How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) and multiplication of floating point values (as in `3.14*3.14`).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the `length` function, which acts in the same way on a list of integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and translation, in the form of inference rules (as in [DM82]). The translation rules provide a semantics for type classes. They also provide one possible implementation technique: if desired, the new system could be added to an existing language with Hindley/Milner types simply by writing a pre-processor.

Two places where the issues of ad-hoc polymorphism arise are the definition of operators for arithmetic and equality. Below we examine the approaches to these three problems adopted by Standard ML and Miranda; not only do the approaches differ between the two languages, they also differ within a single language. But as we shall see, type classes provide a uniform mechanism that can address these problems.

*Authors' address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. Electronic mail: `wadler`, `blott@cs.glasgow.ac.uk`.

Published in: *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[1] Miranda is a trademark of Research Software Limited.

This work grew out of the efforts of the Haskell committee to design a lazy functional programming language[2]. One of the goals of the Haskell committee was to adopt "off the shelf" solutions to problems wherever possible. We were a little surprised to realise that arithmetic and equality were areas where no standard solution was available! Type classes were developed as an attempt to find a better solution to these problems; the solution was judged successful enough to be included in the Haskell design. However, type classes should be judged independently of Haskell; they could just as well be incorporated into another language, such as Standard ML.

Type classes appear to be closely related to issues that arise in object-oriented programming, bounded quantification of types, and abstract data types [CW85, MP85, Rey85]. Some of the connections are outlined below, but more work is required to understand these relations fully.

A type system very similar to ours has been discovered independently by Stefan Kaes [Kae88]. Our work improves on Kaes' in several ways, notably by the introduction of type classes to group related operators, and by providing a better translation method.

This paper is divided into two parts: the body gives an informal introduction to type classes, while the appendix gives a more formal description. Section 2 motivates the new system by describing limitations of *ad-hoc* polymorphism as it is used in Standard ML and Miranda. Section 3 introduces type classes by means of a simple example. Section 4 illustrates how the example of Section 3 may be translated into an equivalent program without type classes. Section 5 presents a second example, the definition of an overloaded equality function. Section 6 describes subclasses. Section 7 discusses related work and concludes. Appendix A presents inference rules for typing and translation.

2 Limitations of ad-hoc polymorphism

This section motivates our treatment of *ad-hoc* polymorphism, by examining problems that arise with arithmetic and equality in Standard ML and Miranda.

Arithmetic. In the simplest approach to overloading, basic operations such as addition and multiplication are overloaded, but functions defined in terms of them are not. For example, although one can write `3*3` and `3.14*3.14`, one *cannot* define

```
square x  =  x*x
```

and then write terms such as

```
square 3
square 3.14
```

This is the approach taken in Standard ML. (Incidentally, it is interesting to note that although Standard ML includes overloading of arithmetic operators, its formal definition is deliberately ambiguous about how this overloading is resolved [HMT88, page 71], and different versions of Standard ML resolve overloading in different ways.)

A more general approach is to allow the above equation to stand for the definition of two overloaded versions of `square`, with types `Int -> Int` and `Float -> Float`. But consider the function:

```
squares (x, y, z)
    =  (square x, square y, square z)
```

Since each of `x`, `y`, and `z` might, independently, have either type `Int` or type `Float`, there are eight possible overloaded versions of this function. In general, there may be exponential growth in the number of translations, and this is one reason why such solutions are not widely used.

In Miranda, this problem is side-stepped by not overloading arithmetic operations. Miranda provides only the floating point type (named "`num`"), and there is no way to use the type system to indicate that an operation is restricted to integers.

Equality. The history of the equality operation is checkered: it has been treated as overloaded, fully polymorphic, and partly polymorphic.

The first approach to equality is to make it overloaded, just like multiplication. In particular, equality may be overloaded on every monotype that *admits equality*, i.e., does not contain an abstract type or a function type. In such a language, one may write `3*4 == 12` to denote equality over integers, or `'a' == 'b'` to denote equality over characters. But one *cannot* define a function `member` by the equations

```
member [] y     = False
member (x:xs) y = (x == y) \/ member xs y
```

[2] The Haskell committee includes: Arvind, Brian Boutel, Jon Fairbairn, Joe Fasel, Paul Hudak, John Hughes, Thomas Johnsson, Dick Kieburtz, Simon Peyton Jones, Rishiyur Nikhil, Mike Reeve, Philip Wadler, David Wise, and Jonathan Young.

and then write terms such as

```
member [1,2,3] 2
member "Haskell" 'k'
```

(We abbreviate a list of characters `['a','b','c']` as `"abc"`.) This is the approach taken in the first version of Standard ML [Mil84].

A second approach is to make equality fully polymorphic. In this case, its type is

```
(==) :: a -> a -> Bool
```

where `a` is a type variable ranging over every type. The type of the `member` function is now

```
member :: [a] -> a -> Bool
```

(We write `[a]` for the type "list of `a`".) This means that applying equality to functions or abstract types does not generate a type error. This is the approach taken in Miranda: if equality is applied on a function type, the result is a run-time error; if equality is applied on an abstract type, the result is to test the *representation* for equality. This last may be considered a bug, as it violates the principle of abstraction.

A third approach is to make equality polymorphic in a limited way. In this case, its type is

$$\texttt{(==) ::}\ \ \texttt{a}_{\texttt{(==)}}\texttt{->}\ \texttt{a}_{\texttt{(==)}}\texttt{->}\ \texttt{Bool}$$

where $\texttt{a}_{\texttt{(==)}}$ is a type variable ranging only over types that admit equality. The type of the `member` function is now

$$\texttt{member ::}\ \ \texttt{[a}_{\texttt{(==)}}\texttt{]}\ \texttt{->}\ \texttt{a}_{\texttt{(==)}}\texttt{->}\ \texttt{Bool}$$

Applying equality, or `member`, on a function type or abstract type is now a type error. This is the approach currently taken in Standard ML, where $\texttt{a}_{\texttt{(==)}}$ is written `''a`, and called an "eqtype variable".

Polymorphic equality places certain demands upon the implementor of the run-time system. For instance, in Standard ML reference types are tested for equality differently from other types, so it must be possible at run-time to distinguish references from other pointers.

Object-oriented programming. It would be nice if polymorphic equality could be extended to include user-defined equality operations over abstract types. To implement this, we would need to require that every object carry with it a pointer to a *method*, a procedure for performing the equality test. If we are to have more than one operation with this property, then each object should carry with it a pointer to a *dictionary* of appropriate methods. This is exactly the approach used in object-oriented programming [GR83].

In the case of polymorphic equality, this means that *both* arguments of the equality function will contain a pointer to the same dictionary (since they are both of the same type). This suggests that perhaps dictionaries should be passed around independently of objects; now polymorphic equality would be passed one dictionary and two objects (minus dictionaries). This is the intuition behind type classes and the translation method described here.

3 An introductory example

We will now introduce type classes by means of an example.

Say that we wish to overload `(+)`, `(*)`, and `negate` (unary minus) on types `Int` and `Float`. To do so, we introduce a new *type class*, called `Num`, as shown in the `class` declaration in Figure 1. This declaration may be read as stating "a type `a` belongs to class `Num` if there are functions named `(+)`, `(*)`, and `negate`, of the appropriate types, defined on it."

We may now declare instances of this class, as shown by the two `instance` declarations in Figure 1. The assertion `Num Int` may be read "there are functions named `(+)`, `(*)`, and `negate`, of the appropriate types, defined on `Int`". The instance declaration justifies this assertion by giving appropriate bindings for the three functions. The type inference algorithm must verify that these bindings do have the appropriate type, i.e., that `addInt` has type `Int->Int->Int`, and similarly for `mulInt` and `negInt`. (We assume that `addInt`, `mulInt`, and `negInt` are defined in the standard prelude.) The instance `Num Float` is declared similarly.

A word on notational conventions: Type class names and type constructor names begin with a capital letter, and type variable names begin with a small letter. Here, `Num` is a type class, `Int` and `Float` are type constructors, and `a` is a type variable.

We may now define

```
square x  =  x * x
```

There exists an algorithm that can infer the type of `square` from this definition (it is outlined in the appendix). It derives the type:

```
square :: Num a => a -> a
```

```
class  Num a  where
   (+), (*)  ::  a -> a -> a
   negate    ::  a -> a

instance  Num Int  where
   (+)     =  addInt
   (*)     =  mulInt
   negate  =  negInt

instance  Num Float  where
   (+)     =  addFloat
   (*)     =  mulFloat
   negate  =  negFloat

square             :: Num a => a -> a
square x           =  x * x

squares            :: Num a, Num b, Num c => (a,b,c) -> (a,b,c)
squares (x, y, z)  =  (square x, square y, square z)
```

Figure 1: Definition of arithmetic operations

```
data  NumD a  =  NumDict (a -> a -> a) (a -> a -> a) (a -> a)

add (NumDict a m n)  =  a
mul (NumDict a m n)  =  m
neg (NumDict a m n)  =  n

numDInt    :: NumD Int
numDInt    =  NumDict addInt mulInt negInt

numDFloat  :: NumD Float
numDFloat  =  NumDict addFloat mulFloat negFloat

square'         :: NumD a -> a -> a
square' numDa x  =  mul numDa x x

squares'  :: (NumD a, NumD b, NumD c) -> (a,b,c) -> (a,b,c)
squares' (numDa, numDb, numDc) (x, y, z)
    =  (square' numDa x, square' numDb y, square' numDc z)
```

Figure 2: Translation of arithmetic operations

This is read, "`square` has type `a -> a`, for every `a` such that `a` belongs to class `Num` (i.e., such that `(+)`, `(*)`, and `negate` are defined on `a`)." We can now write terms such as

```
square 3
square 3.14
```

and an appropriate type will be derived for each (`Int` for the first expression, `Float` for the second). On the other hand, writing `square 'x'` will yield a type error at compile time, because `Char` has not been asserted (via an instance declaration) to be a numeric type.

Finally, if we define the function `squares` mentioned previously, then the type given in Figure 1 will be inferred. This type may be read, "`squares` has the type `(a,b,c) -> (a,b,c)` for every `a`, `b`, and `c` such that `a`, `b`, and `c` belong to class `Num`". (We write `(a,b,c)` for the type that is the cartesian product of `a`, `b`, and `c`.) So `squares` has one type, not eight. Terms such as

```
squares (1, 2, 3.14)
```

are legal, and derive an appropriate type.

4 Translation

One feature of this form of overloading is that it is possible at compile-time to translate any program containing class and instance declarations to an equivalent program that does not. The equivalent program will have a valid Hindley/Milner type.

The translation method will be illustrated by means of an example. Figure 2 shows the translation of the declarations in Figure 1.

For each class declaration we introduce a new type, corresponding to an appropriate "method dictionary" for that class, and functions to access the methods in the dictionary. In this case, corresponding to the class `Num` we introduce the type `NumD` as shown in Figure 2. The `data` declaration defines `NumD` to be a type constructor for a new type. Values of this type are created using the value constructor `NumDict`, and have three components of the types shown. The functions `add`, `mul`, and `neg` take a value of type `NumD` and return its first, second, and third component, respectively.

Each instance of the class `Num` is translated into the declaration of a value of type `NumD`. Thus, corresponding to the instance `Num Int` we declare a data structure of type `NumD Int`, and similarly for `Float`.

Each term of the form `x+y`, `x*y`, and `negate x` is now replaced by a corresponding term, as follows:

```
x+y       -->  add numD x y
x*y       -->  mul numD x y
negate x  -->  neg numD x
```

where `numD` is an appropriate dictionary. How is the appropriate dictionary determined? By its type. For example, we have the following translations:

```
3 * 3
  -->  mul numDInt 3 3
3.14 * 3.14
  -->  mul numDFloat 3.14 3.14
```

As an optimisation, it is easy for the compiler to perform beta reductions to transform these into `mulInt 3 3` and `mulFloat 3.14 3.14`, respectively.

If the type of a function contains a class, then this is translated into a dictionary that is passed at run-time. For example, here is the definition of `square` with its type

```
square          :: Num a => a -> a
square x        =  x * x
```

This translates to

```
square'         :: NumD a -> a -> a
square' numD x  =  mul numD x x
```

Each application of `square` must be translated to pass in the appropriate extra parameter:

```
square 3
  -->  square' numDInt 3
square 3.2
  -->  square' numDFloat 3
```

Finally, the translation of `squares` is also shown in Figure 2. Just as there is one type, rather than eight, there is only one translation, rather than eight. Exponential growth is avoided.

5 A further example: equality

This section shows how to define equality using class and instance declarations. Type classes serve as a straightforward generalisation of the "eqtype variables" used in Standard ML. Unlike Standard ML, this mechanism allows the user to extend equality over abstract types in a straightforward way. And, unlike Standard ML, this mechanism can be translated out at compile time, so it places no special demands on the implementor of the run-time system.

```
class  Eq a  where
  (==) :: a -> a -> bool

instance  Eq Int  where
  (==)  =  eqInt

instance  Eq Char  where
  (==)  =  eqChar

member                :: Eq a => [a] -> a -> Bool
member [] y           =  False
member (x:xs) y       =  (x == y) \/ member xs y

instance  Eq a, Eq b => Eq (a,b)  where
  (u,v) == (x,y)      =  (u == x) & (v == y)

instance  Eq a => Eq [a]  where
  [] == []            =  True
  [] == y:ys          =  False
  x:xs  ==  []        =  False
  x:xs  ==  y:ys      =  (x == y) & (xs == ys)

data  Set a  =  MkSet [a]

instance  Eq a => Eq (Set a)  where
  MkSet xs == MkSet ys  =  and (map (member xs) ys)
                  & and (map (member ys) xs)
```

Figure 3: Definition of equality

The definition is summarised in Figure 3. We begin by declaring a class, `Eq`, containing a single operator, `(==)`, and instances `Eq Int` and `Eq Char` of this class.

We then define the `member` function in the usual way, as shown in Figure 3. The type of `member` need not be given explicitly, as it can be inferred. The inferred type is:

```
member :: Eq a => [a] -> a -> Bool
```

This is read "`member` has type `[a] -> a -> Bool`, for every type `a` such that `a` is in class `Eq` (i.e., such that equality is defined on `a`)" (This is exactly equivalent to the Standard ML type `''a list->''a->bool`, where `''a` is an "eqtype variable".) We may now write terms such as

```
member [1,2,3] 2
member "Haskell" 'k'
```

which both evaluate to `True`.

Next, we give an instance defining equality over pairs. The first line of this instance reads, "for every `a` and `b` such that `a` is in class `Eq` and `b` is in class `Eq`, the pair `(a,b)` is also in class `Eq`." In other words, "if equality is defined on `a` and equality is defined on `b`, then equality is defined on `(a,b)`." The instance defines equality on pairs in terms of equality on the two components, in the usual way.

Similarly, it is possible to define equality over lists. The first line of this instance reads, "if equality is defined on `a`, then equality is defined on type 'list of `a`'." We may now write terms such as

```
"hello" == "goodbye"
[[1,2,3],[4,5,6]]  ==  []
member ["Haskell", "Alonzo"] "Moses"
```

which all evaluate to `False`.

The final data declaration defines a new type constructor `Set` and a new value constructor `MkSet`. If a module exports `Set` but hides `MkSet`, then outside of the module the representation of `Set` will not be accessible; this is the mechanism used in Haskell to define abstract data types. The final instance defines equality over sets. The first line of this instance reads, "if equality is defined on `a`, then equality is

```
data  EqD a     =  EqDict (a -> a -> Bool)

eq (EqDict e)   =  e

eqDInt          :: EqD Int
eqDInt          =  EqDict eqInt

eqDChar         :: EqD Int
eqDChar         =  EqDict eqChar

member'                          :: EqD a -> [a] -> a -> Bool
member' eqDa [] y                =  False
member' eqDa (x:xs) y            =  eq eqDa x y \/ member' eqDa xs y

eqDPair                          :: (EqD a, EqD b) -> EqD (a,b)
eqDPair (eqDa,eqDb)              =  EqDict (eqPair (eqDa,eqDb))

eqPair                           :: (EqD a, EqD b) -> (a,b) -> (a,b) -> Bool
eqPair (eqDa,eqDb) (x,y) (u,v)   =  eq eqDa x u & eq eqDb y v

eqDList                          :: EqD a -> EqD [a]
eqDList eqDa                     =  EqDict (eqList eqDa)

eqList                           :: EqD a -> [a] -> [a] -> Bool
eqList eqDa [] []                =  True
eqList eqDa [] (y:ys)            =  False
eqList eqDa (x:xs) []            =  False
eqList eqDa (x:xs) (y:ys)        =  eq eqDa x y & eq (eqDList eqDa) xs ys
```

Figure 4: Translation of equality

defined on type 'set of `a`'." In this case, sets are represented in terms of lists, and two sets are taken to be equal if every member of the first is a member of the second, and vice-versa. (The definition uses standard functions `map`, which applies a function to every element of a list, and `and`, which returns the conjunction of a list of booleans.) Because set equality is defined in terms of `member`, and `member` uses overloaded equality, it is valid to apply equality to sets of integers, sets of lists of integers, and even sets of sets of integers.

This last example shows how the type class mechanism allows overloaded functions to be defined over abstract data types in a natural way. In particular, this provides an improvement over the treatment of equality provided in Standard ML or Miranda.

5.1 Translation of equality

We now consider how the translation mechanism applies to the equality example.

Figure 4 shows the translation of the declarations in Figure 3. The first part of the translation introduces nothing new, and is similar to the translation in Section 4.

We begin by defining a dicitionary `EqD` corresponding to the class `Eq`. In this case, the class contains only one operation, `(==)`, so the dictionary has only one entry. The selector function `eq` takes a dictionary of type `EqD a` and returns the one entry, of type `a->a->Bool`. Corresponding to the instances `Eq Int` and `Eq Char` we define two dictionaries of types `EqD Int` and `EqD Char`, containing the appropriate equality functions, and the function `member` is translated to `member'` in a straightforward way. Here are three terms and their translations:

```
3*4 == 12
  -->  eq eqDInt (mul numDInt 3 4) 12

member [1,2,3] 2
  -->  member' eqDInt [1,2,3] 2

member "Haskell" 'k'
  -->  member' eqDChar "Haskell" 'k'
```

The translation of the instance declaration for equality over lists is a little trickier. Recall that the instance declaration begins

```
instance  Eq a => Eq [a]  where
  ...
```

This states that equality is defined over type `[a]` if equality is defined over type `a`. Corresponding to this, the instance dictionary for type `[a]` is parameterised by a dictionary for type `a`, and so has the type

```
eqDList :: EqD a -> EqD [a]
```

The remainder of the translation is shown in Figure 4, as is the translation for equality over pairs. Here are three terms and their translations:

```
"hello" == "goodbye"
  -->  eq (eqDList eqDChar)
           "hello"
           "goodbye"

[[1,2,3],[4,5,6]]  ==  []
  -->  eq (eqDList (eqDList eqDInt))
           [[1,2,3],[4,5,6]]
           []

member ["Haskell", "Alonzo"] "Moses"
  -->  member' (eqDList eqDChar)
         ["Haskell", "Alonzo"]
         "Moses"
```

As an optimisation, it is easy for the compiler to perform beta reductions to transform terms of the form `eq (eqDList eqD)` into `eqList eqD`, where `eqD` is any dictionary for equality. This optimisation may be applied to the first two examples above, and also to the definition of `eqList` itself in Figure 4.

It is worthwhile to compare the efficiency of this translation technique with polymorphic equality as found in Standard ML or Miranda. The individual operations, such as `eqInt` are slightly more efficient than polymorphic equality, because the type of the argument is known in advance. On the other hand, operations such as `member` and `eqList` must explicitly pass an equality operation around, an overhead that polymorphic equality avoids. Further experience is needed to asses the trade-off between these costs.

6 Subclasses

In the preceeding, `Num` and `Eq` were considered as completely separate classes. If we want to use both numerical and equality operations, then these each appear in the type separately:

```
memsq :: Eq a, Num a => [a]->a->Bool
memsq xs x  =  member xs (square x)
```

As a practical matter, this seems a bit odd—we would expect every data type that has `(+)`, `(*)`, and `negate` defined on it to have `(==)` defined as well; but not the converse. Thus it seems sensible to make `Num` a subclass of `Eq`.

We can do this as follows:

```
class  Eq a => Num a  where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
```

This asserts that `a` may belong to class `Num` only if it also belongs to class `Eq`. In other words, `Num` is a subclass of `Eq`, or, equivalently, `Eq` is a superclass of `Num`. The instance declarations remain the same as before—but the instance declaration `Num Int` is only valid if there is also an instance declaration `Eq Int` active within the same scope.

From this it follows that whenever a type contains `Num a` it must also contain `Eq a`; therefore as a convenient abbreviation we permit `Eq a` to be omitted from a type whenever `Num a` is present. Thus, for the type of `memsq` we could now write

```
memsq :: Num a => [a]->a->Bool
```

The qualifier `Eq a` no longer needs to be mentioned, because it is implied by `Num a`.

In general, each class may have any number of sub or superclasses. Here is a contrived example:

```
class  Top a  where
  fun1 :: a -> a

class  Top a => Left a  where
  fun2 :: a -> a

class  Top a => Right a  where
  fun3 :: a -> a

class  Left a, Right a => Bottom a
  where
  fun4 :: a -> a
```

The relationships among these types can be diagrammed as follows:

```
       Top
      /   \
     /     \
```

```
      Left   Right
        \     /
         \   /
         Bottom
```

Although multiple superclasses pose some problems for the usual means of implementing object-oriented languages, they pose no problems for the translation scheme outlined here. The translation simply assures that the appropriate dictionaries are passed at run-time; no special hashing schemes are required, as in some object-oriented systems.

7 Conclusion

It is natural to think of adding assertions to the class declaration, specifying properties that each instance must satisfy:

```
class  Eq a  where
  (==)  ::  a  -> a -> Bool
  %  (==) is an equivalence relation

class  Num a  where
  zero, one ::  a
  (+), (*)  ::  a -> a -> a
  negate    ::  a -> a
  %  (zero, one, (+), (*), negate)
  %    form a ring
```

It is valid for any proof to rely on these properties, so long as one proves that they hold for each instance declaration. Here the assertions have simply been written as comments; a more sophisticated system could perhaps verify or use such assertions. This suggests a relation between classes and object-oriented programming of a different sort, since class declarations now begin to resemble object declarations in OBJ [FGJM85].

It is possible to have overloaded constants, such as `zero` and `one` in the above example. However, unrestricted overloading of constants leads to situations where the overloading cannot be resolved without providing extra type information. For instance, the expression `one * one` is meaningless unless it is used in a context that specifies whether its result is an `Int` or a `Float`. For this reason, we have been careful in this paper to use constants that are not overloaded: `3` has type `Int`, and `3.14` has type `Float`. A more general treatment of constants seems to require coercion between subtypes.

It is reasonable to allow a class to apply to more than one type variable. For instance, we might have

```
class  Coerce a b  where
  coerce :: a -> b

instance  Coerce Int Float  where
  coerce = convertIntToFloat
```

In this case, the assertion `Coerce a b` might be taken as equivalent to the assertion that `a` is a subtype of `b`. This suggests a relation between this work and work on bounded quantification and on subtypes (see [CW85, Rey85] for excellent surveys of work in this area, and [Wan87, Car88] for more recent work).

Type classes may be thought of as a kind of bounded quantifier, limiting the types that a type variable may instantiate to. But unlike other approaches to bounded quantification, type classes do not introduce any implicit coercions (such as from subtype `Int` to supertype `Float`, or from a record with fields `x`, `y`, and `z` to a record with fields `x` and `y`). Further exploration of the relationship between type classes and these other approaches is likely to be fruitful.

Type classes also may be thought of as a kind of abstract data type. Each type class specifies a collection of functions and their types, but not how they are to be implemented. In a way, each type class corresponds to an abstract data type with many implementations, one for each instance declaration. Again, exploration of the relationship between type classes and current work on abstract data types [CW85, MP85, Rey85] appears to be called for.

We have already referred to the work of Kaes. One advance of our work over his is the conceptual and notational benefit of grouping overloaded functions into classes. In addition, our system is more general; Kaes cannot handle overloadings involving more than one type variable, such as the `coerce` example above. Finally, our translation rules are an improvement over his. Kaes outlines two sets of translation rules (which he calls "semantics"), one static and one dynamic. His dynamic semantics is more limited in power than the language described here; his static semantics appears similar in power, but, unlike the translation described here, can greatly increase the size of a program.

One drawback of our translation method is that it introduces new parameters to be passed at run-time, corresponding to method dictionaries. It may be possible to eliminate some of these costs by using partial evaluation [BEJ88] to generate versions of functions specialised for certain dictionaries; this would reduce run time at the cost of increasing code size. Further work is needed to assess the trade-offs between our approach (with or without partial evaluation) and other techniques.

It is clear from the above that many issues remain to be explored, and many tradeoffs remain to be assessed. We look forward to the practical experience with type classes that Haskell will provide.

Acknowledgements. The important idea that overloading might be reflected in the type of a function was suggested (in a rather different form) by Joe Fasel. For discussion and comments, we are also grateful to: Luca Cardelli, Bob Harper, Paul Hudak, John Hughes, Stefan Kaes, John Launchbury, John Mitchell, Kevin Mitchell, Nick Rothwell, Mads Tofte, David Watt, the members of the Haskell committee, and the members of IFIP 2.8.

A Typing and translation rules

This appendix presents the formal typing and translation rules, one set of rules performing both typing and translation. The rules are an extension of those given by Damas and Milner [DM82].

A.1 Language

To present the typing and translation rules for overloading, it is helpful to use a slightly simpler language that captures the essential issues. We will use a language with the usual constructs (identifiers, applications, lambda abstractions, and **let** expressions), plus two new constructs, **over** and **inst** expressions, that correspond to `class` and `instance` declarations, respectively. The syntax of expressions and types is given in Figure 5.

An **over** expression

$$\textbf{over}\ x :: \sigma\ \textbf{in}\ e$$

declares x to be an overloaded identifier. Within the scope of this declaration, there may be one or more corresponding **inst** expressions

$$\textbf{inst}\ x :: \sigma' = e_0\ \textbf{in}\ e_1$$

where the type σ' is an instance of the type σ (a notion to be made precise later). Unlike lambda and **let** expressions, the bound variables in **over** and **inst** expressions may not be redeclared in a smaller scope. Also unlike lambda and **let** expressions, **over** and **inst** expressions must contain explicit types; the types in other expressions will be inferred by the rules given here.

As an example, a portion of the definition of equality given in Figure 3 is shown in Figure 6. In this figure, and in the rest of this appendix, we use $Eq\,\tau$ as an abbreviation for the type $\tau \rightarrow \tau \rightarrow Bool$.

As a second example, a portion of the definition of arithmetic operators given in Figure 1 is shown in Figure 7. In this figure we use $Num\,\tau$ as an abbreviation for the type

$$(\tau \rightarrow \tau \rightarrow \tau,\ \tau \rightarrow \tau \rightarrow \tau,\ \tau \rightarrow \tau)$$

In translating to the formal language, we have grouped the three operators together into a "dictionary". This is straightforward, and independent of the central issue: how to resolve overloading.

A.2 Types

The Damas/Milner system distinguishes between types (written τ) and type schemes (written σ). Our system adds a third syntactic group, *predicated types*. The syntax of these is given in Figure 5.

In the full language, we wrote types such as

```
member :: Eq a => [a] -> a -> Bool
```

In the simplified language, we write this in the form

$$member :: \forall\alpha.\ (eq :: Eq\,\alpha).\ [\alpha] \rightarrow \alpha \rightarrow Bool$$

The restriction `Eq a` can be read "equality is defined on type `a`" and the corresponding restriction $(eq :: Eq\,\alpha)$ can be read "*eq* must have an instance of type $Eq\,\alpha$".

In general, we refer to $(x :: \tau).\,\rho$ as a *predicated type* and $(x :: \tau)$ as a *predicate*.

We will give rules for deriving *typings* of the form

$$A \vdash e :: \sigma \setminus \overline{e}$$

This can be read as, "under the set of assumptions A, the expression e has well-typing σ with translation $\overline{e}$". Each typing also includes a translation, so the rules derive typing\translation pairs. It is possible to present the typing rules without reference to the translation, simply by deleting the '$\setminus\overline{e}$' portion from all rules. It is not, however, possible to present the translation rules independently, since typing controls the translation. For example, the introduction and elimination of predicates in types controls the introduction and elimination of lambda abstractions in translations.

Identifiers	x		
Expressions	e	$::=$	x
		$\mid$	$e_0\ e_1$
		$\mid$	$\lambda x.\, e$
		$\mid$	$\textbf{let}\ x = e_0\ \textbf{in}\ e_1$
		$\mid$	$\textbf{over}\ x :: \sigma\ \textbf{in}\ e$
		$\mid$	$\textbf{inst}\ x :: \sigma = e_0\ \textbf{in}\ e_1$
Type Variables	α		
Type Constructors	χ		
Types	τ	$::=$	$(\tau \rightarrow \tau') \mid \alpha \mid \chi(\tau_1 \ldots \tau_n)$
Predicated Types	ρ	$::=$	$(x :: \tau).\, \rho \mid \tau$
Type-schemes	σ	$::=$	$\forall \alpha.\, \sigma \mid \rho$

Figure 5: Syntax of expressions and types

$$
\begin{array}{l}
\textbf{over}\ eq :: \forall\alpha.\ Eq\,\alpha\ \textbf{in} \\
\textbf{inst}\ eq\ \ :: Eq\ Int = eqInt\ \textbf{in} \\
\textbf{inst}\ eq\ \ :: Eq\ Char = eqChar\ \textbf{in} \\
\textbf{inst}\ eq\ \ :: \forall\alpha.\forall\beta.(eq :: Eq\,\alpha).(eq :: Eq\,\beta).Eq\,(\alpha,\beta) \\
\qquad\qquad = \lambda p.\lambda q.\ eq\ (fst\ p)\ (fst\ q) \wedge eq\ (snd\ p)\ (snd\ q)\ \ \textbf{in} \\
eq\ (1, \text{`a'})\ (2, \text{`b'})
\end{array}
$$

Figure 6: Definition of equality, formalised

$$
\begin{array}{l}
\textbf{over}\ numD :: \forall\alpha.\ Num\,\alpha\ \textbf{in} \\
\textbf{inst}\ numD\ \ :: Num\ Int = (addInt, mulInt, negInt)\ \textbf{in} \\
\textbf{inst}\ numD\ \ :: Num\ Float = (addFloat, mulFloat, negFloat)\ \textbf{in} \\
\textbf{let}\ (+) \qquad = fst\ \ numD\ \textbf{in} \\
\textbf{let}\ (*) \qquad = snd\ \ numD\ \textbf{in} \\
\textbf{let}\ negate = thd\ \ numD\ \textbf{in} \\
\textbf{let}\ square = \lambda x.\ x * x\ \textbf{in} \\
square\ 3
\end{array}
$$

Figure 7: Definition of arithmetic operations, formalised

$$
\begin{array}{l}
(eq ::_o \forall\alpha. Eq\,\alpha),\\
(eq ::_i Eq\ Int \setminus eq_{(Eq\ Int)}),\\
(eq ::_i Eq\ Char \setminus eq_{(Eq\ Char)}),\\
(eq ::_i \forall\alpha.\forall\beta.(eq :: Eq\,\alpha).(eq :: Eq\,\beta).Eq\,(\alpha,\beta) \setminus eq_{(\forall\alpha.\forall\beta.(eq::Eq\,\alpha).(eq::Eq\,\beta).Eq\,(\alpha,\beta))}),\\
(eq :: Eq\,\alpha \setminus eq_{(Eq\,\alpha)}),\\
(eq :: Eq\,\beta \setminus eq_{(Eq\,\beta)}),\\
(p :: (\alpha,\beta) \setminus p),\\
(q :: (\alpha,\beta) \setminus q)
\end{array}
$$

Figure 8: Some assumptions

TAUT $A, (x :: \sigma \setminus \overline{x}) \vdash x :: \sigma \setminus \overline{x}$

TAUT $A, (x ::_i \sigma \setminus \overline{x}) \vdash x :: \sigma \setminus \overline{x}$

SPEC
$$\frac{A \vdash e :: \forall\alpha.\ \sigma \setminus \overline{e}}{A \vdash e :: [\alpha \setminus \tau]\sigma \setminus \overline{e}}$$

GEN
$$\frac{\begin{array}{l} A \vdash e :: \sigma \setminus \overline{e} \\ \alpha \text{ not free in } A \end{array}}{A \vdash e :: \forall\alpha.\ \sigma \setminus \overline{e}}$$

COMB
$$\frac{\begin{array}{l} A \vdash e :: (\tau' \to \tau) \setminus \overline{e} \\ A \vdash e' :: \tau' \setminus \overline{e}' \end{array}}{A \vdash (e\ e') :: \tau \setminus (\overline{e}\ \overline{e}')}$$

ABS
$$\frac{A_x, (x :: \tau' \setminus x) \vdash e :: \tau \setminus \overline{e}}{A \vdash (\lambda x.\, e) :: (\tau' \to \tau) \setminus (\lambda x.\, \overline{e})}$$

LET
$$\frac{\begin{array}{l} A \vdash e :: \sigma \setminus \overline{e} \\ A_x, (x :: \sigma \setminus x) \vdash e' :: \tau \setminus \overline{e}' \end{array}}{A \vdash (\textbf{let}\ x = e\ \textbf{in}\ e') :: \tau \setminus (\textbf{let}\ x = \overline{e}\ \textbf{in}\ \overline{e}')}$$

Figure 9: Typing and translation rules, part 1

A.3 Assumptions

Typing is done in the context of a set of assumptions, A. The assumptions bind typing and translation information to the free identifiers in an expression. This includes identifiers bound in lambda and **let** expression, and overloaded identifiers. Although we write them as sequences, assumptions are sets, and therefore the order is irrelevant.

There are three forms of binding in an assumption list:

- $(x ::_o \sigma)$ is used for overloaded identifiers;
- $(x ::_i \sigma \setminus x_\sigma)$ is used for declared instances of overloaded identifiers; and
- $(x :: \sigma \setminus \overline{x})$ is used for lambda and **let** bound variables, and assumed instances of overloaded identifiers.

In $(x :: \sigma \setminus \overline{x})$ and $(x ::_i \sigma \setminus \overline{x})$, the identifier $\overline{x}$ is the translation of x. If x is not an overloaded identifier (that is, if x is bound by a lambda or let expression), then the assumption for x has the form $(x :: \sigma \setminus x)$, so x simply translates as itself.

Figure 8 shows the assumptions available when applying the inference rules to the expression

$$\lambda p.\ \lambda q.\ eq\ (fst\ p)\ (fstq) \wedge eq\ (snd\ p)\ (sndq)$$

in Figure 6. There are three $(::_i)$ bindings, corresponding to the three instance declarations, and two $(::)$ bindings for the two bound variables, and two $(::)$ bindings corresponding to assumed instances of equality. (We shall see later how assumed instances are introduced by the **PRED** rule.)

A.4 Instances

Given a set of assumptions A, we define an instance relation between type-schemes,

$$\sigma \geq_A \sigma'.$$

This can be read as "σ is more general than σ' under assumptions A". This is the same as the relationship defined by Damas and Milner, but extended to apply to predicated types.

Only certain sets of assumptions are valid. The definition of validity depends on the $\geq_A$ relation, so there is a (well-founded) mutual recursion between the definition of valid assumptions and the definition of $\geq_A$. We give the definition of $\geq_A$ in this section, and the definition of valid assumptions in the next.

The instance relation

$$\sigma \geq_A \sigma'$$

where $\sigma = \forall\alpha_1 \ldots \alpha_n . \rho$ and $\sigma' = \forall\beta_1 \ldots \beta_m . \rho'$, is defined as follows:

$\sigma \geq_A \sigma'$ iff
(1) β_i is not free in σ and
(2) $\exists\tau_1, \ldots, \tau_n .\ [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\rho \geq_A \rho'$

This part is similar to the definition in Damas/Milner. The bound variables of σ are specialised and the resulting predicated types are compared.

Define $\rho \geq_A \rho'$ iff the type part of ρ equals the type part of ρ' (the same condition as Damas/Milner), and for every predicate $(x :: \tau)$ in ρ, either

- there is a predicate of the form $(x :: \tau)$ in ρ' (i.e. the predicate appears in both types); or
- the predicate can be *eliminated* under assumptions A.

A predicate $(x :: \tau)$ can be eliminated under A iff either

- $(x :: \tau \setminus \overline{x})$ is in A; or
- $(x ::_i \sigma' \setminus \overline{x})$ is in A and $\sigma' \geq_A \tau$.

For example, if A_0 is the set of assumptions in Figure 8, then

$$\begin{aligned}&(\forall\alpha.\ (eq :: Eq\ \alpha).\ [\alpha] \to \alpha \to Bool)\\ &\qquad \geq_{A_0} ([Int] \to Int \to Bool)\end{aligned}$$

holds. On the other hand,

$$\begin{aligned}&(\forall\alpha.\ (eq :: Eq\ \alpha).\ [\alpha] \to \alpha \to Bool)\\ &\qquad \geq_{A_0} ([Float] \to Float \to Bool)\end{aligned}$$

does *not* hold, since A_0 contains no binding asserting that eq has an instance at type $Float$.

Two type-schemes are *unifiable* if they overlap, that is, if there exists a type that is an instance of both under some set of assumptions. We say that σ and σ' are unifiable if there exists a type τ and valid set of assumptions A such that

$$\sigma \geq_A \tau \ \wedge\ \sigma' \geq_A \tau$$

We write $\sigma \# \sigma'$ if σ and σ' are *not* unifiable.

$$\textbf{PRED}\quad \frac{A, (x :: \tau \setminus x_\tau) \vdash e :: \rho \setminus \overline{e}}{A \vdash e :: (x :: \tau).\, \rho \setminus (\lambda x_\tau.\, \overline{e})} \quad (x ::_o \sigma) \in A$$

$$\textbf{REL}\quad \frac{\begin{array}{l} A \vdash e :: (x :: \tau).\, \rho \setminus \overline{e} \\ A \vdash x :: \tau \setminus \overline{e}' \end{array}}{A \vdash e :: \rho \setminus (\overline{e}\ \overline{e}')} \quad (x ::_o \sigma) \in A$$

$$\textbf{OVER}\quad \frac{A_x, (x ::_o \sigma) \vdash e :: \tau \setminus \overline{e}}{A \vdash (\textbf{over}\ x :: \sigma\ \textbf{in}\ e) :: \tau \setminus \overline{e}}$$

$$\textbf{INST}\quad \frac{\begin{array}{l} A, (x ::_i \sigma' \setminus x_{\sigma'}) \vdash e' :: \sigma' \setminus \overline{e'} \\ A, (x ::_i \sigma' \setminus x_{\sigma'}) \vdash e :: \tau \setminus \overline{e} \end{array}}{A \vdash (\textbf{inst}\ x :: \sigma' = e'\ \textbf{in}\ e) :: \tau \setminus (\textbf{let}\ x_{\sigma'} = \overline{e}'\ \textbf{in}\ \overline{e})} \quad (x ::_o \sigma) \in A$$

Figure 10: Typing and translation rules, part 2

$$\begin{array}{l}
\textbf{let}\ eq_{(Eq\ Int)} = eqInt\ \textbf{in} \\
\textbf{let}\ eq_{(Eq\ Char)} = eqChar\ \textbf{in} \\
\textbf{let}\ eq_{(\forall\alpha.\forall\beta.(eq::Eq\ \alpha).(eq::Eq\ \beta).Eq\ (\alpha,\beta))} \\
\quad = \lambda eq_{(Eq\ \alpha)}.\lambda eq_{(Eq\ \beta)}.\lambda p.\lambda q. \\
\qquad eq_{(Eq\ \alpha)}\ (fst\ p)\ (fst\ q)\ \wedge\ eq_{(Eq\ \beta)}\ (snd\ p)\ (snd\ q)\ \ \textbf{in} \\
eq_{(\forall\alpha.\forall\beta.(eq::Eq\ \alpha).(eq::Eq\ \beta).Eq\ (\alpha,\beta))}\ eq_{(Eq\ Int)}\ eq_{(Eq\ Char)}\ (1, \text{`a'})\ (2, \text{`b'})
\end{array}$$

Figure 11: Translation of equality, formalised

$$\begin{array}{ll}
A_1 : & (eq ::_o \forall\alpha.Eq\ \alpha) \\
& (eqInt :: Eq\ Int \setminus eqInt) \\
& (eqChar :: Eq\ Int \setminus eqChar) \\
\\
e_1 : & \textbf{inst}\ eq :: Eq\ Int = eqInt\ \textbf{in} \\
& \textbf{inst}\ eq :: Eq\ Char = eqChar\ \textbf{in} \\
& eq
\end{array}$$

Figure 12: A problematic expression

A.5 Valid assumptions

All sets of assumptions used within proofs must be valid. The valid sets of assumptions are inductively defined as follows:

- *Empty.* The empty assumption set, $\{\}$, is valid.
- *Normal identifier.* If A is a valid assumption set, x is an identifier that does not appear in A, and σ is a type scheme, then

  $$A,\ (x :: \sigma \setminus x)$$

  is a valid assumption set.
- *Overloaded identifier.* If A is a valid assumption set, x is an identifier that does not appear in A, σ is a type scheme, and $\tau_1, \ldots, \tau_m$ are types and $\sigma_1, \ldots, \sigma_n$ are types schemes such that
  - $\sigma \geq_A \sigma_i$, for i from 1 to n, and
  - $\sigma \geq_A \tau_i$, for i from 1 to m, and
  - $\sigma_i \# \sigma_j$, for distinct i, j from 1 to n

  then

  $$\begin{array}{l} A,\ (x ::_o \sigma), \\ (x ::_i \sigma_1 \setminus x_{\sigma_1}), \ldots, (x ::_i \sigma_n \setminus x_{\sigma_n}), \\ (x :: \tau_1 \setminus x_{\tau_1}), \ldots, (x :: \tau_m \setminus x_{\tau_m}) \end{array}$$

  is a valid assumption set.

For example, the assumptions in Figure 8 are a valid set. However, this set would be invalid if augmented with the binding

$$(eq ::_i \forall\gamma.Eq\,(Char, \gamma) \setminus eq_{(\forall\gamma.Eq\,(Char,\gamma))})$$

as this instance overlaps with one already in the set.

A.6 Inference rules

We now give inference rules that characterise well-typings of the form

$$A \vdash e :: \sigma \setminus \overline{e}$$

The rules break into two groups, shown in Figures 9 and 10. The first group is based directly on the Damas/Milner rules (Figure 9). There are two small differences: translations have been added to each rule in a straightforward way, and there are two **TAUT** rules instead of one (one rule for $(::)$ bindings and one for $(::_i)$ bindings).

For example, let A_0 be the set of assumptions shown in Figure 8, together with assumptions about the types of integer and character constants. Then the above rules are sufficient to derive that

$$\begin{array}{l} A_0 \vdash (eq\ 1\ 2) :: Bool \setminus (eq_{(Eq\,Int)}\ 1\ 2) \\ A_0 \vdash (eq\ \text{'a'}\ \text{'b'}) :: Bool \setminus (eq_{(Eq\,Char)}\ \text{'a'}\ \text{'b'}) \end{array}$$

That is, these rules alone are sufficient to resolve simple overloading.

More complicated uses of overloading require the remaining four rules, shown in Figure 10. The first two deal with the introduction and elimination of predicates, and the second two deal with the **over** and **inst** constructs.

As we have seen, expressions with types that contain classes (that is, expressions with predicated types) are translated to lambda abstractions that require a dictionary to be passed at run-time. This idea is encapsulated in the **PRED** ("predicate") and **REL** ("release") rules. The **PRED** and **REL** rules introduce and eliminate predicates analogously to the way that the **GEN** and **SPEC** rules introduce and eliminate bound type variables. In particular, the **PRED** rule adds a predicate to a type (and has a lambda expression as its translation) and the **REL** rule removes a predicate from a type (and has an application as its translation).

The **OVER** rule types **over** expressions adding the appropriate $(::_o)$ binding to the environment, and the **INST** rule types **inst** expressions adding the appropriate $(::_i)$ binding to the environment. The validity condition on sets of assumptions ensures that overloaded identifiers are only instanced at valid types.

Notice that none of the translations contain **over** or **inst** expressions, therefore, they contain no overloading. It is easy to verify that the translations are themselves well-typed in the Hindley/Milner system.

For example, the program in Figure 6 is translated by these rules into the program in Figure 11. The reader can easily verify that this corresponds to the translation from Figure 3 to Figure 4. We have thus shown how to formalise the typing and transformation ideas that were presented informally in the body of the paper.

A.7 Principal typings

Given A and e, we call σ a *principal type scheme* for e under A iff

- $A \vdash e :: \sigma \setminus \overline{e}$; and

- for every σ', if $A \vdash e :: \sigma' \setminus \overline{e}'$ then $\sigma \geq_A \sigma'$

A key result in the Hindley/Milner system is that every expression e that has a well-typing has a principal type scheme.

We conjecture that for every valid set of assumptions A and every expression e containing no **over** or **inst** expressions, if e has a well-typing under A then e has a principal type scheme under A.

For example, let A_0 be the set of assumptions in Figure 8. Then the typing

$$A_0 \vdash eq :: \forall\alpha.Eq\,\alpha \setminus eq_{(Eq\;alpha)}$$

is principal. Examples of non-principal typings are

$$A_0 \vdash eq :: Eq\;Int \setminus eq_{(Eq\;Int)}$$
$$A_0 \vdash eq :: Eq\;Char \setminus eq_{(Eq\;Char)}$$

Each of these is an instance of the principal typing under assumptions A_0.

The existence of principal types is problematic for expressions that contain **over** and **inst** expressions. For example, let A_1 and e_1 be the assumption set and expression in Figure 12. Then it is possible to derive the typings

$$A_1 \vdash e_1 :: Eq\;Int \setminus eqInt$$
$$A_1 \vdash e_1 :: Eq\;Char \setminus eqChar$$

But there is no principal type! One possible resolution of this is to require that **over** and **inst** declarations have global scope. It remains an open question whether there is some less drastic restriction that still ensures the existence of principal types.

References

[BEJ88] D. Bjørner, A. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, North-Holland, 1988 (to appear).

[CW85] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4, December 1985.

[Car88] L. Cardelli, Structural subtyping and the notion of power type. In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[FGJM85] K. Futasagi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, Principles of OBJ2. In *Proceedings of the 12'th Annual Symposium on Principles of Programming Languages*, January 1985.

[GR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, pp. 29–60, December 1969.

[HMM86] R. Harper, D. MacQueen, and R. Milner, Standard ML. Report ECS-LFCS-86-2, Edinburgh University, Computer Science Dept., 1986.

[HMT88] R. Harper, R. Milner, and M. Tofte, The definition of Standard ML, version 2. Report ECS-LFCS-88-62, Edinburgh University, Computer Science Dept., 1988.

[Kae88] S. Kaes, Parametric polymorphism. In *Proceedings of the 2'nd European Symposium on Programming*, Nancy, France, March 1988. LNCS 300, Springer-Verlag, 1988.

[Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, pp. 348–375, 1978.

[Mil84] R. Milner, A proposal for Standard ML. In *Proceedings of the Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Mil87] R. Milner, Changes to the Standard ML core language. Report ECS-LFCS-87-33, Edinburgh University, Computer Science Dept., 1987.

[MP85] J. C. Mitchell and G. D. Plotkin, Abstract types have existential type. In *Proceedings of the 12'th Annual Symposium on Principles of Programming Languages*, January 1985.

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[Str67] C. Strachey, Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[Wan87] M. Wand, Complete type inference for simple objects. In *Proceedings of the Symposium on Logic in Computer Science*, Ithaca, NY, June 1987. IEEE Computer Society Press, 1987.