A Second Look at Overloading*

Martin Odersky
Universität Karlsruhe†
(odersky@ira.uka.de)

Philip Wadler
University of Glasgow‡
(wadler@dcs.glasgow.ac.uk)

Martin Wehr
Universität Karlsruhe
(wehr@ira.uka.de)

Abstract

We study a minimal extension of the Hindley/Milner system that supports overloading and polymorphic records. We show that the type system is sound with respect to a standard untyped compositional semantics. We also show that every typable term in this system has a principal type and give an algorithm to reconstruct that type.

1 Introduction

Arithmetic, equality, showing a value as a string: three operations guaranteed to give a language designer nightmares. Usually they are dealt with by some form of overloading; but which form is best?

Even if we limit our attention to languages based on the highly successful Hindley/Milner type system, we find many differing treatments of overloading. The same language may treat different operators differently; different languages may treat the same operator differently; and the same language may treat the same operator differently over time. For instance, in Miranda arithmetic is defined only on a single numeric type; equality is a polymorphic function defined at all types, including abstract types where it breaks the abstraction barrier; and the show function may be defined by the user for new types. In the first version of SML equality was simply overloaded at all monomorphic types; while the second version introduced special equality type variables.

Type classes were introduced into Haskell in order to provide a uniform framework for overloading [WB89]. It must have been an idea whose time had come, as it was independently described by Kaes [Kae88]. Since then type classes have attracted considerable attention, with many refinements and variants being described [NS91, NP93, HHPW94, Aug93, PJ93, Jon92b, CHO92, Jon93]. They have also attracted some criticism [App93].

In our view, one of the most serious criticisms of type classes is that a program cannot be assigned a meaning independent of its types. A consequence of this is that two of the most celebrated properties of the Hindley/Milner type system are not satisfied in the presence of type classes: there is no semantic soundness result, and the principal types result holds only in a weak form.

The semantic soundness result shows a correspondence between the typed static semantics of program and its untyped dynamic semantics. It is summarised by Milner's catchphrase 'well typed programs cannot go wrong'. One cannot even formulate such a result for type classes, as no untyped dynamic semantics exists.

The principal type result shows that every typable program has a single most general type. This is also true for type classes. However, much of the utility of this result arises from another property of the Hindley/Milner system: every typeable program remains typeable if all type declarations are removed from it, so type declarations are never required. This fails for type classes: some programs are inherently ambiguous, and require type declarations for disambiguation. Put another way: under Hindley/Milner, a program is untypeable only if it may have no meaning; under type classes, a program may be untypeable because it has too many meanings.

The absence of these properties is not merely the lack of a technical nicety: they arise because the meaning of a program cannot be understood separately from its type. This reduces the range of ways of understanding programs available to a programmer, and reduces the range of ways of implementing programs available to a compiler.

Restricting type classes By a simple restriction to type classes, we may ensure that a program possesses a meaning that can be determined independently of its type.

Recall that a type class limits a type variable, say a, to range over only those types on which an overloaded operator is defined; the overloaded operator may have any type involving a. Here are some examples, representing in simplified form parts of the Haskell standard prelude.

```
class (Num a) where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  neg :: a -> a
  fromInteger :: Integer -> a

class (Eq a) where
  (==) :: (Eq a) => a -> a -> Bool

class (Text a) where
  show :: a -> String
  showList :: [a] -> String
  read :: String -> a
```

*To appear in: *7'th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.

†Institut für Programmstrukturen, 76128 Karlsruhe, Germany.
‡Department of Computing Science, Glasgow G12 8QQ, Scotland.

For instance, the first of these states that type `a` belongs to class `Num` only when there are operators `(+)`, `(*)`, `neg`, and `fromInteger` of the specified types defined for `a`.

The restriction is as follows: for a type class over a type variable `a`, each overloaded operator must have a type of the form `a -> t`, where `t` may itself involve `a`. In the above, `(+)`, `(*)`, `neg`, `(==)`, and `show` satisfy this restriction, while `fromInteger`, `showList`, and `read` do not.

Remarkably, this simple restriction enables one to construct an untyped dynamic semantics, and ensures that no ambiguity can arise: hence type soundness and the strong form of principal types do hold. The resulting system is still powerful enough to handle the overloading of arithmetic, equality, and showing a value as a string, but not powerful enough to handle the overloading of numerical constants or reading a string as a value. The latter are perhaps less essential than the former: neither Miranda nor SML support overloading of the latter sort, and Kaes considered only this restricted form of overloading in his original paper [Kae88].

As an example of the value of this restriction, consider the phrase `[] == []`. In Haskell, this phrase as it stands is ambiguous, and hence meaningless: one must disambiguate by specifying the type of the list elements. This is because the meaning of the program is given by the translation `eqList eqElt [] []`, where `eqList` is equality on lists, and `eqElt` is equality over on the list elements.

In our restricted system, we are guaranteed that the phrase `[] == []` has a meaning independent of types; and that all valid translations yield this meaning. The implementor has a choice: overloading may be implemented by run-time branching, corresponding to the untyped dynamic semantics of Section 3, or by compile-time translation, corresponding to the typed static semantics of Section 4. In the latter case, a valid translation of the program is `eqList undef [] []`, where `undef` is the function that is everywhere undefined; this is because coherence guarantees that if the program doesn't force a translation, then any translation will do. For unrestricted Haskell the compiler writer must choose a translation, because there is no dynamic semantics, and must choose `eqElt` rather that `undef`, because there is no suitable coherence result.

Thus, our restriction of type classes ensures additional useful properties that hold. These additional properties in turn make it possible for us to consider a generalisation of type classes.

Generalising type classes Type classes constrain type variables to range over types at which certain overloaded operators are defined. This appears to be closely related to bounded polymorphism, which constrains type variables to range over types that are subtypes of a given type [CW85, BTCGS91]. Indeed, one can use type classes to mimic bounded polymorphism for the usual subtyping relation on records [Pet94]. But, annoyingly, this mimicry works only for monomorphic records; type classes are not quite powerful enough to handle polymorphic records.

For instance, one would expect the operations `xcoord` and `ycoord` to apply to any record type that contains those fields, for instance it should apply both to a type `Point` containing just those two fields, and to a type `CPoint` that contains both those fields plus a colour. Here is how one can mimic such records in Haskell .

```
class (Pointed a) where
  xcoord :: a -> Float
  ycoord :: a -> Float

data  Point  = MkPoint Float Float
data  CPoint = MkCPoint Float Float Colour

instance  Pointed Point where
  xcoord (MkPoint x y) = x
  ycoord (MkPoint x y) = y

instance  Pointed CPoint where
  xcoord (MkCPoint x y c) = x
  ycoord (MkCPoint x y c) = y

distance :: (Pointed a) => a -> Float
distance p = sqrt (sqr (xcoord p) + sqr (ycoord p))
```

Function `distance` computes the distance of a point from the origin. The type signature is optional, as it may be inferred given only the `class` declaration and the function body.

Note, alas, that this mimicry depends on each field of the record having a monomorphic type that can appear in the class declaration. The polymorphic equivalent of the above would be to have operations `first` and `second` that return the corresponding components of either a pair or a triple, where these may have any type rather than being restricted to `Float`. But there is no way to do this in Haskell.

The source of this problem is class declarations. For `xcoord`, the instances

```
xcoord :: Point -> Float
xcoord :: CPoint -> Float
```

can arise as instantiations of the class declaration

```
xcoord :: a -> Float .
```

But for `first` the instances

```
first :: (a,b) -> a
first :: (a,b,c) -> a
```

have no corresponding class declaration.

We solve this problem by getting rid of class declarations. Instead of declaring that a group of operators belong to a class and specifying a type declaration, we only specify that an operator is overloaded and give no type declaration.

Here is the previous example in our new notation.

```
over xcoord
over ycoord

data  Point  = MkPoint Float Float
data  CPoint = MkCPoint Float Float Colour

inst xcoord :: Point -> Float
     xcoord (MkPoint x y) = x

inst ycoord :: Point -> Float
     ycoord (MkPoint x y) = y

inst xcoord :: CPoint -> Float
     xcoord (MkCPoint x y c) = x

inst ycoord :: CPoint -> Float
     ycoord (MkCPoint x y c) = y

distance :: (xcoord,ycoord::a->Float) => a -> Float
distance p = sqrt (sqr (xcoord p) + sqr (ycoord p))
```

Again, the type declaration for `distance` may be inferred from its body (ignoring, for simplicity, the overloading of `sqrt`, `sqr`, and `+`).

Furthermore, it is now possible to overload `first` and `second` on polymorphic pairs and triples.

```
over first
over second
over third

inst first :: (a,b) -> a
     first (x,y) = x

inst second :: (a,b) -> b
     second (x,y) = x

inst first :: (a,b,c) -> a
     first (x,y,z) = x

inst second :: (a,b,c) -> b
     second (x,y,z) = y

inst third :: (a,b,c) -> c
     third (x,y,z) = c

demo :: (first::a->b,second::a->c) => a -> (c,b)
demo r = (second r, first r)
```

Function `demo` takes a pair or triple and returns its second and first components, in that order. Again, its type can be inferred.

In short, eliminating class declarations makes type classes powerful enough to model bounded polymorphism.

Eliminating class declarations means one need no longer decide in advance which operations belong together in a class. In many situations, this will be a positive advantage. For instance, if we're dealing with pairs we only want `first` and `second` grouped together, but if we're dealing with triples we'll want `third` as well. As a further example, consider the difficulties that the Haskell designers had deciding how to group numeric operators into classes. This design is still argued: should `+` and `*` be in a 'ring' class? The problem is exacerbated because there is no mechanism in Haskell whereby a user may break a given class into smaller classes.

On the other hand, eliminating class declarations means that inferred types become more verbose: the type of every overloaded operator must be mentioned. Records provide some relief here, since they allow us to group related operations together, using a common overloaded identifier for them all. This is explained in more detail in Section 5.

Contributions of this work We combine the above restrictions and generalisations of type classes to define System O, a type system for overloading with the following properties.

- System O possesses an untyped dynamic semantics, and satisfies a corresponding type soundness theorem.
- System O has a strong principal types property. It is never necessary to add type declarations to disambiguate a program.
- As with type classes, there is a standard dictionary transform which takes well-typed programs in System O into equivalent well-typed programs in the Hindley/Milner system.
- System O is powerful enough to model a limited form of F-bounded polymorphism over records, including polymorphic records.

We believe that this makes System O an interesting alternative to type classes.

Related work. Overloading in polymorphic programming languages has first been studied by Kaes [Kae88] and Wadler and Blott [WB89]. Similar concepts can be found in earlier work in symbolic algebra [JT81]. This paper is very much in the tradition of Kaes in that overloading is restricted to functions. It can be seen as a simplification of his system that gets rid of all syntactic declarations of predicates or type classes. We extend the scope of his work by a proof of type soundness and the relationship to record typing.

Much of the later work on overloading is driven by the design and implementation of Haskell's type classes, *e.g.* Nipkow et al. [NS91, NP93] on type reconstruction, Augustsson [Aug93] and Peterson and Jones [PJ93] on implementations, and Hall, Hammond, Peyton Jones and Wadler [HHPW94] on the formal definition of type classes in Haskell. We have already compared our system to that of Haskell.

Other generalisations of Haskell type classes have been proposed. Wadler and Blott, and Jones, consider type classes with multiple type variables [WB89, Jon92b]. Chen, Hudak and Odersky's parametric type classes [CHO92] also have multiple type variables, but a functional dependence is imposed between a primary class variable and dependent parameters. Parametric type classes can model container classes and records. Constructor classes generalize type classes to type constructors [Jon93]. Constructor classes are very good at modeling containers with operations that mediate between similar containers with different element types. We consider it an important problem to determine whether our type system can be generalized to type constructors.

All systems discussed so far implement an *open world* approach, where even empty classes, which do not have any instances at all, are considered legal. This approach works well in a system with separate compilation, where the type checker does not have complete knowledge of instance declarations. By contrast, the *closed world* approach of *e.g.* [Rou90, Smi91, Kae92] rules out empty type schemes. Duggan and Ophel [DO94] support both approaches by distinguishing between open and closed kinds. Volpano [Vol93] has argued that many previously known open world systems are unsound. Volpano's negative results arise because he works with an untyped dynamic semantics for programs with type classes. We have argued here that this is not permissible for Haskell-like programs. Also, by proving type soundness with respect to the untyped dynamic semantics of System O, we show that Volpano's critique does not apply to open world systems in general.

An alternative treatment of overloading regards it as a special case of dynamic typing, using a *typecase* construct to discriminate between overloaded variants [DRW95, HM95]. A semantics along these lines was studied by Thatte [Tha94]. Thatte's semantics maps programs to an explicitly typed polymorphic language similar to XML [MH88]. Type classes denote sets of recursive types in this language. By contrast, our semantics maps to an untyped language where types and type schemes denote ideals.

Outline The rest of this paper is organized as follows. Section 2 presents syntax and typing rules of System O. Section 3 develops a compositional semantics and proves a type soundness theorem. Section 4 discusses the dictionary passing transform. Section 5 presents an encoding of a polymorphic record calculus. Section 6 discusses type reconstruction and the principal type property. Section 7 concludes.

Unique variables	u	$\in \mathcal{U}$	
Overloaded variables	o	$\in \mathcal{O}$	
Constructors	k	$\in \mathcal{K} = \bigcup\{\mathcal{K}_D \mid D \in \mathcal{D}\}$	
Variables	x	$= u \mid o \mid k$	
Terms	e	$= x \mid \lambda u.e \mid e\,e' \mid \mathsf{let}\ u = e\ \mathsf{in}\ e'$	
Programs	p	$= e \mid \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$	
Type variables	α	$\in \mathcal{A}$	
Datatype constructors	D	$\in \mathcal{D}$	
Type constructors	T	$\in \mathcal{T} = D \cup \{\rightarrow\}$	
Types	τ	$= \alpha \mid \tau \rightarrow \tau' \mid D\,\tau_1 \ldots \tau_n$	where $n = \text{arity}(D)$
Type schemes	σ	$= \tau \mid \forall\alpha.\pi_\alpha \Rightarrow \sigma$	
Constraints on α	π_α	$= o_1 : \alpha \rightarrow \tau_1, \ldots, o_n : \alpha \rightarrow \tau_n$	($n \geq 0$, with $o_1, \ldots, o_n$ distinct)
Typotheses	Γ	$= x_1 : \sigma_1, \ldots, x_n : \sigma_n$	($n \geq 0$)

Figure 1: Abstract syntax of System O.

(TAUT) $$\Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$\frac{\Gamma \vdash x_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash x_n : \sigma_n}{\Gamma \vdash x_1 : \sigma_1, \ldots, x_n : \sigma_n} \quad \text{(SET)}$$

(∀I) $$\frac{\Gamma, \pi_\alpha \vdash e : \sigma \qquad (\alpha \notin \text{tv}(\Gamma))}{\Gamma \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma \qquad \Gamma \vdash [\tau/\alpha]\pi_\alpha}{\Gamma \vdash e : [\tau/\alpha]\sigma} \quad (\forall\text{E})$$

(→I) $$\frac{\Gamma, u : \tau \vdash e : \tau'}{\Gamma \vdash \lambda u.e : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e : \tau' \rightarrow \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\,e' : \tau} \quad (\rightarrow\text{E})$$

(LET) $$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, u : \sigma \vdash e' : \tau}{\Gamma \vdash \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

$$\frac{\begin{array}{c}(o : \sigma_{T'} \in \Gamma \;\Rightarrow\; T \neq T') \\ \Gamma \vdash e : \sigma_T \qquad \Gamma, o : \sigma_T \vdash p : \sigma'\end{array}}{\Gamma \vdash \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p : \sigma'} \quad \text{(INST)}$$

Figure 2: Typing rules for System O.

2 Type System

We base our discussion on a simple functional language with overloaded identifiers. Figure 1 gives the syntax of terms and types. We split the variable alphabet into subalphabets $\mathcal{U}$ for *unique* variables, ranged over by u, $\mathcal{O}$ for *overloaded* variables, ranged over by o, and $\mathcal{K}$ for data constructors, ranged over by k. The letter x ranges over both unique and overloaded variables as well as constructors. We assume that every non-overloaded variable u is bound at most once in a program.

The syntax of *terms* is identical to the language *Exp* in [Mil78]. A *program* consists of a sequence of instance declarations and a term. An *instance declaration* ($\mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$) overloads the meaning of the identifier o with the function given by e on all arguments that are constructed from the type constructor T.

A *type* τ is a type variable, a function type, or a datatype. Datatypes are constructed from *datatype constructors* D. For simplicity, we assume that all value constructors and selectors of a datatype $D\ \tau_1\ \ldots\ \tau_n$ are predefined, with bindings in some fixed initial typothesis Γ_0. With user-defined type declarations, we would simply collect in Γ_0 all selectors and constructors actually declared in a given program. Let $\mathcal{K}_D$ be the set of all value constructors that yield a value in $D\,\tau_1, \ldots, \tau_n$ for some types $\tau_1, \ldots, \tau_n$.

We assume that there exists a *bottom* datatype $\perp\!\!\!\perp \in D$ with $\mathcal{K}_{\perp\!\!\!\perp} = \emptyset$. Note that this type is present in Miranda, where it is written (), but is absent in Haskell, where () has a value constructor, also written (). We let T range over datatype constructors as well as the function type constructor $(\rightarrow)$, writing $(\rightarrow)\,\tau\,\tau'$ as a synonym for $\tau \rightarrow \tau'$.

A *type scheme* σ consists of a type τ and quantifiers for some of the type variables in τ. Unlike with Hindley/Milner polymorphism, a quantified variable α comes with a *constraint* π_α, which is a (possibly empty) set of bindings $o : \alpha \rightarrow \tau$. An overloaded variable o can appear at most once in a constraint. Constraints restrict the instance types of a type scheme by requiring that overloaded identifiers are defined at given types. The Hindley/Milner type scheme $\forall\alpha.\sigma$ is regarded as syntactic sugar for $\forall\alpha.() \Rightarrow \sigma$.

Figure 2 defines the typing rules of System O. The type system is identical to the original Hindley/Milner system, as presented in in [DM82], except for two modifications.

- In rule (∀I), the constraint π_α on the introduced bound variable α is traded between typothesis and type scheme. Rule (∀E) has as a premise an instantiation of the eliminated constraint. Constraints are derived using rule (SET). Note that this makes rules (∀I) and (∀E) symmetric to rules (→I) and (→E).
- There is an additional rule (INST) for instance dec-

larations. The rule is similar to (LET), except that the overloaded variable o has an explicit type scheme σ_T and it is required that the type constructor T is different in each instantiation of a variable o.

We let σ_T range over closed type schemes that have T as outermost argument type constructor:

$$\begin{array}{rcll} \sigma_T & = & T\,\alpha_1 \ldots \alpha_n \to \tau & (\mathrm{tv}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}) \\ & | & \forall\alpha.\pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \subseteq \mathrm{tv}(\sigma'_T)). \end{array}$$

The explicit declaration of σ_T in rule (INST) is necessary to ensure that principal types always exist. Without it, one might declare an instance declaration such as

$$\mathsf{inst}\ o = \lambda x.x\ \mathsf{in}\ p$$

where the type constructor on which o is overloaded cannot be determined uniquely.

The syntactic restrictions on type schemes σ_T enforce three properties: First, overloaded instances must work uniformly for all arguments of a given type constructor. Second the argument type must determine the result type uniquely. Finally, all constraints must apply to component types of the argument. The restrictions are necessary to ensure termination of the type reconstruction algorithm. An example is given in Section 6.

The syntactic restrictions on type schemes σ_T also explain why the overloaded variables of a constraint π_α must be pairwise different. A monomorphic argument to an overloaded function completely determines the instance type of that function. Hence, for any argument type τ and overloaded variable o, there can be only one instance type of o on arguments of type τ. By embodying this rule in the form of type variable constraints we enforce it at the earliest possible time.

Example 2.1 The following program fragment gives instance declarations for the equality function `(==)`. We adapt our notation to Haskell's conventions, writing `::` instead of : in a typing; writing `(o::a->t1)=>t2` instead of $\forall\alpha.(o : a \to \tau_1) \Rightarrow \tau_2$; and writing `inst o :: s; o = e` instead of $\mathsf{inst}\ o : \sigma = e$.

```
inst (==) :: Int -> Int -> Bool
     (==) =  primEqInt

listEq :: ((==)::a->a->Bool) => [a]->[a]->Bool
listEq [] []           =  True
listEq (x:xs) (y:ys)   =  x == y && listEq xs ys

inst (==) :: ((==):: a->a->Bool) => [a]->[a]->Bool
     (==) =  listEq
```

Note that using `(==)` directly in the second instance declaration would not work, since instance declarations are not recursive. An extension of System O to recursive instance declaration would be worthwhile but is omitted here for simplicity.

Example 2.2 The following example demonstrates an object-oriented style of programming, and shows where we are more expressive than Haskell's type classes. We write instances of a polymorphic class `Set`, with a member test and operations to compute the union, intersection, and difference of two sets. In Haskell, only sets of a fixed element type could be expressed. The example uses the record extension of Section 5; look there for an explanation of record syntax.

```
type Set a sa
    =  (union, inters, diff :: sa -> sa,
        member :: a -> Bool |}

inst set :: ((==)::a->a->Bool) => [a] -> Set a [a]
     set xs =
       (union  = \ys -> xs ++ ys,
        inters = \ys -> [y | y <- ys | y `elem` xs],
        diff   = \ys -> xs \\ ys,
        member = \y  -> y `elem` xs)

inst set :: ((==),(<):: a->a->Bool)
                      => Tree a -> Set a (Tree a)
     set =  ...
```

m Here are some functions that work with sets.

```
union :: (set: sa -> Set a sa) => sa -> sa -> sa
union xs ys = #union (set xs) ys

diff :: (set: sa -> Set a sa) => sa -> sa -> sa
diff xs ys = #diff (set xs) ys

simdiff :: (set: sa -> Set a sa) => sa -> sa -> sa
simdiff xs ys =  union (diff xs ys) (diff ys xs)
```

3 Semantics

We now give a compositional semantics of System O and show that typings are sound with respect it. The semantics specifies lazy evaluation of functions, except for overloaded functions, which are strict in their first argument. Alternatively, we could have assumed strict evaluation uniformly for all functions, with little change in our definitions and no change in our results.

The meaning of a term is a value in the CPO $\mathcal{V}$, where $\mathcal{V}$ is the least solution of the equation

$$\mathcal{V} \;=\; \mathbf{W}_\perp \;+\; \mathcal{V} \to \mathcal{V} \;+\; \sum_{k \in \mathcal{K}} (k\ \mathcal{V}_1\ \ldots\ \mathcal{V}_{\mathrm{arity}(k)})_\perp.$$

Here, $(+)$ and $\sum$ denote coalesced sums[1] and $\mathcal{V} \to \mathcal{V}$ is the continuous function space. The value $\mathbf{W}$ denotes a type error – it is often pronounced "wrong". We will show that the meaning of a well-typed program is always different from "wrong".

The meaning function $[\![\cdot]\!]$ on terms is given in Figure 3. It takes as arguments a term and an environment η and yields an element of $\mathcal{V}$.The environment η maps unique variables to arbitrary elements of $\mathcal{V}$, and it maps overloaded variables to strict functions:

$$\eta : \mathcal{U} \to \mathcal{V}\ \ \cup\ \ \mathcal{O} \to (\mathcal{V} \circ\!\!\to \mathcal{V}).$$

The notation $\eta[x := v]$ stands for extension of the environment η by the binding of x to v.

Note that our semantics is more "lazy" in detecting wrong terms than Milner's semantics [Mil78]. Milner's semantics always maps a function application $f\,\mathbf{W}$ to $\mathbf{W}$ whereas in our semantics $f\,\mathbf{W} = \mathbf{W}$ only if f is strict. Our semantics correspond better to the dynamic type checking which would in practice be performed when an argument is evaluated. We anticipate no change in our results if Milner's stricter error checking is adopted.

We now give a meaning to types. We start with types that do not contain type variables, also called *monotypes*. We use μ to range over monotypes. Following [Mil78] and

[1] Injection and projection functions for sums will generally be left implicit to avoid clutter.

$$[\![x]\!]\eta \;=\; \eta(x)$$

$$[\![\lambda u.e]\!]\eta \;=\; \lambda v.[\![e]\!]\eta[u := v]$$

$$[\![k\ M_1 \ldots M_n]\!]\eta \;=\; k\ ([\![M_1]\!]\eta) \ldots ([\![M_n]\!]\eta), \quad \text{where } n = \text{arity}(k)$$

$$[\![e\,e']\!]\eta \;=\; \text{if } [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \text{ then } ([\![e]\!]\eta)([\![e']\!]\eta) \text{ else } \mathbf{W}$$

$$[\![\text{let } u = e \text{ in } e']\!]\eta \;=\; [\![e']\!]\eta[u := [\![e]\!]\eta]$$

$$\begin{aligned}&[\![\text{inst } o : \sigma_T = e \text{ in } p]\!]\eta \;=\\ &\quad \text{if } [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \text{ then}\\ &\qquad [\![p]\!]\eta[o := extend(T, [\![e]\!]\eta, \eta(o))]\\ &\quad \text{else } \mathbf{W}\end{aligned}$$

where

$$\begin{aligned}&extend((\to), f, g) =\\ &\quad \lambda v.\text{if } v \in \mathcal{V} \to \mathcal{V} \text{ then } f(v) \text{ else } g(v)\\ &extend(D, f, g) =\\ &\quad \lambda v.\text{if } \exists k \in \mathcal{K}_D.v \in k\ \underbrace{\mathcal{V} \ldots \mathcal{V}}_{\text{arity}(k)} \text{ then } f(v) \text{ else } g(v).\end{aligned}$$

Figure 3: Semantics of terms.

[MPS86], we let monotypes denote ideals. For our purposes, an ideal I is a set of values in $\mathcal{V}$ which does not contain $\mathbf{W}$, is downward-closed and is limit-closed. That is, $y \in I$ whenever $y \leq x$ and $x \in I$, and $\bigsqcup X \in I$ whenever $x \in I$ for all elements x of the directed set X.

The meaning function $[\![\cdot]\!]$ takes a monotype μ to an ideal. It is defined as follows.

$$\begin{aligned}&[\![D\ \mu_1 \ldots \mu_m]\!] \;=\\ &\quad \{\bot\} \cup \bigcup \{k\ [\![\mu_1']\!] \ \ldots\ [\![\mu_n']\!]\\ &\qquad\qquad \mid \Gamma_0 \vdash k : \mu_1' \to \ldots \to \mu_n' \to D\ \mu_1 \ldots \mu_m\}\\ &[\![\mu_1 \to \mu_2]\!] \;=\\ &\quad \{f \in \mathcal{V} \to \mathcal{V} \mid v \in [\![\mu_1]\!] \Rightarrow f\,v \in [\![\mu_2]\!]\}.\end{aligned}$$

Proposition 3.1 Let μ be a monotype. Then $[\![\mu]\!]$ is an ideal.

Proof: A straightforward induction on the structure of μ. □

When trying to extend the meaning function to type schemes we encounter the difficulty that instances of a constrained type scheme $\forall\alpha.\pi_\alpha \Rightarrow \sigma$ depend on the overloaded instances in the environment. This is accounted for by indexing the meaning function for type schemes with an environment.

Definition. A monotype μ is a *semantic instance* of a type scheme σ in an environment η, written $\eta \models \mu \preceq \sigma$, iff this can be derived from the two rules below.

(a) $\eta \models \mu \preceq \mu$.

(b) $\eta \models \mu \preceq (\forall\alpha.\pi_\alpha \Rightarrow \sigma)$
if there is a monotype μ' such that $\eta \models \mu \preceq [\mu'/\alpha]\sigma$ and $\eta(o) \in [\![[\mu'/\alpha]\tau]\!]$, for all $o : \tau \in \pi_\alpha$.

Definition. The meaning $[\![\sigma]\!]_\eta$ of a closed type scheme σ is given by

$$[\![\sigma]\!]_\eta \;=\; \bigcap \{[\![\mu]\!] \mid \eta \models \mu \preceq \sigma\}.$$

Definition. $\eta \models e_1 : \sigma_1, \ldots, e_n : \sigma_n$ iff $[\![e_i]\!]\eta \in [\![\sigma_i]\!]\eta$, for $i = 1, \ldots, n$.

The meaning of type schemes is compatible with the meaning of types:

Proposition 3.2 Let μ be a monotype, and let η be an environment. Then $[\![\mu]\!]_\eta = [\![\mu]\!]$.

Proof: Direct from the definitions of $[\![\sigma]\!]_\eta$ and $\preceq$. □

We now show that type schemes denote ideals. The proof needs two facts about the bottom type $\perp\!\!\!\perp$.

Lemma 3.3 Let η be an environment.
(a) $\eta \models o : \perp\!\!\!\perp \to \mu$, for any variable o, monotype μ.
(b) Let $\sigma = \forall\alpha_1.\pi_{\alpha\,1} \Rightarrow \ldots \forall\alpha_n.\pi_{\alpha\,n} \Rightarrow \tau$ be a type scheme. Then $\eta \models [\perp\!\!\!\perp/\alpha_1, \ldots, \perp\!\!\!\perp/\alpha_n]\tau \preceq \sigma$.

Proof: (a) Assume $v \in [\![\perp\!\!\!\perp]\!]$. Since $\perp\!\!\!\perp$ does not have any constructors, $[\![\perp\!\!\!\perp]\!] = \{\bot\}$, hence $v = \bot$. Since $\eta(o)$ is a strict function, $\eta(o)v = \bot$, which is an element of every monotype.
(b) Follows from the definition of $\preceq$ and (a). □

Proposition 3.4 Let σ be a type scheme and let η be an environment. Then $[\![\sigma]\!]_\eta$ is an ideal.

Proof: The closure properties are shown by straightforward inductions on the structure of σ. It remains to be shown that $\mathbf{W} \notin [\![\sigma]\!]$. By Lemma 3.3(b) there is a monotype μ such that $\eta \models \mu \preceq \sigma$. Hence, $[\![\sigma]\!]_\eta \subseteq [\![\mu]\!]$. But $[\![\mu]\!]$ is an ideal and therefore does not contain $\mathbf{W}$. □

Proposition 3.4 expresses an important property of our semantics: every type scheme is an ideal, even if it contains a type variable constraint $o : \alpha \to \tau$, where o does not have any explicitly declared instances at all. Consequently, there is no need to rule out such a type scheme statically. This corresponds to Haskell's "open world" approach to type-checking, as opposed to the "closed world" approach of *e.g.* [Smi91]. Interestingly, the only thing that distinguishes those two approaches in the semantics of type schemes is the absence or presence of the bottom type $\perp\!\!\!\perp$.

We now show that System O is sound, *i.e.* that syntactic type judgements $\Gamma \vdash p : \sigma$ are reflected by semantic type judgements $\Gamma \models p : \sigma$.

Definition. Let e be a term, let Γ be a closed typothesis, and let σ be a closed type scheme. Then $\Gamma \models e : \sigma$ iff, for all environments η, $\eta \models \Gamma$ implies $\eta \models e : \sigma$.

As a first step, we prove a soundness theorem for terms. This needs an auxiliary lemma, whose proof is straightforward.

Lemma 3.5 If $\eta \models e : \sigma$ and $\eta \models \mu \preceq \sigma$ then $\eta \models e : \mu$.

Theorem 3.6 (Type Soundness for Terms) Let $\Gamma \vdash e : \sigma$ be a valid typing judgement and let S be a substitution such that $S\Gamma$ and $S\sigma$ are closed. Then $S\Gamma \models e : S\sigma$.

Proof: Assume $\Gamma \vdash e : \sigma$ and $\eta \models S\Gamma$. We do an induction on the derivation of $\Gamma \vdash e : \sigma$. We only show cases ($\forall$I), ($\forall$E), whose corresponding inference rules differ from the Hindley/Milner system. The proofs of the other rules are similar to the treatment in [Mil78].

Case ($\forall I$): Then the last step in the derivation is

$$\frac{\Gamma, \pi_\alpha \;\vdash\; e : \sigma' \qquad \alpha \notin \text{tv}(\Gamma)}{\Gamma \;\vdash\; e : \forall\alpha.\pi_\alpha \Rightarrow \sigma'}$$

$$\text{(TAUT)}\quad \Gamma \vdash u : \sigma \;\succ\; u \quad (u : \sigma \in \Gamma) \qquad \Gamma \vdash k : \sigma \;\succ\; u \quad (k : \sigma \in \Gamma) \qquad \Gamma \vdash o : \sigma \;\succ\; u_{o,\sigma} \quad (o : \sigma \in \Gamma)$$

$$\text{(}\forall\text{I)}\quad \frac{\Gamma, o_1 : \tau_1, \ldots, o_n : \tau_n \vdash e : \sigma \;\succ\; e^* \qquad \alpha \notin \mathrm{tv}(\Gamma)}{\begin{array}{l}\Gamma \;\vdash\; e : \forall\alpha.(o_1 : \tau_1, \ldots, o_n : \tau_n) \Rightarrow \sigma \\ \quad\succ\; \lambda u_{o_1,\tau_1}. \ldots \lambda u_{o_n,\tau_n}.e^*\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \;\vdash\; e : \forall\alpha.(o_1 : \tau_1, \ldots, o_n : \tau_n) \Rightarrow \sigma \;\succ\; e^* \\ \Gamma \;\vdash\; o_i : [\tau/\alpha]\tau_i \;\succ\; e_i^* \qquad (i = 1, \ldots, n)\end{array}}{\begin{array}{l}\Gamma \;\vdash\; e : [\tau/\alpha]\sigma \\ \quad\succ\; e^*\, e_1^* \ldots e_n^*\end{array}}\quad\text{(}\forall\text{E)}$$

$$(\to\text{I})\quad \frac{\Gamma, u : \tau \vdash e : \tau' \;\succ\; e^*}{\begin{array}{l}\Gamma \;\vdash\; \lambda u.e : \tau \to \tau \\ \quad\succ\; \lambda u.e^*\end{array}}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \;\succ\; e_1^* \qquad \Gamma \vdash e_2 : \tau' \;\succ\; e_2^*}{\begin{array}{l}\Gamma \;\vdash\; e_1\, e_2 : \tau \\ \quad\succ\; e_1^* e_2^*\end{array}}\quad(\to\text{E})$$

$$\text{(LET)}\quad \frac{\Gamma \vdash e_1 : \sigma \;\succ\; e_1^* \qquad \Gamma, u : \sigma \vdash e_2 : \tau \;\succ\; e_2^*}{\begin{array}{l}\Gamma \;\vdash\; \mathsf{let}\ u = e_1\ \mathsf{in}\ e_2 : \tau \\ \quad\succ\; \mathsf{let}\ u = e_1^*\ \mathsf{in}\ e_2^* : \tau\end{array}}$$

$$\frac{\begin{array}{c}o : \sigma_{T'} \in \Gamma \;\Rightarrow\; T \neq T' \\ \Gamma \vdash e : \sigma_T \;\succ\; e^* \qquad \Gamma, o : \sigma_T \vdash p : \sigma' \;\succ\; p^*\end{array}}{\begin{array}{l}\Gamma \;\vdash\; \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p : \sigma' \\ \quad\succ\; \mathsf{let}\ u_{o,\sigma_T} = e^*\ \mathsf{in}\ p^*\end{array}}\quad\text{(INST)}$$

Figure 4: The dictionary passing transform

for some α, π_α, σ' with $\sigma = \forall\alpha.\pi_\alpha \Rightarrow \sigma'$. We have to show that $e \in [\![\mu]\!]$, for all μ such that $\eta \models \mu \preceq \forall\alpha.S\pi_\alpha \Rightarrow S\sigma'$. Pick an arbitrary such μ. By definition of $(\preceq)$, there exists a μ' such that $\eta \models [\mu'/\alpha](S\pi_\alpha)$ and $\eta \models \mu \preceq [\mu'/\alpha](S\sigma')$. Let $S' = [\mu'/\alpha] \circ S$. Then $\eta \models S'\pi_\alpha$ and $\eta \models \mu \preceq S'\sigma'$. Since $\alpha \notin \mathrm{tv}(\Gamma)$, $\eta \models S'\Gamma$ and therefore $\eta \models S'(\Gamma, \pi_\alpha)$. Then by the induction hypothesis, $\eta \models e : S'\sigma'$. It follows with Lemma 3.5 that $\eta \models e : \mu$.

Case ($\forall E$): Then the last step in the derivation is

$$\frac{\Gamma \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma' \qquad \Gamma \vdash [\tau/\alpha]\pi_\alpha}{\Gamma \vdash e : [\tau/\alpha]\sigma'}$$

for some α, π_α, σ', τ with $\sigma = [\tau/\alpha]\sigma'$. We have to show that $e \in [\![\mu]\!]$, for all μ such that $\eta \models \mu \preceq [S\tau/\alpha]S\sigma'$. Pick an arbitrary such μ. By the induction hypothesis, $\eta \models e : \forall\alpha.S\pi_\alpha \Rightarrow S\sigma'$ and $\eta \models [S\tau/\alpha](S\pi_\alpha)$. It follows with the definition of $\preceq$ that $\eta \models \mu \preceq \forall\alpha.S\pi_\alpha \Rightarrow S\sigma'$. Then by Lemma 3.5, $\eta \models e : \mu$. □

We now extend the type soundness theorem to whole programs that can contain instance declarations.

Theorem 3.7 (Type Soundness for Programs)
Let $\Gamma \vdash p : \sigma$ be a valid closed typing judgement. Then $\Gamma \models p : \sigma$.

Proof: By induction on the structure of p. If p is a term, the result follows from Theorem 3.6. Otherwise p is an instance declaration at top-level. Then the last step in the derivation of $\Gamma \vdash p : \sigma$ is

$$\frac{\begin{array}{c}o : \sigma_{T'} \in \Gamma \;\Rightarrow\; T \neq T' \\ \Gamma \vdash e : \sigma_T \qquad \Gamma, o : \sigma_T \vdash p : \sigma\end{array}}{\Gamma \vdash \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p' : \sigma}$$

for some type scheme σ_T. We have to show that $\eta \models \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p' : \sigma$. By Theorem 3.6, $\eta \models e : \sigma_T$, which implies that $[\![e]\!]_\eta$ is a function. Therefore, $[\![p]\!]\eta = [\![p']\!]\eta[o := f]$ where $f = \mathit{extend}(T, [\![e]\!]\eta, \eta(o))$.

Our next step is to show that $f \in [\![\sigma_T]\!]_\eta$. Let μ be such that $\eta \models \mu \preceq \sigma_T$. Then $\mu = T\mu_1, \ldots, \mu_n \to \mu'$, for some monotypes $\mu_1, \ldots, \mu_n, \mu'$. Now assume that $v \in [\![T\mu_1, \ldots, \mu_n]\!]$. If $v = \bot$ then $f\,v = \bot \in [\![\mu']\!]$. Otherwise, by the definition of *extend*, $f\,v = [\![e]\!]\eta\,v$, and $[\![e]\!]\eta\,v \in [\![\mu']\!]$. In both cases $f\,v \in [\![\mu']\!]$. Since $v \in [\![T\mu_1, \ldots, \mu_n]\!]$ was arbitrary, we have $f \in [\![\mu]\!]$. Since μ was arbitrary, this implies $f \in [\![\sigma_T]\!]_\eta$

It follows that $\eta[o := f] \models o : \sigma_T$. Furthermore, since $\eta \models \Gamma$, and Γ contains by the premise of rule (INST) no binding $o : \sigma_T$, we have that $\eta[o := f] \models \Gamma$. Taken together, $\eta[o := f] \models \Gamma, o : \sigma_T$. By the induction hypothesis, $\eta[o := f] \models p' : \sigma$, which implies the proposition. □

A corollary of this theorem supports the slogan that "well typed programs do not go wrong".

Corollary 3.8 Let $\Gamma \vdash p : \sigma$ be a valid closed typing judgement and let η be an environment. If $\eta \models \Gamma$ then $[\![p]\!]\eta \neq \mathbf{W}$.

Proof: Immediate from Theorem 3.7 and Proposition 3.4. □

4 Translation

This section studies the "dictionary passing" transform from System O to the Hindley/Milner system. Its central idea is to convert a term of type $\forall\alpha.\pi_\alpha \Rightarrow \tau$ to a function that takes as arguments implementations of the overloaded variables in π_α. These arguments are also called "dictionaries".

The target language of the translation is the Hindley/Milner system, which is obtained from System O by eliminating overloaded variables o, instance declarations, and constraints π_α in type schemes. The translation of terms is given in Figure 4. It is formulated as a function of type derivations, where we augment type judgements with an additional component e^* that defines the translation of a term or program p, e.g. $\Gamma \vdash p : \sigma \succ p^*$. To ensure the coherence of the translation, we assume that the overloaded identifiers o_i in a type variable constraint $\{o_1 : \alpha \to \tau_1, \ldots, o_n : \alpha \to \tau_n\}$ are always ordered lexicographically.

Types and type schemes are translated as follows.

$$\begin{array}{rcl}\tau^* &=& \tau \\ (\forall\alpha.\epsilon \Rightarrow \sigma)^* &=& \forall\alpha.\sigma^* \\ (\forall\alpha.o : \alpha \to \tau, \pi_\alpha \Rightarrow \sigma)^* &=& \forall\alpha.(\alpha \to \tau) \to (\forall\pi_\alpha \Rightarrow \sigma)^*\end{array}$$

The last clause violates our type syntax in that a type scheme can be generated as the result part of an arrow.

This is compensated by defining

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

Bindings and typotheses are translated as follows.

$$\begin{array}{rcl} (u:\sigma)^* & = & u:\sigma^* \\ (o:\sigma)^* & = & u_{o,\sigma}:\sigma^*. \\ o_1:\sigma_1,\ \ldots,\ o_n:\sigma_n & = & (o_1:\sigma_1)^*,\ \ldots,\ (o_n:\sigma_n)^*. \end{array}$$

This translates an overloaded variable o to a new unique variable $u_{o,\sigma}$, whose identity depends on both the name o and its type scheme, σ.

Each derivation rule $\Gamma \vdash p : \sigma$ in System O corresponds to a derivation of translated typotheses, terms and type schemes in the Hindley/Milner system. One therefore has:

Proposition 4.1 If $\Gamma \vdash p : \sigma \succ p^*$ is valid then $\Gamma^* \vdash p^* : \sigma^*$ is valid in the Hindley/Milner system

We believe that the translation preserves semantics in the following sense.

Conjecture Let p be a program, μ be a monotype, and let η be an environment. Let Γ be a typothesis which does not contain overloaded variables. If $\Gamma \vdash p : \mu \succ p^*$ and $\eta \models \Gamma$ then $[\![p]\!]\eta = [\![p^*]\!]\eta$.

Although the above claim seems clearly correct, its formal proof is not trivial. Note that coherence of the translation would follow immediately from the above conjecture. Coherence, again, is a property that appears obvious but is notoriously tricky to demonstrate [Blo91, Jon92a], so it is perhaps not surprising that the above conjecture shares this property.

5 Relationship with Record Typing

In this section we study an extension of our type system with a simple polymorphic record calculus similar to Ohori's [Oho92]. Figure 5 details the extended calculus. We add to System O

- record types $\{l_1 : \tau_1,\ \ldots,\ l_n : \tau_n\}$,
- record expressions $\{l_1 = e_1, ..., l_n = e_n\}$, and
- selector functions $\#l$.

It would be easy to add record updates, as in the work of Ohori, but more difficult to handle record extension, as in the work of Wand [Wan87] or Rémy [Rem89]. Jones [Jon92a] has shown how to embed Rémy's system of extensible records by extending unification to an AC theory for records and using (multi-parameter) type classes for stating the absence of fields in a record. Both updates and extensions are however omitted here for simplicity.

Leaving open for the moment the type of selector functions, the system presented so far corresponds roughly to the way records are defined in Standard ML. Selectors are treated in Standard ML as overloaded functions. As with all overloaded functions, the type of the argument of a selector has to be known statically; if it isn't, an overloading resolution error results.

Our record extension also treats selectors as overloaded functions but uses the overloading concept of System O. The most general type scheme of a selector $\#l$ is

$$\forall\beta.\forall\alpha.(\alpha \le \{l : \beta\}) \Rightarrow \alpha \to \beta.$$

This says that $\#l$ can be applied to records that have a field $l : \tau$, in which case it will yield a value of type τ. The type scheme uses a *subtype constraint* $\alpha \le \rho$. Subtype constraints are validated using the subtyping rules in Figure 5. In all other respects, they behave just like overloading constraints $o : \alpha \to \tau$.

Example 5.1 The following program is typable in System O (where the typing of max is added for convenience).

$$\begin{array}{l} \mathsf{let}\ \ \mathit{max}\ :\ \forall\beta.((<) : \beta \to \beta \to \mathsf{bool}) \Rightarrow \\ \qquad\qquad\quad \forall\alpha.(\alpha \le \{\mathit{key} : \beta\}) \Rightarrow \alpha \to \alpha \to \alpha \\ \qquad\qquad = \lambda x.\lambda y.\mathsf{if}\ \#\mathit{key}\ x < \#\mathit{key}\ y\ \mathsf{then}\ y\ \mathsf{else}\ x \\ \mathsf{in} \\ \qquad \mathit{max}\ \{\mathit{key} = 1, \mathit{data} = a\}\ \{\mathit{key} = 2, \mathit{data} = b\} \end{array}$$

In Standard ML, the same program would not be typable since neither the argument type of the selector $\#key$ nor the argument type of the overloaded function $(<)$ are statically known.

Note that the bound variable in a subtype constraint can also appear in the constraining record type, as in

$$\forall\alpha.(\alpha \le \{l : \alpha \to \mathsf{bool}\}) \Rightarrow [\alpha]$$

Hence, we have a limited form of F-bounded polymorphism [CCH+89] — limited since our calculus lacks the subsumption and contravariance rules often associated with bounded polymorphism [CW85]. It remains to be seen how suitable our system is for modeling object-oriented programming. Some recent developments in object-oriented programming languages seem to go in the same direction, by restricting subtyping to abstract classes [SOM93].

We now show that the record extension adds nothing essentially new to our language. We do this by presenting an encoding from System O with records to plain System O. The source of the encoding is a program with records, where we assume that the labels $l_1,\ \ldots,\ l_n$ of all record expressions $\{l_1 = e_1, ..., l_n = e_n\}$ in the source program are sorted lexicographically (if they are not, just rearrange fields). The details of the encoding are as follows.

1. Every record-field label l in a program is represented by an overloaded variable, which is also called l.

2. For every record expression $\{l_1 = e_1, ..., l_n = e_n\}$ in a program, we add a fresh n-ary datatype $R_{l_1 \ldots l_n}$ with a constructor of the same name and selectors as given by the declaration

$$\mathsf{data}\ R_{l_1 \ldots l_n}\ \alpha_1 \ldots \alpha_n = R_{l_1 \ldots l_n}\ \alpha_1 \ldots \alpha_n.$$

3. For every datatype $R_{l_1 \ldots l_n}$ created in Step 2 and every label l_i $(i = 1, ..., n)$, we add an instance declaration

$$\begin{array}{l} \mathsf{inst}\ l_i\ :\ \forall\alpha_1 \ldots \alpha_n . R_{l_1 \ldots l_n}\ \alpha_1 \ldots \alpha_n \to \alpha_i \\ \qquad\quad = \lambda(R_{l_1 \ldots l_n}\ x_1 \ldots x_n).x_i \end{array}$$

(where the pattern notation in the formal parameter is used for convenience).

4. A record expression $\{l_1 = e_1, ..., l_n = e_n\}$ now translates to $R_{l_1 \ldots l_n}\ e_1 \ldots e_n$.

5. A selector function $\#l$ translates to l.

6. A record type $\{l_1 : \tau_1, ..., l_n : \tau_n\}$ is translated to $R_{l_1 \ldots l_n}\ \tau_1 \ldots \tau_n$.

Additional Syntax

Field labels	l	$\in$	$\mathcal{L}$	
Terms	e	$=$	$\ldots \mid \#l \mid \{l_1 = e_1, \ldots, l_n = e_n\}$	$(n \geq 0)$
Record types	ρ	$=$	$\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$	$(n \geq 0$, with $l_1, \ldots, l_n$ distinct)
Types	τ	$=$	$\ldots \mid \rho$	
Constraints on α	π_α	$=$	$\ldots \mid \alpha \leq \rho$	
Typotheses	Γ	$=$	$\ldots \mid \alpha \leq \rho$	

Subtyping Rules

$$\text{(Taut)} \quad \Gamma, \alpha \leq \rho \vdash \alpha \leq \rho$$

$$\Gamma \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1} : \tau_{n+1}, \ldots, l_{n+k} : \tau_{n+k}\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad \text{(Rec)}$$

Additional Typing Rules

$$(\{\}\mathrm{I}) \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\Gamma \vdash \#l : \forall\beta.\forall\alpha \leq \{l : \beta\}.\alpha \to \beta \quad (\{\}\mathrm{E})$$

Figure 5: Extension with record types.

7. A subtype constraint $\alpha \leq \{l_1 : \tau_1, ..., l_n : \tau_n\}$ becomes an overloading constraint $l_1 : \alpha \to \tau_1, \ldots, l_n : \alpha \to \tau_n$.

Let $e^\dagger$, $\sigma^\dagger$, or $\Gamma^\dagger$ be the result of applying this translation to a term e, a type scheme σ, or a typothesis Γ. Then one has:

Proposition 5.2 $\Gamma \vdash e : \tau$ iff $\Gamma^\dagger \vdash e^\dagger : \tau^\dagger$.

Proposition 5.2 enables us to extend the type soundness and principal type properties of System O to its record extension without having to validate them again. It also points to an implementation scheme for records, given an implementation scheme for overloaded identifiers.

Example 5.3 The program of Example 5.1 translates to

$$\begin{array}{ll}
\mathsf{inst}\ \mathit{data} & : \forall\alpha\forall\beta.R_{\mathsf{data,key}}\ \alpha\ \beta \to \alpha \\
& = \lambda R_{\mathsf{data,key}}\ x\ y.\ x\ \mathsf{in} \\
\mathsf{inst}\ \mathit{key} & : \forall\alpha\forall\beta.R_{\mathsf{data,key}}\ \alpha\ \beta \to \beta \\
& = \lambda R_{\mathsf{data,key}}\ x\ y.\ y\ \mathsf{in} \\
\mathsf{let}\ \ \mathit{max} & : \forall\beta.((<) : \beta \to \beta \to \mathsf{bool}) \Rightarrow \\
& \quad \forall\alpha.(\mathit{key} : \alpha \to \beta) \Rightarrow \alpha \to \alpha \to \alpha \\
& = \lambda x.\lambda y.\mathsf{if}\ \mathit{key}\ x < \mathit{key}\ y\ \mathsf{then}\ y\ \mathsf{else}\ x \\
\mathsf{in} & \\
& \mathit{max}\ (R_{\mathsf{data,key}}\ 1\ a)\ (R_{\mathsf{data,key}}\ 2\ b)
\end{array}$$

Records can help to contain the number of overloaded identifiers in type signatures. The idea is to put related operations in a record which is constructed with a single overloaded identifier. The next example expresses shows how to model a simplified `Num` class in this way. In the Haskell-like syntax we use parentheses `(...)` instead of braces `{...}` for records.

```
type Num a = (plus :: a -> a -> a,
              minus:: a -> a -> a,
              neg  :: a -> a)
over num
inst num :: Int -> Num Int
     num =  ...

(+), (-) :: (num :: a -> Num a) => a -> a -> a
neg      :: (num :: a -> Num a) => a -> a
```

```
(+) x y = #plus  (num x) x y
(-) x y = #minus (num x) x y
neg x   = #neg   (num x) x
```

Note the similarity to dictionary passing. One shortcoming of this scheme with respect to Haskell's class declarations concerns subclassing. For instance, we could not pass a variable of type `(num :: a -> Num a) => a` to a function of type

```
(num :: a -> (plus  :: a -> a -> Bool,
              minus :: a -> a -> Bool)) => a -> b
```

Even without introducing full subtyping on records it may be helpful to supplement our system with some way for dealing with this common case. Further experience will be required to determine this.

6 Type Reconstruction

Figures 6 and 7 present type reconstruction and unification algorithm for System O. Compared to Milner's algorithm $\mathcal{W}$ [Mil78] there are two extensions.

- The case of binding a type variable in the unification algorithm is extended. To bind a type variable α to a type τ the constraints of Γ_α have to be satisfied. The function *mkinst* ensures that type τ statisfies the constraints Γ_α.
- The function *tp* is extended with a branch for instance declarations $\mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$. In this case it must be checked that the inferred type σ_T' for the overloading term e is less general then the given type σ_T.

We now state soundness and completeness results for the algorithms *unify* and *tp*. The proofs of these results are along the lines of [Che94]; they are omitted here.

We use the following abbreviations:

$$\begin{array}{lll}
\Gamma_\alpha & = & \{o : \alpha \to \tau \mid o : \alpha \to \tau \in \Gamma\} \\
\Gamma_A & = & \cup_{\alpha \in A}\, \Gamma_\alpha
\end{array}$$

where A is a set of type variables.

Definition. A *configuration* is a pair of a typothesis Γ and a substitution S such that, for all $\alpha \in \mathrm{dom}(S)$, $\Gamma_\alpha = \emptyset$.

$$
\begin{array}{l}
unify \; : \; (\tau, \tau) \to (\Gamma, S) \to (\Gamma, S) \\
unify\,(\tau_1, \tau_2)\,(\Gamma, S) \;\; = \;\; \mathsf{case}\ (S\tau_1, S\tau_2)\ \mathsf{of} \\
\quad (\alpha, \alpha) \Rightarrow \\
\quad\quad (\Gamma, S) \\
\quad (\alpha, \tau), (\tau, \alpha)\ \mathsf{where}\ \alpha \notin \mathrm{tv}(\tau) \Rightarrow \\
\quad\quad foldr\ mkinst\ (\Gamma \backslash \Gamma_\alpha, [\tau/\alpha] \circ S)\ \Gamma_\alpha \\
\quad (T\,\overline{\tau}_1, T\,\overline{\tau}_2) \Rightarrow \\
\quad\quad foldr\ unify\ (\Gamma, S)\ (zip\ (\overline{\tau}_1, \overline{\tau}_2))
\end{array}
$$

$$
\begin{array}{l}
mkinst \; : \; (o : \alpha \to \tau) \to (\Gamma, S) \to (\Gamma, S) \\
mkinst\,(o : \alpha \to \tau)\,(\Gamma, S) \;\; = \;\; \mathsf{case}\ S\alpha\ \mathsf{of} \\
\quad \beta \Rightarrow \\
\quad\quad \mathsf{if}\ \exists o : \beta \to \tau' \in \Gamma \\
\quad\quad \mathsf{then}\ \ unify\ (\tau, \tau')\ (\Gamma, S) \\
\quad\quad \mathsf{else}\ \ (\Gamma \cup \{o : \beta \to [\beta/\alpha]\tau\}, S) \\
\quad T\,\overline{\tau} \Rightarrow \\
\quad\quad \mathsf{case}\ \{newinst\ (\sigma_T, \Gamma, S) \mid o : \sigma_T \in \Gamma\}\ \mathsf{of} \\
\quad\quad\quad \{(\tau_1, \Gamma_1, S_1)\} \Rightarrow unify\ (\alpha \to \tau, \tau_1)\ (\Gamma_1, S_1)
\end{array}
$$

Figure 6: Algorithm for constrained unification

Definition. The following defines a preorder $\preceq$ on substitutions and configurations and a preorder $\preceq_\Gamma$ on type schemes. If $X \preceq Y$ we say that Y is *more general* than X.

- $S' \preceq S$ iff there is a substitution R such that $S' = R \circ S$.
- $(\Gamma', S') \preceq (\Gamma, S)$ iff $S' \preceq S$, $S'\Gamma' \;\vdash\; S'\Gamma_{\mathrm{dom}(S')}$ and $\Gamma' \supseteq \Gamma \setminus \Gamma_{\mathrm{dom}(S')}$.
- $\sigma' \preceq_\Gamma \sigma$ iff, for all $u \notin dom(\Gamma)$, $\Gamma \;\vdash\; u : \sigma$ implies $\Gamma \;\vdash\; u : \sigma'$.

Definition. A *constrained unification problem* is a pair of tuples $(\tau_1, \tau_2)(\Gamma, S)$ where τ_1, τ_2 are types and (Γ, S) is a configuration.

A configuration (Γ', S') is called a *unifying configuration* for $(\tau_1, \tau_2)(\Gamma, S)$ iff $(\Gamma', S') \preceq (\Gamma, S)$ and $S'\tau_1 = S'\tau_2$.

The unifying configuration (Γ', S') is *most general* iff $(\Gamma'', S'') \preceq (\Gamma', S')$, for every other unifying configuration (Γ'', S'').

Definition. A *typing problem* is a triple (p, Γ, S) where (Γ, S) is a configuration and p is a term or program with $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$.

A *typing solution* of a typing problem (p, Γ, S) is a triple (σ, Γ', S') where $(\Gamma', S') \preceq (\Gamma, S)$ and $S'\Gamma' \;\vdash\; p : S'\sigma$.

The typing solution (σ, Γ', S') is *most general* iff for every other typing solution $(\sigma'', \Gamma'', S'')$ it holds $(\Gamma'', S'') \preceq (\Gamma', S')$ and $S''\sigma'' \preceq_{S''\Gamma''} S''\sigma$.

Theorem 6.1 Let $(\tau_1, \tau_2)(\Gamma, S)$ be a constrained unification problem
(*a*) If $unify(\tau_1, \tau_2)(\Gamma, S) = (\Gamma', S')$ then (Γ', S') is a most general unifying configuration for $(\tau_1, \tau_2)(\Gamma, S)$.
(*b*) If $unify(\tau_1, \tau_2)(\Gamma, S)$ fails then there exists no unifying configuration for $(\tau_1, \tau_2)(\Gamma, S)$.

Theorem 6.2 Let (p, Γ, S) be a typing problem.
(*a*) If $tp\ (p, \Gamma, S) = (\sigma, \Gamma', S')$ then (σ, Γ', S') is a most general solution of (p, Γ, S).
(*b*) If $tp\ (p, \Gamma, S)$ fails, then (p, Γ, S) has no solution.

As a corollary of Theorem 6.2, we get that every typable program has a principal type, which is found by *tp*.

Corollary 6.3 (Principal Types) Let (p, Γ, id) be a typing problem such that $\mathrm{tv}(\Gamma) = \emptyset$.
(*a*) Assume $gen\ (tp\ (p, \Gamma, id)) \;=\; (\sigma', \Gamma', S)$ and let $\sigma = S\sigma'$. Then

$$
\begin{array}{ll}
\Gamma \;\vdash\; p : \sigma & \text{and} \\
\Gamma \;\vdash\; p : \sigma'' \Rightarrow \sigma'' \preceq_\Gamma \sigma, & \text{for all type schemes } \sigma''.
\end{array}
$$

(*b*) If $tp\ (p, \Gamma, id)$ fails then there is no type scheme σ such that $\Gamma \;\vdash\; p : \sigma$.

The termination of *unify* and *mkinst* critically depends on the form of overloaded type schemes σ_T:

$$
\begin{array}{rcll}
\sigma_T & = & T\,\alpha_1 \ldots \alpha_n \to \tau & (\mathrm{tv}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}) \\
& \mid & \forall \alpha . \pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \subseteq \mathrm{tv}(\sigma'_T)).
\end{array}
$$

We show with an example why σ_T needs to be parametric in the arguments of T. Consider the following program, where $k \in \mathcal{K}_T$.

$$
\begin{array}{rcl}
p & = & \mathsf{let}\ \ (;)\,x\,y = y\ \mathsf{in} \\
& & \mathsf{inst}\,o : \forall \alpha . o : \alpha \to \alpha \Rightarrow T(T\alpha) \to \alpha \\
& & \quad = \lambda k(k\,x).o\,x \\
& & \mathsf{in} \quad \lambda x.\lambda y.\lambda f.\ o\,x\,; o\,y\,; f\,(k\,y)\,; f x
\end{array}
$$

Then computation of $tp(p, \emptyset, id)$ leads to a call $tp(f\,x, \Gamma, S)$ with $x : \alpha, y : \beta, f : T\beta \to \delta \in \Gamma$. This leads in turn to a call $unify(\alpha, T\beta)(\Gamma, S)$ where the following assumptions hold:

- $\sigma_T \;=\; \forall \alpha . o : \alpha \to \alpha \Rightarrow T(T\alpha) \to \alpha$
- $\Gamma \;\supseteq\; \{o : \alpha \to \alpha, o : \beta \to \beta, o : \sigma_T\}$,
- S is a substitution with $\alpha, \beta \notin dom(S)$.

Unfolding *unify* gives $mkinst(o : \alpha \to \alpha)(\Gamma \setminus \Gamma_\alpha, S')$ where $S' = [T\beta/\alpha] \circ S$, which leads in turn to the following two calls:

1. $newinst(\sigma_T, \Gamma \setminus \Gamma_\alpha, S') \;=\; (T(T\gamma) \to \gamma, \Gamma', S')$ where $\Gamma' \;\supseteq\; \{o : \beta \to \beta, o : \gamma \to \gamma, o : \sigma_T\}$ and γ is a fresh type variable, and
2. $unify(\alpha \to \alpha, T(T\gamma) \to \gamma)(\Gamma', S')$.

Since $S'\alpha \;=\; T\beta$, unfolding of (2) results in an attempt to unify $T\beta$ and $T(T\gamma)$), which leads to the call $unify(\beta, T\gamma)(\Gamma', S')$. This is equivalent to the original call $unify(\alpha, T\beta)(\Gamma, S)$ modulo renaming of α, β to β, γ. Hence, *unify* would loop in this situation.

The need for the other restrictions on σ_T are shown by similar constructions. It remains to be seen whether a more general system is feasible that lifts these restrictions, *e.g.* by extending unification to regular trees [Kae92].

7 Conclusion

We have shown that a rather modest extension to the Hindley/Milner system is enough to support both overloading and polymorphic records with a limited form of F-bounded polymorphism. The resulting system stays firmly in the tradition of ML typing, with type soundness and principal type properties completely analogous to the Hindley/Milner system.

$$
\begin{array}{lcl}
newinst & : & (\sigma, \Gamma, S) \to (\tau, \Gamma, S) \\
\multicolumn{3}{l}{newinst\ (\forall\alpha.\pi_\alpha \Rightarrow \sigma, \Gamma, S)} \\
 & = & \mathsf{let}\ \ \beta \text{ a new type variable} \\
 & & \mathsf{in}\ \ \ newinst \\
 & & \qquad ([\beta/\alpha]\sigma, \Gamma \cup [\beta/\alpha]\pi_\alpha, S) \\
\multicolumn{3}{l}{newinst\ (\tau, \Gamma, S)} \\
 & = & (\tau, \Gamma, S)
\end{array}
$$

$$
\begin{array}{lcl}
skolemize & : & (\sigma, \Gamma, S) \to (\tau, \Gamma, S) \\
\multicolumn{3}{l}{skolemize\ (\forall\alpha.\pi_\alpha \Rightarrow \sigma, \Gamma, S)} \\
 & = & \mathsf{let}\ \ T \text{ a new 0-ary type constructor} \\
 & & \mathsf{in}\ \ \ skolemize \\
 & & \qquad ([T/\alpha]\sigma, \Gamma \cup [T/\alpha]\pi_\alpha, S) \\
\multicolumn{3}{l}{skolemize\ (\tau, \Gamma, S)} \\
 & = & (\tau, \Gamma, S)
\end{array}
$$

$$
\begin{array}{lcl}
gen & : & (\tau, \Gamma, S) \to (\sigma, \Gamma, S) \\
gen\ (\sigma, \Gamma, S) & = & \mathsf{if}\ \exists\alpha.\alpha \in \mathrm{tv}(S\sigma) \setminus \mathrm{tv}(S(\Gamma \backslash \Gamma_\alpha)) \\
 & & \mathsf{then}\ \ gen\ (\forall\alpha.\Gamma_\alpha \Rightarrow \sigma, \Gamma \backslash \Gamma_\alpha, S) \\
 & & \mathsf{else}\ \ (\sigma, \Gamma, S)
\end{array}
$$

$$
\begin{array}{lcl}
tp & : & (p, \Gamma, S) \to (\tau, \Gamma, S) \\
tp\ (u, \Gamma, S) & = & \mathsf{if}\ u : \sigma \in \Gamma \\
 & & \mathsf{then}\ newinst\ (\sigma, \Gamma, S) \\
tp\ (o, \Gamma, S) & = & newinst\ (\forall\beta\forall\alpha.(o : \alpha \to \beta) \Rightarrow \alpha \to \beta, \Gamma, S)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{tp\ (\lambda u.e, \Gamma, S)} \\
 = & \mathsf{let} & \alpha \text{ a new type variable} \\
 & & (\tau, \Gamma_1, S_1) = tp\ (e, \Gamma \cup \{u : \alpha\}, S) \\
 & \mathsf{in} & (\alpha \to \tau, \Gamma_1, S_1)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{tp\ (e\,e', \Gamma, S)} \\
 = & \mathsf{let} & (\tau_1, \Gamma_1, S_1) = tp\ (e, \Gamma, S) \\
 & & (\tau_2, \Gamma_2, S_2) = tp\ (e', \Gamma_1, S_1) \\
 & & \alpha \text{ a new type variable} \\
 & & (\Gamma_3, S_3) = unify\ (\tau_1, \tau_2 \to \alpha)\ (\Gamma_2, S_2) \\
 & \mathsf{in} & (\alpha, \Gamma_3, S_3)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{tp\ (\mathsf{let}\ u = e\ \mathsf{in}\ e', \Gamma, S)} \\
 = & \mathsf{let} & (\sigma, \Gamma_1, S_1) = gen\ (tp\ (e, \Gamma, S)) \\
 & \mathsf{in} & tp\ (e', \Gamma_1 \cup \{u : \sigma\}, S_1)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{tp\ (\mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p, \Gamma, S)} \\
 = & \mathsf{let} & (\sigma'_T, \Gamma_1, S_1) = gen\ (tp\ (e, \Gamma, S)) \\
 & & (\tau_2, \Gamma_2, S_2)\ =\ skolemize\ (\sigma_T, \Gamma_1, S_1) \\
 & & (\tau_3, \Gamma_3, S_3)\ =\ newinst\ (\sigma'_T, \Gamma_2, S_2) \\
 & \mathsf{in} & \mathsf{if}\ \forall o : \sigma_{T'} \in \Gamma.\ T \neq T' \wedge \\
 & & \quad unify(\tau_2, \tau_3)(\Gamma_3, S_3) \text{ defined } \mathsf{then} \\
 & & \qquad tp\ (p, \Gamma_1 \cup \{o : \sigma_T\}, S_1)
\end{array}
$$

Figure 7: Type reconstruction algorithm for System O

The encoding of a polymorphic record calculus in System O indicates that there might be some deeper relationships between F-bounded polymorphism and overloading. This is also suggested by the similarities between the dictionary transform for type classes and the Penn translation for bounded polymorphism [BTCGS91]. A study of these relationships remains a topic for future work.

Acknowledgments We are grateful to Kung Chen and John Maraist for valuable comments on previous drafts of this paper. The section on records was motivated in part by a discussion led by Simon Peyton Jones, Mark Jones and others on the Haskell mailing list. Many other discussions with numerous participants have also contributed to this work.

References

[App93] Andrew W. Appel. A critique of standard ML. *Journal of Functional Programming*, 3(4), 1993.

[Aug93] Lennart Augustsson. Implementing Haskell overloading. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 65–73, June 1993.

[Blo91] Stephen Blott. *An Approach to Overloading with Polymorphism*. PhD thesis, Department of Computer Science, University of Glasgow, Sept 1991.

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Che94] Kung Chen. *A Parametric Extension of Haskell's Type Classes*. PhD thesis, Yale University, New Haven, Connecticut, December 1994. YALEU/DCS/RR-1057.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, June 1992.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, January 1982.

[DO94] Dominic Duggan and John Ophel. Kinded parametric overloading. Technical Report CS-94-35, University of Waterloo, September 1994.

[DRW95] Catherine Dubois, Francois Rouaix, and Pierre Weis. Extensional polymorphism. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 118–129, January 1995.

[HHPW94] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *Proc. 5th European Symposium on Programming*, pages 241–256, 1994. Springer LNCS 788.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

[Jon92a] Mark P. Jones. *Qualified Types: Theory and Practice*. D.phil. thesis, Oxford University, September 1992.

[Jon92b] Mark P. Jones. A theory of qualified types. In *Proc. 4th European Symposium on Programming*, pages 287–306, February 1992. Springer LNCS 582.

[Jon93] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 52–61, June 1993.

[JT81] R.D. Jenks and B.M. Trager. A language for computational algebra. In *Proc. ACM Symposium on Symbolic and Algebraic Manipulation*, pages 22–29, 1981.

[Kae88] Stefan Kaes. Parametric overloading. In *Proc. 2nd European Symposium on Programming*. Springer-Verlag, 1988. Springer LNCS 300.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 193–204, June 1992.

[MH88] John C. Mitchell and Robert Harper. The essence of ML. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM, ACM Press, January 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 409–418, 1993.

[NS91] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 1–14, August 1991. Springer LNCS 523.

[Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 154–165, January 1992.

[Pet94] John Peterson. Structures in Yale Haskell. draft paper, 1994.

[PJ93] John Peterson and Mark Jones. Implementing type classes. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 227–236, June 1993. SIGPLAN Notices 28(6).

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM, January 1989.

[Rou90] François Rouaix. Safe run-time overloading. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 355–366, January 1990.

[Smi91] Geoffrey S. Smith. *Polymorphic type inference for languages with overloading and subtyping*. PhD thesis, Cornell University, Ithaca, NY, August 1991.

[SOM93] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. In *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993.

[Tha94] Satish R. Thatte. Semantics of type classes revisited. In *Proc. Conference on Lisp and Functional Programming*, pages 208–219, 1994.

[Vol93] Dennis Volpano. A critique of type systems for global overloading. Computer Science Technical Report NPSCS-94-006, Naval Postgraduate School, October 1993.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proc.IEEE Symposium on Logic in Computer Science*, pages 37–44, June 1987.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.