Type Classes with Existential Types

Konstantin Läufer

Department of Mathematical and Computer Sciences
Loyola University of Chicago
6525 North Sheridan Road
Chicago, IL 60626, USA
laufer@math.luc.edu
https://wwwmath.luc.edu/~laufer/

October 26, 1995

Abstract

We argue that the novel combination of type classes and existential types in a single language yields significant expressive power. We explore this combination in the context of higher-order functional languages with static typing, parametric polymorphism, algebraic data types, and Hindley-Milner type inference. Adding existential types to an existing functional language that already features type classes requires only a minor syntactic extension.

We first demonstrate how to provide existential quantification over type classes by extending the syntax of algebraic data type definitions and give examples of possible uses. We then develop a type system and a type inference algorithm for the resulting language. Finally, we present a formal semantics by translation to an implicitly-typed second-order λ-calculus and show that the type system is semantically sound.

Our extension has been implemented in the Chalmers Haskell B. system, and all examples from this paper have been developed using this system.

1 Introduction

In this paper, we combine two independent programming language constructs, *type classes* and *existential types*, in a single language. We explore this combination in the context of higher-order functional languages with static typing, parametric polymorphism, algebraic data types, and Hindley-Milner type inference. Although the combination of these features results in significant expressive power, adding existential types to an existing functional language that already provides type classes, such as Haskell, requires only a minor syntactic extension.

We first demonstrate how existential types over type classes can be added to the language by extending the syntax of algebraic data type definitions. We give examples illustrating how we express

- first-class abstract data types with user-defined interfaces,
- aggregates of different implementations of the same abstract data type,
- dynamic dispatching of operations with respect to the representation type, and
- separate interface and implementation hierarchies.

We then develop a type system and a type inference algorithm for the resulting language. Furthermore, we give a formal semantics by translation to an implicitly-typed second-order λ-calculus and show that the type system is semantically sound. Proofs for the various theorems can be found in the author's PhD thesis (Läufer, 1992).

For the sake of concreteness, our language is presented here as an extension to Haskell (Hudak et al., 1992), but other languages with similar type systems can be extended analogously. Our proposal has been implemented in the Chalmers Haskell B. system (Augustsson, 1993a). Thus all examples from this paper have been developed and tested using the `hbi` interpreter and are given in Haskell syntax.

In the remainder of this paper, Section 2 describes how algebraic data types can be extended with existential quantification over type classes. Section 3 contains a collection of examples. Section 4 presents a formal language that combines type classes and existential types. Section 5 and Section 6 develop a type system and a type inference algorithm for our language. Section 7 illustrates the translation of our language into a suitable target language. Section 8 concludes with an outlook on related and future work.

2 Algebraic Data Types with Existential Quantification over Type Classes

This section describes how abstract data types with user-defined interfaces can be provided by extending the syntax of algebraic data type definitions. While our extension can be applied to any language based on a polymorphic type system with algebraic data types and type classes, it has been implemented in the Chalmers Haskell B. system (Augustsson, 1993a) and is presented as an extension to Haskell.

Type classes (Kaes, 1988, Wadler and Blott, 1989) provide a systematic approach to ad-hoc operator overloading. Each *class declaration* introduces a new class name C and one or more new overloaded functions $f_1, \ldots, f_n$. Each type that supports a group of overloaded functions is declared as an *instance* of the corresponding class. Algorithmic type inference of principal types in the style of Hindley and Milner is possible for ML-like languages extended with type classes, such as Haskell (Chen et al., 1992, Nipkow and Prehofer, 1993).

Existential types (Mitchell and Plotkin, 1988, Cardelli and Wegner, 1985) are a formalization of the concept of abstract data type, such as the package in Ada (United States Department of Defense, 1983), the cluster in CLU (Liskov and Guttag, 1986), and the module in Modula-2 (Wirth, 1985). By stating that a value v has the existential(ly quantified) type $\exists\alpha.\tau$, we mean that v has type $[\tilde{\tau}/\alpha]\tau$ for some unique, anonymous type $\tilde{\tau}$. The existentially quantified type variable α stands for the representation type of the abstract data type. Existential types are introduced via the `pack` construct and eliminated via the `abstype` or `open` construct (Mitchell and Plotkin, 1988, Cardelli and Wegner, 1985). Abstract data types expressed in this way are first-class in the sense that their instances can be treated like ordinary values.

Perry (1990) incorporates abstract data types into a statically-typed functional language by allowing the component types of algebraic data types to be existentially quantified. The `pack` construct then corresponds to the application of a value constructor, and the `open` construct corresponds to pattern matching against a value constructor. Mitchell and Plotkin (1988) observe that a given expression can have many different existential types; thus each application of `pack` must state the resulting existential type intended. Perry's system satisfies this requirement by associating each existential type with a specific value constructor. Läufer and Odersky (1994) give a formal treatment of a related system and point out a technical error that occurs in Perry's published work (1990), though not in his implemented system. Our current work is based on Läufer and Odersky's formalism.

We now combine type classes and existential types as follows: By constraining the existentially quantified type variables to belong to certain type classes, we can require that the representation types support certain operations. It is in this sense that type classes serve as *interfaces* of abstract data types, as suggested by Läufer and Odersky (1991). Following Perry (1990), we syntactically extend algebraic data type definitions by dropping the restriction that only type variables that are *bound* as formal type parameters may appear in the component types of the data type. Any *free* type variable in a data type declaration is considered local to and existentially quantified in the component types of the value constructor in which it appears. Just as universally quantified type variables can be constrained by a *context* c of the form $(C_1\ u_1, \ldots, C_n\ u_n)$, stating that type variable u_i belongs to type class C_i, existentially quantified type variables can be constrained by *local contexts* $c_1, \ldots, c_n$ for the value constructors in which they appear. Thus the general form of a data type declaration is as follows:

$$\texttt{data}\ [c \Rightarrow]\ \ T\ \ u_1 \ldots u_k\ \ =\ \ [c_1 \Rightarrow] K_1\ t_{11} \ldots t_{1k_1}\ \ |\ \ \ldots\ \ |\ \ [c_n \Rightarrow] K_n\ t_{n1} \ldots t_{1k_n}$$

A value of such a data type is constructed by applying a value constructor K_i to values whose types are instances of the component types $t_{i1}, \ldots, t_{ik_i}$ of K_i. Both bound and free type variables in the component types must be replaced with types that satisfy the global context c and the local context c_i. Thus the type of a value constructor K_i is universally quantified over all bound type variables and those free type variables that appear in the component types of K_i:

$$K_i :: \forall u_1 \ldots u_k v_1 \ldots v_l .\ \tilde{c}_i \Rightarrow t_{i1} \to \ldots \to t_{ik_i} \to (T\ u_1 \ldots u_k)$$

In this type, $v_1, \ldots, v_l$ are all free type variables in the types $t_{i1}, \ldots, t_{ik_i}$ except $u_1, \ldots, u_{k_i}$, and $\tilde{c}_i$ is the largest subset of $c \cup c_i$ that constrains only those type variables free in the types $t_{i1}, \ldots, t_{ik_i}$. These free type variables are existentially quantified in the component types of the constructed value: As the value is constructed, we lose all information about the types the type variables are instantiated with, except that these instance types satisfy the combined context $\tilde{c}_i$.

A value of a data type is decomposed by pattern matching against the constructor used in constructing the value. When a value constructor K appears in a pattern $K\ \ x_1 \ldots x_m$, each existentially quantified type variable in the component types of K stands for some unknown type that satisfies the combined context $\tilde{c}$ for K and is replaced with a fresh, anonymous type in the types of the bound variables $x_1, \ldots, x_m$. Nothing is known about the anonymous types except that they satisfy the local context for the constructor K. Since the true identities of the anonymous types are not known statically, they must not escape the scope of the bound variables $x_1, \ldots, x_m$.

3 Examples

The following examples illustrate various applications of existential quantification combined with type classes. All examples were developed and tested using the Chalmers Haskell B. interpreter `hbi` (Augustsson, 1993a). We assume that all numerals have type `Int`, except in the last example, where they have type `Float`.

Lists of different implementations of the same abstract data type

The first two examples are variations on a simple example of existential quantification provided by Cardelli and Wegner (1985). The sole purpose of these examples is to illustrate how the type system works. Consider the following algebraic data type declaration:

```
data KEY a = MakeKey a (a -> Int)
```

This declaration introduces a new type constructor `KEY` with formal type parameter `a`. The type of the value constructor `MakeKey` is universally quantified over `a`:

```
MakeKey :: a -> (a -> Int) -> KEY a
```

By contrast, the declaration

```
data KEY = MakeKey a (a -> Int)
```

introduces a new (parameterless) type `KEY` and a value constructor `MakeKey` of type

```
MakeKey :: a -> (a -> Int) -> KEY
```

Since all applications of `MakeKey` have the same result type, `KEY`, we can construct the following list:

```
hetList = [MakeKey 5       id,
           MakeKey [1,2,3] length,
           MakeKey True    (\x -> if x then 1 else 0),
           MakeKey 9       (+ 1)]
```

Although `hetList` is a homogeneous list of type `[KEY]`, each of its elements can have a different representation type `a`.

We use pattern matching to extract the two components of a value of type `KEY`. The type variable `a` in the component type of the constructor `MakeKey` is existentially quantified. Thus the argument type of `f` and the type of `x` in the following function definition are identical, giving the function the type `KEY -> Int`:

```
whatkey (MakeKey x f) = f x
```

Since the anonymous types that replace existentially quantified type variables must not escape the scope in which they are introduced, the following function definition is ill-typed:

```
value (MakeKey x f) = x
```

The function `hetMin` finds the minimum of a list of `KEY`s with respect to the integer value obtained by applying the function `whatkey`:

```
hetMin [x]    = x
hetMin (x:xs) = let y = hetMin xs in
                    if whatkey x < whatkey y then x else y
```

and the expression `whatkey (hetMin hetList)` evaluates to `1`.

We observe that `KEY` is an abstract data type whose implementations *explicitly* bundle together a value of some type and a method on that type returning an `Int`. Each element of `hetList` may be viewed as a different implementation of the same abstract data type. Two implementations of `KEY` differ either in representation types, for example the second and third elements, or in methods, such as the first and last elements of `hetList`. Given a value of type `KEY`, we do not know its representation type, but it is guaranteed that we can safely apply the second component (the method) to the first component (the value).

By contrast, the traditional approach to dealing with heterogeneous lists in statically-typed languages is to introduce an algebraic data type with a separate constructor for each type allowed in the list. There are several disadvantages to this approach: First, one has to remember and use a number of different constructors. Second, the component types of the algebraic data type are not abstract; consequently, any operation can be applied to the components as long as it is type-correct. Finally and most importantly, algebraic types of this form do not support extension; when a new case is added to an algebraic type, every function that operates on the type has to be changed to include the new case. Our examples show that algebraic data types with existential component types simultaneously cure all three of these drawbacks.

Type classes as interfaces of abstract data types

Type classes provide a way of associating methods with a type in such a way that the methods are *implicitly* available for any value of this type. For example, the following type class `Key` specifies that its instances must implement a method `getkey` returning an integer:

```
class Key a where
  getkey :: a -> Int
```

Each instance type of `Key` declares how it implements the method `getkey`, for example:

```
instance Key Int  where getkey = id
instance Key Bool where getkey = \x -> if x then 1 else 0
instance Key [a]  where getkey = length
```

We can use the type class `Key` to define the *interface* of the abstract data type `KEY` by constraining the existentially quantified type variable `a` to be an instance of the type class `Key`. This interface is expressed by the constructor context `Key a` in the following type declaration:

```
data KEY = (Key a) => MakeKey a
```

We can still define lists of different `KEY`s:

```
hetList = [MakeKey 5, MakeKey [1,2,3], MakeKey True, MakeKey 9]
```

However, any two implementations of `KEY` with the same representation type now share the method implemented in the instance declaration corresponding to that type. Unlike in the explicit case, the method dictionaries are packed implicitly with the component values. This is reflected in the translation scheme presented in Section 7 and corresponds to *dynamic dispatching* in object-oriented programming languages, where a method associated with an object is selected and applied to the object at run time. Unlike the previous example, new implementations of the type `KEY` can only be added by declaring additional instances of the class `Key`.

As in the previous example, we compare different values of type `KEY` by mapping them to integer values. Since the method `getkey` can be applied to the component of any value of type `KEY`, we could now write the function `whatkey` as follows:

```
whatkey (MakeKey x) = getkey x
```

Instead, we choose to take a more systematic approach. We first make the type `KEY` an instance of the class `Key`. The method `getkey` is implemented in the same way as `whatkey`:

```
instance Key KEY where getkey (MakeKey x) = getkey x
```

We then declare `KEY` as an instance of the equality and ordered classes, thus taking advantage of a whole collection of predefined functions. Only the methods `(==)` and `(<=)` need to be implemented here since the class declarations for `Eq` and `Ord` contain default implementations for all their other methods:

```
instance Eq KEY where
  x == y = getkey x == getkey y

instance Ord KEY where
  x <= y = getkey x <= getkey y
```

Using the predefined polymorphic `minimum` function on lists of ordered values, the expression `getkey (minimum hetList)` evaluates to `1`.

Tree search

Mitchell and Plotkin (1988) provide two interesting applications of existential types, which we adapt to our language. The first example is a tree search function whose behavior is directed by a data algebra parameter. If a stack is passed as a parameter, the function performs depth-first search, while a queue parameter results in breadth-first search. Other data algebras could be passed as parameters as well. For example, a priority queue results in "best-first" search, where "best" is determined by the priority queue.

We first define a type class `Tree` that describes trees whose nodes have integer labels. The function `label` returns the label of a node. The function `isleaf` tests whether a node is a leaf, whereas `left` and `right` return the left and right subtrees of any nonleaf.

```
class Tree a where
  label  :: a -> Int
  isleaf :: a -> Bool
  left   :: a -> a
  right  :: a -> a
```

The following abstract data type describes data algebras parameterized by the type of the element stored in the structure. The first component corresponds to the storage structure itself, the second component to the insert operation, and the third component to the delete operation.

```
data STRUCT t = Struct s ((t, s) -> s) (s -> (t, s))
```

We now implement three data algebras, all of which are represented as lists, but have different operations. In the case of a stack, insert and delete correspond to push and pop at the head of the list. In the case of a queue, new elements are inserted at the tail of the list instead.

```
stack  = Struct [] (\(x, s) -> x : s)     (\s -> (head s, tail s))
queue  = Struct [] (\(x, s) -> s ++ [x]) (\s -> (head s, tail s))
```

A priority queue requires an ordering relationship on its element type, as expressed by the context `Ord t`. Insertion takes place at the head of the list, while deletion causes the smallest element according to the ordering relationship to be returned.

```
prique :: (Ord t) => STRUCT t

prique = Struct [] (\(x, s) -> x : s)
                   (\s -> let x = minimum s in (x, s \\ [x]))
```

We now define the `search` function. The first parameter is the tree to be searched, the second parameter is the label of the node we are looking for, and the third parameter is the data algebra that directs the search. Since we assume that at least one node has the desired label, there is no error checking. The result is the first subtree of `start` whose root has the label `goal`.

```
search :: (Tree t) => t -> Int -> STRUCT t -> t

search start goal (Struct empty insert delete) = find (start, empty)
  where
    next (node, st) = if isleaf node then
                        delete st
                      else
                        delete (insert (left node,
                                        insert (right node, st)))
    find (node, st) = if label node == goal then
                        node
                      else
                        find (next (node, st))
```

Perry's system (1990) does not allow the declaration of functions whose result type contains anonymous representation types of abstract data types. Therefore it would be impossible to write the auxiliary functions `next` and `find` as local functions. Instead, one would have to write them as global functions that operate on the encapsulated structure. This is undesirable for several reasons: First, one might not want the auxiliary functions to be accessible outside of `search`. Second, any local variables in `search` shared by `next` and `find` would have to be passed as parameters. Finally, operating on the encapsulated structure requires repeatedly opening and encapsulating the structure.

Since type classes in Haskell cannot be parameterized, we have to settle for trees with a fixed element type. We would have preferred to parameterize the type class tree by its element type. Two extensions of Haskell address the issue of parameterized type classes (Chen et al., 1992, Jones, 1993b). Combining parameterized type classes with existential types would give us abstract data types with parameterized interfaces. We plan to investigate this combination further.

Abstract streams

The second example by Mitchell and Plotkin (1988) shows how to express streams using existential quantification over type classes. The example employs the Sieve of Erastosthenes to produce a stream enumerating all prime numbers.

We first introduce a type class `State` to describe representation types of internal states of integer streams. It resembles the type class `Key` from the first example in this section: the only operation, `value`, produces an integer from a state.

```
class State a where
  value :: a -> Int
```

The following data type specifies that a stream has two components, a start state and a function to produce the next state from the start state. We use existential quantification over the type class `State` to express that we can produce a value of type integer from a state of the stream.

```
data STREAM = (State a) => Stream a (a -> a)
```

We now define possible instance types of `State`. Clearly, any integer can represent the state of a stream of integers. So can any stream of integers, if we interpret the value of the internal state of the stream as the value of the state that the stream represents.

```
instance State Int where
  value = id

instance State STREAM where
  value (Stream s n) = value s
```

The `sift` function removes from a stream of integers `s` all numbers divisible by the first value of `s`.

```
sift (Stream start next) =
  let n = value start
      f state = if (value state) `mod` n == 0 then
                  f (next state)
                else
                  state
  in
      Stream (f start) (\x -> f (next x))
```

Thus, if `s` is the stream 2, 3, 4, … , then the sequence formed by taking the first value of each stream `s`, `sift s`, `sift (sift s))`, … is the sequence of all primes. The stream of these streams can be written as

```
sieve = Stream (Stream 2 (+ 1)) sift
```

Finally, a stream of integers can be converted to an (infinite) list of integers as follows:

```
tolist (Stream start next) =
   (value start) : (tolist (Stream (next start) next))
```

The i-th prime is then given by `(tolist sieve) !! i`.

Composition of a list of functions

The algebraic data types in the preceding examples have only one constructor. Data types with several constructors are possible as well; any existentially quantified type variables are local to the component type of the constructor in which they appear.

The following type describes lists of functions, in which the type of each function would allow it to be composed with the next. For notational convenience, we declare the first constructor as right-associative:

```
infixr `FunCons`

data FunList a b = (a -> c) `FunCons` (FunList c b) | FunOne(a -> b)
```

The universally quantified type variables `a` and `b` correspond to the argument type of the first and the result type of the last function, respectively; the existentially quantified type variable `c` represents the intermediate types arising during the composition of two functions. We can now construct lists of composable functions, for example:

```
funOne  = FunOne id

funList = (\x -> x * x)   `FunCons`
          ((==) 9)        `FunCons`
          (\x -> (x, x))  `FunCons`
          funOne
```

We use a recursive function to apply the function resulting from the successive composition to an argument. Since the recursive call to `apply` has a different type from the one defined, we must enable polymorphic recursion by giving an explicit type signature:

```
apply :: FunList a b -> a -> b

apply (FunOne f)        x = f x
apply (f `FunCons` fl) x = apply fl (f x)
```

Evaluation of the expression `apply funList 3` then results in `(True, True)`. We can even count the number of functions in the composition:

```
numberComposites :: Num c => (FunList a b) -> c

numberComposites (FunOne f) = 1
numberComposites (f `FunCons` fl) = 1 + numberComposites fl
```

The expression `numberComposites funList` evaluates to `4`.

Points and colored points

The following example demonstrates how object-oriented concepts can be modeled using existential quantification over type classes. We start out with a two-level class hierarchy of points and colored points. In object-oriented terminology, this is inheritance at the *interface level*, used to establish relationships between the abstract properties of classes.

```
data Color = Red | Green | Blue deriving Eq

type Pair = (Float, Float)

class Point p where
  move :: p -> Pair -> p
  pos  :: p -> Pair

class (Point p) => ColorPoint p where
  color  :: p -> Color
  paint  :: p -> Color -> p
```

Next, we define two implementations of class `Point`, one based on cartesian coordinates:

```
data CartPoint = C Pair

instance Point CartPoint where
  move (C(x,y)) (dx,dy) = C(x + dx, y + dy)
  pos (C p) = p
```

and one on polar coordinates:

```
data PolarPoint = P Pair

instance Point PolarPoint where
  move p (dx,dy) = cart2polar (x + dx, y + dy)
```

```
        where (x,y) = pos p
              cart2polar (x,y) = P(sqrt(x * x + y * y), atan2 y x)
  pos (P(r,a)) = (r * cos a, r * sin a)
```

As in the previous examples, we define an abstract data type with type class `Point` as its interface:

```
data POINT = (Point p) => MakePoint p
```

By making type `POINT` an instance of class `Point`, we provide dynamic dispatching of the methods in the interface class:

```
instance Point POINT where
  move (MakePoint p) q = MakePoint (move p q)
  pos  (MakePoint p) = pos p
```

We now declare a type constructor that simply adds a field of type `Color` to any instance of the class `Point`:

```
data (Point p) => COLORPOINT p = MakeColPoint p Color
```

We now add colored points to our class hierarchy in two steps. First, we state that any application of the type constructor `COLORPOINT` to an instance of the class `Point` again results in an instance of `Point`:

```
instance (Point p) => Point (COLORPOINT p) where
  move (MakeColPoint p c) d = MakeColPoint (move p d) c
  pos  (MakeColPoint p c) = pos p
```

Second, we state that any application of `COLORPOINT` to an instance of the class `Point` belongs to the class `ColorPoint` as well:

```
instance (Point p) => ColorPoint (COLORPOINT p) where
  color (MakeColPoint p c) = c
  paint (MakeColPoint p c) d = MakeColPoint p d
```

These two instance declarations state that extending any instance type of the class `Point` by a `Color` component results in an instance of the classes `Point` and `ColorPoint`. Thus we automatically get a colored version of both implementations of `Point`, `CartPoint` and `PolarPoint`. In object-oriented terminology, this is inheritance at the *implementation level* for code reuse.

The following list of points contains various combinations of uncolored, colored, cartesian, and polar points:

```
pointList = [MakePoint(P(5,0)),
             MakePoint(MakeColPoint (C(3,4)) Blue),
             MakePoint(C(2,7)),
             MakePoint(MakeColPoint (P(5,pi/4)) Red)]
```

Once we make a colored point abstract by applying the constructor `Point`, we can only apply the operations `move` and `pos` in class `Point`, but not the operations in class `ColorPoint`. Thus we cannot directly obtain any color information from an abstract (colored) point. We could, however, make indirect use of this information if we modified the second last instance declaration such that, for example, red points moved slower and appeared in a closer position than blue points.

We can now evaluate expressions such as

```
map (\p -> move p (1, 2)) pointList
```

resulting in

```
[MakePoint(P(6.32, 0.32)),
 MakePoint(MakeColPoint (C(4.00, 6.00)) Blue),
 MakePoint(C(3.00, 9.00)),
 MakePoint(MakeColPoint (P(7.16, 0.89)) Red)]
```

4 The Formal Language Exell

In this section, we formally describe the syntax of expressions and types in our language, Exell. Exell is an extension of Mini-Haskell (Nipkow and Prehofer, 1993) with user-defined algebraic data types. While Mini-Haskell is not strictly a subset of the Haskell language, it captures the essential features of Haskell relating to type classes. The resulting language is a λ-calculus with **let**-expressions, extended with class, instance, and data type declarations, and with expressions for constructing, examining, and decomposing values belonging to data types. A program is a sequence of declarations followed by an expression. The syntax of Exell expressions is shown in Figure 1. For syntactic convenience, a list of objects $s_1, \ldots, s_n$ is abbreviated by $\overline{s_n}$; for example, $t(\tau_1, \ldots, \tau_n)$ is written as $t(\overline{\tau_n})$.

Identifiers	x, C
Constructors	K
Expressions	$e \;=\; x \mid (e_1 e_2) \mid \lambda x.e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid K \mid \textbf{is}\ K \mid \textbf{let}\ K\ x = e_1\ \textbf{in}\ e_2$
Declarations	$d \;=\; \textbf{data}\ t(\overline{\alpha_n{:}S_n}) = \rho \mid \textbf{class}\ C\ \textbf{is}\ \forall\alpha{:}C.\forall\overline{\alpha_n : S_n}.\tau \mid \textbf{inst}\ t : \left(\overline{S_n}\right)C\ \textbf{is}\ e$
Programs	$p \;=\; d\ p \mid e$

Figure 1: Syntax of Exell expressions

The type syntax of Exell is given in Figure 2. It includes recursive types ρ and user-defined type constructors t. Skolem types κ are used to type identifiers that are bound by a pattern-matching **let**-expression and whose type is existentially quantified. Skolem types are drawn from a collection of uninterpreted ground types; thus a fresh Skolem type is unique and distinct from all other types except itself. Explicit existential types arise only in the component types of user-defined type constructors; thus the syntax for type schemes includes the case $\eta \to \tau$. Unlike Haskell, which allows value constructors to have several curried arguments, Exell uses argument tuples instead. Both approaches are equivalent since all curried arguments must be present when a Haskell constructor is used in a pattern.

In typed languages such as ML, values are classified by types in judgments of the form $e : \tau$. In Exell, as well as in Haskell, types are classified by *sorts* in judgments such as $\tau : S$, stating that type τ is in sort S. Sorts are finite sets of classes, and $\tau : S$ means that τ belongs to each class in S. The empty sort is denoted by $\varnothing$, and the class C is an abbreviation for the singleton sort $\{C\}$.

Type classes	C, D	
Sorts	$S, T = \{C_1, \ldots, C_n\}$	
Type variables	α, β	
Type constructors	t	
Skolem types	κ	
Recursive types	ρ	$= \mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$
Types	τ	$= unit \mid bool \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \rightarrow \tau' \mid \kappa \mid t(\tau_1, \ldots, \tau_n)$
Existential types	η	$= \exists\alpha{:}S.\eta \mid \tau$
Type schemes	σ	$= \forall\alpha{:}S.\sigma \mid \eta \rightarrow \tau \mid \tau$

Figure 2: Syntax of Exell types

Since classes with multiple functions and subclasses are syntactic sugar (Chen et al., 1992), Exell classes are not ordered, and each class declares only one overloaded function. For convenience, we use the same name for the class and the function. Sorts are ordered by their natural subset relationship; semantically, if $S' \subseteq S$, then S' is more general than S.

We illustrate the syntax of Exell by translating the second example from Section 3 into Exell:

class *Key* **is** $\forall\alpha : Key.\ \alpha \rightarrow int$
inst *int* : *Key* **is** $\lambda x.\ x$
inst *bool* : *Key* **is** $\lambda x.$ **if** x 1 0
inst *list* : (Ω) *Key* **is** *length*
data $KEY = \mu\gamma.\ K\ (\exists\beta : Key.\ \beta)$
inst *KEY* : *Key* **is** $\lambda x.$ **let** $K\ z = x$ **in** *Key z*
inst *KEY* : *Eq* **is** $\lambda xy.$ *Eq* (*Key x*) (*Key y*)
inst *KEY* : *Ord* **is** $\lambda xy.$ *Ord* (*Key x*) (*Key y*)
let *hetList* = [*K* 5, *K* [1, 2, 3], *K True*, *K* 9] **in**
    Key (*minimum hetList*)

The `case`-expression and the related pattern matching for function parameters in source-level Haskell syntax correspond to nested **if**-expressions with **is**-expressions as conditions and pattern-matching **let**-expressions for the different cases. The Haskell example

```
data T a = K a | L Int | M
...
case x of K y -> 1
          L z -> z
          M   -> 0
```

translates to Exell as follows:

data $T(\alpha) = \mu\gamma.\ K\ \alpha + L\ int + M\ unit$
. . .
if (**is** $K\ x$)
  1
  (**if** (**is** $L\ x$)
    (**let** $L\ z = x$ **in** z)
    0)

5 Sort Inference and Type Inference for Exell

In this section, we present a syntax-directed type inference system for Exell. The rules for *sort inference* determine to what sort a given type belongs. The rules for *type inference* rely on the rules for sort inference and determine to what type a given expression belongs.

Sort inference

We follow Nipkow and Prehofer (1993) in our treatment of sort inference. The sorts of type variables are recorded in a *sort context* Γ, a mapping from type variables to sorts such that $Dom(\Gamma) = \{\alpha | \Gamma\alpha \neq \emptyset\}$ is finite. Sort contexts are written as $[\alpha_1{:}S_1, \ldots, \alpha_n{:}S_n]$, where $\{\alpha_1, \ldots, \alpha_n\} \supseteq Dom(\Gamma)$ are distinct type variables. The extension of Γ to map a type variable α to a sort S is denoted by $\Gamma[\alpha{:}S]$. For a set of type variables V, the restriction of Γ to variables not in V is $\Gamma \setminus V = [\alpha{:}\Gamma\alpha | \alpha \in Dom(\Gamma) - V]$.

The behavior of type constructors is stated by *arity declarations* of the form $t : (\bar{S}_n)\, S$. Such a declaration means that an application of t to types $\tau_1, \ldots, \tau_n$ of sorts $S_1, \ldots, S_n$, respectively, has the sort S. A set of arity declarations is called a *signature* Σ. A *default* arity declaration of the form $t : (\bar{S}_n)\, \emptyset$ guarantees that a type constructor is always applied to a fixed number n of argument types of the correct sorts; all other arity declarations are associated with instance declarations and have the form $t : (\bar{S}_n)\, C$.

The judgment $\Gamma, \Sigma \vdash \tau : S$ states that the Exell type τ belongs to sort S. The rules for sort inference are given in Figure 3. In the TVAR rule, the sort of a type variable is looked up in the sort context. In the TCON rule, the sort of a type constructor application is determined by looking up a declaration for the type constructor in the signature and checking that the actual type arguments have the correct sorts. In the SUB and UNION rules, the sort assigned to a type is generalized and specialized, respectively.

Type inference

Our type inference system builds on the one given by Nipkow and Prehofer (1993), which extends the original system of Damas and Milner (1982) by the notion of sorts. To facilitate algorithmic type inference, our system is syntax-directed; there is one type rule for each case in the grammar. An *environment* is a finite mapping $E = [x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n]$ from identifiers to type schemes. The

$$\text{TVAR}\quad \frac{\Gamma\alpha = S}{\Gamma, \Sigma \vdash \alpha : S} \qquad \text{TCON}\quad \frac{t : \left(\overline{S_n}\right) S \in \Sigma \qquad \Gamma, \Sigma \vdash \tau_i : S_i \qquad i = 1 \ldots n}{\Gamma, \Sigma \vdash t(\overline{\tau_n}) : S}$$

$$\text{SUB}\quad \frac{\Gamma, \Sigma \vdash \tau : S \qquad S' \subseteq S}{\Gamma, \Sigma \vdash \tau : S'} \qquad \text{UNION}\quad \frac{\Gamma, \Sigma \vdash \tau : S_1 \qquad \Gamma, \Sigma \vdash \tau : S_2}{\Gamma, \Sigma \vdash \tau : S_1 \cup S_2}$$

Figure 3: Type inference rules for classes and sorts

domain of E is $Dom(E) = \{x_1, \ldots, x_n\}$. $E[x{:}\sigma]$ is a new environment that maps x to σ and all y in $Dom(E)$ to $E(y)$. The free type variables in E are given by $FV(E) = FV(\sigma_1) \cup \ldots \cup FV(\sigma_n)$. The free Skolem types in a type τ are given by $FS(\tau)$; FS generalizes to environments analogously to FV. A *substitution* is a finite mapping from type variables to types. Substitutions are denoted by θ and δ; $\varnothing$ denotes the empty substitution. We define $Dom(\theta) = \{\alpha \mid \theta\alpha \neq \alpha\}$.

We define the following two relations between types:

Definition 5.1 The type scheme $\sigma' = \forall\overline{\alpha'_m{:}S'_m}.\tau'$ is a *generic instance* of $\sigma = \forall\overline{\alpha_n{:}S_n}.\tau$ under Γ and Σ, written $\Gamma, \Sigma \vdash \sigma \geq \sigma'$, iff there exists a substitution θ such that

(i) $\theta\tau = \tau'$,

(ii) $Dom(\theta) \subseteq \{\overline{\alpha_n}\}$,

(iii) $\Gamma\left[\overline{\alpha'_m{:}S'_m}\right], \Sigma \vdash \theta\alpha_i : S_i$ for $i = 1 \ldots n$, and

(iv) $\{\overline{\alpha'_m}\} \cap FV(\sigma) = \varnothing$.

Similarly, the type scheme $\sigma' = \forall\overline{\alpha'_m{:}S'_m}.\eta' \rightarrow \tau'$ is a *generic instance* of $\sigma = \forall\overline{\alpha_n{:}S_n}.\eta \rightarrow \tau$ under Γ and Σ iff there exists a substitution θ such that $\theta(\eta \rightarrow \tau) = \eta' \rightarrow \tau'$ and (ii), (iii) and (iv) hold as above.

Definition 5.2 The existential type $\eta' = \exists\overline{\alpha'_m{:}S'_m}.\tau'$ is a *generic generalization* of $\eta = \exists\overline{\alpha_n{:}S_n}.\tau$ under Γ and Σ, written $\Gamma, \Sigma \vdash \eta \leq \eta'$, iff there exists a substitution θ such that

(i) $\theta\tau = \tau'$,

(ii) $Dom(\theta) \subseteq \{\overline{\alpha_n}\}$,

(iii) $\Gamma\left[\overline{\alpha'_m{:}S'_m}\right], \Sigma \vdash \theta\alpha_i : S_i$ for $i = 1 \ldots n$, and

(iv) $\{\overline{\alpha'_m}\} \cap FV(\eta) = \varnothing$.

The following are examples of the ordering relations involving existential quantification:

$$[], \{int : C\} \vdash \forall\alpha : C\,.\ (\exists\beta : D\,.\,\beta \times (\beta \to \alpha)) \to \alpha \geq (\exists\beta : D\,.\,\beta \times (\beta \to int)) \to int$$

$$[], \{int : C, list : (C)\,D\} \vdash \exists\beta : D\,.\,\beta \times (\beta \to int) \leq list(int) \times (list(int) \to int)$$

The judgment $\Gamma, \Sigma, E \vdash p : \tau$ states that the Exell program p is of type τ under the signature Σ and the assumptions in Γ and E.

The typing rules for declarations are shown in Figure 4. The rules for class and instance declarations are the same as in Mini-Haskell. In a class declaration, the environment E is extended by a type declaration for the overloaded function C; according to the expression syntax given in Figure 1, the type $\forall\alpha{:}C.\sigma$ of C must not contain any existential quantifiers. In an instance declaration, the signature Σ is extended by an arity declaration for the type constructor t. In a recursive data type declaration, a default arity declaration for the data type t is added to the signature Σ, and the value constructors K_i are recorded in the environment with explicit universal and existential quantification for use by the CONS, TEST, and PAT rules below. The declaration rules can be applied backwards to build up Σ and E, which are then used to type the expression e.

$$\text{DATA}\ \frac{\Gamma, \Sigma \cup \{t{:}\left(\overline{S_n}\right)\varnothing\}, E\left[K_i{:}\forall\overline{\alpha_n{:}S_n}.\left[t(\overline{\alpha_n})/\beta\right]\eta_i \to t(\overline{\alpha_n})\middle| i = 1\ldots m\right] \vdash p : \tau}{\Gamma, \Sigma, E \vdash \textbf{data}\ t(\overline{\alpha_n{:}S_n}) = \mu\beta.K_1\eta_1 + \ldots + K_m\eta_m\ ; p : \tau}$$

$$\text{CLASS}\ \frac{\Gamma, \Sigma, E\,[C{:}\forall\alpha{:}C.\sigma] \vdash p : \tau}{\Gamma, \Sigma, E \vdash \textbf{class}\ C\ \textbf{is}\ \forall\alpha{:}C.\sigma\ ; p : \tau}$$

$$\text{INST}\ \frac{\begin{array}{cc}\Gamma\left[\overline{\alpha_n{:}S_n}, \overline{\beta_m{:}T_m}\right], \Sigma, E \vdash e : \left[t(\overline{\alpha_n})/\alpha\right]\tau & \{\overline{\beta_m}\} = FV(\tau) - \{\alpha\} \\ E(C) = \forall\alpha{:}C.\forall\overline{\beta_m{:}T_m}.\tau & \Gamma, \Sigma \cup \{t{:}\left(\overline{S_n}\right)C\}, E \vdash p : \tau'\end{array}}{\Gamma, \Sigma, E \vdash \textbf{inst}\ t : \left(\overline{S_n}\right)C\ \textbf{is}\ e\ ; p : \tau'}$$

Figure 4: Type inference rules for declarations

The four typing rules shown in Figure 5 are the same as in the Mini-Haskell system. They are used in the typing of variables, abstractions, applications, and **let**-expressions.

Three new rules are given in Figure 6; they are used to type value constructors, **is**-expressions, and pattern-matching **let**-expressions, respectively. We explain each of the new rules in turn. The CONS rule specifies that the existentially quantified type variables in the argument type of a constructor are replaced with types of appropriate sorts when the constructor is used. Hence these type variables behave as if they were universally quantified, and the type of the constructor

$$\text{VAR}\quad \frac{E(x) = \forall\overline{\alpha_n{:}S_n}.\tau \qquad \Gamma, \Sigma \vdash \tau_i : S_i \qquad i = 1\ldots n}{\Gamma, \Sigma, E \vdash x : [\overline{\tau_n/\alpha_n}]\tau}$$

$$\text{ABS}\quad \frac{\Gamma, \Sigma, E\,[x{:}\tau_2] \vdash e : \tau_1}{\Gamma, \Sigma, E \vdash \lambda x.e : \tau_2 \to \tau_1}$$

$$\text{APP}\quad \frac{\Gamma, \Sigma, E \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma, \Sigma, E \vdash e_2 : \tau_2}{\Gamma, \Sigma, E \vdash (e_1\ e_2) : \tau_1}$$

$$\{\overline{\alpha_n}\} = FV(\tau_1) - FV(E)$$

$$\text{LET}\quad \frac{\Gamma[\overline{\alpha_n{:}S_n}], \Sigma, E \vdash e_1 : \tau_1 \qquad \Gamma, \Sigma, E[x{:}\forall\overline{\alpha_n{:}S_n}.\tau_1] \vdash e_2 : \tau_2}{\Gamma, \Sigma, E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

Figure 5: Type inference rules for expressions

reflects the type given in Section 2. The TEST rule ensures that the predicate **is** K is applied only to arguments whose type is the same as the result type of the constructor K. Finally, the PAT rule governs the typing of pattern-matching **let**-expressions. It requires that the expression e_1 is of the same type as the result type of the constructor K. The body e_2 is typed under the environment extended with a typing for the bound identifier x. The new Skolem types $\kappa_1, \ldots, \kappa_m$ must not appear in the environment E; this ensures that they do not appear in the type of any identifier free in e_2 other than x. The rule also guarantees that the Skolem types do not appear in the result type τ'. Since the Skolem types $\kappa_1, \ldots, \kappa_m$ replace the existentially quantified type variables of sorts $S_1, \ldots, S_m$ in the component type of K, the body of the **let**-expression must be typed under an *extended* signature containing appropriate instance declarations for $\kappa_1, \ldots, \kappa_m$. The pattern-matching **let**-expression is monomorphic in the sense that the type of the bound variable x is not generalized. This restriction is sufficient to guarantee a type-preserving translation into a target language (see Section 7).

The following theorem states that Exell is a conservative extension of Mini-Haskell:

Theorem 5.3 For any Mini-Haskell program p, $\Gamma, \Sigma, E \vdash p : \tau$ iff $\Gamma, \Sigma, E \vdash_{MH} p : \tau$.

The theorem still holds if we extend Mini-Haskell to include recursive data types and pattern-matching **let**-expressions without existential quantification.

6 Computing Principal Types

In this section, we present the type inference algorithm $W_\exists$ and demonstrate that it computes

$$\text{CONS}\quad \frac{\Gamma, \Sigma \vdash E(K) \geq \left(\exists \overline{\beta_m{:}T_m}.\tau\right) \to t(\overline{\tau_n}) \qquad \Gamma, \Sigma \vdash \tau'_i : T_i \qquad i = 1 \ldots m}{\Gamma, \Sigma, E \vdash K : \left[\overline{\tau'_m / \beta_m}\right]\tau \to t(\overline{\tau_n})}$$

$$\text{TEST}\quad \frac{\Gamma, \Sigma \vdash E(K) \geq \eta \to t(\overline{\tau_n})}{\Gamma, \Sigma, E \vdash \mathbf{is}\ K : t(\overline{\tau_n}) \to bool}$$

$$\text{PAT}\quad \frac{\begin{array}{c}\Gamma, \Sigma \vdash E(K) \geq \left(\exists \overline{\beta_m{:}T_m}.\tau\right) \to t(\overline{\tau_n}) \qquad \Gamma, \Sigma, E \vdash e_1 : t(\overline{\tau_n}) \\ T_i = \{C^i_1, \ldots, C^i_{k_i}\} \qquad i = 1 \ldots m \qquad \{\overline{\kappa_m}\} \cap (FS(\tau') \cup FS(E)) = \varnothing \\ \Gamma, \Sigma \cup \bigcup_{i=1}^{m} \{\kappa_i{:}()C^i_1, \ldots, \kappa_i{:}()C^i_{k_i}\}, E\left[x{:}\left[\overline{\kappa_m/\beta_m}\right]\tau\right] \vdash e_2 : \tau'\end{array}}{\Gamma, \Sigma, E \vdash \mathbf{let}\ K\ x = e_1\ \mathbf{in}\ e_2 : \tau'}$$

Figure 6: Type inference rules for expressions involving existential types

principal types for well-typed Exell expressions.

Nipkow and Prehofer (1993) show that *coregular* signatures guarantee unique most general unifiers, which are a sufficient condition for the existence of principal types (see Appendix A). A signature Σ is called coregular if for all type constructors t and all classes C the set

$$D_\Sigma(t, C) = \{\overline{S_n} \mid t{:}\left(\overline{S_n}\right)C \in \Sigma\}$$

is either empty or contains a greatest element w.r.t. the component-wise ordering of the $\overline{S_n}$. If Σ is coregular, let $Dom_\Sigma(t, C)$ return the greatest element of $D_\Sigma(t, C)$ or fail if $D_\Sigma(t, C)$ is empty.

Coregular signatures are guaranteed if we require the following, simpler context condition for the instance declarations in an Exell program:

> For every class C and type constructor t, there is at most one instance declaration **inst** $t : \left(\overline{S_n}\right)C$.

A significant difference between our contribution and Nipkow and Prehofer's work (1993) is that signatures are no longer fixed since the body of a pattern-matching **let**-expression has to be typed under an extended signature. Because the extended signature contains only one instance declaration for each Skolem type, it is also coregular, provided that the original signature satisfied the context condition.

We are now ready to present the type inference algorithm $W_\exists$. The parameters of $W_\exists$ are a set of symbols (type variables and Skolem types) V, a substitution θ, a sort context Γ, a signature Σ, an environment E, and an expression e. $W_\exists$ returns a quadruple $(V, \Gamma, \theta, \tau)$, where $\theta\tau$ is the type of e under the context Γ and the signature Σ. The set V contains all "used" symbols, that is, type variables and Skolem types that occur in τ, θ, or E. Thus a new symbol is one that does not occur in V.

For an environment E and a substitution θ, we define $\theta E = [x{:}\theta(E(x)) \mid x \in Dom(E)]$. We call E a *closed* environment if $FV(E) = \varnothing$. The *free variables* of a substitution θ are given by

$$FV(\theta) = Dom(\theta) \cup \bigcup_{\alpha \in Dom(\theta)} FV(\theta\alpha)$$

A substitution θ *obeys* Γ *in the context of* Γ' *and* Σ, written $\Gamma', \Sigma \vdash \theta : \Gamma$, iff $\Gamma', \Sigma \vdash \theta\alpha : \Gamma\alpha$ for all $\alpha \in Dom(\Gamma)$.

The algorithm follows the syntax-directed type inference rules, hence there is one case for each rule. The cases for declarations are shown in Figure 7, where $geninst_{\Gamma, \Sigma}(\sigma \geq \sigma')$ checks whether σ' is a generic instance of σ. The four cases in Figure 8 are identical to algorithm I in Nipkow and Prehofer's paper (1993); the function $unify_{\Sigma}$ for unification of types with sort constraints is described in Appendix A. The three additional cases given in Figure 9 deal with value constructors, **is**-expressions, and pattern-matching **let**-expressions.

$$W_{\exists}(V, \theta, \Gamma, \Sigma, E, \textbf{data } t(\overline{\alpha_n{:}S_n}) = \mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \; ; p) =$$

$$W_{\exists}\Big(V, \theta, \Gamma, \Sigma \cup \{t : \big(\overline{S_n}\big)\varnothing\},$$

$$E\Big[K_i{:}\forall\overline{\alpha_n{:}S_n}.\,\big[t(\overline{\alpha_n})/\beta\big]\eta_i \to t(\overline{\alpha_n}) \,\Big|\, i = 1\ldots m\Big], p \Big)$$

$$W_{\exists}(V, \theta, \Gamma, \Sigma, E, \textbf{class } C \textbf{ is } \forall\alpha{:}C.\sigma \; ; p) = W_{\exists}(V, \theta, \Gamma, \Sigma, E\,[C{:}\forall\alpha{:}C.\sigma], p)$$

$$W_{\exists}(V, \theta, \Gamma, \Sigma, E, \textbf{inst } t : \big(\overline{S_n}\big)C \textbf{ is } e \; ; p) =$$

$$\begin{array}{l}
\text{let} \quad (V_1, \Gamma_1, \theta_1, \tau_1) = W_{\exists}(V, \theta, \Gamma, \Sigma, E, e) \\
\qquad \{\overline{\alpha_m}\} = FV(\theta_1\tau_1) \\
\qquad \Sigma_1 = \Sigma \cup \{t{:}\big(\overline{S_n}\big)C\} \\
\text{in} \\
\qquad \text{if } geninst_{\Gamma, \Sigma_1}(E(C) \geq \forall\overline{\alpha_m{:}\Gamma_1\alpha_m}.\theta_1\tau_1) \\
\qquad \text{then } W_{\exists}(V_1, \theta, \Gamma, \Sigma_1, E, p)
\end{array}$$

Figure 7: Type inference algorithm for declarations

The remainder of this section presents the lemmas and theorems needed to establish the correctness and principal types results. The first two lemmas state that an instantiation or a generalization still holds after a substitution is applied to both types involved, provided that the substitution obeys the sort context.

$$
\begin{array}{lll}
W_{\exists}(V, \theta, \Gamma, \Sigma, E, x) & = \text{let} & \forall \overline{\alpha_n : S_n}.\tau \;=\; E(x) \\
 & & \{\overline{\beta_n}\} \cap V \;=\; \varnothing \\
 & \text{in} & \left(V \cup \{\overline{\beta_n}\}, \Gamma\left[\overline{\beta_n : S_n}\right], \theta, \left[\overline{\beta_n / \alpha_n}\right]\tau \right) \\
W_{\exists}(V, \theta, \Gamma, \Sigma, E, \lambda x.e) & = \text{let} & \alpha \notin V \\
 & & (V', \Gamma', \theta', \tau) \;=\; W_{\exists}(V \cup \{\alpha\}, \theta, \Gamma, \Sigma, E[x{:}\alpha], e) \\
 & \text{in} & (V', \Gamma', \theta', \alpha \to \tau) \\
W_{\exists}(V, \theta, \Gamma, \Sigma, E, (e_1 e_2)) & = \text{let} & (V_1, \Gamma_1, \theta_1, \tau_1) \;=\; W_{\exists}(V, \theta, \Gamma, \Sigma, E, e_1) \\
 & & (V_2, \Gamma_2, \theta_2, \tau_2) \;=\; W_{\exists}(V_1, \theta_1, \Gamma_1, \Sigma, E, e_2) \\
 & & \alpha \notin V_2 \\
 & & (\Gamma', \theta') \;=\; \mathit{unify}_{\Sigma}(\Gamma_2, \theta_2\tau_1 = \theta_2\tau_2 \to \alpha) \\
 & \text{in} & (V_2 \cup \{\alpha\}, \Gamma', \theta'\theta_2, \alpha) \\
\multicolumn{3}{l}{W_{\exists}(V, \theta, \Gamma, \Sigma, E, \textbf{let } x = e_1 \textbf{ in } e_2) =} \\
 & \text{let} & (V_1, \Gamma_1, \theta_1, \tau_1) \;=\; W_{\exists}(V, \theta, \Gamma, \Sigma, E, e_1) \\
 & & \{\overline{\alpha_n}\} \;=\; FV(\theta_1\tau_1) - FV(\theta_1 E) \\
 & \text{in} & W_{\exists}\left(V_1, \theta_1, \Gamma_1 \setminus \{\overline{\alpha_n}\}, \Sigma, E\left[x{:}\forall \overline{\alpha_n : \Gamma_1\alpha_n}.\theta_1\tau_1\right], e_2\right)
\end{array}
$$

Figure 8: Type inference algorithm for expressions

Lemma 6.1 If $\Gamma, \Sigma \vdash \sigma \geq \tau$ and $\Gamma', \Sigma \vdash \theta : \Gamma$, then $\Gamma', \Sigma \vdash \theta\sigma \geq \theta\tau$.

Lemma 6.2 If $\Gamma, \Sigma \vdash \eta \leq \tau$ and $\Gamma', \Sigma \vdash \theta : \Gamma$, then $\Gamma', \Sigma \vdash \theta\eta \leq \theta\tau$.

The next lemma states that a typing judgment still holds after a substitution is applied to the type of the expression, provided that the substitution obeys the sort context.

Lemma 6.3 If $\Gamma, \Sigma, E \vdash e : \tau$ and $\Gamma', \Sigma \vdash \theta : \Gamma$, then $\Gamma', \Sigma, \theta E \vdash e : \theta\tau$.

The syntactic soundness theorem states that any typing computed by algorithm $W_{\exists}$ can be proved under the type inference rules.

Theorem 6.4 (Syntactic soundness) If $W_{\exists}(V, \theta, \Gamma, \Sigma, E, e) \;=\; (V', \Gamma', \theta', \tau)$ and $Dom(\Gamma) \cup FV(\theta) \cup FV(E) \subseteq V$, then

(i) $\Gamma', \Sigma, \theta' E \vdash e : \theta'\tau$

$$
\begin{array}{ll}
W_\exists(V, \theta, \Gamma, \Sigma, E, K) & = \textsf{let } \forall\overline{\alpha_n : S_n}.\left(\exists\overline{\beta_m : T_m}.\tau\right) \to t(\overline{\alpha_n}) \; = \; E(K) \\
& \qquad \{\overline{\alpha'_n}, \overline{\beta'_m}\} \cap V \; = \; \varnothing \\
& \quad \textsf{in } \; (V \cup \{\overline{\alpha'_n}, \overline{\beta'_m}\}, \Gamma\left[\overline{\alpha'_n : S_n}, \overline{\beta'_m : T_m}\right], \theta, \\
& \qquad\qquad \left[\overline{\alpha'_n / \alpha_n}, \overline{\beta'_m / \beta_m}\right]\tau \to t(\overline{\alpha'_n})) \\
W_\exists(V, \theta, \Gamma, \Sigma, E, \mathbf{is}\ K) & = \textsf{let } \forall\overline{\alpha_n : S_n}.\eta \to t(\overline{\alpha_n}) \; = \; E(K) \\
& \qquad \{\overline{\alpha'_n}\} \cap V \; = \; \varnothing \\
& \quad \textsf{in } \; \left(V \cup \{\overline{\alpha'_n}\}, \Gamma\left[\overline{\alpha'_n : S_n}\right], \theta, t(\overline{\alpha'_n}) \to \mathit{bool}\right)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{W_\exists(V, \theta, \Gamma, \Sigma, E, \mathbf{let}\ K\ x = e_1\ \mathbf{in}\ e_2) =} \\
& \textsf{let } \forall\overline{\alpha_n : S_n}.\left(\exists\overline{\beta_m : T_m}.\tau\right) \to t(\overline{\alpha_n}) \; = \; E(K) \\
& \qquad \{\overline{\alpha'_n}\} \cap V \; = \; \varnothing \\
& \qquad (V_1, \Gamma_1, \theta_1, \tau_1) \; = \; W_\exists(V \cup \{\overline{\alpha'_n}\}, \theta, \Gamma, \Sigma, E, e_1) \\
& \qquad (\Gamma', \theta') \; = \; \mathit{unify}_\Sigma(\Gamma_1, \theta_1\tau_1 = t(\overline{\alpha'_n})) \\
& \qquad \{\overline{\kappa_m}\} \cap V_1 \; = \; \varnothing \\
& \qquad (V_2, \Gamma_2, \theta_2, \tau_2) \; = \; W_\exists\Big(V_1 \cup \{\overline{\kappa_m}\}, \theta'\theta_1, \Gamma', \\
& \qquad\qquad \Sigma \cup \bigcup_{i=1}^{m} \{\kappa_i : ()C \mid C \in T_i\}, E\left[x : \left[\overline{\kappa_m / \beta_m}\right]\theta'\tau\right], e_2\Big) \\
& \textsf{in} \\
& \qquad \textsf{if } \{\overline{\kappa_m}\} \cap (FS(\theta_2\tau_2) \cup FS(\theta_2 E)) \; = \; \varnothing \\
& \qquad \textsf{then } \; (V_2, \Gamma_2, \theta_2, \tau_2)
\end{array}
$$

Figure 9: Type inference algorithm for expressions involving existential types

(ii) $Dom(\Gamma') \cup FV(\theta') \cup FV(E) \subseteq V$

(iii) $\Gamma', \Sigma \vdash \theta' : \Gamma$

Definition 6.5 Let E be a closed environment. The type scheme $\sigma = \forall\overline{\alpha_n{:}S_n}.\tau$ is a *principal type* of an expression e w.r.t Σ and E if $\left[\overline{\alpha_n{:}S_n}\right], \Sigma, E \vdash e : \tau$ and if $\left[\overline{\alpha'_m{:}S'_m}\right], \Sigma, E \vdash e : \tau'$ implies $\left[\overline{\alpha'_m{:}S'_m}\right], \Sigma \vdash \sigma \geq \tau'$ where $\{\overline{\alpha'_m}\} \subseteq FV(\tau')$.

The following lemma and theorem establish syntactic completeness and the principal type property: If it can be proved under the type inference rules that an expression has a type τ, then the type inference algorithm computes a type that is at least as general as τ.

Lemma 6.6 If $\Gamma', \Sigma, E' \vdash e : \tau'$ where $E' = \delta'\theta E$ and $\Gamma', \Sigma \vdash \delta' : \Gamma$, then there exist V and δ_1 such that

(i) $Dom(\Gamma) \cup FV(\theta) \cup FV(E) \subseteq V$

(ii) $W_\exists(V, \theta, \Gamma, \Sigma, E, e) = (V_1, \Gamma_1, \theta_1, \tau_1)$

(iii) $E' = \delta_1\theta_1 E$

(iv) $\tau' = \delta_1\theta_1\tau_1$

(v) $\Gamma', \Sigma \vdash \delta_1 : \Gamma_1$

Theorem 6.7 (Syntactic completeness and principal types) If $\Gamma', \Sigma, E \vdash e : \tau'$ and E is closed, then $W_\exists(\emptyset, \emptyset, [], \Sigma, E, e) = (V, \Gamma, \theta, \tau)$ and $\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\theta\tau$ is a principal type of e w.r.t. Σ and E, where $\{\overline{\alpha_n}\} = FV(\theta\tau)$.

The restriction to closed environments is justified since **class** and **inst** declarations cannot introduce free type variables into E.

7 A Translational Semantics

In this section, we present a semantics for Exell by translation to a suitable target language. Our approach is based on the original compile-time translation scheme by Wadler and Blott (1989), which was further developed by Hall, Hammond, Peyton Jones, and Wadler (1992). The basic idea is to eliminate classes in favor of run-time *method dictionaries* that contain instances for particular types of the overloaded functions associated with a class. An identifier given a polymorphic type scheme in the original environment is typed as a function in the translated environment; the translated type has dictionary arguments for each class that constrains a type variable in the original type, and the result type of the function is the same as the original type. The translation is type-preserving in the sense that every well-typed Exell program translates to a well-typed target program.

Informal treatment of the translation

We first explain the translation informally by translating the second example in Section 3 into an implicitly-typed language with a Haskell-like syntax. As in the original article by Wadler and Blott (1989), we declare a new type for each class declaration to represent the corresponding method dictionaries. In this case, we introduce the type constructor `KeyD` corresponding to the class `Key`. All dictionaries for this class are created using the value constructor `KeyDict`.

```
data KeyD a = KeyDict (a -> Int)
```

The function `getk` selects from a method dictionary of type `KeyD` the (only) method it contains:

```
getk (KeyDict g) = g
```

Each instance declaration of the class `Key` translates to the declaration of a method dictionary of type `KeyD`. Corresponding to the instance `Key Int`, we declare a dictionary of type `KeyD Int`, and so forth:

```
keyDInt     = KeyDict id
keyDBool    = KeyDict (\x -> if x then 1 else 0)
keyDIntList = KeyDict length
```

An application of the function `getkey` to a value translates to an expression selecting the only method from a dictionary of type `KeyD` and applying that method to the value. For example, the expression `getkey [1,2,3]` translates to `(getk KeyDIntList) [1,2,3]` and evaluates to `3`.

Existentially quantified type variables can occur in the component types of algebraic data types. Furthermore, each existentially quantified type variable may be constrained by one or more type classes. The functions required by these type classes are implicitly bundled with the component values and are available when the component values are accessed. This bundling is made explicit in the translation: each type variable that is constrained by one or more type classes requires including one or more dictionaries in the component type of the algebraic data type. In our example, the type variable `a` is an instance of the class `Key`; hence the translated data type `KEY'` contains a dictionary of type `KeyD a` in addition to the value of type `a`:

```
data KEY' = MakeKey' (KeyD a) a
```

Each application of the original value constructor `MakeKey` is translated to an application of the new constructor `MakeKey'`, and an appropriate method dictionary is supplied as an argument:

```
hetList' = [MakeKey' keyDInt 5,     MakeKey' keyDIntList [1,2,3],
            MakeKey' keyDBool True, MakeKey' keyDInt 9]
```

When a value of type `KEY` is decomposed, two components are available: a value of some type `t` and a suitable dictionary of type `KeyD t` containing a function that can be selected and applied to the value. This can be seen in the translated version of the `whatkey` function:

```
whatkey' (MakeKey' keyDa x) = (getk keyDa) x
```

It is not surprising that our translation of the second example in Section 3 almost exactly results in the first example, where a method was provided explicitly as a component of the data type.

Finding a suitable target language

Translation into an ML-like language is insufficient since the dictionaries are passed as function arguments but used polymorphically. The following Haskell program illustrates this problem:

```
class C a where f :: a -> b -> (a, b)
h x = (f x 3, f x True)
```

According to the original translation scheme (Wadler and Blott, 1989) used in Haskell B. (Augustsson, 1993b), this program is translated to the following program with explicit method dictionaries instead of classes:

```
h = \C -> \x -> ((f C) x 3), (f C) x True))
```

The expression `f C` is the component of the dictionary `C` corresponding to the overloaded function `f`. The problem is that `C` is used in two different polymorphic instances! Therefore the translated program cannot be typed in the ML system.

Hall et al. (1992) solve this problem by translating Haskell into the second-order λ-calculus. Similarly, we translate Exell into the language *MPS*, an implicitly typed second-order λ-calculus with existentially quantified and recursive types (MacQueen et al., 1986), whose properties are summarized in Appendix B. Since the *MPS* type system is semantically sound, we can prove semantic soundness of the Exell type system by showing that our translation is type-preserving. We use MPS as our target language instead of the (explicitly typed) second-order λ-calculus for two reasons: First, it directly supports existentially quantified and recursive types. Second, since it is implicitly typed, it provides a concise notation and closely corresponds to the way programs are executed in the sense that no type information is maintained at run-time.

Formal aspects of the translation

In our translation scheme, we assume that the classes in a sort are always listed in the same, fixed order, for example, in their order of declaration. A method dictionary for a class C and a type constructor t is a single function C_t corresponding to the *instance function* implementing the overloaded class function for that type constructor; in the case of a parameterized type constructor, the method dictionary takes additional dictionaries as parameters corresponding to the sorts of the formal type parameters. The function $d_\Sigma(\tau, C)$ generates the method dictionary for type τ and class C under signature Σ:

$$d_\Sigma(\alpha, C) \quad = \alpha^C$$

$$d_\Sigma(t(\overline{\tau_n}), C) = C_t\ d_\Sigma(\tau_1, C_1^1)\ldots d_\Sigma(\tau_1, C_{k_1}^1)\ldots d_\Sigma(\tau_n, C_1^n)\ldots d_\Sigma(\tau_n, C_{k_n}^n) \text{ where}$$

$$\{\overline{S_n}\} = Dom_\Sigma(t, C) \text{ and } S_i = \{C_1^i, \ldots, C_{k_i}^i\} \text{ for } i = 1\ldots n.$$

A class definition of the form **class** C **is** $\forall\alpha{:}C.\sigma$ gives rise to a corresponding type definition $C(\alpha) = MPS(\sigma)$. Thus we can type the dictionaries as

$$[] \vdash_{\text{MPS}} d_\Sigma(\tau, C) : C(MPS(\tau))$$

The type translation function MPS is defined as follows, assuming $S_i = \{C_1^i, \ldots, C_{k_i}^i\}$ for $i = 1\ldots n$:

$$
\begin{aligned}
MPS(\forall\overline{\alpha_n{:}S_n}.\tau) &= \forall\overline{\alpha_n}.C^1_1(\alpha_1) \to \ldots \to C^1_{k_1}(\alpha_1) \to \ldots \to C^n_{k_n}(\alpha_n) \to \ldots \to C^n_{k_n}(\alpha_n) \to MPS(\tau)\\
MPS(\exists\overline{\alpha_n{:}S_n}.\tau) &= \exists\overline{\alpha_n}.\left(C^1_1(\alpha_1) \times \ldots \times C^1_{k_1}(\alpha_1) \right) \times \ldots \times \left(C^n_{k_n}(\alpha_n) \times \ldots \times C^n_{k_n}(\alpha_n) \right) \times MPS(\tau)\\
MPS(t(\overline{\tau_n})) &= \left[\overline{MPS(\tau_n)/\alpha_n}\right] (\mu\beta.\ldots + K\ MPS(\eta) + \ldots) \text{ if } t(\overline{\alpha_n{:}S_n}) = \mu\beta.\ldots + K\eta + \ldots\\
MPS(\tau) &= \tau \text{ for all other types } \tau
\end{aligned}
$$

The type translation function extends to environments as follows:

$$MPS(E) = [x{:}MPS(E(x)) \mid x \in Dom(E)]$$

Thus, if an identifier x has the type $\forall\overline{\alpha_n{:}S_n}.\tau$ in E, it has the type $MPS(E)(x) = MPS(\forall\overline{\alpha_n{:}S_n}.\tau)$ in the translated environment. $MPS(\forall\overline{\alpha_n{:}S_n}.\tau)$ is a polymorphic function type with parameter types $C^i_j(\alpha_i)$ and result type $MPS(\tau)$; in the type system of our target language, the parameters can again be used polymorphically. Although the type translation function MPS does not appear explicitly in the typing rules below, it captures the implicit types of the dictionaries generated by the translation.

The syntax-directed translation scheme is integrated in the type inference rules. The judgment $\Gamma, \Sigma, E \vdash p : \tau \Rightarrow \tilde{p}$ states that the Exell program p has type τ and translation $\tilde{p}$ under the signature Σ and the assumptions in Γ and E. We now discuss the translation rules in turn:

Figure 10 shows the translation rules for declarations. A data type declaration is not translated at all. The overloaded function in a class declaration is translated to the identity function that receives and returns a method dictionary. An instance declaration for a class C and a type constructor t is translated to a binding for the corresponding instance function C_t.

Figure 11 contains the translation rules for Haskell expressions. Abstractions and applications are translated to abstractions and applications of their translated subexpressions. An identifier is provided with method dictionaries for each class required by the sorted type variables in its type scheme. According to the definition of method dictionaries, a free variable in a type results in corresponding free identifiers in the dictionary for that type. In a **let**-expression, any free identifiers that arise from generic type variables in the translation of the bound expression are captured by λ-bindings.

Figure 12 gives the translation rules for Exell expressions involving existential quantification and recursive data types. A value constructor K is translated to a function that attaches the tag K to its component value; in addition, dictionaries for the existentially quantified type variables must be grouped with the component value. This technique has its analogy in object-oriented programming languages, where an object includes a method dispatch table; in typical implementations, however, references to dictionaries instead of copies of dictionaries are included in the objects. An **is**-expression is translated to a function that inspects the tag of a pair. The translation of a pattern-matching **let**-expression first requires the tag to be inspected; then the method dictionary components are bound to identifiers for the instance functions corresponding to the Skolem types; finally the value component is bound to the original bound identifier.

$$\text{DATA}\quad \frac{\Gamma, \Sigma \cup \{t{:}\left(\overline{S_n}\right)\varnothing\}, E\left[K_i{:}\forall\overline{\alpha_n{:}S_n}.\left[t(\overline{\alpha_n})/\beta\right]\eta_i \to t(\overline{\alpha_n}) \,\middle|\, i = 1\ldots m\right] \vdash p : \tau \Rightarrow \tilde{p}}{\Gamma, \Sigma, E \vdash \textbf{data}\ t(\overline{\alpha_n{:}S_n}) = \mu\beta.K_1\eta_1 + \ldots + K_m\eta_m\ ;\ p : \tau \Rightarrow \tilde{p}}$$

$$\text{CLASS}\quad \frac{\Gamma, \Sigma, E\,[C{:}\forall\alpha{:}C.\sigma] \vdash p : \tau \Rightarrow \tilde{p}}{\Gamma, \Sigma, E \vdash \textbf{class}\ C\ \textbf{is}\ \forall\alpha{:}C.\sigma\ ;\ p : \tau \Rightarrow \textbf{let}\ C = \lambda x.x\ \textbf{in}\ \tilde{p}}$$

$$\begin{array}{c}
\Gamma\left[\overline{\alpha_n{:}S_n}, \overline{\beta_m{:}T_m}\right], \Sigma, E \vdash e : \left[t(\overline{\alpha_n})/\alpha\right]\tau \Rightarrow \tilde{e} \qquad \{\overline{\beta_m}\} = FV(\tau) - \{\alpha\} \\[1ex]
S_i = \{C_1^i, \ldots, C_{k_i}^i\} \qquad i = 1\ldots n \qquad T_j = \{D_1^j, \ldots, D_{l_j}^j\} \qquad j = 1\ldots m
\end{array}$$

$$\text{INST}\quad \frac{E(C) = \forall\alpha{:}C.\forall\overline{\beta_m{:}T_m}.\tau \qquad \Gamma, \Sigma \cup \{t{:}\left(\overline{S_n}\right)C\}, E \vdash p : \tau' \Rightarrow \tilde{p}}{\begin{array}{l}\Gamma, \Sigma, E \vdash \textbf{inst}\ t : \left(\overline{S_n}\right)C\ \textbf{is}\ e\ ;\ p : \tau' \Rightarrow \\ \qquad \textbf{let}\ C_t = \lambda\alpha_1^{C_1^1}\ldots\alpha_1^{C_{k_1}^1}\ldots\alpha_n^{C_1^n}\ldots\alpha_n^{C_{k_n}^n}\beta_1^{D_1^1}\ldots\beta_1^{D_{l_1}^1}\ldots\beta_m^{D_1^m}\ldots\beta_m^{D_{l_m}^m}.\tilde{e}\ \textbf{in}\ \tilde{p}\end{array}}$$

Figure 10: Translation rules for declarations

$$\text{VAR}\quad \frac{E(x) = \forall\overline{\alpha_n{:}S_n}.\tau \qquad \Gamma, \Sigma \vdash \tau_i : S_i \qquad S_i = \{C_1^i, \ldots, C_{k_i}^i\} \qquad i = 1\ldots n}{\Gamma, \Sigma, E \vdash x : \left[\overline{\tau_n/\alpha_n}\right]\tau \Rightarrow x\ d_\Sigma(\tau_1, C_1^1)\ldots d_\Sigma(\tau_1, C_{k_1}^1)\ldots d_\Sigma(\tau_n, C_1^n)\ldots d_\Sigma(\tau_n, C_{k_n}^n)}$$

$$\text{ABS}\quad \frac{\Gamma, \Sigma, E\,[x{:}\tau_2] \vdash e : \tau_1 \Rightarrow \tilde{e}}{\Gamma, \Sigma, E \vdash \lambda x.e : \tau_2 \to \tau_1 \Rightarrow \lambda x.\tilde{e}}$$

$$\text{APP}\quad \frac{\Gamma, \Sigma, E \vdash e_1 : \tau_2 \to \tau_1 \Rightarrow \tilde{e}_1 \qquad \Gamma, \Sigma, E \vdash e_2 : \tau_2 \Rightarrow \tilde{e}_2}{\Gamma, \Sigma, E \vdash (e_1\ e_2) : \tau_1 \Rightarrow (\tilde{e}_1\ \tilde{e}_2)}$$

$$\{\overline{\alpha_n}\} = FV(\tau_1) - FV(E) \qquad S_i = \{C_1^i, \ldots, C_{k_i}^i\} \qquad i = 1\ldots n$$

$$\text{LET}\quad \frac{\Gamma\left[\overline{\alpha_n{:}S_n}\right], \Sigma, E \vdash e_1 : \tau_1 \Rightarrow \tilde{e}_1 \qquad \Gamma, \Sigma, E\left[x{:}\forall\overline{\alpha_n{:}S_n}.\tau_1\right] \vdash e_2 : \tau_2 \Rightarrow \tilde{e}_2}{\Gamma, \Sigma, E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \Rightarrow \textbf{let}\ x = \lambda\alpha_1^{C_1^1}\ldots\alpha_1^{C_{k_1}^1}\ldots\alpha_n^{C_1^n}\ldots\alpha_n^{C_{k_n}^n}.\tilde{e}_1\ \textbf{in}\ e_2}$$

Figure 11: Translation rules for expressions

Type preservation and ambiguity

The type preservation theorem states that the Exell type system is sound w.r.t. its translational

$$\text{CONS}\quad \frac{\Gamma, \Sigma \vdash E(K) \geq \left(\exists \overline{\beta_m{:}T_m}.\tau\right) \to t(\overline{\tau_n}) \qquad \Gamma, \Sigma \vdash \tau'_i : T_i \quad T_i = \{C^i_1, \ldots, C^i_{k_i}\} \quad i = 1\ldots m}{\begin{array}{l}\Gamma, \Sigma, E \vdash K : \left[\overline{\tau'_m/\beta_m}\right]\tau \to t(\overline{\tau_n}) \Rightarrow \\ \quad \lambda x.\left(K, \left(\left(d_\Sigma(\tau'_1, C^1_1), \ldots, d_\Sigma(\tau'_1, C^1_{k_1})\right), \ldots, \left(d_\Sigma(\tau'_m, C^m_1), \ldots, d_\Sigma(\tau'_m, C^m_{k_m})\right), x\right)\right)\end{array}}$$

$$\text{TEST}\quad \frac{\Gamma, \Sigma \vdash E(K) \geq \eta \to t(\overline{\tau_n})}{\Gamma, \Sigma, E \vdash \textbf{is } K : t(\overline{\tau_n}) \to bool \Rightarrow \lambda x.\left(\pi^2_1 x = K\right)}$$

$$\text{PAT}\quad \frac{\begin{array}{c}\Gamma, \Sigma \vdash E(K) \geq \left(\exists \overline{\beta_m{:}T_m}.\tau\right) \to t(\overline{\tau_n}) \qquad \Gamma, \Sigma, E \vdash e_1 : t(\overline{\tau_n}) \Rightarrow \tilde{e}_1 \\ T_i = \{C^i_1, \ldots, C^i_{k_i}\} \qquad i = 1\ldots m \qquad \{\overline{\kappa_m}\} \cap (FS(\tau') \cup FS(E)) = \varnothing \\ \Gamma, \Sigma \cup \bigcup_{i=1}^{m} \{\kappa_i{:}()C^i_1, \ldots, \kappa_i{:}()C^i_{k_i}\}, E\left[x{:}\left[\overline{\kappa_m/\beta_m}\right]\tau\right] \vdash e_2 : \tau' \Rightarrow \tilde{e}_2\end{array}}{\begin{array}{l}\Gamma, \Sigma, E \vdash \textbf{let } K\ x = e_1 \textbf{ in } e_2 : \tau' \Rightarrow \\ \quad \textbf{if } \pi_1\tilde{e}_1 \neq K \textbf{ then fail else} \\ \quad\quad \textbf{let } \tilde{x} = \pi_2\tilde{e}_1; C^i_{j\kappa_i} = \pi_j\pi_i\tilde{x}\big|_{j = 1\ldots k_i,\, i = 1\ldots m}; x = \pi_{m+1}\tilde{x} \textbf{ in } \tilde{e}_2\end{array}}$$

Figure 12: Translation rules for expressions involving existential types

semantics: every translation of a well-typed Exell program results in a well-typed MPS program.

Theorem 7.1 If $\Gamma, \Sigma, E \vdash p : \tau \Rightarrow \tilde{p}$, where $Dom(\Gamma) = \{\overline{\alpha_n}\}$ and $\Gamma\alpha_i = \{C^i_1, \ldots, C^i_{k_i}\}$ for $i = 1\ldots n$, then

$$FV(\tilde{p}) - FV(p) = \left\{\alpha_1^{C^1_1}, \ldots, \alpha_1^{C^1_{k_1}}, \ldots, \alpha_n^{C^n_1}, \ldots, \alpha_n^{C^n_{k_n}}\right\} \text{ and}$$

$$MPS(E) \vdash_{\text{MPS}} \lambda\alpha_1^{C^1_1}\ldots\alpha_1^{C^1_{k_1}}\ldots\alpha_n^{C^n_1}\ldots\alpha_n^{C^n_{k_n}}.\tilde{p} : MPS(\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\tau).$$

The practical implication of this result is that we do not need to carry around any type information after the translation. Therefore it is type-safe to evaluate the translated programs in an untyped language. A similar theorem for Mini-Haskell was given by Nipkow and Snelting (1991).

As in Haskell, there are terms in Exell that do not have a unique semantics. For example, the term `show (read "True")` either evaluates to the string `"True"` or fails, depending on the choice of the type of the subexpression `read "True"`. This problem is called *(semantic) incoherence* (Blott, 1991, Jones, 1993a, Thatte, 1994) and occurs when unbound method dictionaries are generated during the translation. Our goal is to establish a simple syntactic criterium for typings leading to a coherent semantics for a term. We observe that unbound method dictionaries in the translation correspond to sorted type variables that do not occur in the expression type or in the environment. Since no dictionary variables are ever generated for type variables belonging to the empty sort $\emptyset$, only type variables in Γ that are not free in τ or E can lead to incoherence. Following Blott (1991), we define *(syntactic) ambiguity* for typings, type schemes, and environments:

Definition 7.2 A typing $\Gamma, \Sigma, E \vdash p : \tau$ is *unambiguous* iff $Dom(\Gamma) \subseteq FV(\tau) \cup FV(E)$.

Definition 7.3 A type scheme $\sigma = \forall \overline{\alpha_n : S_n} . \sigma'$, where $\sigma' = \tau$ or $\sigma' = \eta \to \tau$, is *unambiguous* iff $\{\alpha_1, \ldots, \alpha_n\} \subseteq FV(\sigma')$.

Definition 7.4 An environment E is *unambiguous* iff $E(x)$ is unambiguous for all $x \in Dom(E)$.

Blott's coherence result (1991) states that a syntactically unambiguous typing leads to a coherent semantics. In the presence of existential quantification, application of a value constructor causes incoherence when there is a choice of possible representation types that replace the corresponding existentially quantified type variables. The representation types would then contain type variables that are recorded in the sort context, but neither occur in the result type nor in the environment. Thus the incoherence would again be apparent syntactically. We therefore expect Blott's result to extend to Exell, as expressed in the following conjecture:

Conjecture 7.5 If E is unambiguous, Σ is coregular, and $\Gamma, \Sigma, E \vdash p : \tau$ is unambiguous, then for any two translations $\Gamma, \Sigma, E \vdash p : \tau \Rightarrow \tilde{p}_1$ and $\Gamma, \Sigma, E \vdash p : \tau \Rightarrow \tilde{p}_2$, $[\![\tilde{p}_1]\!] = [\![\tilde{p}_2]\!]$ with respect to the semantics of MPS defined in Appendix B.

As pointed out by Jones (1993a), a significant limitation of his and Blott's work is that their coherence results apply only to languages with a nonstrict or call-by-name semantics. Jones suggests extending these results to languages with a strict or call-by-value semantics by reworking them under an axiomatization of equality suitable for a strict semantics.

8 Conclusion and Related Work

We have demonstrated how type classes and existential types can be combined in a functional language with a static, polymorphic type system and algebraic data type declarations. Furthermore, we have given a formal type system and a type inference algorithm for our language. Finally, we have presented a semantics via a type-preserving translation of our extended language into an implicitly typed second-order λ-calculus.

Perry (1990) was the first to address Hindley-Milner type inference for existential types in the Hope+C system. We follow Perry in allowing existential quantification in the component types of algebraic data types. There is a significant difference between Perry's and our treatment of Skolem types. Following Mitchell and Plotkin (1988), we require in the PAT rule that Skolem types do not escape the pattern-matching **let**-expression in which they are introduced. Perry, on the other hand, places no restriction on his equivalent DES rule, but disallows any Skolem types in the result type of a function in his ABS rule. His approach leads to a more restrictive system in practice: it is not possible in Hope+C to define local functions that operate on the internal representation of an abstract data type (see also the tree search example in Section 3). Cardelli and Leroy (1990) also describe a formal calculus that allows Skolem types to escape the expression in which they are introduced when an existential type is eliminated, provided that they do not escape the scope of the (existentially typed) identifier with which they are associated. While their approach requires restrictions in the type rules for constructs that introduce bound identifiers, it does admit Skolem types in the result types of functions.

Läufer and Odersky (1991) employ Haskell type classes as interfaces of abstract data types by using them to constrain existentially quantified type variables. By applying their idea to existentially quantified type variables in the component types of algebraic data types, we avoid having to include the operations on abstract data types explicitly in the component types. Läufer and Odersky (1994) present an extension of ML with existentially quantified types; in their language, as in Perry's (1990), operations on abstract data types must be included explicitly in the component types of algebraic data types. Following Cardelli and Leroy (1990), Läufer and Odersky (1994) then describe a further extension of ML that allows Skolem types to escape the expression in which they are introduced. If we drop the class and instance constructs from our language and avoid overloading in the initial environment, we obtain Läufer and Odersky's language (1994) as a special case. However, the PAT rule in our type system is monomorphic to facilitate the proof of soundness for our translational semantics, whereas the corresponding rule in their system is polymorphic. While this condition is minor in practice, strictly speaking, our system includes only a subsystem of theirs with a monomorphic PAT rule. We conjecture that an extension of our system with a polymorphic PAT rule is still semantically sound and expect to prove this in the future.

Chen, Hudak, and Odersky (1992) present an extension of Haskell with parameterized type classes. Jones (1993b) describes a related extension of Haskell with type constructor classes. We plan to extend Exell analogously to obtain abstract data types with parameterized interfaces.

Mitchell, Meldal, and Madhav (1991) describe the possibility of treating Standard ML modules as first-class values but do not address the issue of type inference. By hiding the type components of a structure, the type of the structure itself is implicitly coerced from a strong (dependent) sum type to a weak (existentially quantified) sum type. Harper and Lillibridge (1994) and independently Leroy (1994) further explore this idea in a new treatment of the Standard ML module system. In their approach, structures have weak sum types and act as first-class values. Thus stratification of types into different universes of "small" types and "large" strong sum types is no longer necessary. However, the module-based approaches are semantically complex and lack support for type inference.

Statically-typed object-oriented languages such as C++ (Stroustrup, 1986) have been successful at expressing heterogeneous aggregates via extensible subtype hierarchies. However, most of these languages lack type inference and parametric polymorphism. Furthermore, they do not support separate interface and implementation hierarchies. Baumgartner and Russo (1994) propose a solution that separates the two hierarchies in C++. Since our work is based on type classes, it provides a similar separation, but allows only a limited form of reuse at the implementation level via conditional instance declarations, as illustrated in Section 3.

Pierce and Turner (1993) describe an object-oriented language based on existential quantification instead of recursive record types. Their language builds on an extension of F_ω to include subtyping and seems sufficiently powerful to model most features in typical object-oriented languages, including reference to the methods of the superclass and private instance variables, which are not supported in Exell. However, their language is explicitly typed and highly complex; consequently, algorithmic type inference is not considered.

Rémy (1994) investigates the idea of object-oriented programming in an extension of ML by modifying the type system to include recursive types, existential and universal types, and mutable extensible records. His work supports single and multiple inheritance and access to methods of the superclasses. However, unlike our work, there is no support for an interface hierarchy; thus lists containing, for example, both points and color points cannot be built.

Acknowledgments

I would like to thank Martin Odersky for sharing his insights with me through numerous discussions. This work has greatly benefited from conversations with Stefan Kaes, Tobias Nipkow, and Phil Wadler. Lennart Augustsson's extension of Haskell B. with existential quantification made it possible to develop and test the examples contained in this paper. I would like to thank the anonymous referees of FPCA '93 and JFP for their valuable feedback on earlier versions of this work and for suggesting the interesting example involving the composition of a list of functions.

References

Augustsson, L. 1993a. Haskell B. user manual. Distributed with the HBC compiler.

Augustsson, L. 1993b. Implementing Haskell overloading. In *Proc. Functional Programming Languages and Computer Architecture*. ACM.

Baumgartner, G. and Russo, V. 1994. Signatures: A C++ extension for type abstraction and subtype polymorphism. *Software: Practice & Experience*. To appear.

Blott, S. 1991. *An Approach to Overloading with Polymorphism*. PhD thesis, Department of Computing Science, University of Glasgow.

Cardelli, L. and Leroy, X. 1990. Abstract types and the dot notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Sea of Galilee, Israel.

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

Chen, K., Hudak, P., and Odersky, M. 1992. Parametric type classes. In *Proc. ACM Conf. Lisp and Functional Programming*.

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

Hall, C., Hammond, K., Peyton Jones, S., and Wadler, P. 1992. Type classes in Haskell. Technical report, Department of Computer Science, University of Glasgow.

Harper, R. and Lillibridge, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 123–137.

Hudak, P., Peyton-Jones, S., Wadler, P., et al. 1992. Report on the programming language Haskell: A non-strict, purely functional language, Version 1.2. *ACM SIGPLAN Notices*, 27(5).

Jones, M. 1993a. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Department of Computer Science, Yale University.

Jones, M. 1993b. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. Functional Programming Languages and Computer Architecture*. ACM.

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

Läufer, K. 1992. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University. Available as Technical Report 622, December 1992, from New York University, Department of Computer Science.

Läufer, K. and Odersky, M. 1991. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, pages 148–162. Workshops in Computing series, Springer-Verlag.

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

Leroy, X. 1994. Manifest types, modules, and separate compilation. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 109–123.

Liskov, B. and Guttag, J. 1986. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Mass.

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

Mitchell, J., Meldal, S., and Madhav, N. 1991. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*.

Mitchell, J. and Plotkin, G. 1988. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502.

Nipkow, T. and Prehofer, C. 1993. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*.

Nipkow, T. and Snelting, G. 1991. Type classes and overloading resolution via order-sorted unification. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–14. Lecture Notes in Computer Science 523, Springer-Verlag.

Perry, N. 1990. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College.

Pierce, B. and Turner, D. 1993. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*.

Rémy, D. 1994. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Proc. Intl. Symp. Theoretical Aspects of Computer Software (TACS)*. Lecture Notes in Computer Science, Springer-Verlag.

Stroustrup, B. 1986. *The C++ Programming Language*. Addison-Wesley.

Thatte, S. 1994. Semantics of type classes revisited. In *Proc. ACM Conf. Lisp and Functional Programming*.

United States Department of Defense 1983. *Reference Manual for the ADA Programming Language*. Springer-Verlag.

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

Wirth, N. 1985. *Programming in Modula-2*. Springer-Verlag, New York, 3rd edition.

A Unification of Types with Sort Constraints

We summarize Nipkow and Prehofer's results (1993) on unification of types with sort constraints. An important difference in our work is that signatures are no longer fixed since the body of a pattern-matching **let**-expression has to be typed under an extended signature. Therefore all functions depending on Σ have to be parameterized by Σ.

We define the following generality ordering on context-substitution pairs:

$(\Gamma, \theta) \geq (\Gamma', \theta')$ iff $\Gamma', \Sigma \vdash \theta : \Gamma$ and there is a substitution δ such that $\delta\theta = \theta'$.

We can express sorted unification as unsorted unification plus constraint solving. Given a coregular signature Σ, this has the following form:

$$\begin{aligned} unify_\Sigma(\Gamma, \tau_1 = \tau_2) \; = \; & \text{let } \theta = mgu(\tau_1 = \tau_2) \\ & \quad \Gamma_c = \bigcup_{\alpha \in Dom(\theta)} constrain_\Sigma(\theta\alpha, \Gamma\alpha) \\ & \text{in } \; (\Gamma_c \cup (\Gamma \setminus Dom(\theta)), \theta) \end{aligned}$$

where

$$\begin{aligned} constrain_\Sigma(\alpha, S) &= [\alpha{:}S] \\ constrain_\Sigma(t(\overline{\tau_n}), S) &= \bigcup_{C \in S} constr_\Sigma(\overline{\tau_n}, Dom_\Sigma(t, C)) \\ constr_\Sigma(\overline{\tau_n}, \overline{S_n}) &= \bigcup_{i = 1\ldots n} constrain_\Sigma(\tau_i, S_i) \end{aligned}$$

The following two theorems give a precise characterization of those signatures that allow principal types:

Theorem A.1 If Σ is coregular, then $unify_\Sigma$ computes a *most general unifier*.

Theorem A.2 $unify_\Sigma$ is *unitary* iff Σ is coregular.

B The *MPS* Polymorphic λ-Calculus

MacQueen, Plotkin, and Sethi (1986) give a semantics for an implicitly typed, second-order polymorphic λ-calculus based on order ideals over domains. The expression syntax, the type syntax, and the type inference system of the language *MPS* are shown in Figure 13. The type *con* consists of all constructor tags K. Whereas the Exell inference rules follow the expression syntax, the *MPS* inference rules follow the type syntax; this facilitates the proof of Theorem 7.1. We allow the form **let** $x = e_1$ **in** e_2 as syntactic sugar for the substitution $[e_1/x]\,e_2$.

The denotational semantics of *MPS* expressions is given as follows:

$$\begin{aligned} [\![x]\!]\rho &= \rho(x) \\ [\![\lambda x.e]\!]\rho &= \lambda a \in V.\, [\![e]\!]\rho\,[x{:}a] \\ [\![e\; e']\!]\rho &= \text{if } e \in V \to V \text{ then } ([\![e]\!]\rho)\,([\![e']\!]\rho) \text{ else wrong} \end{aligned}$$

where ρ is an environment mapping identifiers to values. The denotational semantics of *MPS* types is given by the ideal expressions corresponding to the syntactic type expressions; for example, the semantics of universal and existential quantification is:

$$[\![\forall\alpha.\sigma]\!]\psi = \textstyle\bigsqcap_{I \in \Im(D),\, \text{wrong} \notin I} [\![\sigma]\!]\psi\,[\alpha{:}I]$$

$$[\![\exists\alpha.\sigma]\!]\psi = \textstyle\bigsqcup_{I \in \Im(D),\, \text{wrong} \notin I} [\![\sigma]\!]\psi\,[\alpha{:}I]$$

$$\text{Expressions} \quad e \;=\; x \mid (e_1 e_2) \mid \lambda x.e$$

$$\text{Types} \quad \sigma \;=\; con \mid bool \mid int \mid \alpha \mid \sigma \to \sigma' \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \forall\alpha.\sigma \mid \exists\alpha.\sigma \mid \mu\alpha.\sigma$$

$$\text{V} \quad \frac{E(x) = \sigma}{E \vdash x : \sigma}$$

$$\to\text{I} \quad \frac{E\,[x{:}\sigma_2] \vdash e : \sigma_1}{E \vdash \lambda x.e : \sigma_2 \to \sigma_1} \qquad \to\text{E} \quad \frac{E \vdash e_1 : \sigma_2 \to \sigma_1 \qquad E \vdash e_2 : \sigma_2}{E \vdash (e_1 e_2) : \sigma_1}$$

$$+\text{I} \quad \frac{E \vdash e : \sigma_{1/2}}{E \vdash e : \sigma_1 + \sigma_2} \qquad +\text{E} \quad \frac{E \vdash e_1 : \sigma_1 + \sigma_2 \quad E\,[x{:}\sigma_1] \vdash e_2 : \tau \quad E\,[x{:}\sigma_2] \vdash e_2 : \tau}{E \vdash [e_1/x]\,e_2 : \tau}$$

$$\forall\text{I} \quad \frac{E \vdash e : \sigma \qquad \alpha \notin FV(E)}{E \vdash e : \forall\alpha.\sigma} \qquad \forall\text{E} \quad \frac{E \vdash e : \forall\alpha.\sigma}{E \vdash e : [\tau/\alpha]\,\sigma}$$

$$\exists\text{I} \quad \frac{E \vdash e : [\tau/\alpha]\,\sigma}{E \vdash e : \exists\alpha.\sigma} \qquad \exists\text{E} \quad \frac{E \vdash e_1 : \exists\alpha.\sigma \quad E\,[x{:}\sigma] \vdash e_2 : \tau \quad \alpha \notin FV(E) \cup FV(\tau)}{E \vdash [e_1/x]\,e_2 : \tau}$$

$$\mu\text{I} \quad \frac{E \vdash e : [\mu\alpha.\sigma/\alpha]\,\sigma}{E \vdash e : \mu\alpha.\sigma} \qquad \mu\text{E} \quad \frac{E \vdash e : \mu\alpha.\sigma}{E \vdash e : [\mu\alpha.\sigma/\alpha]\,\sigma}$$

Figure 13: Syntax and type inference system of *MPS*

where ψ is a type environment mapping type variables to ideals I over the domain D.

We define $E \models e : \sigma$ to mean that for all ρ and ψ such that $[\![\, x \,]\!]\rho \in [\![\, E(x) \,]\!]\psi$ for all $x \in Dom(E)$ we have $[\![\, e \,]\!]\rho \in [\![\, \sigma \,]\!]\psi$.

MPS is semantically sound with respect to its type inference system; this is expressed by the following theorem:

Theorem B.1 If $E \vdash_{\text{MPS}} e : \sigma$ then $E \models e : \sigma$.

This theorem says that if we can statically infer a type for an expression, then the dynamic type of an expression is the same as the static type. Consequently, no well-typed *MPS* expression evaluates to `wrong`.