First-class Polymorphism with Type Inference

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

Abstract

Languages like ML and Haskell encourage the view of values as *first-class* entities that can be passed as arguments or results of functions, or stored as components of data structures. The same languages offer *parametric polymorphism*, which allows the use of values that behave uniformly over a range of different types. But the combination of these features is not supported—polymorphic values are not first-class. This restriction is sometimes attributed to the dependence of such languages on *type inference*, in contrast to more expressive, explicitly typed languages, like System F, that do support first-class polymorphism.

This paper uses relationships between types and logic to develop a type system, FCP, that supports first-class polymorphism, type inference, and also first-class abstract datatypes. The immediate result is a more expressive language, but there are also long term implications for language design.

1 Introduction

Programming languages gain flexibility and orthogonality by allowing values to be treated as *first-class* entities. Such values can be passed as arguments or results of functions, or be stored and retrieved as components of data structures. In functional languages, the former often implies the latter; values are stored in a data structure by passing them as arguments to *constructor* functions, and retrieved using *selector* functions.

In languages like ML [20] and Haskell [8], new types of first-class value are specified by giving the names and types for their constructors. For example, a definition:

$$\textbf{data}\ \mathit{List}\ a \ = \ \mathit{Nil} \ \mid \ \mathit{Cons}\ a\ (\mathit{List}\ a)$$

introduces a new, unary type constructor, *List*, with two constructor functions:

$$\begin{array}{lcl} \mathit{Nil} & :: & \forall a.\mathit{List}\ a \\ \mathit{Cons} & :: & \forall a.a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a. \end{array}$$

Pattern matching allows the definition of selectors:

$$\begin{array}{lcl} \mathit{head} & :: & \forall a.\mathit{List}\ a \rightarrow a \\ \mathit{head}\ (\mathit{Cons}\ x\ \mathit{xs}) & = & x \end{array}$$

and other useful operations on lists, such as:

$$\begin{array}{lcl} \mathit{length} & :: & \forall a.\mathit{List}\ a \rightarrow \mathit{Int} \\ \mathit{length}\ \mathit{Nil} & = & 0 \\ \mathit{length}\ (\mathit{Cons}\ x\ \mathit{xs}) & = & 1 + \mathit{length}\ \mathit{xs}. \end{array}$$

All of these functions have *polymorphic* types, which indicate that they work in a uniform manner, independently of the type of elements in the lists concerned. Indeed, examples like these are often used to illustrate the flexibility of polymorphic type systems.

An important property of languages based on the Hindley-Milner type system [6, 19] is that most general, or *principal*, types for functions like these can be inferred automatically from the types of the constructors *Nil* and *Cons*. There is no need for further type annotations. At the same time, the typing discipline provides a guarantee of *soundness* or type security; the execution of a well-typed program will not "go wrong." Combining these attractive features, the Hindley-Milner type system has been adopted as the basis for a number of different programming languages.

However, the Hindley-Milner type system has a significant limitation: polymorphic values are not first-class. Formally, this is captured by making a distinction between monomorphic types and polymorphic *type schemes*. Universal quantifiers, signalling polymorphism,

To appear in the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15-17, 1997.

can only appear at the outermost level of a type scheme, and quantified variables can only be instantiated with monomorphic types. To put it another way: first-class values have monomorphic type.

In many applications, this limitation is considered a reasonable price to pay for the convenience of type inference. For example, it is often enough to be able to pass particular monomorphic instances of polymorphic values to and from functions, rather than the polymorphic values themselves. On the other hand, comparisons with explicitly typed languages, like System F [4, 29], that do support first-class polymorphism, reveal significant differences in expressiveness. We will see a number of practical examples in later sections that use first-class polymorphism in essential ways, but cannot be coded in a standard Hindley-Milner type system.

1.1 This paper

In this paper we show that polymorphic values *can* be used as first-class objects in a language with an effective type inference algorithm, provided that we are prepared to package them up as datatype components. The constructs that are used to build and extract these components serve as an alternative to type annotations; they allow us to define and use first-class polymorphic values without sacrificing the simplicity and convenience of type inference. Although the notation is different, the use of special constructs to package and unwrap polymorphic values is not new. For example, System F uses type abstraction to build polymorphic values and type application to instantiate them. However, in contrast with our approach, the standard presentation of System F requires explicit type annotations for *every* λ-bound variable, and the type inference problems for implicitly and partially typed variations of System F are undecidable [34, 1, 26].

Our approach is inspired by rules from predicate calculus that are used to convert logical formulae to *prenex* form, with all quantifiers at the outermost level. This leads to a system that allows both universal and existential quantifiers in the types of datatype components. The former provides support for first-class polymorphism, while the latter can be used to deal with examples of first-class abstract datatypes [21] in the style suggested by Perry [23] and refined by Läufer [14, 16].

The main subjects covered in the remaining sections of this paper are as follows:

- Section 2 shows how ideas from logic can be used to develop a type system that allows datatypes with components whose types include universal or existential quantifiers. We refer to this type system as FCP, a mnemonic for **F**irst-**C**lass **P**olymorphism.
- Section 3 illustrates the expressiveness of FCP. Our examples include a representation for booleans and Church numerals, a facility for using monads [32] as first-class values, a treatment of stacks as abstract datatypes, and the Pierce-Turner representation of objects using existential types [27]. These examples have been tested using a prototype implementation of FCP developed by the author.
- Section 4 gives a formal definition of FCP, including the treatment of special syntax for constructors and pattern matching. Type inference for FCP is described in Section 5.
- Section 6 highlights a correspondence between uses of constructors and selectors in FCP, and uses of type abstraction and application, respectively, in System F. By defining an appropriate collection of datatypes, we show that every System F program can be implemented by a term in FCP.
- Section 7 describes the background for FCP and discusses its relationship to other, largely orthogonal extensions of Hindley-Milner typing.

2 Quantified component types

Consider a simple datatype definition:

$$\textbf{data}\ T\ a \quad = \quad C\ \tau,$$

which introduces a new type constructor T, with a constructor C and an easily-defined selector unC of type:

$$\begin{array}{lcl} C & :: & \forall a.\tau \to T\ a \\ unC & :: & \forall a.T\ a \to \tau. \end{array}$$

In both types, the quantified variable a ranges over arbitrary monomorphic types, so the components of any data structure that we build or access using these functions must have monomorphic types.

The purpose of this section is to show how these ideas can be extended to datatypes with quantified component types. Specifically, we are interested in a system that allows definitions of the form:

$$\textbf{data}\ T\ a \quad = \quad C\ (\mathcal{Q}x.\tau),$$

for some quantifier $\mathcal{Q} \in \{\forall, \exists\}$. This would give constructor and selector functions with types of the form:

$$\begin{array}{lcl} C & :: & \forall a.(\mathcal{Q}x.\tau) \to T\ a \\ unC & :: & \forall a.T\ a \to (\mathcal{Q}x.\tau). \end{array}$$

A general treatment of the nested quantifiers in these types would make it difficult to deal with type inference, and could lead to undecidability [34, 1, 26]. But our goal is more modest—nested quantifiers are used only in the types of constructors and selectors—so it is reasonable to hope that we can make some progress. Indeed, we are not the first to consider extensions of the Hindley-Milner type system that support datatypes with quantified component types. For example, Perry [23] and Läufer [14] have each described extensions of Milner's type inference algorithm that allow datatypes with existentially quantified components. Rémy [28] used an extension of ML with both universally and existentially quantified datatype components to model aspects of object-oriented languages, but did not discuss type inference. Recent work by Odersky and Läufer [22] has similar goals to the present paper and permits universal quantification of datatype fields, with an encoding to simulate existential quantification. However, their approach requires a significant extension of the usual type inference mechanisms—replacing unification with instantiation. We believe that the approach described here is simpler, achieving the same degree of expressiveness, but based on techniques that are easier to use and easier to implement.

2.1 Eliminating nested quantifiers

If we cannot work with nested quantifiers, then perhaps we can find a way to eliminate them. There is a well-known procedure in predicate logic for converting an arbitrary formula, possibly with nested quantifiers, to an equivalent formula in *prenex* form with all quantifiers at the outer-most level. The process is justified by the following equivalences from classical logic, all subject to the condition that x does not appear free in P:

$$
\begin{aligned}
(\forall x.P \Rightarrow Q) &= P \Rightarrow (\forall x.Q) && (1)\\
(\forall x.Q \Rightarrow P) &= (\exists x.Q) \Rightarrow P && (2)\\
(\exists x.P \Rightarrow Q) &= P \Rightarrow (\exists x.Q) && (3)\\
(\exists x.Q \Rightarrow P) &= (\forall x.Q) \Rightarrow P && (4)
\end{aligned}
$$

Using the Curry-Howard isomorphism [7] as a bridge between logic and type theory, we can use Equation 1 to justify the equivalence:

$$(\forall a.\forall x.T\ a \rightarrow \tau) \quad = \quad \forall a.T\ a \rightarrow (\forall x.\tau)\,.$$

This result tells us that we can convert an arbitrary term of one type to a corresponding term of the other type. In this case, the equivalence allows us to deal with selectors for datatypes with universally quantified components, that is, with functions whose types are of the form on the right hand side, by treating them as values of the prenex form type on the left hand side.

In a similar way, Equation 2 suggests a simple treatment for constructors of datatypes with existentially quantified components:

$$(\forall a.\forall x.\tau \rightarrow T\ a) \quad = \quad \forall a.(\exists x.\tau) \rightarrow T\ a.$$

We might hope that the two remaining equations could be used to deal with selectors for datatypes with existentially quantified components, and with constructors for datatypes with polymorphic components, respectively. But the Curry-Howard isomorphism deals with the relationship between type theory and *intuitionistic* rather than classical predicate calculus, and Equations 3 and 4 are not valid in this setting. Instead, we are forced to make do with weaker implications. For example, the closest that we can get to the classical equivalence of Equation 3 is the implication:

$$(\exists x.P \Rightarrow Q) \quad \Rightarrow \quad P \Rightarrow (\exists x.Q)\,. \qquad (3')$$

A term with the type shown on the right-hand side is a function that, for each argument of type P, returns a result of some type $[\tau/x]Q$. The choice of the otherwise-unspecified witness type τ may depend on the particular value that the function is applied to; different argument values may produce results of different types. This is exactly the behaviour that we expect for a selector function for a datatype with an existentially quantified component. But compare this with the type on the left-hand side of the implication; the position of the quantifier in the formula $(\exists x.P \Rightarrow Q)$ indicates that the choice of a witness type τ is independent of any particular argument value of type P. Clearly, the two types are not equivalent, although the implication tells us that we can convert an arbitrary term of the left-hand type to a term of the right-hand type.

In a similar way, the closest that intuitionistic logic gets to the classical equivalence of Equation 4 is an implication:

$$(\exists x.Q \Rightarrow P) \quad \Rightarrow \quad (\forall x.Q) \Rightarrow P. \qquad (4')$$

It follows that certain terms of type $\forall a.\exists x.(\tau \rightarrow T\ a)$ can be substituted for terms of type $\forall a.(\forall x.\tau) \rightarrow T\ a$, the type of a constructor function for a datatype with a polymorphic component. Unfortunately, it does not help us to determine which terms of the latter type can be represented in this way. And even if that were not a problem, we would still need some form of top-level, existential quantification—a feature that is not usually supported in Hindley-Milner style type systems.

We find ourselves in a rather frustrating situation. If we are restricted to a language of types that allows only outermost quantification, then we can select from, but not construct datatypes with polymorphic components, and we can construct, but not select from datatypes with existentially quantified components. We cannot meet our goals if constructors and selectors are to be treated as normal, first-class functions.

2.2 Special syntax for constructors

The problems that we have seen are the result of restricting our attention to types in prenex form. In the context of logic, such restrictions seem rather artificial; given $(\forall x.Q) \Rightarrow P$ as a hypothesis, we can try to construct a proof of $(\forall x.Q)$ and then deduce P as a conclusion, as in the following derivation:

$$\dfrac{A \vdash (\forall x.Q) \Rightarrow P \qquad \dfrac{A \vdash Q \quad x \notin A}{A \vdash \forall x.Q}\,(\forall I)}{A \vdash P}\,(\to E)$$

Switching back from logic to terms and types, this is exactly the structure that we need to deal with a constructor for a datatype with a universally quantified component[1]:

$$\dfrac{K : (\forall x.\tau) \to T\ a \qquad \dfrac{A \vdash E : \tau \quad x \notin TV(A)}{A \vdash E : \forall x.\tau}\,(\forall I)}{A \vdash K\ E : T\ a}\,(\to E)$$

For this rule, there is no need to treat the constructor K, with its non-prenex form type, as a normal, first-class function. Instead, we view the application of K to E as a completely new syntactic construct—and as a clear hint for the type-checker.

The remaining problem, to provide access to a value stored in a datatype with an existentially quantified component, can also be dealt with by introducing a new syntactic construct. In this case, our inspiration comes from the elimination rule for existential quantification:

$$\dfrac{\exists x.P \quad \forall x.P \Rightarrow Q \quad x \notin Q}{Q}$$

If we consider a constructor $K : (\forall x.\tau) \to T\ a$, then we obtain a typing rule:

$$\dfrac{\dfrac{A \vdash E : T\ a}{A \vdash K^{-1}\ E : \exists x.\tau} \qquad \dfrac{\dfrac{A, x{:}\tau \vdash E' : \tau'}{A \vdash \lambda z.E' : \tau \to \tau'}\,(\to I)}{A \vdash \lambda z.E' : \forall x.\tau \to \tau'}\,(\forall I)}{A \vdash (\mathbf{case}\ E\ \mathbf{of}\ (K\ z) \to E') : \tau'}\,(\exists E)$$

with the side condition that $x \notin TV(A, \tau')$. An attractive feature of this approach is that soundness of our typing rules follows directly from the soundness of the corresponding rules in intuitionistic predicate logic.

We now have the key to understanding FCP—the type system introduced in this paper. By abandoning the first-class status of constructors and introducing new syntactic constructs in their place, we obtain the tools that we need to construct and access datatypes with universally or existentially quantified components.

[1] The notation $TV(A)$ used here denotes the set of type variables appearing free in A.

3 Examples

Before going on to the formal definition of FCP, we pause for some examples. These serve both to illustrate the expressiveness of the system, and to show the notation that is used in our prototype implementation, which is an extension of the Hugs [12] implementation of Haskell 1.3 [24].

The first example is an implementation for booleans using polymorphism and functions. Apart from passing them around as normal first-class entities, the only way that boolean values can be used is to make choices between alternatives. This leads us to a representation for booleans as functions of type $\forall a.a \to a \to a$, and there are only two interesting functions of this type: `true`, which always returns the first of its two arguments, and `false`, which always returns the second of its arguments. We can make this idea concrete using the definitions in Figure 1. Our implementation adopts the convention that variables like `a` in the definition of the `Boolean` datatype that are not bound on the left hand side of the definition are instead bound by an implicit universal quantifier. So we can (almost) think of `B` as a constructor of type $(\forall a.a \to a \to a) \to Boolean$. The corresponding selector function, `cond`, is just the familiar conditional and can be used to define standard operators like `and` and `or`. We have included type signature declarations for the operators defined here, and in later examples, as a form of documentation. However, they are not strictly necessary; the same types would have been obtained by the type inference algorithm.

```
data Boolean = B (a -> a -> a)

true, false :: Boolean
true         = B (\t f -> t)
false        = B (\t f -> f)

cond        :: Boolean -> a -> a -> a
cond (B b)   = b

and, or     :: Boolean -> Boolean -> Boolean
and x y      = cond x y false
or  x y      = cond x true y
```

Figure 1: An encoding of boolean values

This treatment of booleans is an example of well-known techniques that are used to encode standard datatypes in λ-calculus. The FCP encodings of natural numbers (Church numerals) in Figure 2, and of lists in Figure 3, are obtained in a similar way.

```
data Church = Ch ((a->a) -> (a->a))

unCh       :: Church -> (a -> a) -> a -> a
unCh (Ch n) = n

zero, one  :: Church
zero        = Ch (\f x -> x)
one         = Ch (\f x -> f x)

succ       :: Church -> Church
succ n      = Ch (\f x -> unCh n f (f x))

pred       :: Church -> Church
pred n      = fst (unCh n s z)
 where s (x,y) = (y,succ y)
       z       = (error "pred zero", zero)

iszero     :: Church -> Boolean
iszero n    = unCh n (\x -> false) true

add, mul   :: Church -> Church -> Church
add n m     = unCh n succ m
mul n m     = unCh n (add m) zero
```

Figure 2: An encoding of Church numerals

```
data List a = L ((a -> b -> b) -> b -> b)

fold :: List a -> (a -> b -> b) -> b -> b
fold (L f)
       = f

nil  :: List a
nil   = L (\c n -> n)

cons :: a -> List a -> List a
cons x xs
      = L (\c n -> c x (fold xs c n))

hd   :: List a -> a
hd l  = fold l (\x xs -> x) (error "hd []")

tl   :: List a -> List a
tl l  = fst (fold l c n)
 where c x (l,t) = (t, cons x t)
       n         = (error "tl []", nil)
```

Figure 3: An encoding of lists and folds

```
data Monad m
 = MkMonad (a -> m a)
           (m a -> (a -> m b) -> m b)

unit (MkMonad u b) = u
bind (MkMonad u b) = b

join      :: Monad m -> m (m a) -> m a
join m xss = bind m xss id

listMonad :: Monad [ ]
listMonad  = MkMonad unit bind
 where unit x        = [x]
       bind []     f = []
       bind (x:xs) f = f x ++ bind xs f

data Maybe a = Just a | Nothing

maybeMonad :: Monad Maybe
maybeMonad  = MkMonad unit bind
 where unit x          = Just x
       bind Nothing  f = Nothing
       bind (Just x) f = f x
```

Figure 4: Monads as first-class values

The next example, in Figure 4, demonstrates the combination of FCP and higher-order polymorphism in our prototype implementation, to provide a type-safe representation for monads [32] as first-class values. Much has been achieved using constructor class overloading [11] to explore the use of monads and monad transformers [2, 18]. As this example suggests, the same experiments can be repeated in FCP by packaging these items up as first-class values. In fact, FCP has exactly the features needed to describe the implementation of constructor classes by a source-to-source translation.

The ability to reify monads as first-class data structures was also an important part of Steele's work to construct programming language interpreters from reusable building blocks [31]. These ideas can be expressed very neatly in FCP, using essentially the same definitions as in Figure 4. By comparison, Steele found that he had to use a program specializer to circumvent problems caused by limitations in the type system of the version of Haskell that he was using at the time.

As an application of existential types, Figure 5 shows a portion of an implementation of an abstract stack datatype. The definition of `testExpr` produces a list of integers by mapping an operation over a list of stacks, each of which could have a different internal representa-

```
data Stack a
      = Stack xs              -- self
              (a -> xs -> xs) -- push
              (xs -> xs)      -- pop
              (xs -> a)       -- top
              (xs -> Bool)    -- empty

makeListStack :: [a] -> Stack a
makeListStack xs
 = Stack xs (:) tail head null

push          :: a -> Stack a -> Stack a
push x (Stack self push' pop top empty)
 = Stack (push' x self) push' pop top empty

top          :: Stack a -> a
top (Stack self push pop top' empty)
 = top' self

testExpr :: [Int]
testExpr  = map (top . push 1)
                [makeListStack [1,2,3],
                 makeListStack [4,5]]
```

Figure 5: A simple encoding of stack packages

tion, hidden by an existential quantifier. Note that our prototype implementation adopts the convention that variables beginning with an `x` are existentially quantified, avoiding the need for an explicit quantifier[2].

Our final example, in Figure 6, is an implementation of objects using the representation described by Pierce and Turner [27], and demonstrating the use of both existentially and universally quantified datatype components. For example, in this encoding, objects are represented by values of type `Obj m`, with a state component and a collection of methods; the type of the state, `xs`, is hidden using an existential quantifier. Classes, on the other hand, are represented by functions that are polymorphic in the 'final representation type', `f`, for objects of that class. Other items defined in Figure 6 include a function for constructing `new` instances of a class; a specific example—the ubiquitous `pointClass`; and the `ext` operator that is used to describe inheritance. The `NT` type—which might be used in a more general setting to describe natural transformations—is of particular interest here. While Pierce and Turner rely on higher-order subtyping, we use values of type `NT p q` as explicit coer-

[2] This convention is acceptable in a prototype, but is also rather ugly. We hope to find a more attractive alternative for use in future language designs.

```
data Obj m = MkObj xs          -- state
                   (m xs)      -- methods

data Class m s
 = MkClass ((f -> s)           -- extract
             -> (f -> s -> f)  -- overwrite
                -> m f         -- self
                   -> m f)

new             :: Class m s -> s -> Obj m
new (MkClass c) s = MkObj s m
 where m = c (\r -> r) (\_ r -> r) m

-- Natural transformations:
data NT m n = MkNT { coerce :: m a -> n a }

-- A specific example:
data PointM s = MkP{ set :: (s -> Int -> s),
                     get :: (s -> Int) }

point'Set     :: NT p PointM ->
                   (Obj p -> Int -> Obj p)
point'Set st (MkObj s m) i
 = MkObj (set (coerce st m) s i) m

pointClass :: Class PointM Int
pointClass = MkClass (\extr over self ->
              MkP{ set = \r i -> over r i,
                   get = \r   -> extr r })

-- Inheritance:
data Inc s n r
 = MkInc ((f -> r)          -- extract
          -> (f -> r -> f) -- overwrite
             -> s f         -- super methods
                -> n f      -- self  methods
                   -> n f) -- new methods

ext :: NT p q
       -> Class q s              -- super
          -> Inc q p r           -- increment
             -> (r -> s)         -- extract
                -> (r -> s -> r)-- overwrite
                   -> Class p r -- new class
ext st (MkClass sup) (MkInc inc) get put
 = MkClass (\g p self ->
    inc g p (sup (\s -> get (g s))
                 (\s t -> p s (put (g s) t))
                 (coerce st self))
           self)
```

Figure 6: The Pierce and Turner encoding of objects

cions; another alternative would have been to use (multiple parameter) constructor classes, but these are not currently supported in Hugs. It is beyond the scope of this paper to explain the Pierce and Turner encoding in any further detail; instead, we refer the interested reader to the original paper [27].

4 Formal Development

Our formal presentation of FCP begins with the Hindley-Milner type system whose type language, term language, and typing rules are summarized in Figure 7. The extensions that are needed to accommodate FCP are summarized in Figure 8.

Type Language:

$$\begin{array}{lcll} \sigma & ::= & \forall t.\sigma & \textit{polymorphic type} \\ & | & \tau & \textit{monotype} \\ \tau & ::= & t & \textit{type variable} \\ & | & \tau \to \tau' & \textit{function type} \end{array}$$

Term Language:

$$\begin{array}{lcll} E & ::= & x & \textit{variables} \\ & | & E\,E & \textit{application} \\ & | & \lambda x.E & \textit{abstraction} \\ & | & \mathbf{let}\ x = E\ \mathbf{in}\ E & \textit{local definition} \end{array}$$

Typing Rules:

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

$$(\to E) \quad \frac{A \vdash E : \tau' \to \tau \quad A \vdash E' : \tau'}{A \vdash E\,E' : \tau}$$

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

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

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

$$(\forall I) \quad \frac{A \vdash E : \sigma \quad t \notin TV(A)}{A \vdash E : \forall t.\sigma}$$

Figure 7: The Hindley-Milner Type System

Mechanisms for defining new datatypes and their associated constructor functions are clearly going to play an important part in any practical system based on FCP. For the formal development, we will assume that the type language of our system has been extended with a collection of new datatype constructors, T, and that the term language has been similarly extended with a family of constructor function constants, K. However, it is not necessary for us to go any further here in specifying how this information might be extracted from the notation used in particular source program texts. We also assume that each constructor K has been assigned a closed type of the form: $\sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\nu) \to \nu'$, where α, β, and γ represent (possibly empty) sequences of quantified variables. We use the notation:

$$K : (\forall\alpha.\exists\beta.\tau') \to \tau$$

to indicate that the type $(\forall\alpha.\exists\beta.\tau') \to \tau$ can be obtained by instantiating the variables γ in σ_K to particular types. Note that we do not make any assumptions about uses of the type constructors, T, in the types τ and τ', so the type inference mechanisms described here can be used in more general ways than our current focus on datatypes and constructor functions might suggest. For example, our framework could also be used to type Launchbury and Peyton Jones' `runST` construct [17], and Gill *et al.*'s `build` construct [3].

Type Language:

$$\begin{array}{lcll} \tau & ::= & T\ \tau_1\ \ldots\ \tau_n & \textit{datatype, arity}(T) = n \\ & | & \ldots & \end{array}$$

Term Language:

$$\begin{array}{lcll} E & ::= & K\ e & \textit{construction} \\ & | & \lambda(K\ x).E & \textit{decomposition} \\ & | & \ldots & \end{array}$$

Typing Rules, for each $K : (\forall\alpha.\exists\beta.\tau') \to \tau$:

$$(\mathit{make}) \quad \frac{A \vdash E : [\nu/\beta]\tau' \quad \alpha \notin TV(A)}{A \vdash K\ E : \tau}$$

$$(\mathit{break}) \quad \frac{A, x{:}[\nu/\alpha]\tau' \vdash E : \tau'' \quad \beta \notin TV(A, \tau'', \nu)}{A \vdash (\lambda(K\ x).E) : \tau \to \tau''}$$

Figure 8: Extensions for FCP

Note that we allow both universal and existential quantifiers in the type of datatype components; this avoids the need to treat constructors with existentially and universally quantified components as separate cases, although it does require slightly more complicated typing rules. The ordering of the quantifiers, placing existentials inside the universals is fairly arbitrary, but we have found this to be the most convenient choice in practical applications. Of course, programmers can control the ordering of quantifiers by embedding values of one datatype inside another.

To simplify the presentation, we have assumed that each constructor has precisely one argument. Constructors with multiple arguments can be modelled by extending the language with n-ary tuples; in fact, this could be achieved with an FCP encoding of pairs:

```
data Pair a b = MkPair ((a -> b -> c) -> c)
```

Construction and decomposition are described by FCP terms with a syntax that was chosen to reflect the close analogy with normal function application and abstraction. A more conventional, but complex notation for decomposition uses a **case** or **match** construct. For example, an expression of the form:

$$\textbf{case}\ E\ \textbf{of}\ (K\ x) \rightarrow E'$$

could be treated as syntactic sugar for $(\lambda(K\ z).E')\ E$. Pattern matching against values from datatypes with multiple constructors can be described using a 'fatbar' operator, as described by Peyton Jones [25].

The type checking rules for FCP are again justified by appealing to the underlying logic, as explained in Section 2.2. The actual derivations are complicated by the presence of both existential and universal quantifiers. For example, assuming $K : (\forall\alpha.\exists\beta.\tau') \rightarrow \tau$, the (*make*) rule is obtained from the derivation:

$$\dfrac{\dfrac{\dfrac{A \vdash E : [\nu/\beta]\tau'}{A \vdash E : \exists\beta.\tau'} \quad \alpha \notin TV(A)}{A \vdash E : \forall\alpha.\exists\beta.\tau'}}{A \vdash K\ E : \tau}$$

The derivation of (*break*) is a little more complex, but follows the same basic pattern as in Section 2.2. To simplify the presentation, we omit the terms at each node of the proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, \tau \vdash \tau}}{A, \tau \vdash \forall\alpha.\exists\beta.\tau'}\ \text{(a)}}{A, \tau \vdash [\nu/\alpha]\exists\beta.\tau'}}{A, \tau \vdash \exists\beta.[\nu/\alpha]\tau'}\ \text{(b)} \qquad \dfrac{\dfrac{A, \tau, [\nu/\alpha]\tau' \vdash \tau''}{A, \tau \vdash [\nu/\alpha]\tau' \rightarrow \tau''}}{A, \tau \vdash \forall\beta.[\nu/\alpha]\tau' \rightarrow \tau''}\ \text{(c)}}{\dfrac{A, \tau \vdash \tau''}{A \vdash \tau \rightarrow \tau''}}\ \text{(d)}$$

The step labelled (a) corresponds to stripping the constructor K from a value of type τ to obtain a value of type $\forall\alpha.\exists\beta.\tau'$. The side conditions $\beta \notin TV(\nu)$, $\beta \notin TV(A)$, and $\beta \notin TV(\tau'')$ are used in steps (b), (c), and (d), respectively.

From a language design perspective, it may be preferable to allow only one kind of local quantifier in the type of any given constructor; none of the examples in this paper actually use both kinds of quantification in a single constructor. It is easy to obtain the appropriate typing rules for this as special case of the (*make*) and (*break*) rules. For example, a constructor $U : (\forall\alpha.\tau') \rightarrow \tau$ with universally quantified components can be dealt with using the rules:

$$\dfrac{A \vdash E : \tau' \quad \alpha \notin TV(A)}{A \vdash K\ E : \tau} \qquad \dfrac{A, x{:}[\nu/\alpha]\tau' \vdash E : \tau''}{A \vdash (\lambda(K\ x).E) : \tau \rightarrow \tau''}$$

In a similar way, a constructor $K : (\exists\beta.\tau') \rightarrow \tau$ with existentially quantified components can be dealt with using the rules:

$$\dfrac{A \vdash E : [\nu/\beta]\tau'}{A \vdash K\ E : \tau} \qquad \dfrac{A, x{:}\tau' \vdash E : \tau'' \quad \beta \notin TV(A, \tau'')}{A \vdash (\lambda(K\ x).E) : \tau \rightarrow \tau''}$$

Another option would have been to include only universal quantification in the formal description, and to have dealt with existential quantifiers using an encoding:

$$\exists x.P \quad = \quad \forall y.(\forall x.P \Rightarrow y) \Rightarrow y.$$

We have not followed this approach in the current paper; first, because the encoding can sometimes be a bit awkward to use in practice, and second because we wanted to provide a reasonably symmetric treatment of the two forms of quantification.

5 Type inference for FCP

In this section, we describe a type inference algorithm for FCP, based on Milner's algorithm W [19]. The algorithm is both sound and complete with respect to the typing rules for FCP in Section 4. In other words, the algorithm succeeds if, and only if the input term is well-typed, in which case it calculates a principal type for that term. This is important because it allows programmers to take advantage of the expressiveness of FCP, without giving up on the convenience of type inference.

5.1 Unification

As usual, unification [30] plays a central role in the type inference process. Given two types τ and τ', the goal of a unification algorithm is to find the most general substitution U such that $U\tau = U\tau'$. In this paper, we use a modest extension of the standard algorithm that takes a set of variables, V, as an additional parameter. The algorithm treats each of these variables as a constant that can only be unified with itself, or with other variables that are not in V. As a result, none of the variables in V will be bound by the substitution U produced as a result of unification; in other words, the restriction of U to V, written $U|_V$, will be the identity substitution, id, on V. This is particularly important in the rules for constructing and decomposing values of datatypes with universally and existentially quantified components, respectively.

The algorithm is a straightforward modification of Robinson's original algorithm. The presentation in Figure 9 uses judgements of the form:

$$\tau \stackrel{U}{\sim} \tau' \bmod V$$

to indicate that the unification algorithm succeeds with a most general unifier U for the types τ and τ, such that $U|_V = id$. The substitution U is most general in the sense that, if $S\tau = S\tau'$ and $S|_V = id$, then there is a substitution R such that $S = RU$. Moreover, it is easy to show that the algorithm fails only if there are no substitutions S that satisfy these properties.

$$(id) \qquad \tau \stackrel{id}{\sim} \tau \bmod V$$

$$(var) \qquad \left.\begin{array}{l} \alpha \stackrel{[\tau/\alpha]}{\sim} \tau \bmod V \\ \tau \stackrel{[\tau/\alpha]}{\sim} \alpha \bmod V \end{array}\right\} \alpha \notin V \cup TV(\tau)$$

$$(fun) \qquad \frac{\tau \stackrel{U}{\sim} \nu \bmod V \qquad U\tau' \stackrel{U'}{\sim} U\nu' \bmod V}{(\tau \to \tau') \stackrel{U'U}{\sim} (\nu \to \nu') \bmod V}$$

Figure 9: Rules for unification.

5.2 A type inference algorithm

The rules in Figure 10 give a type inference algorithm for FCP, with one rule for each of the six syntactic constructs in the language. A successful invocation of the type inference algorithm is represented using a judgement of the form:

$$TA \stackrel{W}{\vdash} E : \tau \bmod V,$$

where the type assignment A, the term E, and the set of variables V are the inputs, and the substitution T and type τ are the outputs. The following theorem is important because it tells us that any typing produced by the type inference algorithm corresponds to a valid typing derivation under the original typing rules.

Theorem 1 (Soundness) *If $TA \stackrel{W}{\vdash} E : \tau \bmod V$, then $TA \vdash E : \tau$ and $T|_V = id$.*

Conversely, the following theorem indicates that, if a term has type τ under the original rules, then the type inference algorithm will succeed, and the inferred type will be at least as general as τ.

Theorem 2 (Completeness) *If $SA \vdash E : \tau$ and $S|_V = id$, then $TA \stackrel{W}{\vdash} E : \nu \bmod V$ for some T, ν, and there is a substitution R such that $S \approx RT$[3] and $\tau = R\nu$.*

For example, consider the special case of a well-typed expression E in a top-level environment A with $V = \emptyset$, and $TV(A) = \emptyset$ (and hence $SA = A$, for any substitution S). Together, the two theorems above tell us that the type inference algorithm will succeed with a typing $A \vdash E : \nu$ and that every possible type of E in A can be expressed as a substitution instance of ν.

6 From System F to FCP

The examples of first-class polymorphism and abstract datatypes that we have seen in previous sections can also be programmed directly in explicitly typed languages like System F, the polymorphic λ-calculus of Girard [4] and Reynolds [29], or higher-order variants like Fω. In fact, the FCP implementations for booleans, numbers, and lists in previous sections are direct translations of the standard System F encodings for these datatypes. The main difference is that FCP programs use constructor and selector functions where the System F uses type abstraction and type application, respectively. For example, compare our definition of the successor function:

$$succ \;=\; \lambda n.Ch\ (\lambda f.\lambda x.unCh\ n\ f\ (f\ x))$$

with the System F implementation:

$$\begin{array}{rcl} succ & = & \lambda n : Church. \\ & & \Lambda a.\lambda f : (a \to a).\lambda x : a.n\ a\ f\ (f\ x). \end{array}$$

Note also that the use of constructors and selectors in FCP provides enough additional type information to avoid the need for the type annotations on λ-bound variables in System F.

In the remainder of this section we show that any System F program can be expressed in FCP, essentially by replacing type abstractions and applications with constructors and selectors. A complication occurs because some System F types can be represented in several different ways in FCP, but we deal with this by defining a family of *conversions* that allow us to switch between different representations of any given type.

For reference, we summarize the type language, term language, and typing rules of System F in Figure 11.

[3] $S \approx RT$ means that the substitutions S and RT are equal, ignoring 'new' variables introduced during type checking.

$$(var)^{\mathrm{W}} \quad \frac{(x:\forall\alpha.\tau) \in A \quad \beta \text{ new}}{A \stackrel{\mathrm{W}}{\vdash} x : [\beta/\alpha]\tau \bmod V}$$

$$(\rightarrow E)^{\mathrm{W}} \quad \frac{TA \stackrel{\mathrm{W}}{\vdash} E : \tau \bmod V \quad T'\,TA \stackrel{\mathrm{W}}{\vdash} F : \tau' \bmod V \quad T'\tau \stackrel{U}{\sim} \tau' \rightarrow \alpha \bmod V \quad \alpha \text{ new}}{UT'\,TA \stackrel{\mathrm{W}}{\vdash} EF : U\alpha \bmod V}$$

$$(\rightarrow I)^{\mathrm{W}} \quad \frac{T(A_x, x:\alpha) \stackrel{\mathrm{W}}{\vdash} E : \tau \bmod V \quad \alpha \text{ new}}{TA \stackrel{\mathrm{W}}{\vdash} \lambda x.E : T\alpha \rightarrow \tau \bmod V}$$

$$(let)^{\mathrm{W}} \quad \frac{TA \stackrel{\mathrm{W}}{\vdash} E : \tau \bmod V \quad \sigma = Gen(TA, \tau) \quad T'(TA_x, x:\sigma) \stackrel{\mathrm{W}}{\vdash} F : \tau' \bmod V}{T'\,TA \stackrel{\mathrm{W}}{\vdash} (\textbf{let } x = E \textbf{ in } F) : \tau'}$$

$$(make)^{\mathrm{W}} \quad \frac{\sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \rightarrow \tau' \quad \alpha,\beta,\gamma \text{ new} \quad TA \stackrel{\mathrm{W}}{\vdash} E : \nu \bmod V \quad \nu \stackrel{U}{\sim} \tau \bmod V \cup \{\alpha\} \quad \alpha \notin TV(UTA)}{UTA \stackrel{\mathrm{W}}{\vdash} (K\ E) : U\tau' \bmod V}$$

$$(break)^{\mathrm{W}} \quad \frac{\sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \rightarrow \tau' \quad \alpha,\beta,\gamma \text{ new} \quad T(A_x, x:\tau) \stackrel{\mathrm{W}}{\vdash} E : \nu \bmod V \cup \{\beta\} \quad \beta \notin TV(TA, \nu, T\alpha)}{TA \stackrel{\mathrm{W}}{\vdash} (\lambda(K\ x).E) : T\tau \rightarrow \nu \bmod V}$$

Figure 10: Type inference algorithm W.

6.1 System F types to FCP

We begin by defining a mapping that associates each System F type σ with an FCP type σ^*. The mapping requires the definition of a family of datatypes, one for each type of the form $\forall t.\sigma$:

$$\textbf{data } T_{\forall t.\sigma}\ \alpha_1\ \ldots\ \alpha_n \quad = \quad Mk_{\forall t.\sigma}\ \sigma^*.$$

The parameters $\alpha_1, \ldots, \alpha_n$ used here are the free variables of $\forall t.\sigma$. Obviously, the set of type constructors $T_{\forall t.\sigma}$ is infinite, but only finitely many will be required in the translation of a given program. The required mapping can now be defined:

$$\begin{array}{lcl} t^* & = & t \\ (\sigma_1 \rightarrow \sigma_2)^* & = & \sigma_1^* \rightarrow \sigma_2^* \\ (\forall t.\sigma)^* & = & T_{\forall t.\sigma}\ \alpha_1\ \ldots\ \alpha_n \end{array}$$

(Again, the parameters $\alpha_1, \ldots, \alpha_n$ in the last line are the free variables of $\forall t.\sigma$.) It is easy to verify that this mapping takes the standard System F encodings of the boolean, number and list types to the corresponding FCP versions given in previous sections:

$$\begin{array}{lcl} (\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha)^* & = & \texttt{Boolean} \\ (\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)^* & = & \texttt{Church} \\ (\forall\beta.(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta)^* & = & \texttt{List}\ \alpha \end{array}$$

For each constructor function $Mk_{\forall t.\sigma}$, we can define a selector function:

$$\begin{array}{lcl} Mk_{\forall t.\sigma}^{-1} & :: & \forall\alpha_1 \ldots \forall\alpha_n.\forall t.(\forall t.\sigma)^* \rightarrow \sigma^* \\ Mk_{\forall t.\sigma}^{-1}\ (Mk_{\forall t.\sigma}\ x) & = & x \end{array}$$

The superscript on the selector is intended to emphasize a view of the constructor and selector as mutual inverses in an isomorphism that describes the packaging and unpackaging of polymorphic values. Of course, the choice of names for the type and value constructors and for the selectors is fairly arbitrary; in specific cases, we can adopt more intuitive naming schemes.

6.2 Conversion between FCP types

There are often several ways to represent a System F type in FCP, and the mapping described above gives us only one of several alternatives. For example, the encoding of $\sigma_1 = \forall t.(t \rightarrow Int)$ is $\sigma_1^* = S$ where:

$$\textbf{data } S \quad = \quad D\ (t \rightarrow Int).$$

Another way to obtain an encoding for σ_1 is to start with the more general type, $\sigma_2 = \forall t.(t \rightarrow s)$ with an encoding $\sigma_2^* = T\ s$ where:

$$\textbf{data } T\ s \quad = \quad C\ (t \rightarrow s)$$

and to note that $\sigma_1 = [Int/s]\sigma_2$. Thus System F values of type σ_1 could be represented as FCP values of type S, or of type $T\ Int$. Fortunately, although the two types are not equal, it is easy to repackage a value of either type as a value of the other by using an appropriate combination of constructors and selectors. The follow-

Type Language:

$$\begin{array}{rcll} \sigma & ::= & t & \textit{type variables} \\ & | & \sigma \to \sigma & \textit{function types} \\ & | & \forall t.\sigma & \textit{polymorphic types} \end{array}$$

Term Language:

$$\begin{array}{rcll} E & ::= & x & \textit{variables} \\ & | & E\,E & \textit{application} \\ & | & \lambda x\!:\!\sigma.E & \textit{abstraction} \\ & | & E\,\sigma & \textit{type application} \\ & | & \Lambda t.E & \textit{type abstraction} \end{array}$$

Typing Rules:

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

$$(\to E)\quad \frac{A \vdash E : \sigma' \to \sigma \quad A \vdash E' : \sigma'}{A \vdash E\,E' : \sigma}$$

$$(\to I)\quad \frac{A_x, x\!:\!\sigma' \vdash E : \sigma}{A \vdash \lambda x\!:\!\sigma.E : \sigma' \to \sigma}$$

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

$$(\forall I)\quad \frac{A \vdash E : \sigma \quad t \notin TV(A)}{A \vdash \Lambda t.E : \forall t.\sigma}$$

Figure 11: System F

ing typing derivations show the steps that are needed:

$$\cfrac{\cfrac{A \vdash E : T\ Int}{A \vdash C^{-1}\ E : t \to Int}}{A \vdash D\ (C^{-1}\ E) : S} \qquad \cfrac{\cfrac{A \vdash E : S}{A \vdash D^{-1}\ E : t \to Int}}{A \vdash C\ (D^{-1}\ E) : T\ Int}$$

In fact, we can generalize this construction to deal with any pair of System F types in which one is obtained from the other by substituting for a free variable:

Theorem 3 *For any System F types σ_1 and σ_2, any type variable t, and any FCP derivations:*

$$A \vdash E : ([\sigma_1/t]\sigma_2)^* \qquad A \vdash E' : [\sigma_1^*/t]\sigma_2^*$$

there is an effective algorithm for constructing FCP terms C and C' such that:

$$A \vdash C : [\sigma_1^*/t]\sigma_2^* \qquad A \vdash C' : ([\sigma_1/t]\sigma_2)^*$$

and $Erase(C) =_{\beta\eta} Erase(E)$, $Erase(C') =_{\beta\eta} Erase(E')$.

We refer to the terms C and C' as *conversions* of E and E', respectively. The notation $Erase(E)$ denotes the λ-term obtained from the FCP term E by deleting any occurrences of constructor or selector function symbols; this provides a semantics for E in some underlying λ-calculus (as in Milner's work [19]) where constructors and selectors for $T_{\forall t.\sigma}$ types are interpreted as identities. The significance of the theorem is that it allows us to turn a derivation for a term of type $[\sigma_1^*/t]\sigma_2^*$ into a derivation for a semantically equivalent term of type $([\sigma_1/t]\sigma_2)^*$. The proof, a construction of the conversions C and C', is a straightforward induction on σ_2. However, it is worth mentioning that the existence of conversions in both directions between the two types $[\sigma_1^*/t]\sigma_2^*$ and $([\sigma_1/t]\sigma_2)^*$ is essential because it allows us to deal with the anti-monotonicity in the first argument of the function type constructor.

6.3 System F terms to FCP

It remains to show how arbitrary System F programs can be converted to equivalent programs in FCP. Given a System F term E with type σ under the assumptions A, our goal is to find a corresponding FCP term E^* with type σ^* under the assumptions in A^*, where:

$$A^* = \{\, (x\!:\!\sigma^*) \mid (x\!:\!\sigma) \in A \,\}.$$

The existence of suitable translations is guaranteed by the following result:

Theorem 4 *For any System F derivation $A \vdash E : \sigma$, there is an FCP term E^* with $Erase_{\mathrm{F}}(E) =_{\beta\eta} Erase(E^*)$ such that $A^* \vdash E^* : \sigma^*$.*

The notation $Erase_{\mathrm{F}}(E)$ used here denotes the System F erasure of E, which is the λ-term obtained from E by deleting any uses of type abstraction or application. The proof of the theorem is by structural induction. Most of the cases are straightforward, but conversions are needed to deal with derivations involving the rule $(\forall E)$. To see this, consider a System F derivation that ends with the instantiation of a polymorphic type:

$$\cfrac{\cfrac{\vdots}{A \vdash E : \forall t.\sigma}}{A \vdash E\,\sigma' : [\sigma'/t]\sigma}$$

It follows by induction that $A^* \vdash E^* : (\forall t.\sigma)^*$ and hence that $A^* \vdash Mk_{\forall t.\sigma}^{-1}\ E^* : [\sigma'^*/t]\sigma^*$; now we can take a conversion to obtain the required typing.

We have now shown that arbitrary System F programs can be expressed in FCP. For example, the implementations of booleans, numbers and lists in previous sections can all be obtained in this way. The general encoding that we have used in this section can be a little awkward to use; for example, it treats each universal quantifier separately. But this will not cause problems in practice because programmers can make direct use of FCP and need not be restricted to the $T_{\forall t.\sigma}$ family of datatypes.

7 Conclusion and Discussion

We have described a modest extension of the Hindley-Milner type system that offers both the convenience of type inference and the expressiveness of first-class polymorphism. The conventional Hindley-Milner type system is already powerful enough for many programming tasks, but FCP provides an extra degree of flexibility that is useful in particular situations, without compromising the benefits of simple type inference. A prototype type-checker for FCP has been implemented as an extension of the Hugs implementation of Haskell. This has been used to test the programs included in the body of the paper, and seems to work well in practice.

7.1 Three dimensions of type inference

In general terms, the work described in this paper continues a program of research to explore extensions of the Hindley-Milner type system [6, 19]. Figure 12 illustrates three, largely orthogonal dimensions of the design space that we have explored in work to date: overloading, higher-order polymorphism, and, the main subject of the current paper, first-class polymorphism.

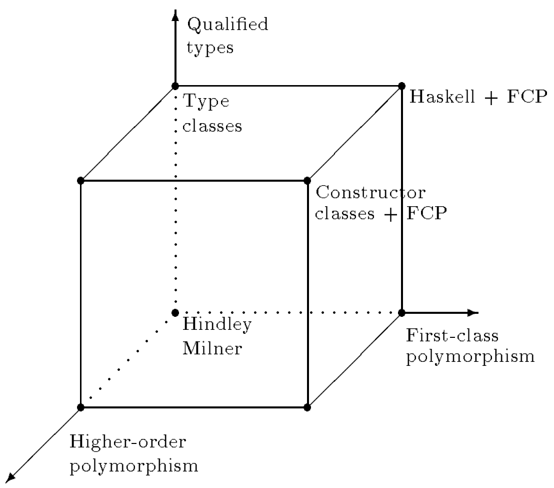

Figure 12: Extensions of Hindley-Milner typing

Type class overloading, in the form described by Kaes [13], and Wadler and Blott [33], was probably the first of these three extensions to be incorporated in a widely used programming language. The semantics of the system of type classes in the paper by Wadler and Blott was explained by a type-directed, source level translation into the Hindley-Milner type system; extra *dictionary* parameters were introduced to specify the meaning of overloaded operators at particular types. As such, type class overloading provides a convenient form of implicit parameterization. This has proved to be particularly useful for dealing with equality and arithmetic operations in the presence of a polymorphic type system.

Type classes were adopted as part of the design for Haskell [9], but the early proposals were extended in subtle ways to allow class definitions like the following:

```
class Foo a where
  bar :: a -> b -> [b]
```

The variable `b` in the type of the `bar` member is bound by an implicit universal quantifier, and suggests an implementation in which dictionary components can have polymorphic types. So, while Haskell type classes can be explained by a translation into FCP, they cannot, in general, be implemented by a translation into the standard Hindley-Milner type system. Implementors of Haskell side-stepped the problems of augmenting the language with general FCP-like constructs by choosing either a more powerful language like System F [5], or a simple untyped language [10] as a target for the translation of source programs.

The introduction of constructor classes [11] lead to the discovery of several new uses for overloading. Some of the best known applications—for example, using functors or monads—were inspired by work in category theory [2, 18]. As a simple example, a class of functors might be defined as follows:

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

The system of constructor classes was described as a combination of overloading and higher-order polymorphism, the latter referring to the ability to use variables ranging over different kinds of type constructors—like `f` in the example above. It is surprisingly easy to generalize the standard results for type inference in a Hindley-Milner framework to the higher-order case, so long as the language of type constructors is restricted to avoid problems with undecidable, higher-order unification. But, by itself, the resulting system is not particularly useful; in practice, just as there are no interesting terms of type $\forall\alpha.\forall\beta.\alpha \to \beta$, there are also no interesting terms of type $\forall\alpha.\forall f.\alpha \to f\ \alpha$. With hindsight, we can see that the expressiveness of constructor classes actually comes from the combination of higher-order polymorphism and hidden uses of first-class polymorphism.

For example, the `map` function described above would not be useful without the implicit universal quantification over the variables `a` and `b`. With observations like this, we are now in a better position to distinguish between the syntactic convenience of overloading and the expressiveness of first-class polymorphism.

7.2 Areas for future work

There is still much work to be done:

- Further dimensions: It is clear that there are many more dimensions to type inference than the three that we have focused on here, an observation that is supported by the extensive literature on the subject. For example, we consider the problems of tackling various notions of *extensibility* to be a particularly interesting direction for future work.
- Language design problems: For example, there is a significant overlap in the facilities for defining datatypes in FCP, classes in Haskell, and modules in Standard ML. This might prompt a search for more orthogonal language mechanisms supporting the same features.
- Technical problems: For example, we have not fully addressed the problems of integrating FCP with type class overloading to allow datatypes with components whose types are qualified by class constraints. To the best of our knowledge, only the special case of datatypes with existentially quantified components has been considered in work to date [15].

Type inference can be thought of as a compromise between the convenience of programming without type annotations and the expressiveness of explicitly-typed languages. Practical systems include elements of both extremes; for example, while languages like Standard ML and Haskell do not require type information for each λ-bound variable, the types assigned to constants and constructor functions play a critical role in the type inference process. Perhaps the most important aspect of the FCP type system introduced in this paper is the new perspective that it provides in helping us to understand a small part of the design space. However, we hope that it will also play a useful role in practical programming language designs.

Acknowledgements

Thanks to my friends and colleagues in the functional programming group at Nottingham for their valuable input to the development of the work described in this paper. A particular thank you to Benedict R. Gaster for his help in preparing the object example in Section 3.

References

[1] H.-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

[2] L. Duponcheel and E. Meijer. On the expressive power of constructor classes. In *Proceedings of the 1994 Glasgow Functional Programming Workshop*, Ayr, September 1994.

[3] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

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

[5] C. Hall, K. Hammond, W. Partain, S. Peyton Jones, and P. Wadler. The Glasgow Haskell compiler: a retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland*, July 1992. Springer Verlag Workshops in computing series.

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

[7] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

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

[9] P. Hudak and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0). Technical report, University of Glasgow, April 1990.

[10] M. P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

[11] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[12] M. P. Jones. Hugs 1.3 user manual. Technical Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, August 1996.

[13] S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88: European Symposium on Programming, Nancy, France*, New York, 1988. Springer-Verlag. Lecture Notes in Computer Science, 300.

[14] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, July 1992.

[15] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[16] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[17] J. Launchbury and S. P. Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[18] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 1995.

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

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

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

[22] M. Odersky and K. Läufer. Putting type annotations to work. In *Conference record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–67, St. Petersburg Beach, FL, January 1996. ACM press.

[23] N. Perry. *The implementation of practical functional programming languages*. PhD thesis, Imperial College, 1991.

[24] J. Peterson and K. Hammond (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0). Research Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996.

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

[26] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993.

[27] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[28] D. Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 321–346. Springer-Verlag, April 1994.

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

[30] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 1965.

[31] G. L. Steele Jr. Building interpreters by composing monads. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, OR, January 1994.

[32] P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.

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

[34] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, July 1994.