From Hindley-Milner Types to First-Class Structures

Mark P. Jones
Department of Computer Science, University of Nottingham,
University Park, Nottingham NG7 2RD, England.
mpj@cs.nott.ac.uk

Abstract

We describe extensions of the Hindley-Milner type system to support higher-order polymorphism and first-class structures with polymorphic components. The combination of these features results in a 'core language' that rivals the expressiveness of the Standard ML module system in some respects and exceeds it in others.

1 Introduction

The Hindley-Milner type system [13, 28, 8], hereafter referred to as HM, represents a significant and highly influential step in the design and development of programming language type systems. The main reason for this is that it combines the following features in a single framework:

- Type security: soundness results guarantee that well-typed programs cannot 'go wrong'.
- Flexibility: polymorphism allows the use and definition of functions that behave uniformly over all types.
- Type inference: the existence of principal types, and algorithms to compute them, provides a direct way to identify well-typed terms without the use of type annotations.

Note that it is this combination, rather than the individual features themselves, that make HM particularly interesting. For example, there are many different type systems supporting polymorphism of one form or another, while soundness is usually regarded as an essential prerequisite in the design of any type system. HM is also quite attractive from the perspective of language implementors:

- The type inference algorithm is easy to implement and behaves well in practice.
- Polymorphism is easy to implement, packaging function arguments as boxed values—a uniform representation that is independent of the type of values involved. One of the main attractions of this approach is the ease with which it permits true separate compilation.

As a result, HM has been used as a basis for several widely used functional languages including Standard ML (SML) [29] and Haskell [14].

We should also recognize that HM has some significant limitations, including:

- Polymorphism is restricted to the second-order case, i.e. the form of polymorphism used in the polymorphic λ-calculus [38], also known as System F [9, 10]. Higher-order polymorphism of the form provided in F_ω, allowing the specification of functions that behave uniformly over all type constructors is not supported.

- The price that we pay for the convenience of type inference is the inability to define or use functions with polymorphic arguments. By contrast, arguments with polymorphic type can be used in system F and F_ω.

Despite these limitations, and thanks to the use of polymorphism and higher-order functions, HM does permit a limited form of abstraction. For example, the definition of a polymorphic sort procedure may be parameterized by a suitable comparison function. But the need for mechanisms to support large-scale, modular programming have prompted the development of more complex systems, including HM as a 'core'. The best known example of this is the SML module system which has a precise formal definition [29] and has proved to be very useful in practice [4]. Unfortunately, such systems can be quite complicated and may not be fully integrated with the core language. For example, type-theoretic treatments of the SML module system are usually based on the use of dependent types [26, 24, 30]. But it is not easy to discern the role of dependent types in the formal definition of SML [29], which, instead, uses a semantics based on freshly generated tokens or *stamps* to account for the concepts of *sharing* and *generativity*. Recent work by Leroy [22] provides a more type-theoretic treatment of the same concepts but, once again, relies on non-trivial extensions of the underlying dependent type theory. Additional machinery is necessary to extend the SML module system beyond its current definition, for example, to support modules as first-class values [31, 20], higher-order modules [12, 34, 41, 25, 7], polymorphism [19], and *transluscent sums* or *manifest types* [11, 21]. This last item is motivated in part by the desire to admit a form of separate compilation; this is not possible with the current definition of SML [2]. These extended systems are undoubtedly very powerful, but they are also rather complex.

In the remaining sections, we show how relatively modest changes to HM can overcome the limitations above: i.e. to support higher-order polymorphism (Section 2) and to package up polymorphic values in first-class *structures* that can be used as function arguments (Section 3). The resulting system, referred to here as XHM, is of interest in its own right as a study of the boundaries of type inference and also as an implementation language for type and constructor classes [43, 17]. However, in this paper, we focus on the possibility of using structures as the basis for a module system. Unlike the SML module system, we do not make any separation between the core and module languages. For example, structures can be assigned polymorphic types and used as function arguments to implement higher-order modules. Another important difference is that XHM structures do not contain type components (Section 4). The paper ends with a summary of background and related work (Section 5) and with conclusions and suggestions for future work (Section 6).

2 Higher-order HM

The extension of HM to support higher-order polymorphism is surprisingly straightforward. In this section, we review the treatment of higher-order HM suggested by [17], and speculate why this idea has only recently received any significant level of interest.

In standard HM, the type of an object may include variables representing a fixed, but arbitrary type. For example, a polymorphic identity function can be defined using:

$$\begin{array}{lcl} id & :: & a \to a \\ id\ x & = & x \end{array}$$

To emphasize the fact that the variable a represents an arbitrary type, it is common to write the type of id as $\forall a.a \to a$, using explicit universal quantification.

To extend HM to support higher-order polymorphism, we need to allow the use of variables to represent, not just arbitrary types, but also type constructors, etc. Following [17], we use a system of kinds to distinguish between different forms of type constructor, given by the grammar:

$$\begin{array}{rcll} \kappa & ::= & * & \textit{the kind of all (mono)types} \\ & | & \kappa_1 \to \kappa_2 & \textit{function kinds} \end{array}$$

Intuitively, the kind $\kappa_1 \to \kappa_2$ represents constructors which take something of kind κ_1 and return something of kind κ_2. This choice of notation is motivated by Barendregt's description of generalized type systems [3].

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

$$\begin{array}{rcll} C^\kappa & ::= & \chi^\kappa & \textit{constants} \\ & | & \alpha^\kappa & \textit{variables} \\ & | & C^{\kappa' \to \kappa}\ C^{\kappa'} & \textit{applications} \end{array}$$

This corresponds very closely to the way that most type expressions are already written in Haskell. For example, *List a* is an application of the constructor constant *List* to the constructor variable a. In addition, each constructor constant has a corresponding kind. For example, writing $(\to)$ for the function space constructor and $(,)$ for pairing we have:

$$\begin{array}{lcl} \textit{Int},\ \textit{Float},\ () & :: & * \\ \textit{List} & :: & * \to * \\ (\to),\ (,) & :: & * \to * \to * \end{array}$$

The kinds of constructor applications are given by:

$$\frac{C :: \kappa' \to \kappa \qquad C' :: \kappa'}{C\ C' :: \kappa}$$

The task of checking that a type expression is well-formed can now be reformulated as the task of checking that a constructor expression has kind $*$. The apparent mismatch between the explicitly kinded constructor expressions specified above and the implicit kinding used in examples can be resolved by a process of kind inference; i.e. by using standard techniques to infer kinds for user defined constructors without the need for programmer-supplied kind annotations [17].

The set of type schemes is described by the following grammar:

$$\begin{array}{rcll} \tau & ::= & C^* & \textit{monotypes} \\ \sigma & ::= & \forall \alpha^\kappa.\sigma & \textit{polymorphic types} \\ & | & \tau & \end{array}$$

With these definitions in place, we can use the standard notation to specify the typing rules of higher-order HM in Figure 1. Note the use of the symbols τ and σ to restrict the application of certain rules to types or type schemes, respectively. Most of the rules are the same as in HM. In rule $(\forall I)$, the condition $\alpha^\kappa \notin CV(A)$ is needed to ensure that we do not universally quantify over a variable which is constrained by the type assignment A, where the expression $CV(X)$ denotes the set of all constructor variables appearing free in X.

The definition of a type inference algorithm for higher-order HM, and proof of its soundness and completeness, follow almost directly from the corresponding definitions for standard HM. The only complication is in the extension of the unification algorithm to constructors of arbitrary kind. The critical observation there is that there are no non-trivial equivalences between

$$(var) \quad \frac{(x:\sigma) \in A}{A \vdash x : \sigma}$$

$$(\rightarrow E) \quad \frac{A \vdash E : \tau' \rightarrow \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau}$$

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

$$(let) \quad \frac{A \vdash E : \sigma \quad A_x, x:\sigma \vdash F : \tau}{A \vdash (\textbf{let } x = E \textbf{ in } F) : \tau}$$

$$(\forall E) \quad \frac{A \vdash E : \forall \alpha^\kappa.\sigma \quad C \in C^\kappa}{A \vdash E : [C/\alpha^\kappa]\sigma}$$

$$(\forall I) \quad \frac{A \vdash E : \sigma \quad \alpha^\kappa \notin CV(A)}{A \vdash E : \forall \alpha^\kappa.\sigma}$$

Figure 1: Typing rules for higher-order HM

constructor expressions, and that there is no way to define arbitrary abstractions over constructor variables. This limits the power of the type system quite significantly although it does not have any real impact on the use of XHM to describe modular structure. More importantly, this restriction enables us to avoid the need for higher-order unification which would have resulted in an undecidable type system. To avoid unnecessary distraction from the subject of this paper, we refer the interested reader to [17] for further description and formalization of the topics mentioned here.

Given that the extension of HM to the higher-order case is so straightforward, we might speculate why this idea does not appear to have been explored in any detail elsewhere. One possibility is that the notation for types in some languages forces a first-order view of types. We have been fortunate that the concrete syntax for languages like Haskell encourages us to think of type expressions like $T\ a\ b$ rather than $T(a, b)$.

Another possible explanation is that, by itself, pure higher-order polymorphism seems too general for practical applications. For example, it is hard to think of any useful functions with type $\forall a.\forall m.a \rightarrow m\ a$[1]. The only possibility is the function $\lambda x.\bot$ which, apart from having almost no practical use, can be treated as having the more general type $\forall a.\forall b.a \rightarrow b$ without the need for higher-order polymorphism.

Perhaps the most interesting aspect of higher-order HM is the way that it enables us to define new datatypes with both types and constructors as parameters:

$$\begin{array}{lcl}
\textbf{data } \mathit{Rec}\ f & = & \mathit{In}\ (f\ (\mathit{Rec}\ f)) \\
\textbf{data } \mathit{ListF}\ a\ b & = & \mathit{Nil} \mid \mathit{Cons}\ a\ b \\
\textbf{type } \mathit{List}\ a & = & \mathit{Rec}\ (\mathit{ListF}\ a) \\
\textbf{data } \mathit{StateM}\ m\ s\ a & = & \mathit{STM}\ (a \rightarrow m\ (a, s))
\end{array}$$

The first three examples here can be used to provide a general framework for constructing recursive datatypes and corresponding recursion schemes [27]. The fourth example can be used to describe a parameterized state monad [18, 23].

[1] An obvious way to formalize arguments like this would be to use higher-order analogues of well-known parametricity conditions [39] or free-theorems [42].

The reader may like to check the following kinds for each of the type constructors introduced above.

$$\begin{array}{lcl} \mathit{Rec} & :: & (* \to *) \to * \\ \mathit{ListF} & :: & * \to * \to * \\ \mathit{List} & :: & * \to * \\ \mathit{StateM} & :: & (* \to *) \to * \to * \to * \end{array}$$

All of these kinds can be determined automatically without the use of kind annotations. As a final comment, it is worth noting that the implementation of this weak form of higher-order polymorphism is straightforward, and that experience with practical implementations, for example, Gofer, suggests that it is also very natural from a programmer's perspective.

3 Polymorphic values in function arguments

In this section, we describe the second ingredient of XHM: the ability to support function arguments with polymorphic components. Although this may seem to be a rather big step, we start with an example to show that it is in fact already permitted in a system with type or constructor classes where overloaded values are implicitly parameterized by dictionary structures. However, the most important part of this is the use of explicit type information, not the overloading mechanisms involved. From this observation, we move on to a more general discussion of signatures and structures in XHM and to some simple examples to illustrate its use in practice.

3.1 A motivating example using constructor classes

As far as we know, the only other example of work where HM has been extended to higher-order polymorphism is in the study of constructor classes [17] where it is combined with overloading. Mechanisms for overloading provide a form of implicit parameterization that is appropriate when the meaning of a particular symbol is uniquely determined by the types of values that are involved. For example, the treatment of a type constructor as a (category-theoretic) functor in the canonical manner is a good application. But there are also situations where the use of overloading is hard to justify. It would be hard to believe that overloading, by itself, was responsible for the success of higher-order polymorphism in the development of constructor classes.

Closer examination reveals that the real reason for the success of constructor classes is the ability to deal with (implicit) function parameters containing values of polymorphic type. To illustrate this claim, consider the following definitions[2]:

$$\begin{array}{l} \textbf{class}\ \mathit{Unit}\ m\ \textbf{where} \\ \quad \mathit{unit} \quad :: \quad a \to m\ a \\ \\ \mathit{twice} \quad :: \quad \mathit{Unit}\ m \Rightarrow a \to m\ (m\ a) \\ \mathit{twice}\ x \quad = \quad \mathit{unit}\ (\mathit{unit}\ x) \end{array}$$

Note that the definition of *twice* would not be well-typed if *unit* was passed as an explicit parameter as in:

$$\mathit{twice}\ \mathit{unit}\ x \quad = \quad \mathit{unit}\ (\mathit{unit}\ x)$$

In this case, the only possible type for *twice* would be $\forall a.(a \to a) \to (a \to a)$. However, using the dictionary based techniques of Wadler and Blott [43], the constructor class program above

[2]The `Unit` class defined her can be thought of as a cut-down version of the `Monad` class used elsewhere [17, 23] to support the use of monads in functional programming.

is implemented by translating it to obtain the following definitions:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Unit}\ m & = & \{\, \mathit{unit} :: \forall a.a \to m\ a \,\} \\ \\ \mathit{twice} & :: & \mathit{Unit}\ m \to a \to m\ a \\ \mathit{twice}\ u\ x & = & u.\mathit{unit}\ (u.\mathit{unit}\ x) \end{array}$$

We use the notation $\{\ldots\}$ to represent a record of named components and the familiar notation $r.l$ to denote the extraction of a field l from a record r. The important point in this definition is that the *unit* component of the u record that is passed as an argument to *twice* has a polymorphic type. By instantiating the quantified type variable to a for the inner call of *unit* and to $m\ a$ for the outer call, we obtain the required type. What makes this work is the fact that the *unit* function is declared as a global function with a polymorphic type that allows the type variable a in the example above to be instantiated to different types. It is this additional type information, not the use of overloading, that is important for the work described in this paper. The approach suggested by Wadler and Blott is to translate programs with implicit overloading into a language that makes the use of dictionary structures explicit. In essence, one of the main claims of this paper is that the target of this translation for the system of constructor classes is a useful and powerful language in its own right.

3.2 Records, signatures and structures

To complete the formal description of XHM, we extend the higher-order HM of the previous section to include record types (i.e. additional elements of C^*) of the form:

$$\{\, x_1 :: \sigma_1; \ldots; x_n :: \sigma_n \,\}$$

For convenience, we will often abbreviate record types like this using expressions of the form $\{\, x_i :: \sigma_i \,\}$, where i ranges over some implicit set of indices. We will refer to record types of this form as signatures.

Unlike signatures in the SML module system, signatures in XHM may contain free variables. For example, the variable m appears free in the signature $\mathit{Unit}\ m$ above. Alternatively, we can think of *Unit* as a parameterized signature; in fact, for the purposes of kind inference, *Unit* is treated as having kind $(* \to *) \to *$. On the other hand, SML also permits the declaration of type constructors in signatures, with corresponding definitions in matching structures[3]. We discuss the tradeoffs between signature parameters in XHM and type components in SML more fully below in Section 4.

XHM signatures are also very much like records in SML, with one important difference: the components of an SML record cannot have a polymorphic type. In this way, XHM can be viewed as an attempt to unify the module and record languages of SML. It is also possible that the XHM system can be further extended to support fully polymorphic extensible modules using the approaches suggested by [16, 35], for example. We leave this as a topic for future research. A previous attempt to explain the ML module system using record types has been presented by Aponte [1], but does not make any use of a higher-order type system and differs substantially from the approach described in this paper.

[3]For completeness, we should mention that SML structures may also include definitions of exceptions; we will not address the use of exceptions in this paper.

Structure values will be written using the syntax:

$$\begin{array}{lcl}\textbf{struct} & & \\ \quad x_1 & = & \langle\text{implementation of } x_1\rangle \\ & \vdots & \\ \quad x_n & = & \langle\text{implementation of } x_n\rangle\end{array}$$

and will also be abbreviated using expressions of the form $\textbf{struct}\ \{\, x_i = E_i \,\}$, again with i ranging over some implicit set of indices. We have chosen this particular syntax because it fits well with the Haskell layout rule in the prototype implementation; in fact, as the examples below illustrate, we actually allow a more liberal syntax that allows the use of function arguments and pattern matching on the left hand side of equations.

The typing rules for structures are unlikely to cause many surprises. First the rule for constructing structures:

$$\frac{A \vdash E_i : \sigma_i}{A \vdash \textbf{struct}\ \{\, x_i = E_i \,\} : \{\, x_i :: \sigma_i \,\}}$$

It is useful to think of structure expressions as being like local definitions that bind several variables, as in $\textbf{let}\ \{\, x_i = E_i \,\}\ \textbf{in}\ F$ for example, except that the expression over which the bindings are scoped is omitted and the result is the value of the bindings themselves, preserved as a structure value.

A second rule is needed to describe the typing of a component that has been extracted from a structure value. This is also straightforward:

$$\frac{A \vdash E : \{\, x_i :: \sigma_i \,\}}{A \vdash E.x_j : \sigma_j}$$

An alternative, but equivalent, approach is to think of selectors as functions:

$$(\_.x_j) \quad :: \quad \forall v.\{\, x_i :: \sigma_i \,\} \to \sigma_j$$

where v represents the free variables, or parameters, of the signature $\{\, x_i :: \sigma_i \,\}$. If we think of signatures as abstract datatypes, for example, forgetting the definition of *Unit* as a signature type in the previous section and thinking of $Unit\ m$ as some primitive type, then we can write the type of selectors in a more concrete form:

$$(\_.unit) \quad :: \quad \forall m.Unit\ m \to (\forall a.a \to m\ a)$$

Since a does not appear free in the constructor expression $Unit\ m$, we can move the quantifier for a outwards to obtain an equivalent higher-order HM type for the selector:

$$(\_.unit) \quad :: \quad \forall m.\forall a.Unit\ m \to a \to m\ a$$

This gives a strong hint to the implementation of XHM type checking, and also to the proof that XHM programs are sound and have principal type schemes. All that is necessary is sufficient explicit type information to determine which selector type is appropriate in any particular context.

3.3 The use of explicit type information

The need for explicit type information in programs using records will already be familiar to SML programmers; the definition of SML requires that the shape of any record—i.e. a complete

list of its fields—can be determined at compile-time. What we have described here is a natural generalization of this; we require not only the names of all of the fields, but also the types for every field that is referenced. It is fairly easy to arrange for this mechanism to reduce to the SML treatment of records when a polymorphic type has not been specified.

Type annotations are not necessary in many simple examples. For example, the following program type checks without any additional type information:

$$\begin{array}{lcl} f\ x & = & \textbf{struct} \\ & & \quad h\ z = [z] \\ & & \quad u = x \\ v\ y & = & m.h\ (m.h\ m.u) \\ & & \textbf{where}\ m = f\ y \end{array}$$

In this case, we can calculate the following types for the components in the structure value in the definition of f:

$$\begin{array}{lcl} h & :: & \forall t.t \to [t] \\ u & :: & a \end{array}$$

where a is the type of the argument x. Thus:

$$f \quad :: \quad \forall a.a \to \{\, h :: \forall t.t \to [t];\ u :: a \,\}$$

and it follows that v has type $\forall a.a \to [[a]]$.

In practice, explicit type annotations are typically only required for the definition of recursive or mutually recursive structures (which are not permitted by the SML module system) or for functions that manipulate structure values (corresponding to SML functor definitions where explicit type information is also required in SML, or to higher-order or first-class modules which are not supported by SML). For example, the type signature accompanying the following definition cannot be omitted[4]:

$$\begin{array}{lcl} \mathit{makeUnit} & :: & a \to \mathit{Unit}\ m \to m\ a \\ \mathit{makeUnit}\ x\ u & = & u.\mathit{unit}\ x \end{array}$$

On the other hand, we are free to store values of some type $\mathit{Unit}\ m$ in data structures such as lists and to use many higher order functions, for example $\lambda z.\mathit{map}\ (\mathit{makeUnit}\ z)$, without further type annotations.

Some may question the need for explicit type information in a language that is based on a system like HM, but we do not expect that this will have any significant impact on programmers:

- Some form of explicit type information is already necessary in many languages based on HM. For example, this includes the overloading mechanisms of Haskell and the treatment of records, arithmetic, and structures in SML.
- Explicit type annotations are only required in situations where they would already be required by programs using the SML module system, or in programs that cannot be written with SML modules.
- Despite the fact that it is not necessary, the use of type annotations in implicitly typed languages like ML and Haskell is widely recognized as 'good programming style', and many programmers already routinely include type declarations in their source code. The type assigned to a value serves as a useful form of program documentation. In addition, this gives a simple way to check that the programmer-supplied type signatures, reflecting intentions about the way an object will be used, are consistent with the types obtained by type inference.

[4]Unless some alternative method is used to specify the type of u!

3.4 Simple applications of XHM structures

The examples given above have already demonstrated the use of XHM structures as first-class values. In this section, we give some examples to illustrate other applications of these ideas.

One of the standard examples in texts on abstract datatypes is the definition of a type *cpx* of complex numbers together with a collection of operations for manipulating values of this type. It is often convenient to package these operations together, as described by the signature:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Complex}\ \mathit{cpx} & = & \{\, \mathit{mkCart}, \mathit{mkPolar} & :: \quad \mathit{Float} \to \mathit{Float} \to \mathit{cpx}; \\ & & \mathit{re},\ \mathit{im} & :: \quad \mathit{cpx} \to \mathit{Float}; \\ & & \mathit{mag},\ \mathit{phase} & :: \quad \mathit{cpx} \to \mathit{Float}; \ldots \,\} \end{array}$$

There are two obvious ways to implement complex numbers using pairs of floating point numbers and choosing either a cartesian or polar representation:

$$\begin{array}{lcll} \mathit{rectCpx} & = & \textbf{struct} & \\ & & \mathit{mkCart}\ x\ y & = \quad (x, y) \\ & & \mathit{mkPolar}\ r\ \theta & = \quad (r\cos\theta, r\sin\theta) \\ & & \mathit{re}\ (x,\ y) & = \quad x \\ & & \ldots & \end{array}$$

$$\begin{array}{lcll} \mathit{polarCpx} & = & \textbf{struct} & \\ & & \mathit{mkCart}\ x\ y & = \quad (\mathit{sqrt}\ (x^2 + y^2), \mathit{atan2}\ y\ x) \\ & & \mathit{mkPolar}\ r\ \theta & = \quad (r, \theta) \\ & & \mathit{re}\ (r,\ \theta) & = \quad r\cos\theta \\ & & \ldots & \end{array}$$

As it stands, the implementation type of both complex number packages is the same and is captured explicitly in the type of each; $\mathit{Complex}\ (\mathit{Float}, \mathit{Float})$. Later, in Section 4.3, we will show how to make these types abstract, either completely by using an existential, or partially by giving them names without revealing how they are implemented.

In fact, polymorphism already provides one form of abstraction. To illustrate this, suppose that we want to define a package of complex number arithmetic. We start with a signature for a general arithmetic package, considerably simplified for the purposes of this paper:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Arith}\ a & = & \{\, \mathit{plus} & :: \quad a \to a \to a; \\ & & \mathit{neg} & :: \quad a \to a; \ldots \,\} \end{array}$$

Now we can describe the construction of a complex arithmetic package from an arbitrary complex number package using the following function:

$$\begin{array}{lcl} \mathit{compArith} & :: & \mathit{Complex}\ t \to \mathit{Arith}\ t \\ \mathit{compArith}\ c & = & \textbf{struct} \\ & & \quad \mathit{plus}\ z_1\ z_2 \quad = \quad c.\mathit{mkCart}\ (c.\mathit{re}\ z_1 + c.\mathit{re}\ z_2)\ (c.\mathit{im}\ z_1 + c.\mathit{im}\ z_2) \\ & & \quad \ldots \end{array}$$

Since the same variable, t, appears as a parameter to both the the *Complex* and *Arith* signatures, it is clear that the type of values that the arithmetic operations in the result can be applied to is the same as the type of complex numbers provided as an argument. At the same time, the fact that t is universally quantified ensures that the definition of *compArith* cannot make any assumptions about the implementation of complex numbers.

The definition of *compArith* corresponds to a functor in SML, but there is no need for a special syntax for functor or structure definitions in XHM; the examples above use only the features of the core language which just happen to involve structure values. The use of the same type variable, t, in both argument and result signatures corresponds to a sharing specification[5], often considered one of the most difficult aspects of the SML module system [36]. In this setting, sharing seems more familiar, simply identifying two types by using the same name for each.

For some applications, it is necessary to use type constructors rather than simple types as signature parameters. For example, a specification of queue datatypes, similar in spirit to those in Paulson's textbook [36], might be given by the following signature:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Queue}\ q & = & \{\, \mathit{empty} & :: \quad q\ a; \\ & & \mathit{enq} & :: \quad q\ a \to a \to q\ a; \\ & & \mathit{null} & :: \quad q\ a \to \mathit{Bool}; \\ & & \mathit{hd} & :: \quad q\ a \to a; \\ & & \mathit{deq} & :: \quad q\ a \to q\ a\,\} \end{array}$$

The variable q used here ranges over unary type constructors that correspond to functions mapping types to types, i.e. over constructors of kind $* \to *$. We will also treat the variable a appearing free in the type of each of the queue operators above as if it had been bound by an explicit universal quantifier. For example, the type of the *empty* component in a structure of type $\mathit{Queue}\ q$ is $\forall a.q\ a$.

The following structure describes the standard implementation of queues using lists:

$$\begin{array}{lcll} \mathit{listQueue} & :: & \mathit{Queue\ List} & \\ \mathit{listQueue} & = & \textbf{struct} & \\ & & \quad \mathit{empty} & = \quad \mathit{Nil} \\ & & \quad \mathit{enq}\ n\ q & = \quad q{+\!\!+}[n] \\ & & \quad \mathit{null}\ \mathit{Nil} & = \quad \mathit{True} \\ & & \quad \mathit{null}\ (\mathit{Cons}\ n\ ns) & = \quad \mathit{False} \\ & & \quad \mathit{hd}\ (\mathit{Cons}\ n\ ns) & = \quad n \\ & & \quad \mathit{deq}\ (\mathit{Cons}\ n\ ns) & = \quad ns \end{array}$$

We hope that the reader will have already recognized that the type declaration here is provided for documentation only, and is not required to determine the type of *listQueue*.

4 Structures and signatures for modular programming?

One of the main goals of this section is to address the question of whether XHM can be used as the basis for a module system with signatures and structures corresponding to interfaces and implementations, respectively. The term 'module system' is already widely used in several different ways with meanings including:

- A mechanism to support separate compilation and namespace management.
- A mechanism to enable the decomposition of large programs into small, reusable units in a way that is resistant to small changes in the program.
- A mechanism for defining abstractions.

[5]SML sharing specifications can also be used to specify equalities between structure values. We do not attempt to deal with this in XHM.

In this and following sections, we argue that XHM is consistent with such goals: Separate compilation is ensured by maintaining a clear separation between types and values and program decomposition is supported by the use of higher-order and nested polymorphism. Powerful abstractions can be defined using parameterized structures. On the other hand, XHM does not by itself allow the definition of abstract datatypes, although this can be dealt with using other methods.

We pay particularly close attention to comparisons with the SML module system. The most significant difference between these two systems is the fact that XHM uses parameterized signatures while SML allows type components in signature and structure definitions. It is certainly true that the inclusion of type components can be a very powerful tool; indeed, without careful restrictions, this may prove too powerful, leading to intractability and undecidable type checking. However, we argue that much can be accomplished *without* type components. This, we believe, is also more in the spirit of HM than systems that include or manipulate type components in modules. For example, in Milner's original work [28], types are used as a purely semantic notion, representing subsets of a semantic domain, not as any concrete form of value.

4.1 Lifting type definitions

We start with an important observation, that type definitions in a module can be lifted to the top-level, which helps to explain why it is possible to omit type components from modular structures. For example, consider the following SML fragment:

```
structure s
  = struct
      type T       =  Int
      data List a  =  Nil | Cons a (List a)
      ...
    end
```

Despite appearances, the type synonym T and the type constructor *List* are *not* local to the definition of s. At any point in the program where s is in scope, these type constructors can be accessed by the names $s.T$ and $s.List$, respectively. Renaming any references to these types and their constructors in the body of s, we can lift these definitions to the top-level, to obtain the following XHM definitions:

```
type s.T       =  Int
data s.List a  =  s.Nil | s.Cons a (s.List a)
s              =  struct
                    ...
```

In effect, all that the datatype definitions in the original SML program accomplish is to define new top-level datatypes in which the type and value constructor names are decorated with the name of the enclosing structure.

In some situations, renaming is not sufficient to allow type definitions to be lifted to the top-level. For example, the *List* datatype in the following functor definition involves a 'free variable'—the type $x.T$, a component of the argument structure x:

```
functor f(x:SIG) : SIG'
  = struct
      data List  =  Nil | Cons x.T List
      ...
    end
```

The solution in this case is to add an extra parameter to the datatype definition before moving it to the top-level, as shown on the right. This is just a form of λ-lifting [15, 37]:

$$\begin{array}{lcl} \textbf{data}\ f.List\ t & = & f.Nil \\ & | & f.Cons\ t\ f.List \\ f & :: & SIG\ t \to SIG'\ (f.List\ t) \\ f\ x & = & \ldots \end{array}$$

Notice how the parameterized signatures in the type for f capture the relationship between the types involved in the argument x and those involved in the result $f\ x$.

It is also important to mention that, since the form of higher-order polymorphism described in Section 2 allows type constructors to be used as both signature and datatype parameters, the same technique can be used to deal with type constructor components of functor arguments.

4.2 Generativity

Readers with experience of the SML module system will realize that the simple lifting of the datatype f in the functor definition at the end of the previous section is not quite correct. The reason for this is that SML adopts a notion of generativity, producing a new type constructor each time the functor is applied to an argument. Thus two definitions:

$$\begin{array}{lcl} \textbf{structure}\ s_1 & = & f(x) \\ \textbf{structure}\ s_2 & = & f(x) \end{array}$$

will produce structures with incomparable type components. In truth, when we use an SML functor to 'generate' a new datatype, we are in fact constructing a new instance of a fixed datatype, which is then hidden, in essence, by a form of existential quantification. There is no way to express the *List* type produced by applying f to an appropriate argument structure in the notation of SML, so we are forced to package up instantiation of the actual implementation type, $f.List$, and hiding of the resulting type as a single operation.

Lifting type definitions to the top-level allows us to express the type components of the result of functor applications; for the example above, both s_1 and s_2 have type component $f.List\ t$, assuming that x has type $SIG\ t$. We are then free to treat the question of whether we wish to conceal these implementation types as a separate concern. Various methods for achieving this are described below in Section 4.3. Note that higher-order polymorphism is essential, for the general case, to express type components that depend on type constructors of higher kinds in functor arguments.

We see then that, in XHM, there is no need for the same notions of generativity used in SML. Of course, this only affects the static properties of the system. We would normally expect the definitions above to produce two distinct copies of the same structure; these may not be semantically equivalent in a language with side effects, for example, if the generated structures include state components.

4.3 Abstraction

Abstraction—the ability to hide information about the implementation of a module and protect against misuse—is an important feature of module systems that has been mentioned only briefly in the discussion above. The system described in this paper allows us to distinguish between two different forms of abstraction:

- The independence of the implementation of a module from the implementation of its imports.
- The ability to hide details about the type or type constructor parameters in the signature of a structure.

The *compArith* example in Section 3.4 has already illustrated how the first of these can be expressed very elegantly in XHM using polymorphism,

For the second, we can identify at least two different levels of hiding that may be useful. One alternative is to provide a name for a datatype, but to conceal the details of its implementation, such as the constructor functions of an algebraic datatype or the expansion of a type synonym. This is just a matter of scoping, and does not require any changes to the underlying type theory; indeed, it is perhaps best dealt with at the level of compilation units rather than in the core language itself.

For example, we might use a construct:

$$\begin{array}{l}\textbf{export } \mathit{Cpx}, \mathit{cpx} \textbf{ from} \\ \quad \begin{array}{lcl}\textbf{type } \mathit{Cpx} & = & (\mathit{Float}, \mathit{Float}) \\ \mathit{cpx} & :: & \mathit{Complex}\ \mathit{Cpx} \\ \mathit{cpx} & = & \ldots\end{array}\end{array}$$

The intention here is that the definition of the *cpx* structure and the name of the *Cpx* type are exported, but that the implementation of *Cpx* is not. Outside this definition, a programmer can be sure that *compArith cpx* has type *Arith Cpx*, and that there is no danger of confusing this with any other complex number package that happens to use pairs of floating point numbers as a representation of complex numbers. Several languages support similar constructs, either through explicit namespace management primitives (as in Haskell) or more specific features such as the **abstype** construct in SML. While this approach can be quite useful, it is limited to top-level definitions. For example, we cannot simulate the behaviour of SML functor application outlined above where a new type is generated each time the functor is applied.

A second alternative is to use existential types, concealing not just the implementation, but also the name of an abstract type. We strongly believe that XHM can be extended in a modular fashion to support existential types, for example using dot-notation [5], and the combination of type inference and existential typing that has been explored by Läufer [20]. Note that this can be used to simulate the effects of datatype generativity; for example, if f is assigned a type of the form $\mathit{SIG}\ t \to \exists t'.\mathit{SIG}'\ t'$, then the definitions of structures s_1 and s_2 in the previous section will produce distinct types $s_1.t'$ and $s_2.t'$.

We believe that both of these approaches are useful in their own right. However, neither coincides exactly with the form of abstract datatypes provided by SML; the latter falls somewhere between the two extremes of named and existentially quantified abstract datatypes. It remains as an interesting problem to determine whether there is a modular extension of XHM which provides the same form of abstraction as SML.

4.4 Polymorphic modules

In recent work, Kahrs [19] has shown that it is not possible to define certain kinds of polymorphic module in SML. Fortunately, this problem does not occur with XHM modules. In fact, a direct translation of Kahrs' SML code to our framework already provides the desired form of polymorphism. For reasons of space, we will restrict ourselves to a rather shorter example using

the following SML signature:

$$\begin{array}{l}\textbf{signature } I \\ = \textbf{ sig} \\ \quad \textbf{type } T \\ \quad id \quad :: \quad T \to T \\ \textbf{end}\end{array}$$

The problem identified by Kahrs is caused by the fact that the type component in any structure matching I must be fixed to some specific type. In particular, it cannot be a variable, and hence it is impossible to define a structure s that matches I such that $s.id$ is the polymorphic identity function.

The corresponding definition in XHM is:

$$\textbf{type } I\ t \quad = \quad \{\, id :: t \to t \,\}$$

But in this case, we can define a structure:

$$\begin{array}{lll} s & = & \textbf{struct} \\ & & \quad id\ x \quad = \quad x \end{array}$$

which has type $\forall t.I\ t$ and hence $s.id$ can be used as a polymorphic identity function. We refer the reader to [19] for examples of more useful applications of this form of polymorphism.

4.5 Practical concerns

Practical experience with the SML module system suggests a number of useful features for module system designs and we will take the opportunity for a brief discussion of some of these ideas here.

First, we may be concerned that, in realistic applications, the number of type components in a module may become quite large and that the corresponding parameterized signature would become rather cumbersome and awkward. The easiest way to overcome this problem is to extend the languages of constructors and kinds in the Higher-order HM system to include records (i.e. labelled products) of the form:

$$(t_1 = C^{\kappa_1}, \ldots, t_n = C^{\kappa n}) :: (t_1 :: \kappa_1, \ldots, t_n :: \kappa_n)$$

and to allow constructor variables ranging over such kinds. This gives the flexibility to package several constructors into a single signature parameter. Apart from reducing the number of parameters that have to be written, this also makes source code more resistant to change if, for example, a new type parameter is added. Type sharing constraints can also be described quite nicely in this framework. One possibility is to use a form of qualified types:

$$\mathit{someprogram} :: (r.x = s.y) \Rightarrow \mathit{SIG}\ r \to \mathit{SIG}'\ s$$

to indicate that the x and y fields of r and s, respectively, are equal. Alternatively, we might adopt a notation based on row variables to express the same constraints:

$$\mathit{someprogram} :: \mathit{SIG}\ (r \mid x = a) \to \mathit{SIG}'\ (s \mid y = a).$$

Subsumption is another useful feature of the SML module system which allows unwanted components of functor arguments to be ignored. Explicit type information is used to support this. We believe that much the same techniques would also be applicable here, although we have not yet attempted to establish this formally. Another interesting idea is to extend work on extensible

records to provide a collection of operators on structures, for example, using $r \setminus l$ to specify the structure obtained by removing the l component from r, or $(r \mid x = v)$ to describe the extension of a structure r with a new component x. This would provide an explicit alternative to the implicit treatment of subsumption.

Finally, it is clear from our earlier discussions that XHM structures have much in common with type classes in languages like Haskell. However, further work is needed to obtain a language design that combines these two features in an elegant and orthogonal manner.

5 Type-theoretic background

To set the contributions of this paper in perspective, this section reviews some of the previous attempts to provide a type-theoretic foundation for modular programming, concentrating in particular, on two of the the most important features of such systems: abstraction and separate compilation.

5.1 Existential typing and abstraction

One way to formalize the process of hiding the implementation of an abstract datatype is to use an existential type [33, 6]:

$$\textbf{type}\ \mathit{Complex}' \quad = \quad \exists \mathit{cpx}.\mathit{Complex}\ \mathit{cpx}.$$

Informally, the existential typing indicates that there is a type *cpx* with the operations listed above defined on it, but prohibits the programmer from making any assumptions about the implementation type. Formally, the properties of existentials are described by the following typing rules, based on standard rules for existential quantifiers in logic:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t.\tau)}$$

This is often described as the introduction rule. Note that the implementation type τ' for the abstract type is discarded and does not appear anywhere in the conclusion.

At the same time, the requirement that N has a polymorphic type in the following elimination rule ensures that we do not make any assumptions about the now hidden implementation type since the definition behaves uniformly for all choices of t:

$$\frac{\Gamma \vdash M : \exists t.\tau \quad \Gamma \vdash N : \forall t.\tau \to \tau' \quad t \notin TV(\tau')}{\Gamma \vdash \textbf{open}\ M\ \textbf{in}\ N : \tau'}$$

Existential types completely hide the identity of implementation types. For example, the types *cpx* and *cpx*′ in the body of the following expression cannot be identified, even though they both come from the same term c of type *Complex*:

$$\begin{array}{l}\textbf{open}\ c\ \textbf{in}\ \Lambda \mathit{cpx}.\lambda x : \mathit{Complex}\ \mathit{cpx}. \\ \quad \textbf{open}\ c\ \textbf{in}\ \Lambda \mathit{cpx}'.\lambda y : \mathit{Complex}\ \mathit{cpx}'. \\ \quad\quad \ldots\end{array}$$

To emphasize this behaviour, suppose that we define an abstract data type of arithmetic operations, and attempt to reconstruct the *compArith* function from Section 3.4:

$$\begin{array}{lcl} \textbf{type } \mathit{Arith}' & = & \exists a.\mathit{Arith}\ a \\ \\ \mathit{compArith} & :: & \mathit{Complex}' \to \mathit{Arith}' \\ \mathit{compArith}\ c & = & \textbf{open } c \textbf{ in } \Lambda \mathit{cpx}.\lambda p : \mathit{ComplexOps}\ \mathit{cpx}. \\ & & \quad \{\mathit{add}{=}\ldots\} \end{array}$$

Given an implementation c of type *Complex′*, we can use the expression *compArith* c to obtain a package for arithmetic on complex numbers. However, this has no practical use because the typing rules for existentials make it impossible to construct any values to which the *add* and *neg* functions of the resulting package can be applied! The type system does not capture the equivalence of the type of complex numbers used in c with the type of values that can be manipulated by *compArith* c.

An alternative approach to existential typing, using *dot notation* in place of the **open** construct described above, has been investigated by Cardelli and Leroy [5]. The dot notation allows us to identify the implementation types of two packages if they have the 'same name'. This avoids the first problem illustrated above, but not the second. Dot notation is also limited by the unavoidably conservative notions of 'same name' that are needed to ensure decidability of type checking, and is not very well-behaved under simple program transformations.

5.2 Dependent types and separate compilation

Motivated by problems with existential types, similar to those described above, MacQueen [24] argued that dependent types provide a better basis for modular programming. In this framework, structures are represented by pairs $\langle \tau, M \rangle$ containing both a type component τ and a term M whose type may depend on the choice of τ. Structures of this form can be treated as elements of a dependent sum type, described informally by:

$$\Sigma t.f(t) = \{\, \langle \tau, M \rangle \mid M \text{ has type } f(\tau) \,\}.$$

The typing rules for dependent sums are standard (see [26], for example) and can be written in the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash \langle \tau', M \rangle : (\Sigma t.\tau)} \qquad \frac{\Gamma \vdash M : (\Sigma t.f(t))}{\Gamma \vdash \mathit{snd}\ M : f(\mathit{fst}\ M)}$$

The introduction rule on the left is very similar to the corresponding rule for existentials except that the implementation type, τ', is captured in the structure $\langle \tau', M \rangle$ in the conclusion. The elimination rule on the right indicates that, if M is a structure of type $\Sigma t.f(t)$, then the second component, *snd* M, of M has type $f(\mathit{fst}\ M)$, where *fst* M is the first component of M.

In this setting, the **open** M **in** N construct used in the treatment of existentials can be replaced by the term N (*fst* M) (*snd* M), but this is not *abstraction preserving* in the sense of Mitchell [32]. Informally, dependent sums are more powerful than existentials because of the ability to name the type component *fst* M of a structure M. Of course, this also means that the type component of a structure is no longer abstract.

In a sense, a simple treatment of modules using dependent types is actually *too* powerful for practical systems because it interferes with separate compilation. More precisely, it makes it more difficult to separate compile-time type-checking from run-time evaluation. To illustrate this, we recast the previous definitions of complex number and arithmetic types using dependent

sums to obtain:

$$\begin{array}{lcl} \textbf{type } \mathit{Complex}' & = & \Sigma \mathit{cpx}.\mathit{Complex}\ \mathit{cpx} \\ \textbf{type } \mathit{Arith}' & = & \Sigma a.\mathit{Arith}\ a \\ \\ \mathit{compArith} & :: & \mathit{Complex}' \to \mathit{Arith}' \\ \mathit{compArith}\ c & = & \langle \mathit{fst}\ c, \{\ \mathit{add} = \ldots\ \} \rangle \end{array}$$

At first glance, this definition suffers from the same problems as the previous version using existentials; the type $\mathit{Complex}' \to \mathit{Arith}'$ does not reflect the fact that the type components of the argument and result structures are the same. However, this information can be obtained by carrying out a limited degree of evaluation during type checking. For example, if $c = \langle \tau, M \rangle$, then:

$$\mathit{fst}\ (\mathit{compArith}\ c) = \mathit{fst}\ \langle \tau, \{\ \mathit{add} = \ldots\ \} \rangle = \tau.$$

To ensure that static type checking is possible, it is important to distinguish compile-time evaluation of this kind from arbitrary run-time execution of a program. Unfortunately, treating a functor as a function of type $(\Sigma s.f(s)) \to (\Sigma t.g(t))$ does not reflect this separation; in general, a type of this form may include elements in which the type component of the result depends on the value component of the argument. As an alternative, Harper, Mitchell and Moggi [12, 34] have shown that a suitable *phase distinction* can be established by modelling functors from $\Sigma s.f(s)$ to $\Sigma t.g(t)$ as values of type $\Sigma h.(\forall s.f(s) \to g(h(s)))$ where h ranges over functions from types to types, corresponding to the compile-time part of the functor.

As the example above shows, it is sometimes necessary to inspect the implementation of a structure to find the value of a type component. Not surprisingly, this means that it is not possible to provide true separate compilation for SML [2]. Even the 'smartest recompilation' scheme proposed by Shao and Appel [40] does not permit true separate compilation because it delays some type checking, and hence the detection of some type errors, to link-time.

The need for type sharing constraints in functor definitions is, in fact, motivated by similar problems; since it is impossible to evaluate the formal parameters of a functor, we must instead supply the required identities between type components using explicit sharing equations. Further extensions to the basic theory of dependent types are needed to deal with this, and other ideas including generativity, polymorphism, abstraction, higher-order modules, and modules as first-class values.

5.3 Translucent sums and manifest types

Recent proposals for *translucent sums* by Harper and Lillibridge [11] and *manifest types* by Leroy [21] provide a compromise between existential typing and dependent sums, allowing the programmer to include additional type information in the signature for a structure. For these systems, we use an introduction rule of the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t = \tau'.\tau)}$$

Notice that, unlike the previous cases, the implementation type τ' appears in the inferred type although this can be hidden by coercing it to a standard existential type:

$$\frac{\Gamma \vdash M : (\exists t = \tau'.\tau)}{\Gamma \vdash M : (\exists t.\tau)}$$

These systems provide better support for abstraction and separate compilation than Standard ML. However, the underlying theories are quite complex and unfamiliar, relying on the use of dependent types.

5.4 Comparison with XHM

For all of the examples described above, the type of a module, package or structure is given by an expression of the form $\mathrm{Q}t.f(t)$ for some parameterized record type $f(t)$ and some quantifier Q. Most of the problems described above are caused by the fact that these quantifiers can be 'overly protective', limiting the ability to propagate type information. In the previous sections of this paper, we have shown how to use record types of the form $f(t)$ as signatures, treating quantifiers as a separate issue.

6 Conclusion

We have presented an extension of the Hindley-Milner type system that provides:

- Support for higher-order polymorphism.
- Support for function arguments with polymorphic components, sometimes requiring explicit type information to guide the main type inference process.
- A clear separation between static and dynamic semantics.

This leads to a module system in which:

- Structures are first-class values.
- Higher-order modules (i.e. first-class functors) are admitted. Leroy [21] gives several examples of higher-order modules, some of which can only by typed using his system of manifest types, while others require different higher-order extensions of SML [25]. All of these examples, including those in [25], can be typed in XHM.
- Polymorphic modules and structures may be defined.
- True separate compilation is possible: the interface to a separately compiled structure is completely determined by its signature, so there is no need to inspect its implementation.
- Since the module language is based on HM, we believe that it will be easy to use, particularly for programmers that are already familiar with the core languages of Standard ML, Haskell or similar languages.
- We avoid the need for some of the more complicated aspects of the Standard ML type system such as generativity and sharing.

By contrast, none of these is possible with the SML module system without, often substantial, modification. One of the main topics for future research is to investigate the role of implicit subsumption; i.e. the ability to discard elements from a structure as a result of signature matching in SML. We believe that this can be accomplished using a simple form of subtyping, guided by type annotations, or otherwise by extending XHM with an mechanism for controlling the set of bindings that are exported from a structure.

We are currently working on a practical implementation of the ideas presented in this paper for a Haskell-like language. The main goal of this work is to provide the functionality of SML-style modules while preserving the basic character of Haskell, including type classes, side-effect free, call-by-name/lazy semantics, implicit mutual recursion, and so forth. We intend to use our prototype to investigate the use of XHM modules in realistic applications, and to assess how well the ideas described in this paper fare in a practical module system.

Acknowledgements

Many of the ideas presented in this paper were developed while the author was a member of the Department of Computer Science, Yale University, supported in part by a grant from ARPA, contract number N00014-91-J-4043.
My thanks to Paul Hudak, Sheng Liang and, in particular, Dan Rabin and Xavier Leroy for their valuable comments and suggestions during the development of the ideas presented in this paper.

References

[1] María Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993.

[2] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[3] H. Barendregt. Introduction to generalised type systems. *Journal of functional programming*, 1, April 1991.

[4] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM conference on Lisp and Functional Programming*, Orlando, FL, June 1994.

[5] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report report 56, DEC SRC, 1990.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[7] Pierre Crégut and David MacQueen. An implementation of higher-order functors. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, FL, June 1994.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, N.M., January 1982.

[9] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie de types. In Fenstad, editor, *Proceedings of the Scandanavian logic symposium*. North Holland, 1971.

[10] Jean-Yves Girard. The system F of variable types, 15 years later. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, chapter 7, pages 87–126. Addison Wesley, 1990.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[12] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[13] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[14] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[15] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture*, pages 190–205, New York, 1985. Springer-Verlag. Lecture Notes in Computer Science, 201.

[16] Mark P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[17] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

[18] M.P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA, December 1993.

[19] Stefan Kahrs. First-class polymorphism for ml. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.

[20] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.

[21] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[22] Xavier Leroy. A syntactic theory of type generativity and sharing. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, FL, June 1994.

[23] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.

[24] David MacQueen. Using dependent types to express modular structure. In *13th Annual ACM Symposium on Principles of Programming languages*, pages 277–286, St. Petersburg Beach, F.L., January 1986.

[25] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.

[26] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*. North Holland, Amsterdam, 1982.

[27] Erik Meijer and Mark P. Jones. Gofer goes bananas. In preparation, 1994.

[28] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[29] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[30] John Mitchell and Robert Harper. The essence of ml. In *Fiftheenth ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1988.

[31] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[32] John C. Mitchell. On abstraction and the expressive power of programming languages. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, New York, September 1991. Springer-Verlag. Lecture Notes in Computer Science, 526.

[33] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[34] Eugenio Moggi. A category-theoretic account of program modules. In *Summer conference on category theory and computer science*, pages 101–117, New York, 1989. Springer-Verlag. Lecture Notes in Computer Science, 389.

[35] Atsushi Ohori. A compilation method for ml-style polymorphic record calculi. In *Proceedings 19th Symposium on Principles of Programming Languages*. ACM, January 1992.

[36] L.C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[37] S.L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, 1987.

[38] J.C. Reynolds. Towards a theory of type structure. In *Paris colloquium on programming*, New York, 1974. Springer-Verlag. Lecture Notes in Computer Science, 19.

[39] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, Amsterdam, 1983. North-Holland.

[40] Z. Shao and A. Appel. Smartest recompilation. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993.

[41] Mads Tofte. Principal signatures for higher-order program modules. In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, January 1992.

[42] P. Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89*, Imperial College, London, September 1989.

[43] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.