Type Classes are Signatures of Abstract Types

In Proceedings of the Phoenix/Esprit Seminar and Workshop on Declarative Programming, Nov. 1991

Konstantin Läufer*
New York University
251 Mercer Street
New York, NY 10012, USA
laufer@cs.nyu.edu

Martin Odersky
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
odersky@watson.ibm.com

Abstract

We present an extension of Haskell's type class concept in which a type class is identified with the signature of an abstract type. As shown by Mitchell and Plotkin, abstract types can be expressed using existential quantification. Unlike in Mitchell and Plotkin's work, an abstract type does not come with one — and only one — implementation. Rather, any concrete type can be declared to be an implementation by a clause that corresponds to an instance declaration in Haskell. We introduce F-bounded existential quantification, where an abstract type has the form:

$$\exists\alpha.C(\alpha).\tau(\alpha).$$

Here, $C(\alpha)$ is a set of constraints that restricts the range of the bound variable α, and $\tau(\alpha)$ is a type constructed from α. The expression reads "some type $\tau(\alpha)$, where α is some arbitrary fixed type satisfying constraints $C(\alpha)$". The constraint set C corresponds to a type class. Just like a type class, it contains declarations for overloaded identifiers as well as conformity clauses that declare one abstract type to be more specific than another.

The generalization of type classes to abstract types has the advantage of greater expressiveness: We can model polymorphic abstract types and heterogeneous data structures, concepts which cannot be expressed in Haskell. An example of a polymorphic abstract type is $\forall\alpha.Bag\ \alpha$, the abstract type of all bags with elements of type α. In Haskell, we would either have to fix the element type, or we would have to fix the implementation of Bag.

Our extension shares the desirable properties of the type class approach in that it is fully static and in that type reconstruction is feasible.

*Supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J-1110

1 Introduction

Recently, many researchers have looked at the problem of systematic overloading resolution [8, 19, 1, 5]. All these approaches use predicates (called *type classes* in [19]) which assert that certain overloaded operations with given signatures are defined. Functions can be made dependent on these assertions using a form of bounded universal type quantification, in which the bound variable is restricted to range over the instances of a type class. In [19] it was noted that type classes and abstract data types [14, 3] are similar in that they both define a signature without its implementation. Because of this correspondence, Wadler and Blott called for a closer exploration of the relationship between type classes and abstract data types.

Here we describe the results of our exploration of this relationship. We show that type classes can indeed be generalized to a form of abstract data types, by identifying them with type signatures. Furthermore, the generalization is not simply a recast of the type class concept into another formalism; it allows us to model several new concepts. Among these are polymorphic abstract types and heterogeneous data structures.

Polymorphic abstract types are useful for modeling signatures which depend on other types. The abstract type $\forall\alpha.Bag\ \alpha$, for instance, cannot be represented as a type class, but it can be represented in our type system.

Heterogeneous data structures are useful in extensible software systems. An example is a window handler which provides central bookkeeping for windows of various types. In an extensible system, window types can be defined in clients as well as in the handler itself. Therefore, the type of any global data structure (say, one containing all active windows) cannot be a simple, finite sum type. What is needed is a heterogeneous structure whose elements are instances of an abstract type "Window".

Programming languages usually support heterogeneous data by adopting an object-oriented subtyping rule (it is no coincidence that the interest in object-oriented programming has spread in parallel with the use of windowing operating systems). Subtyping is not without problems, however. First, pervasive subtyping prevents the definition of polymorphic homogeneous data structures, which are useful in many circumstances. The function *maximum*, for instance, which finds the maximum of element a list, makes sense only on homogeneous lists whose elements are all of the same, ordered type. Another, more serious, problem stems from the subtype rule for method functions. Because the first argument to a method is implicit, and the others are subjected to the contravariance rule, method arguments are treated asymmetrically. This leads to counter-intuitive subtyping rules for methods with additional arguments of type "like current".

Several newer approaches try to overcome the problems caused by the contravariance rule in object-oriented programming. Descriptive classes [17] and F-bounded polymorphism [1] are in concept very similar to Haskell's type classes. Other methods, which generalize the subtyping concept, introduce "like current" parameters, which are checked type correct at runtime [4], or replace the subtyping rule by something rather more complex [16]. None of these latter approaches is both fully static and safe. A system that describes heterogeneous data types directly is [18]. Thatte uses partial types instead of existential quantification, which leads to less precise typings and requires runtime type checking in general. By contrast, our system follows the type class approach in that it is fully static; no type-related errors can occur at runtime. Another desirable property that our system shares with type classes is the existence of a type reconstruction algorithm.

The rest of this paper is organized as follows: Section 2 gives an informal overview of of abstract types. Section 3 discusses heterogeneous data structures, Section 4 sketches a runtime model, and Section 5 concludes. The formal typing rules are given in the Appendix.

2 Abstract Types

Following [14, 3], we identify abstract types with existential types. We use a bounded form of existential quantification. An abstract type is of the form

$$\exists\alpha.C(\alpha).\tau(\alpha).$$

Here, $C(\alpha)$ is a set of constraints which restricts the range of the bound variable α. $\tau(\alpha)$ is a type constructed from α. In the simplest case, $\tau = \alpha$. As an example, in our framework the abstract type of values admitting an order relation is

$$\exists\alpha\ .\ [(<), (\leq) : \alpha \to \alpha \to Bool]\ .\ \alpha,$$

which reads "some (concrete) type α, on which $(<)$ and $(\leq)$ are defined".

On the other hand, the unbound abstract type

$$\exists\alpha\ .\ .\ (\alpha \to \alpha \to Bool) \times \alpha$$

describes all pairs whose components are a function of type $\alpha \to \alpha \to Bool$ and a value of some (unconstrained) type α. Thus each value of this abstract type might have a different first (and second) component.

An example of an existential type with a more complex type component is:

$$\exists\alpha\ .\ [(<), (\leq) : \alpha \to \alpha \to Bool]\ .\ List\ (\alpha \times \alpha),$$

the type of all lists whose elements are homogeneous pairs whose components have both the same ordered type.

By contrast,

$$\begin{aligned} List\ ((\exists\alpha\ .\ [(<), (\leq) : \alpha \to \alpha \to Bool]\ .\ \alpha)\ \times \\ (\exists\alpha\ .\ [(<), (\leq) : \alpha \to \alpha \to Bool]\ .\ \alpha)) \end{aligned}$$

gives us a list of heterogeneous pairs, whose components are are arbitrary — possibly different — ordered types.

2.1 Abstract Type Declarations

In the concrete syntax of our example language, we use keywords `some` ... `where` instead of existential quantifiers. An abstract type is expressed as follows:

$$\tau = \texttt{some}\ a\ \texttt{where}\ C\ \texttt{in}\ \tau'$$

The bound variable a is a *placeholder*; it stands for all implementations of the abstract type. We adopt the convention that the signature part following `where` can be omitted if it is empty and that the type part following `in` can be omitted if it is just the placeholder.

The signature part corresponds to a type class. It consists of an arbitrary number of conformity clauses and definitions of overloaded identifiers. A conformity clause is of the form $\alpha :: \tau$, where α is the placeholder and τ is another abstract type. It reads: "Every implementation of the type being defined is also an implementation of τ". A definition of an overloaded identifier is of the form $x : \sigma$. It states that an identifier x of type $[\tau =: \alpha]\sigma$ is declared for all implementations τ of the defined type. Here, $[\tau =: \alpha]$ denotes substitution of τ for α.

Example 2.1 Some simple abstract type declarations are:

```
Eq    = some a where
          (=): a -> a -> Bool

Ord   = some a where
          a :: Eq
          (<), (<=) : a -> a -> Bool

Point = some p where
          p   :: Ord
          x, y : p -> Coord
          move : p -> (Coord * Coord) -> p
```

So far, we have used abstract types in ways which could just as well have been expressed with type classes. Abstract types are more general than type classes, however, since they are identified with types. Being types, they can be used in a more flexible way than predicates on types can. Two examples are the possibility to define values of an abstract type, and the possibility to construct abstract types which are parameterized by other types. Values of abstract types are important in connection with heterogeneous data structures (see Section 3). An example of a parameterized abstract type is the following:

Example 2.2 `Bag` is a type constructor which, given an element type `a`, yields the abstract type of bags of `a`.

```
a :: Eq => Bag a = some b where
    b :: Eq
    emptybag     : b
    singleton    : a -> b
    union,
    intersection,
    difference   : b -> b -> b
    elements     : b -> List a
```

2.2 The Implementation Relation

We have seen that an abstract type defines the interface of some data structure; it does not specify the data structure's implementation. To specify that a given type τ is an implementation of an abstract type σ, we use a variant of a conformity clause, in which the implementations of all functions in σ's signature are given. Syntax:

$$\begin{array}{lcl} \mathit{impl} & ::= & [\mathit{cond} \Rightarrow] \ \tau :: \sigma \ \texttt{where} \ x_1 = e_1, \ldots, x_n = e_n \\ \mathit{cond} & ::= & [\mathit{cond},] \ \alpha :: \sigma \end{array}$$

Notes:

1. Every concrete or abstract type may occur in arbitrarily many implementation clauses. Hence, an abstract type can be implemented by several concrete types, and a concrete type can be the implementation of several abstract types.

2. Implementations can be conditional.

3. Each implementing expression e_i must be of type $[\tau =: \alpha]\tau_i$, where the declaration $x_i : \tau_i$ forms part of the signature of σ and α is σ's placeholder variable.

Example 2.3 Lists conditionally implement the `Eq` abstract type:

```
a :: Eq => List a :: Eq where
    (=) = listeq

listeq : List a -> List a -> Bool
listeq = ...
```

In the same style we can state that `List` conditionally implements the `Ord` abstract type. Lists of elements with equality also implement bags; a simple, albeit inefficient implementation would be:

```
a :: Ord => List a :: Bag a where
    (=) xs ys              = sort xs = sort ys
    emptybag               = []
    singleton x            = [x]
    union                  = (++)
    intersection           = isect
    difference             = (--)
    elements               = id

isect [] ys     = []
isect (x:xs) ys = (if x `in` ys then [x] else [])
                  ++ isect xs (ys -- [x])
```

Note that `List` implements the abstract type `Eq` in two different ways: Once directly, meaning lexical ordering, and another time indirectly, via `Bag`. Having two different implementations of equality raises the question which implementation is meant by `(=)` We disambiguate by the rule that left-hand side occurrences of `(=)` refer to the newly defined type (i.e. `Bag`), whereas right-hand side occurrences refer to the implementing type (i.e. `List`). As a consequence, the definitions of a signature are not known in the expressions that implement the signature. In our example, `intersection` had to be defined in terms of an auxiliary, recursive function `isect`. This is admittedly rather crude, but the only route open to us if we want to avoid atrocities like CLU's **rep** and **cvt**.

2.3 F-Bounded Polymorphism

Like existential types, universal types are bounded, having the form:

$$\forall\alpha.C(\alpha).\tau(\alpha)$$

C is a set of conformity constraints. A conformity constraint $\alpha :: \tau$ restricts the bound variable α to range only over implementations of abstract type τ. Again, this is similar to the predicated types in [19]. It is strictly more expressive than universal polymorphism with type class bounds since bounds are types and therefore can be parameterized.

Example 2.4 A function to compute the symmetric difference between two bags:

$$\texttt{symdiff} : \forall\alpha.\forall\beta\ .\ (\beta :: Bag\ \alpha)\ .\ \beta \times \beta \to \beta$$

```
symdiff b1 b2 = union (difference (b1, b2), difference (b2, b1))
```

In other type systems such as the ones of Haskell, ML [12], or XML+ [13], we either have to fix the element type α, or we have to fix the implementation of `Bag`, whereas our system allows to abstract on both element type and implementation.

2.4 Abstract Types are Large

Not all types have equal status in the Hindley/Milner system. Universally quantified types are second class citizens, since they cannot be substituted for a type variable. The reason for this restriction is that type reconstruction is conjectured to be undecidable if quantifiers range over polymorphic as well as monomorphic types [7]. In [10], the types over which quantifiers range are called small types, and all other types are called large types. Universally quantified types would be considered large by that definition. As far as existential types go, we have a choice. To see the distinction, consider the type expression (from [3])

$$\exists\alpha.\alpha \times \alpha.$$

If abstract types are large, this expression denotes the type of pairs where both elements have the same, unknown type. On the other hand, if abstract types are small, we can substitute $\exists\alpha.\alpha$ for α and write:

$$\exists\alpha.\alpha \times \alpha \quad = \quad (\exists\alpha.\alpha) \times (\exists\alpha.\alpha)$$

Hence, we see that the abstract type $\exists\alpha.\alpha \times \alpha$ stands now for pairs whose elements have arbitrary, possibly different types.

In this work, we have chosen abstract types to be large. That is, quantifiers do not range over abstract types, and abstract types cannot be substituted for type variables. This makes automatic type reconstruction possible (an inference algorithm is given in [9]). For the general case, with quantifiers ranging over abstract as well as concrete types, the problem of type inference is still open.

One consequence of abstract types being large is that the form of types taking part in the proof of a typing is restricted. The typing rules in Appendix A force all existential quantifiers to occur only at the outermost level, as element types of an algebraic type, or as function results. While we regard the syntax of types required by the typing rules as their "normal form," we use abstract types more liberally in our programming examples. The examples can be "de-sugared" to normal form using laws (P1) – (P3):

(P1) $(\exists\alpha.C.\tau) \to \tau' \ = \ \forall\alpha.C.(\tau \to \tau')$

(P2) $(\exists\alpha.C.\eta_1) \times \eta_2 \ = \ \exists\alpha.C.\eta_1 \times \eta_2$ if α is not free in η_2
$\eta_1 \times (\exists\alpha.C.\eta_2) \ = \ \exists\alpha.C.\eta_1 \times \eta_2$ if α is not free in η_1

(P3) $\exists\alpha.(\alpha :: \exists\beta.C.\beta).\alpha \ = \ \exists\alpha.C.\alpha$

3 Heterogeneous Data Types

Abstract types with more than one implementation give us heterogeneous data structures for free. If we declare a data structure with a field of an abstract type, every occurrence of this field can be represented by a different implementation of the abstract type. That is, the structure is (boundedly) heterogeneous.

Example 3.1 A heterogeneous list of windows.

Assume the following hierarchy of window types to be given:

```
Window      = some w where
                handle  : w -> Event -> w
                display : w -> DisplayList
                ...

TextWindow  = some w where w :: Window
                ...

GraphWindow = some w where w :: Window
                ...
```

We can then declare a type `WindowList` in terms of the abstract type `Window`. Note that unlike `List`, `WindowList` is a type, not a type constructor, since there are no variables on the left hand side of the declaration.

```
WindowList  = WNil | WCons (Window * WindowList)
```

Then, in the expression

```
case wl of WCons (win, ws) => ...
```

the pattern `win` is typed `Window`. This means that `win` can be passed to every function which expects an arbitrary implementation of `Window` as argument.

The constructor aspect of the above type declaration is in some sense the reverse of the pattern matching aspect. With properties P1, P2, and P3, we have:

$$
\begin{array}{lcl}
\texttt{WCons} & : & Window \times WindowList \to WindowList \\
& = & \text{(P3)} \\
& & (\exists w \,.\, w :: Window \,.\, w) \times WindowList \to WindowList \\
& = & \text{(P2)} \\
& & (\exists w \,.\, w :: Window \,.\, w \times WindowList) \to WindowList \\
& = & \text{(P1)} \\
& & \forall w \,.\, w :: Window \,.\, (w \times WindowList \to WindowList).
\end{array}
$$

Hence, we see that, first, the quantifier changes between constructors and pattern matching, and, second, that `WCons` indeed may be passed an arbitrary implementation of `Window`. This means that (implementations of) `TextWindows` or `GraphWindows`, for example, can also be included in window lists, or, put in other words, that `WindowList` is a heterogeneous data structure. This solves the extensibility problem described in the introduction. For example, a broadcast function, which passes an event to all windows in a `WindowList` could be expressed as follows:

```
broadcast : WindowList -> Event -> WindowList

broadcast WNil e             = WNil
broadcast (WCons (w, ws)) e = WCons (handle w e, broadcast ws e)
```

4 Runtime Model

Since the implementation of the operations defined in an abstract type is not known statically, we have to maintain this information at runtime, in the form of "method dictionaries". Our runtime model is an extension of the one introduced by Wadler and Blott [19]. They present a scheme for the translation of expressions with type classes into equivalent ML expressions.

Every implementation of an abstract type defines a dictionary, i.e. a tuple which contains the definition of all operations in the type's signature. Dictionaries are passed to polymorphic functions as additional arguments. A dictionary is passed for every conformity constraint in the function's type. A function such as

$$\texttt{redefine} \;:\; \forall w \,.\, (w :: Window) \,.\, w \to w$$

would be translated to a function with two arguments, a dictionary and a window. The additional dictionary arguments are passed at the point in the program text where the polymorphic type is instantiated. An expression such as `redefine tw`, where `tw` is a `TextWindow`, would be translated to (we mark translated versions of original identifiers with an apostrophe):

```
redefine' textWinDict tw.
```

This method is identical to the one described in [19], with conformity constraints replacing instance constraints in Haskell. For abstract types, an extension to the scheme is needed.

We map values of an existential type into dictionary/value tuples, similarly to the memory layout for object-oriented languages. A `WindowList` would thus be translated into a data structure which is decorated with one dictionary table per element:

```
WindowList' = WNil'
            | WCons' (WindowDict, (WindowData, WindowList'))
```

Since existential polymorphism in the type becomes universal polymorphism in its constructors, all constructors already have the needed dictionaries as arguments. The original `WCons`, for example, is of type

$$\texttt{WCons} \;:\; \forall \alpha \,.\, (\alpha :: Window) \,.\, \alpha \times WindowList \to WindowList,$$

and would hence be translated to a function whose signature matches the second alternative in type `WindowList`:

$$\texttt{WCons}' \;:\; WindowDict \times (WindowData \times WindowList') \to WindowList'$$

If a value of an existential types is passed to a function, all needed dictionary components are extracted and passed as additional arguments. Hence, the expression

```
redefine w
  where WCons (w, ws) = wlist,
```

would be translated to

```
redefine' wdict w
  where WCons' (wdict, (w, ws)) = wlist.
```

5 Conclusion

We have introduced a form of abstract data type that extends the type class concept of Haskell and allows us to model heterogeneous data structures, useful for extensible software systems. The type system is fully static, that is, no type-related errors can occur at runtime. Types are reconstructible using a variant of the ML type inference algorithm.

Since abstract types are large, our type discipline is polymorphic over concrete types only. An interesting extension, which we are currently investigating, would be a theory that is is polymorphic over abstract types as well. We could then abstract over properties of types; a concept such as $List\,T$, the list of objects which are instances of the given abstract type T, could be defined. This expression could then be instantiated to yield $List\ Window$, or $List\ Order$, for example. We see two ways to achieve that extension: We could either introduce explicit typing for abstract types in the style of the ML Module system [10], or we could generalize our system by considering existential types to belong to the universe of small types. The problem of type inference in such an extension remains to be solved, however.

Acknowledgements

We would like to thank Ben Goldberg, Fritz Henglein, Kim Marriott, and Satish Thatte for helpful comments and discussions.

A Typing Rules

This section presents the formal typing rules for abstract data types. For the kernel language, these rules are similar to the ones of the Hindley/Milner system [11, 6]. In their overloading and implementation aspects, they are derived from the rules in [19]. New rules for existential quantification and the formal treatment of algebraic data types have been added.

A.1 Language

The example language is designed such that common functional languages can be mapped into it. It consists of expressions of the following forms:

Expressions	e	::=	x	(identifier)
		|	$e\ e'$	(function application)
		|	$\lambda x.e$	(function abstraction)
		|	**let** $x = e$ **in** e'	(local declaration).
		|	**data** σ **in** e	(data declaration)
		|	**over** $x :: \sigma$ **in** e	(overloaded identifier)
		|	**inst** $x :: \sigma = e$ **in** e'	(instance declaration)

Predefined:

$$
\begin{array}{rcl}
\textbf{fix} & : & \forall\alpha.(\alpha \to \alpha) \to \alpha \\
\textbf{if} & : & \forall\alpha.Bool \to \alpha \to \alpha \to \alpha \\
(\cdot\ ,\ \cdot) & : & \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta \\
\textbf{fst} & : & \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\
\textbf{snd} & : & \forall\alpha.\forall\beta.\alpha \times \beta \to \beta
\end{array}
$$

The first four productions describe a conventional λ-calculus language, similar to *Exp* [11] Instead of having a set of primitive types, we augment *Exp* by **data** clauses defining algebraic types (types are discussed in the next subsection). We also add declarations for overloaded identifiers and their instances in the style of [19].

Example A.1 The overloaded identifiers of the abstract types *Eq* and *Bag* are declared by:

$$\textbf{over } eq :: \forall\alpha\ .\ EqSig\ \alpha \textbf{ in}$$
$$\textbf{over } bag :: \forall\alpha\ .\ (eq :_i EqSig\ \alpha)\ .\ \forall\beta\ .\ BagSig\ \alpha\ \beta \textbf{ in } \ldots$$

The lower-case identifiers *eq* and *bag* name the defined operations. That is, *eq* is just the equality function, and *bag* is a tuple of all functions defined in the abstract type `Bag` (see Example 2.2). *EqSig* and *BagSig* are used as shorthands for the signatures of the abstract type. That is,

$$\begin{array}{lcll} EqSig\ \alpha & = & \alpha \to \alpha \to Bool & \\ BagSig\ \alpha\ \beta & = & \beta\times & \text{(emptybag)} \\ & & (\alpha \to \beta)\times & \text{(singleton)} \\ & & (\beta \to \beta \to \beta)\times & \text{(union)} \\ & & \ldots & \end{array}$$

The conformity declaration that $List\ \alpha$ implements $Bag\ \alpha$ is translated as follows:

$$\begin{array}{lcl} \textbf{inst } bag & :: & \forall\alpha\ .\ (eq :_i EqSig\ \alpha)\ .\ BagSig\ \alpha\ (List\ \alpha) \\ & = & (Nil, \\ & & \lambda x.Cons\ x\ Nil, \\ & & \lambda xs.\lambda ys.append\ xs\ ys, \\ & & \ldots) \end{array}$$

More detailed examples are found in [19].

Remark on naming: All variable identifiers defined in a **let** or **over** clause, and all type and constructor identifiers defined in a **data** clause are assumed to be different from each other.

A.2 Types

We augment Hindley/Milner types with algebraic data types and bounded universal and existential quantification:

Type Identifiers	T			
Constructors	K			
Types	τ	::=	α	(variable)
		|	$\tau \to \eta$	(function)
		|	$\tau_1 \times \tau_2$	(product)
		|	$\mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n$	(algebraic type)
Abstract Types	η	::=	$\exists\alpha.\chi.\eta \mid \tau$	
Type Schemes	σ	::=	$\forall\alpha.\chi.\sigma \mid \eta$	
Constraints	χ	::=	$\chi_1.\chi_2$	(conjunction)
		|	$x :_i \tau$	(is-instance)

An algebraic data type is given by a type expression of the form $\mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n$. Here, T is the type's name and $K_1, \ldots, K_n$ are its constructors. For every such algebraic type, we assume injection functions K_i, projection functions $\downarrow K_i$, and test functions $?K_i$ to be given. The types of these function are determined by rules INJ, PROJ, and TEST in Section A.4.

Example A.2 Type $List\ \alpha$ can be written as follows:

$$\forall\alpha\ .\ \mu\ List\ .\ Nil + Cons\ (\alpha \times List)$$

The deduction rules in Section A.4 ensure that only algebraic types declared in a **data** clause take part in a typing.

An abstract type is defined by a type expression of the form $\exists\alpha.\chi.\eta$. Here, η is a (possibly abstract) type, and χ is a constraint which restricts the range of the quantifier. Constraints are finite sets of instance assertions $x :_i \tau$. If the constraint set is empty it can be omitted, dropping one of the two delimiting period signs. Note that this is different from the concrete example language, where constraints were written in the reverse way.

Example A.3 Type $Bag\ \alpha$ can be translated as follows:

$$\forall\alpha\ .\ \exists\beta\ .\ (bag :_i BagSig\ \alpha\ \beta)\ .\ \beta$$

A.3 Assumptions

Assumptions are sets of type predicates.

Assumptions	A	$::=$	$A_1.A_2$	(conjunction)
		$\mid$	$\pi(\sigma)$	(predicate in σ)
Predicates in σ	$\pi(\sigma)$	$::=$	$x :_o \sigma$	(overloaded identifier)
		$\mid$	$x :_i \sigma$	(instance)
		$\mid$	$x : \sigma$	(has-type)
		$\mid$	$!\,\sigma$	(is-type)

A.4 Typing Rules

Logical rules

TAUT
$$\frac{}{A\ \vdash\ \pi(\sigma)} \quad (\pi(\sigma) \in A)$$

AND
$$\frac{A\ \vdash\ \pi_1(\sigma_1) \qquad A\ \vdash\ \pi_2(\sigma_2)}{A\ \vdash\ \pi_1(\sigma_1).\pi_2(\sigma_2)}$$

∀-ELIM
$$\frac{A\ \vdash\ \pi(\forall\alpha.\chi.\sigma) \qquad A\ \vdash\ [\tau =: \alpha]\chi}{A\ \vdash\ \pi([\tau =: \alpha]\sigma)}$$

∀-INTRO
$$\frac{A\ .\ \chi\ \vdash\ \pi(\sigma)}{A\ \vdash\ \pi(\forall\alpha.\chi.\sigma)} \quad (\alpha \notin FV(A))$$

Rules ∀-ELIM and ∀-INTRO are generalized to cover all forms of type predicates in Section A.3, instead of just predicates of the form $e : \tau$. Note that type variables can be instantiated only with concrete types.

$$\text{∃-ELIM} \qquad \frac{A \vdash \pi(\exists\alpha.\chi.\eta)}{A \vdash \pi(\eta)} \quad (\alpha \notin FV(\eta))$$

$$\text{SKOLEM} \qquad \frac{A \vdash \pi(\exists\alpha.\chi.\eta) \qquad A\,.\,\pi([\beta =: \alpha]\eta)\,.\,[\beta =: \alpha]\chi \vdash \pi'(\eta')}{A \vdash \pi'(\exists\alpha.\chi.[\alpha =: \beta]\eta')} \quad \begin{array}{l}(\beta \notin FV(A,\chi,\eta),\\ \alpha \notin FV(\eta'))\end{array}$$

With rule ∃-ELIM, "superfluous" existential quantifiers, which do not bind anything, may be dropped. Rule SKOLEM eliminates an existential quantifier in a subproof, replacing it with a "Skolem" variable (written β in the rule above). There is no rule for introducing an existential quantifier. Loosely speaking, the only way for an existential quantifier to enter a proof of $(\vdash e : \tau)$ is via an explicitly given abstract type in e. Type reconstruction is thus greatly simplified.

Rules for the base language

The rules for the base language (excluding **data** clauses and overloading) are fairly conventional:

$$\text{APP} \qquad \frac{A \vdash e : \tau \to \eta \qquad A \vdash e' : \tau}{A \vdash (e\ e') : \eta}$$

$$\text{ABS} \qquad \frac{A\,.\,(x : \tau) \vdash e : \eta}{A \vdash (\lambda x.e) : \tau \to \eta}$$

$$\text{LET} \qquad \frac{A \vdash e : \sigma \qquad A\,.\,(x : \sigma) \vdash e' : \tau}{A \vdash (\mathbf{let}\ x = e\ \mathbf{in}\ e') : \tau}$$

Rules for algebraic type declarations and pattern matching

$$\text{DATA} \qquad \frac{A\,.\,(!\,\sigma) \vdash e : \tau}{A \vdash (\mathbf{data}\ \sigma\ \mathbf{in}\ e) : \tau}$$

$$\text{INJ} \qquad \frac{A \vdash !\,\mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n}{A \vdash K_i : \mathit{shallow}(\eta_i \to \mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n)}\ (1 \le i \le n)$$

$$\text{PROJ} \qquad \frac{A \vdash !\,\mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n}{A \vdash {\downarrow}K_i : \mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n \to \eta_i} \quad (1 \le i \le n)$$

$$\text{TEST} \qquad \frac{A \vdash !\,\mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n}{A \vdash ?K_i : \mu T.K_1\ \eta_1 + \ldots + K_n\ \eta_n \to \mathit{Bool}} \quad (1 \le i \le n)$$

Rules INJ, PROJ, and TEST give types to injections, projections and test functions which are associated with an algebraic data type. Function *shallow* in INJ converts a function signature of

the form $\eta \to \eta'$ (which is syntactically illegal if η is quantified) to an equivalent shallow function signature [2]. Existential quantifiers in argument position are converted to universal quantifiers over the whole function.

$$\begin{aligned} \mathit{shallow}\ ((\exists\alpha.\chi.\eta) \to \eta') &= \forall\alpha.\chi.\mathit{shallow}\ (\eta \to \eta') \\ \mathit{shallow}\ (\tau \to \eta) &= \tau \to \eta \end{aligned}$$

Rules for overloading

$$\text{OVER} \quad \frac{A\ .\ (x :_o \sigma)\ \vdash\ e : \tau}{A\ \vdash\ (\mathbf{over}\ x :: \sigma\ \mathbf{in}\ e) : \tau}$$

$$\text{INST} \quad \frac{\begin{array}{c} A\ \vdash\ x :_o \sigma \\ A\ \vdash\ e : \sigma \\ A\ .\ (x :_i \sigma)\ \vdash\ e' : \tau \end{array}}{A\ \vdash\ (\mathbf{inst}\ x :: \sigma = e\ \mathbf{in}\ e') : \tau} \quad \begin{array}{l} (\text{for all } (x :_i \sigma') \in A: \\ \sigma, \sigma' \text{ not unifiable}) \end{array}$$

$$\text{ITYPE} \quad \frac{A\ \vdash\ x :_i \sigma}{A\ \vdash\ x : \sigma}$$

The last three rules are essentially equivalent to rules for overloading resolution in [19]. Wadler and Blott's definition of "valid assumption set" corresponds to our "not unifiable" condition in rule INST.

Wadler and Blott conjectured that principal types exist in their system, provided all **over** and **inst** declarations are global. This was subsequently shown correct in [15]. We conjecture that the extended system with existential type shares the principal type properties of type classes.

References

[1] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Functional Programming and Computer Architecture*, pages 273–280, 1989.

[2] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 9(8):147–172, 1987.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[4] W. Cook. A proposal for making Eiffel type-safe. *Computer Journal*, 32(4):305–311, 1989.

[5] G. Cormack and A. Wright. Type-dependent parameter inference. In *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pages 127–136, White Plains, NY, June 1990.

[6] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[7] F. Henglein and H. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Proc. 18th ACM Symp. on Principles of Programming Languages (POPL)*, Orlando, Florida, Jan. 1991.

[8] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symosium on Programming, Lecture Notes in Computer Science, Vol. 300*, pages 131–144, Nancy, France, March 1988. Springer-Verlag.

[9] K. Läufer and M. Odersky. Type inference for an object-oriented extension of ML. NYU-CIMS Report, New York University, Department of Computer Science, in preparation.

[10] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[11] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[12] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[13] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

[14] J. Mitchell and G. Plotkin. Abstract Types have Existential Type. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 37–51. ACM, Jan. 1985.

[15] T. Niphow and G. Snelting. Type classes and overloading resolution via order-sorted unification. Technical Report PI-R8/90, Technische Hochschule Darmstadt, Praktische Informatik, July 1990.

[16] J. Palsberg and M. Schwartzbach. Type substitution for object-oriented programming. In N. Meyrowitz, editor, *Proc. Conf. Object-Oriented Programming: Systems, Languages, and Applications and European Conf. on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, Oct. 1990. ACM Press.

[17] D. Sandberg. An alternative to subclassing. In *Proc. Object-Oriented Programming: Languages, Systems and Applications*, pages 424–428, 1986.

[18] S. Thatte. Type inference with partial types. In *Proc. Int'l Conf. on Algorithms, Languages and Programming*, pages 615–629, 1988.

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