B]) \cong [FA \to FB]$.

One of the key ideas in Chapter 5 is to use the categorical generalization of polynomials to interpret first-order polymorphism. To help provide some intuition behind polynomial categories we first describe the more common idea of polynomial rings in terms of the categorical notion of universality. A more detailed discussion of these and other similar ideas can be found in a wide selection of texts on algebra (see, for example, Mac Lane and Birkhoff's introductory text [LB93]).

Definition B.5 (Algebraic rings) *A* **ring** $\mathbb{R} = (\mathbb{R}, +, *, 1)$ *is a set* $\mathbb{R}$ *with two binary operations, addition and multiplication, and a unit for multiplication, such that:*

- $(\mathbb{R}, +)$ *is an abelian group;*
- $(\mathbb{R}, *, 1)$ *is a monoid; and*
- *Multiplication distributes over addition.*

A **commutative** *ring is one in which the multiplication is commutative.*

Consider the following polynomial with integral coefficients and indeterminate x

$$f = 4x + 67x^2 + 5x^3.$$

It is possible to consider this expression as shorthand for the function $f(x) = 4x + 67x^2 + 5x^3$, which can be evaluated by substituting numbers in place of the variable x. However, polynomials are formal expressions, which may themselves be added and multiplied forming a ring, when the coefficients form a commutative ring $\mathbb{K}$. By the *ring of polynomials* in the indeterminate, x, written $\mathbb{K}[x]$, we mean the set of all symbols $a_o + a_1 x + \cdots + a_n x^n$, where n can be any nonnegative integer and the coefficients a_i are all in the ring K.

The construction of the polynomial ring $\mathbb{K}[x]$ from the commutative ring $\mathbb{K}$ can be characterized using the categorical notion of universality. Intuitively this can be thought of as a 'universal' way of adjoining a new element x to the ring $\mathbb{K}$. Given a commutative ring $\mathbb{K}$, an inclusion arrow $\iota : \mathbb{K} \to \mathbb{K}[x]$, and any ring homomorphism $f : \mathbb{K} \to \mathbb{L}$, there is a unique arrow $\varphi : \mathbb{K}[x] \to \mathbb{L}$ such that $\varphi(x) = d$ and the following diagram commutes:

$$\begin{tikzcd} \mathbb{K} \arrow[r, "\iota"] \arrow[dr, "f"'] & \mathbb{K}[x] \arrow[d, dashed, "\varphi"] \\ & \mathbb{L}, \end{tikzcd}$$

where $\mathbb{L}$ is a commutative ring and $d \in \mathbb{L}$. Intuitively, this definition states that the ring homomorphism f uniquely extends to the ring homomorphism φ, which behaves as f except in the case when it is applied to the indeterminate x, in which case it evaluates to value d. Operationally, φ is simply f with a built-in environment containing the given value for x. Alternatively one can think of φ evaluating the polynomial, with respect to f, at the point $x = d$.

Keeping the notion of polynomial rings in mind we now generalize the notion of adjoining an indeterminate to cartesian and cartesian closed categories.

Definition B.6 (Cartesian (closed) polynomial categories) *Let $\mathbb{C}$ and $\mathbb{D}$ be cartesian (closed) categories, X an object (called an indeterminate), and $F : \mathbb{C} \to \mathbb{D}$ a cartesian (closed) functor, then $\mathbb{C}[X]$ is a polynomial category if there exists an inclusion functor $I : \mathbb{C} \to \mathbb{C}[X]$ and a unique cartesian (closed) functor $G : \mathbb{C}[X] \to \mathbb{D}$, such that $G(X) = D$ and the following diagram commutes:*

$$\begin{tikzcd} \mathbb{C} \arrow[r, "I"] \arrow[dr, "F"'] & \mathbb{C}[X] \arrow[d, dashed, "G"] \\ & \mathbb{D}, \end{tikzcd}$$

where $\mathbb{D}$ is any cartesian (closed) category, $F : \mathbb{C} \to \mathbb{D}$ is a cartesian (closed) functor, and D is any object $D \in \mathbb{D}$.

The following proposition shows that given any cartesian (closed) category one can construct (unique up to isomorphism) the corresponding polynomial category.

Proposition B.7 (Cartesian (closed) polynomial categories) *If* $\mathbb{C}$ *is a cartesian (closed) category,* X *an indeterminate, there exists an inclusion functor* $I : \mathbb{C} \to \mathbb{C}[X]$ *and a unique cartesian (closed) functor* $G : \mathbb{C}[X] \to \mathbb{D}$*, such that* $G(X) = D$ *and the following diagram commutes:*

$$\begin{tikzcd} \mathbb{C} \arrow[r, "I"] \arrow[dr, "F"'] & \mathbb{C}[X] \arrow[d, "G"] \\ & \mathbb{D}, \end{tikzcd}$$

where $\mathbb{D}$ *is a cartesian (closed) category,* $F : \mathbb{C} \to \mathbb{D}$ *is a cartesian (closed) functor, and* $D \in \mathbb{D}$*.*

It is straightforward to generalize the notion of a polynomial category in one indeterminate to many by observing that the isomorphism $\mathbb{C}[X, Y] \cong (\mathbb{C}[X])[Y]$ follows directly by application of Proposition B.7.

We now introduce the notion of a substitution functor, which provides a method for instantiating an indeterminate $X \in \mathbb{C}[X]$ to an object in the underlying category $\mathbb{C}$.

Lemma B.8 (Cartesian (closed) substitution functor) *If* $\mathbb{C}$ *is a cartesian (closed) category* $\mathbb{C}$ *and* C *and object of* $\mathbb{C}$*, then the cartesian (closed) functor* $G : \mathbb{C}[X] \to \mathbb{C}$*, with* $GI = 1_{\mathbb{C}}$ *and* $G(X) = C$*, is uniquely determined. (Note that the functor* I *is as defined in Proposition B.7).*

Intuitively, a substitution functor is one which behaves as the identity functor on objects of $\mathbb{C}$, while instantiating the indeterminate object X to a particular object of C in $\mathbb{C}$. For example, consider the object $[X \to [X \to X]]$ in $\mathbb{C}[X]$, which is sent to the object $[C \to [C \to C]]$ in $\mathbb{C}$.

Following the generalization of cartesian (closed) polynomials in one indeterminate to many unknowns, we can extend the notion of a substitution functor to more that one argument.

Lemma B.9 *If* $\mathbb{C}[\vec{X}\,]$ *is a polynomial cartesian (closed) category and* $\vec{D} = \langle D_1, \cdots, D_n \rangle$ *is an n-tuple of objects of* $\mathbb{C}[\vec{X}\,]$*, then the substitution functor*

$$S_{\vec{D}} : \mathbb{C}[\vec{X}\,] \to \mathbb{C}[\vec{X}\,],$$

is the unique functor defined such that

$$S(X_1) = D_1, \cdots, S(X_n) = D_n,$$

and commuting with $\mathbb{C} \hookrightarrow \mathbb{C}[\vec{X}\,]$*.*

Index