Interface Types for Haskell

Peter Thiemann and Stefan Wehr

Institut für Informatik, Universität Freiburg, Germany
{thiemann,wehr}@informatik.uni-freiburg.de

Abstract. Interface types are a useful concept in object-oriented programming languages like Java or C#. A clean programming style advocates relying on interfaces without revealing their implementation.
Haskell's type classes provide a closely related facility for stating an interface separately from its implementation. However, there are situations in which no simple mechanism exists to hide the identity of the implementation type of a type class. This work provides such a mechanism through the integration of lightweight interface types into Haskell.
The extension is non-intrusive as no additional syntax is needed and no existing programs are affected. The implementation extends the treatment of higher-rank polymorphism in production Haskell compilers.

1 Introduction

Interfaces in object-oriented programming languages and type classes in Haskell are closely related: both define the types of certain operations without revealing their implementations. In Java, the name of an interface also acts as an *interface type*, whereas the name of a type class can only be used to constrain types. Interface types are a proven tool for ensuring data abstraction and information hiding. In many cases, Haskell type classes can serve the same purpose, but there are situations for which the solutions available in Haskell have severe drawbacks.

Interface types provide a simple and elegant solution in these situations. A modest extension to Haskell provides the simplicity and elegance of interface types: simply allow programmers to use the name of a type class as a first-class type. The compiler then translates such interface types into existentially quantified data types [11] (available in several Haskell compilers such as GHC [3] or Hugs [5]) and generates all the boilerplate code necessary for dealing with these existential types. To keep type inference manageable, we follow the same strategy as type inference algorithms for rank-n types [14] and require type annotations if interface types should be introduced.

Contributions and Outline. A *case study* (Section 2.1) compares several approaches to information hiding in Haskell. It demonstrates that interface types provide the simplest solution. Two further example applications (Section 2.2 and Section 2.3) underline the advantages of interface types.

In Section 3, we *formalize interface types* as an extension of a type system and inference algorithm for rank-n types introduced by Peyton Jones and others [14]. The resulting inference algorithm (explained in Section 4 in terms of a bidirectional type system) is close to the one used in GHC.

A *prototype implementation* of the type inference algorithm is available.[1] We have developed it as an extension of Peyton Jones's implementation of rank-n type inference [14].

Section 5 sketches the translation to System F, the second component needed for implementing interface types in a compiler. Sections 6 and 7 discuss related work and conclude.

2 Motivation

To motivate the need for interface types, we present the results of a case study that compares different approaches to information hiding in the design of a library for database access (Section 2.1). In two additional examples, we show how interface types help in designing a library for sets and graphical user interfaces (Section 2.2 and Section 2.3, respectively).

2.1 Interface Types for Database Access

Consider a programmer designing a Haskell library for accessing databases. Ideally, the public interface of the library makes no commitment to a particular database system and users of the library should not be able to create dependencies on a particular database system (exception to both: opening new connections). Thus, all datatypes describing connections to the database, query results, cursors, and so on should be abstract, and the only way to manipulate them should be through operations provided in the library.

Record Types as Interface Types. As a concrete example, consider the HDBC package [4]. Up to version 1.0.1.2, HDBC provided database operations through a record type similar to the following:[2]

```
module Database.HDBC (Connection(..)) where
data Connection = Connection { dbQuery :: String -> IO [[String]] }
```

HDBC comes with a number of *drivers* that provide support for a specific database system through an operation to create a connection:

```
module Database.HDBC.PostgreSQL (connectPSQL) where
connectPSQL :: String -> IO Connection

module Database.HDBC.Sqlite3 (connectSqlite3) where
connectSqlite3    :: FilePath -> IO Connection
```

[1] `http://www.informatik.uni-freiburg.de/~thiemann/haskell/IFACE/impl.tgz`

[2] We only show those parts of the code relevant to our problem. Modules whose names start with `MyHDBC` are not part of HDBC.

Once a connection is established, the `Connection` datatype ensures that application code works independently of the specific database system. Thus, the design just outlined fulfills the requirements at the beginning of this paragraph.

There is, however, one major disadvantage: the set of database operations is fixed and cannot be extended easily. Suppose that we want to add support for PostgreSQL's [16] asynchronous events.[3] We cannot extend the existing `Connection` datatype because not all database systems support asynchronous events. Thus, we need to create a new datatype:

```
module MyHDBC (ConnectionAE(..)) where
data ConnectionAE = ConnectionAE { dbQuery' :: String -> IO [[String]],
                                   listen   :: String -> IO (),
                                   notify   :: String -> IO () }
```

But now functions operating on `Connection` do not work with `ConnectionAE`, although the latter type supports, in principle, all operations of the former.

Type Classes as Interface Predicates. For this reason, HDBC version 1.1.0.0 replaces the datatype `Connection` with a type class `IConnection`:

```
module Database.HDBC (IConnection(..)) where
class IConnection c where
  dbQuery :: c -> String -> IO [[String]]
```

Support for asynchronous events is now modeled through a subclass of `IConnection`:

```
module MyHDBC (IConnectionAE(..)) where
class IConnection c => IConnectionAE c where
  listen :: c -> String -> IO ()
  notify :: c -> String -> IO ()
```

This way, functions with signatures of the form `IConnection c => .. -> c -> ..` also work when passing an instance of `IConnectionAE` as the `c` argument.

The classes `Connection` and `IConnectionAE` are *not* types, but serve as predicates on type variables. Thus, the connect function provided by a database driver has to return the concrete connection type. For example:

```
module Database.HDBC.Sqlite3 (ConnectionSqlite3(), connectSqlite3) where
data ConnectionSqlite3 = ConnectionSqlite3 {
       sqlite3Query :: String -> IO [[String]] }
instance IConnection ConnectionSqlite3 where
  dbQuery = sqlite3Query
connectSqlite3 :: FilePath -> IO ConnectionSqlite3
```

A concrete return type violates our requirement that application code should not be able to create a dependency on a particular database system: The driver module `Database.HDBC.Sqlite3` exports the datatype `ConnectionSqlite3`. Application code may use this type in function signatures, data type declarations etc.

Is there a Haskell solution to this problem? Simply hiding the `ConnectionSqlite3` type inside the `Database.HDBC.Sqlite3` module is not enough, because a type name is useful for type specifications. There are at least two solutions to this problem, both of which involve advanced typing constructs.

[3] PostgreSQL provides a `listen` and a `notify` operation: `listen` allows processes to register for some event identified by a string, `notify` signals the occurrence of an event.

Existential Types as Interface Types. The first solution uses algebraic datatypes with existential types [11].[4]

```
module Database.HDBC (IConnection(..), ExIConnection(..)) where
-- class IConnection as before
data ExIConnection =
     forall c . IConnection c => ExIConnection c
instance IConnection ExIConnection where
  dbQuery (ExIConnection c) = dbQuery c

module MyHDBC (IConnectionAE(..), ExIConnectionAE(..)) where
-- class IConnectionAE as before
data ExIConnectionAE = forall c . IConnectionAE c => ExIConnectionAE c
instance IConnection ExIConnectionAE where
  dbQuery (ExIConnectionAE c) = dbQuery c
instance IConnectionAE ExIConnectionAE where
  listen (ExIConnectionAE c) = listen c
  notify (ExIConnectionAE c) = notify c
```

With this solution, the module `Database.HDBC.Sqlite3` no longer exports the type `ConnectionSqlite3` and the return type of `connectSqlite3` becomes `ExIConnection`.

However, this solution has some drawbacks:

- A value of type `ExIConnectionAE` cannot be used where a value of type `ExIConnection` is expected. Instead, we have to unpack and re-pack the existential type.
- Writing and maintaining the boilerplate for the datatype declarations `ExIConnection` and `ExIConnectionAE`, as well as the corresponding instance declarations is tedious, especially when the class hierarchy becomes larger.

Rank-2 Types as Interface Types. The second solution is to provide a function that passes the newly created connection to a continuation. Thanks to higher-rank polymorphism [14], the continuation can be given a sensible type. With this approach, the driver for PostgreSQL would look like this:

```
module MyHDBC.PostgreSQL (runWithPSQL) where
data ConnectionPSQL = ConnectionPSQL { psqlQuery  :: String -> IO [[String]],
                                       psqlListen :: String -> IO (),
                                       psqlNotify :: String -> IO () }
instance IConnection ConnectionPSQL where
  dbQuery = psqlQuery
instance IConnectionAE ConnectionPSQL where
  listen = psqlListen
  notify = psqlNotify
connectPSQL :: String -> IO ConnectionPSQL
runWithPSQL :: String -> (forall c. IConnectionAE c => c -> IO t) -> IO t
runWithPSQL s f = do c <- connectPSQL s
                     f c
```

Thanks to the generic instantiation relation for types, this function allows for some unexpected flexibility. Clearly, a function of type

```
psqlWorker :: IConnectionAE c => c -> IO Result
```

can serve as a (second) parameter to `runWithPSQL`. But also

[4] GHC uses the keyword `forall` for existential quantifiers.

```
dbWorker :: IConnection c => c -> IO Result
```

is a type correct second argument to `runWithPSQL`. The flexibility of this approach is appealing, but writing the user code using continuations can be demanding and may obfuscate the code.

Which of the two solutions does HDBC choose? The answer is: *none*. It seems that the benefit of hiding the concrete connection type does not outweigh the complexity of the two solutions.

Type Classes as Interface Types. We propose an alternative solution that is lightweight and easy to use. We consider the name C of a type class as an *interface type* that denotes some unknown instance of the class. Thus, the interface type C stands for the bounded existential type $\exists c \,.\, C\, c \Rightarrow c$.

For example, the interface type `IConnection` represents some unknown instance of the type class `IConnection`. Here is some code for an Sqlite3 driver module following this approach:

```
module Database.HDBC.Sqlite3 (connectSqlite3) where
data ConnectionSqlite3 = ConnectionSqlite3 {
       sqlite3Query :: String -> IO [[String]] }
instance IConnection ConnectionSqlite3 where
  dbQuery = sqlite3Query
connectSqlite3 :: FilePath -> IO IConnection
connectSqlite3 = internConnectSqlite3
internConnectSqlite3 :: FilePath -> IO ConnectionSqlite3
```

Transferring the subclass hierarchy on type classes to a "more polymorphic than" relation on interface types allows values of type `IConnectionAE` to be passed to functions accepting a parameter of type `IConnection` without any explicit conversions. This approach yields the same flexibility with respect to parameter passing as with type classes and continuations using rank-2 types (but without requiring the use of continuations).

Thus, the solution combines the advantages of type classes approach (extensibility, flexibility with respect to parameter passing, ease to use) with the additional benefit that application code cannot directly refer to the implementation type of a connection. Moreover, there is no need to write boilerplate code as with existential types wrapped in data types and there is no need to use continuations as with the rank-2 types approach.

2.2 Interface Types for Sets

Consider a programmer designing a Haskell library for manipulating sets. The library should consist of a public interface for common set operations and various implementations of this interface. For simplicity, we consider only sets of integers with the operations `empty`, `insert`, `contains`, and `union`. We can easily encode the first three operations as methods of a type class `IntSet`:

```
class IntSet s where
  empty    :: s
  insert   :: s -> Int -> s
  contains :: s -> Int -> Bool
```

The signature of the `union` operation is not straightforward, because it should be possible to union two sets of *different* implementations. Thus, the second parameter of `union` should be an arbitrary `IntSet` instance, leading to the signature `union :: IntSet s' => s -> s' -> ?`. But what should the result type be?

When implementing sets using lists, we would like it to be `s'`:

```
instance IntSet [Int] where
  empty        = []
  insert l i   = i:l
  contains l i = i `elem` l
  union l s'   = foldl insert s' l
```

When implementing sets using characteristic functions, we would like it to be `s`:

```
instance IntSet (Int -> Bool) where
  empty        = \i -> False
  insert f i   = \j -> i == j || f j
  contains f i = f i
  union f s'   = \i -> contains f i || contains s' i
```

In general, the result type of `union` is some unknown instance of `IntSet`, which is exactly the kind of interface type introduced in Section 2.1. This choice avoids the declaration of an extra algebraic data type with existential quantification, writing boilerplate instance definitions, and packing and unpacking the existential type. Instead, we simply define the signature of `union` as

```
  -- inside type class IntSet
  union :: s -> IntSet -> IntSet
```

and get the rest for free. Especially, the two instance declarations for `[Int]` and `Int -> Bool` now become valid.

2.3 Interface Types for Graphical User Interfaces

Consider a programmer designing a Haskell library for writing graphical user interfaces. The library should provide several different kinds of widgets: a text input widget, a button widget, a table widget, and so on. It is reasonable to abstract over the common operations of widgets with a type class:

```
class Widget w where
  draw    :: w -> IO ()
  minSize :: w -> (Int,Int)
  name    :: w -> String
```

Some widgets provide additional features. A typical example is focus handling:

```
class Widget w => FocusWidget w where
  setFocus   :: w -> IO ()
  unsetFocus :: w -> IO ()
```

As an example, let us write the representation of a table widget. A table widget is essentially a list of rows, where each row consists of a list of widgets. Additionally, a table stores a second list of all focusable widgets. Clearly, the list of widgets in a row and the list of focusable widgets are heterogeneous. The

element types just happen to be instances of `Widget` or `FocusWidget`. Hence, we need some kind of existential type, again.

As in Section 2.1 and Section 2.2, algebraic datatypes with existential quantifiers are an option. Here is the code with a function that extracts all rows from a table containing at least one focusable widget.

```
data ExWidget      = forall w . Widget w => ExWidget w
data ExFocusWidget = forall w . FocusWidget w => ExFocusWidget w
instance Widget ExWidget where
  draw (ExWidget w)    = draw w
  minSize (ExWidget w) = minSize w
  name (ExWidget w)    = name w
instance Widget ExFocusWidget where
  draw (ExFocusWidget w)    = draw w
  minSize (ExFocusWidget w) = minSize w
  name (ExFocusWidget w)    = name w
instance FocusWidget ExFocusWidget where
  setFocus (ExFocusWidget w)   = setFocus w
  unsetFocus (ExFocusWidget w) = unsetFocus w

instance Eq ExWidget where
  w1 == w2 = name w1 == name w2
instance Eq ExFocusWidget where
  w1 == w2 = name w1 == name w2

data Table = Table { rows :: [[ExWidget]], focusable :: [ExFocusWidget] }

focusableRows :: Table -> [[ExWidget]]
focusableRows tab =
  filter (\row -> any (\w -> w `elem` map asWidget (focusable tab)) row) (rows tab)
  where asWidget (ExFocusWidget w) = ExWidget w
```

With interface types all the boilerplate code vanishes:

```
instance Eq Widget where
  w1 == w2 = name w1 == name w2
instance Eq FocusWidget where
  w1 == w2 = name w1 == name w2

data Table = Table { rows      :: [[Widget]],
                     focusable :: [FocusWidget] }

focusableRows :: Table -> [[Widget]]
focusableRows tab =
  filter (\row -> any (\w -> w `elem` (focusable tab)) row) (rows tab)
```

In the subexpression `w `elem` (focusable tab)`, the compiler has to insert a coercion from `[FocusWidget]` to `[Widget]`. In general, such coercions are generated automatically if the corresponding datatype is an instance of `Functor`. In our concrete example, the datatype is `List`, which is already an instance of `Functor`.

2.4 Restrictions on Interface Types

Interface types are not a panacea. In the preceding section, we have seen that compound datatypes have to be instances of `Functor` if coercions should be generated automatically.

Syntax

$$
\begin{array}{lll}
\textit{expressions} & e, f & ::= x \mid \lambda x.e \mid \lambda(x :: s).e \mid f\, e \mid \texttt{let}\, x = e\, \texttt{in}\, f \mid (e :: s) \\
\textit{types} & s, t & ::= a \mid T\, \overline{t} \mid s \to t \mid \forall \overline{a}.P \Rightarrow t \\
\textit{monotypes} & m & ::= a \mid T\, \overline{m} \mid m \to m \\
\textit{predicates} & P, Q & ::= \texttt{true} \mid P, A
\end{array}
$$

Typing rules

$$\textit{(E-var)}\ P \mid \Gamma(x : s) \vdash x : s \qquad \textit{(E-lam)}\ \frac{P \mid \Gamma(x : m) \vdash e : t}{P \mid \Gamma \vdash \lambda x.e : m \to t}$$

$$\textit{(E-alam)}\ \frac{P \mid \Gamma(x : s) \vdash e : t}{P \mid \Gamma \vdash \lambda(x :: s).e : s \to t} \qquad \textit{(E-app)}\ \frac{P \mid \Gamma \vdash f : s \to t \quad P \mid \Gamma \vdash e : s}{P \mid \Gamma \vdash f\, e : t}$$

$$\textit{(E-ann)}\ \frac{P \mid \Gamma \vdash e : s}{P \mid \Gamma \vdash (e :: s) : s} \qquad \textit{(E-let)}\ \frac{P \mid \Gamma \vdash e : s \quad P \mid \Gamma(x : s) \vdash f : t}{P \mid \Gamma \vdash \texttt{let}\, x = e\, \texttt{in}\, f : t}$$

$$\textit{(E-gen)}\ \frac{P, Q \mid \Gamma \vdash e : s \quad \overline{a} \cap \mathrm{free}(\Gamma) = \emptyset \quad \mathrm{free}(P) \subseteq \mathrm{free}(\Gamma)}{P \mid \Gamma \vdash e : \forall \overline{a}.Q \Rightarrow s}$$

$$\textit{(E-spec)}\ \frac{P \mid \Gamma \vdash e : s \quad P \vdash^{dsk} s \preceq t}{P \mid \Gamma \vdash e : t}$$

Fig. 1: Syntax and static semantics of λ^Q.

Moreover, not every type class makes for a sensible interface type. In particular, the "dispatch type" of the class must appear exactly once negatively in the signatures of the member functions. Without this restriction, it is not possible to derive the instance definition on the interface type automatically. The examples in this section all obey this restriction, but any type class with a "binary method" such as `(==) :: Eq a => a -> a -> Bool` does not.

3 A Language with Interface Types

This section defines a calculus with interface types in two steps. The first step recalls qualified types [6] and extends it with higher-rank polymorphism along the lines of Peyton Jones et al [14]. The second step adds interface types to that calculus and defines and investigates their induced subtyping relation.

The presentation relies on standard notions of free and bound variables, substitution for type variables in a syntactic object $s[a \mapsto m]$, as well as the notation $\overline{a}$ as a shorthand for the sequence $a_1, \ldots, a_n$, for some unspecified $n \geq 0$.

3.1 Qualified Types with Higher-Rank Polymorphism

Fig. 1 contains the syntax and the static semantics of λ^Q, the language of qualified types as considered by Jones [6]. There are expressions e, types t, monotypes m (types that only contain type variables and type constructors), and predicates

Rho-types $r ::= m \mid s \to s$

Weak prenex conversion $\mathrm{pr}(s) = s'$

$$\textit{(N-poly)}\ \frac{\mathrm{pr}(r_1) = \forall \overline{b}.Q \Rightarrow r_2 \quad \overline{a} \cap \overline{b} = \emptyset}{\mathrm{pr}(\forall \overline{a}.P \Rightarrow r_1) = \forall \overline{a}\overline{b}.P, Q \Rightarrow r_2}$$

$$\textit{(N-fun)}\ \frac{\mathrm{pr}(s_2) = \forall \overline{a}.P \Rightarrow r_2 \quad \overline{a} \cap \mathrm{free}(s_1) = \emptyset}{\mathrm{pr}(s_1 \to s_2) = \forall \overline{a}.P \Rightarrow s_1 \to r_2} \qquad \textit{(N-mono)}\ \mathrm{pr}(m) = m$$

Deep skolemization $P \vdash^{dsk} s \preceq r$

$$\textit{(I-dsk)}\ \frac{\mathrm{pr}(s_2) = \forall \overline{a}.Q \Rightarrow r_2 \quad \overline{a} \cap \mathrm{free}(s_1) = \emptyset \quad P, Q \vdash^{dsk^*} s_1 \preceq r_2}{P \vdash^{dsk^*} s_1 \preceq s_2}$$

$$\textit{(I-spec)}\ \frac{P \Vdash Q[\overline{a} \mapsto \overline{m}] \quad P \vdash^{dsk^*} r_1[\overline{a} \mapsto \overline{m}] \preceq r_2}{P \vdash^{dsk^*} \forall \overline{a}.Q \Rightarrow r_1 \preceq r_2}$$

$$\textit{(I-fun)}\ \frac{P \vdash^{dsk} s_3 \preceq s_1 \quad P \vdash^{dsk^*} s_2 \preceq r_4}{P \vdash^{dsk^*} s_1 \to s_2 \preceq s_3 \to r_4} \qquad \textit{(I-tycon)}\ \frac{P \vdash^{dsk^*} \overline{s} \preceq \overline{r}}{P \vdash^{dsk^*} T\,\overline{s} \preceq T\,\overline{r}}$$

$$\textit{(I-mono)}\ P \vdash^{dsk^*} m \preceq m$$

Fig. 2: Instantiation rules for types.

P. Predicates are conjunctions of (at this point) unspecified atomic predicates A. The comma operator "," on predicates stands for conjunction and is assumed to be associative, commutative, and idempotent. Besides function types, the type language includes arbitrary covariant data type constructors T, which are introduced and eliminated with appropriate functions provided in the environment.[5]

The typing judgment and the rules defining its derivability are standard for a language with qualified types. The presentation extends Jones's by allowing arbitrary rank universally quantified types including type qualifications (cf. [10]). These higher-rank types are introduced through explicit type annotations following the initial lead of Odersky and Läufer [12]. Type inference for a language with these features is tricky and incomplete, but manageable [10, 14, 20]. In particular, this language is a core subset of the language implemented in bleeding edge Haskell compilers like GHC.

The rule *(E-spec)* relies on the generic instantiation relation $\preceq$ for types specified in Fig. 2. It generalizes the respective definition of Odersky and Läufer [12] (to qualified types) as well as Jones's ordering relation on constrained type schemes [6] (to higher-ranked types). Its particular formulation of $\preceq$ using deep skolemization is taken from Peyton Jones et al [14], extended with rule *(I-tycon)* that exploits the assumption that type constructors are covariant. A yet more general definition albeit without deep skolemization underlies the system of qualified types for MLF [10].

[5] In Haskell, the covariance requirement boils down to require T to be an instance `Functor`.

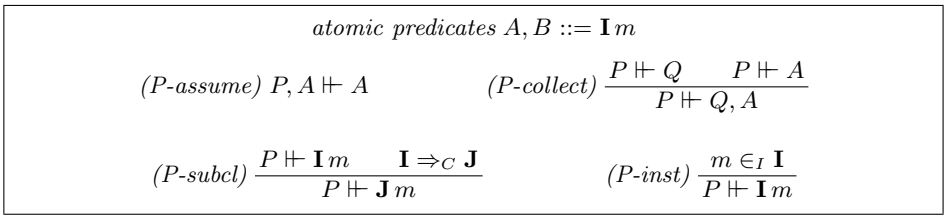

Fig. 3: Entailment for predicates.

So far, the definition is independent of any particular choice of predicates. Thus, it remains to choose a language of atomic predicates and define the entailment relation $P \Vdash Q$. In our case, the atomic predicates are *type class constraints*. Their entailment relation $\Vdash$ relies on specifications of type classes $\mathbf{I}$ and of type class instances, *i.e.*, the subclass relation between two classes $\mathbf{I} \Rightarrow_C \mathbf{J}$ (read: $\mathbf{I}$ is a subclass of $\mathbf{J}$) and the instance relation $m \in_I \mathbf{I}$ between a monotype m and a class $\mathbf{I}$. Their syntactic details are of no concern for this work and we consider the relations as built into the entailment relation $\Vdash$ on predicates which obeys the rules in Fig. 3. To avoid clutter, Haskell's facility of recursively defining $\in_I$ from single instance definitions for type constructors is omitted.

3.2 Interface Types

The language λ^Q serves as a base language for the language λ^I, which augments λ^Q with the definitions in Fig. 4. The only extension of λ^I over λ^Q is the notion of an *interface type* $\mathbf{I}$ and a slightly extended type instantiation relation. Each (single-parameter) type class $\mathbf{I}$ gives rise to an interface type of the same name. This interface type can be interpreted as the existential type $\exists a.\mathbf{I}\, a \Rightarrow a$ in that it stands for one particular element of the set of instances of $\mathbf{I}$. Furthermore, the interface type $\mathbf{I}$ is regarded an instance of $\mathbf{J}$ whenever $\mathbf{I} \Rightarrow_C \mathbf{J}$. This assumption is consistent with the observation that all instances of $\mathbf{I}$ are also instances of $\mathbf{J}$, hence the set of instances of $\mathbf{J}$ includes the instances of $\mathbf{I}$.

There is no explicit constructor for an element of interface type, but any suitable value can be coerced into one through a type annotation by way of the instantiation relation $\preceq$. This practice is analogous to the practice of casting to the interface type, which is the standard way of constructing values of interface type in, say, Java.[6] There is no explicit elimination form for a interface type $\mathbf{I}$, either. Rather, member functions of each class $\mathbf{J}$ where $\mathbf{I} \Rightarrow_C \mathbf{J}$ are directly applicable to a value of interface type $\mathbf{I}$ without explicitly unpacking the existential type.

Section 2 demonstrates that interface types are most useful if they enjoy a subtyping relation as they do in Java. Fig. 5 defines this subtyping relation in the obvious way. Each instance type of $\mathbf{J}$ is a subtype of $\mathbf{J}$ and the subclass

[6] An implementation can easily provide a cast-like operation to introduce interface types. However, type annotations are more flexible because they simplify the conversion of functions that involve interface types.

Extended syntax of types

$$m ::= \cdots \mid \mathbf{I}$$

Additional type instantiation rules

$$\textit{(I-iface)}\ \frac{P \Vdash \mathbf{I}\, m}{P \vdash^{dsk*} m \preceq \mathbf{I}}$$

Additional entailment rules

$$\textit{(P-subint)}\ \frac{\mathbf{I} \Rightarrow_C \mathbf{J}}{P \Vdash \mathbf{J}\, \mathbf{I}}$$

Fig. 4: Extensions for λ^I with respect to Figures 1 and 2.

$$\textit{(S-refl)}\ t \leq t \qquad \textit{(S-trans)}\ \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3} \qquad \textit{(S-subclass)}\ \frac{\mathbf{I} \Rightarrow_C \mathbf{J}}{\mathbf{I} \leq \mathbf{J}}$$

$$\textit{(S-instance)}\ \frac{m \in_I \mathbf{J}}{m \leq \mathbf{J}} \qquad \textit{(S-tycon)}\ \frac{\overline{s} \leq \overline{t}}{T\,\overline{s} \leq T\,\overline{t}} \qquad \textit{(S-fun)}\ \frac{t_1 \leq s_1 \quad s_2 \leq t_2}{s_1 \to s_2 \leq t_1 \to t_2}$$

$$\textit{(S-qual)}\ \frac{s \leq t}{\forall \overline{a}.Q \Rightarrow s \leq \forall \overline{a}.Q \Rightarrow t}$$

Fig. 5: Subtyping.

relation induces subtyping among the interface types. The remaining structural rules are standard.

Interestingly, the instantiation relation $\preceq$ already includes the desired subtyping relation $\leq$.

Lemma 1. *The instance relation $P \vdash^{dsk} s \preceq t$ is reflexive and transitive.*

Lemma 2. *1. If $\boldsymbol{I} \Rightarrow_C \boldsymbol{J}$, then $P \Vdash \boldsymbol{I} \preceq \boldsymbol{J}$.*
2. If $m \in_I \boldsymbol{J}$, then $P \Vdash m \preceq \boldsymbol{J}$.

Lemma 3. *$s \leq t$ implies $P \Vdash s \preceq t$.*

4 Inference

The type system presented in Section 2 is in logical form. It is a declarative specification for the acceptable type derivations, but gives no clue how to compute such a derivation, in particular because rules *(E-gen)* and *(E-spec)* allow us to generalize and specialize, respectively, everywhere. Thus, the logical system is suitable for investigating meta-theoretical properties of the system, such as type soundness, but unsuitable for inferring types.

This section introduces a bidirectional version of the system geared at type inference. As the development in this paper parallels the one in the "Practical Type Inference" paper [14, Fig. 8], we omit the intermediate step of forming a syntax-directed system, which would not yield further insight.

Direction $\delta ::= \Uparrow \mid \Downarrow$

Judgment $P \mid \Gamma \vdash_\delta e : r$

$$\textit{(BE-var)}\ \frac{P \vdash_\delta^{inst} s \preceq r}{P \mid \Gamma(x : s) \vdash_\delta x : r} \qquad \textit{(BE-lam1)}\ \frac{P \mid \Gamma(x : m) \vdash_\Uparrow e : r}{P \mid \Gamma \vdash_\Uparrow \lambda x.e : m \to r}$$

$$\textit{(BE-lam2)}\ \frac{P \mid \Gamma(x : s) \vdash_\Downarrow^{poly} e : s'}{P \mid \Gamma \vdash_\Downarrow \lambda x.e : s \to s'} \qquad \textit{(BE-alam1)}\ \frac{P \mid \Gamma(x : s) \vdash_\Uparrow e : r}{P \mid \Gamma \vdash_\Uparrow \lambda(x :: s).e : s \to r}$$

$$\textit{(BE-lam2)}\ \frac{P \mid \Gamma(x : s) \vdash_\Downarrow^{poly} e : s'' \quad P \vdash^{dsk} s' \preceq s}{P \mid \Gamma \vdash_\Downarrow \lambda(x :: s).e : s' \to s''}$$

$$\textit{(BE-app)}\ \frac{P \mid \Gamma \vdash_\Uparrow f : s' \to s'' \quad P \mid \Gamma \vdash_\Downarrow^{poly} e : s' \quad P \vdash_\delta^{inst} s'' \preceq r}{P \mid \Gamma \vdash_\delta f\, e : r}$$

$$\textit{(BE-ann)}\ \frac{P \mid \Gamma \vdash_\Downarrow^{poly} e : s \quad P \vdash_\delta^{inst} s \preceq r}{P \mid \Gamma \vdash_\delta (e :: s) : r}$$

$$\textit{(BE-let)}\ \frac{P \mid \Gamma \vdash_\Uparrow^{poly} e : s \quad P \mid \Gamma(x : s) \vdash_\delta f : r}{P \mid \Gamma \vdash_\delta \texttt{let}\, x = e\, \texttt{in}\, f : r}$$

Generalization judgment $P \mid \Gamma \vdash_\delta^{poly} e : s$

$$\textit{(BE-gen1)}\ \frac{P, Q \mid \Gamma \vdash_\Uparrow e : r \quad \overline{a} = \mathrm{free}(r) \setminus \mathrm{free}(\Gamma) \quad \mathrm{free}(P) \subseteq \mathrm{free}(\Gamma)}{P \mid \Gamma \vdash_\Uparrow^{poly} e : \forall \overline{a}.Q \Rightarrow r}$$

$$\textit{(BE-gen2)}\ \frac{P, Q \mid \Gamma \vdash_\Downarrow e : r \quad \mathrm{pr}(s) = \forall \overline{a}.Q \Rightarrow r \quad \overline{a} \cap \mathrm{free}(\Gamma) = \emptyset \quad \mathrm{free}(P) \subseteq \mathrm{free}(\Gamma)}{P \mid \Gamma \vdash_\Downarrow^{poly} e : s}$$

$$\textit{(BE-gen3)}\ \frac{P \Vdash \mathbf{I}\, m \quad P \mid \Gamma \vdash_\Downarrow e : m}{P \mid \Gamma \vdash_\Downarrow^{poly} e : \mathbf{I}}$$

Instantiation judgment $P \mid \Gamma \vdash_\delta^{inst} e : s \preceq r$

$$\textit{(BE-inst1)}\ \frac{P \Vdash Q[\overline{a} \mapsto \overline{m}]}{P \vdash_\Uparrow^{inst} \forall \overline{a}.Q \Rightarrow r \preceq r[\overline{a} \mapsto \overline{m}]} \qquad \textit{(BE-inst2)}\ \frac{P \vdash^{dsk} s \preceq r}{P \vdash_\Downarrow^{inst} s \preceq r}$$

Fig. 6: Bidirectional version of Odersky and Läufer, extended with qualified types and subtyping of interface types.

Fig. 6 displays the rules of the directional system. It relies on the same entailment relation as the logical system as well as on the weak prenex transformation and (extended) deep skolemization judgments from Figures 2 and 4. It extends the system of Peyton Jones et al [14] with the handling of predicates in general, one extra instantiation rule *(I-iface)*, and one extra generalization rule *(BE-*

Type translation

$$|a| = a \qquad |s \to t| = |s| \to |t| \qquad |\mathbf{I}| = W_{\mathbf{I}} \qquad |\forall \overline{a}.P \Rightarrow t| = \forall \overline{a}.|P| \to |t|$$

$$|\texttt{true}| = * \qquad |P, Q| = |P| \times |Q| \qquad |\mathbf{I}\, m| = E_{\mathbf{I}}\{|m|\}$$

Term translation (excerpt)

$$\textit{(TS-tycon)}\ \frac{P \vdash^{dsk^*} h : \overline{s} \preceq \overline{r}}{P \vdash^{dsk^*} map_T\ h : T\,\overline{s} \preceq T\,\overline{r}}$$

$$\textit{(TS-iface)}\ \frac{P \Vdash y : \mathbf{I}\, m}{P \vdash^{dsk^*} \lambda(x :: |m|).K_{\mathbf{I}}\ y\, x : m \preceq \mathbf{I}}$$

$$\textit{(TE-gen3)}\ \frac{\overline{v : A} \Vdash y : \mathbf{I}\, r \quad \overline{v : A} \mid \Gamma \vdash_{\Downarrow} e \rightsquigarrow e' : r}{\overline{v : A} \mid \Gamma \vdash^{poly}_{\Downarrow} e \rightsquigarrow K_{\mathbf{I}}\ y\, e' : \mathbf{I}}$$

Fig. 7: Translation from λ^I to System F. $E_{\mathbf{I}}\{t\}$ is the type of evidence values for class $\mathbf{I}$ at instance t. The type of a wrapper constructor is $K_{\mathbf{I}} : \forall a.E_{\mathbf{I}}\{a\} \to a \to W_{\mathbf{I}}$.

gen3). This rule mimics *(I-iface)*, but it can only occur in a derivation in a place where full instantiation is not desired so that *(I-iface)* is not applicable.

It is straightforward to check that the directional system is sound with respect to the logical system. However, doing so requires extra type annotations because the directional system accepts more programs than the logical one.

Lemma 4. *Suppose that $P \mid \Gamma \vdash^{poly}_{\delta} e : s$. Then there exists some e' such that $P \mid \Gamma \vdash e' : s$ where e' differs from e only in additional type annotations on the bound variables of lambda abstractions.*

Here is an example for a term that type checks in the directional system (extended with integers and booleans), but not in the logical system:

$$\lambda(f :: ((\forall a.a \to a) \to \texttt{int} \times \texttt{bool}) \to (\forall a.a \to a) \to \texttt{int} \times \texttt{bool}).\\ f\,(\lambda id.(id\,5, id\,false))\,(\lambda x.x)$$

It works in the directional system because the rule *(BE-app)* infers a polymorphic type for f, checks its argument against the resulting polymorphic type, and then does the same for the second argument. However, in the logical system, the argument of the function $\lambda id.(id\,5, id\,false)$ cannot receive a polymorphic type.

The directional system extends the logical one (viz. [14, Theorem 4.8]).

Lemma 5. *Suppose that $P \mid \Gamma \vdash e : s$. Then $P \mid \Gamma \vdash^{poly}_{\delta} e : s'$ and $P \vdash^{dsk} s' \preceq s$.*

It is not clear whether the directional system retains the principal types property (viz. [14, Theorem 4.13]).

5 Translation to System F

The last step towards an implementation of interface types in a Haskell compiler is the translation to its internal language, System F, in Fig. 7. Peyton Jones

et al [14] define most parts of this translation by extending their bidirectional system (without predicate handling), so the figure concentrates on the rules and types not present in that work.

This translation maps an atomic predicate $\mathbf{I}\,m$ into *evidence*, which is a variable binding $v : |\mathbf{I}\,m|$. The value of v is a suitable dictionary $D_{\mathbf{I}}\{|m|\} : E_{\mathbf{I}}\{|m|\}$ for the type class $\mathbf{I}$. A value with interface type $\mathbf{I}$ is represented as a System F term of type $W_{\mathbf{I}}$. This term wraps a dictionary of type $E_{\mathbf{I}}\{|m|\}$ and a witness of type $|m|$ using an automatically generated datatype constructor $K_{\mathbf{I}}$.

The translation rules for instantiation have the form $P \vdash h : s \preceq s'$ and yield a System F term h of type $|s| \to |s'|$. The rule *(TS-tycon)* demonstrates why the type constructors T have to be covariant: the translation of the instantiation judgment requires a map operation map_T for each such T.[7] The rule *(TS-iface)* shows the conversion of an instance type m of class $\mathbf{I}$ to its interface type $\mathbf{I}$ by applying the wrapper constructor $K_{\mathbf{I}}$ to the dictionary y yielded by the extended entailment judgment and the value x of type $|m|$.

The translation rules for expressions have the form $P \mid \Gamma \vdash e \rightsquigarrow e' : r$ where e is the source expression and e' its translation. The rule *(TE-gen3)* performs essentially the same task as *(TS-iface)*, but in a term context.

The translation to System F preserves types:

Lemma 6. *Let $\vdash^F$ denote the System F typing judgment.*

- *Suppose that $\overline{(v :: A)} \mid \Gamma \vdash e \rightsquigarrow e' : r$. Then $\overline{(v :: |A|)}, |\Gamma| \vdash^F e' : |r|$.*
- *Suppose that $\overline{(v :: A)} \mid \Gamma \vdash^{poly} e \rightsquigarrow e' : s$. Then $\overline{(v :: |A|)}, |\Gamma| \vdash^F e' : |s|$.*

6 Related Work

There is a lot of work on type inference for first-class polymorphism [7–10,12,14, 18,20,21]. Our inference algorithm directly extends the algorithm for predicative higher-rank types of Peyton Jones and others [14] with support for interface types. The interface type system is also predicative.

Läufer [11] extends algebraic data types with existential quantification constrained over type classes. In Läufer's system, programmers have to explicitly pack and unpack existential types through the standard introduction and elimination constructs of algebraic data types. Our approach translates interface types into algebraic data types with existential quantification. Type annotations serve as the pack operation for interface types. An explicit unpack operation for interface types is not required.

Diatchki and Jones [2] use type class names to provide a functional notation for functional dependencies. The name of a type class C with $n+1$ parameters serves as a type operator that, when applied to n type arguments, represent the $(n+1)$th class parameter, which must be uniquely determined by the other class parameters. Hence, the type $C\,t_1 \ldots t_n$ translates to a fresh type variable a subject to the constraint $C\,t_1 \ldots t_n\,a$. With interface types, the name of a single-parameter type class represents some unknown type that implements the type

[7] In Haskell, T has to be an instance of `Functor`, so map_T becomes `fmap`.

class. An interface type in argument position of some type signature could be handled by a local translation similar to the one used by Diatchki and Jones. An interface type in result position, however, requires a different treatment to cater for the existential nature of interface types.

Oliveira and Sulzmann [13] present a Haskell extension that unifies type classes and GADTs. Their extension also includes a feature that allows class names being used as types. Similar to our work, these types represent some unknown instance of the class. Different from our work, Oliveira and Sulzmann provide neither a type inference algorithm and nor an implementation.

The present authors [22] investigate a language design that extends Java's interface mechanism with the key features of Haskell's type class system. The resulting language generalizes interface types to bounded existential types, following an idea already present with LOOM's hash types [1].

Standard ML's module system [19] allows programmers to hide the implementation of a module behind an interface (*i.e.*, signature). For example, an ML implementation of the database library from Section 2.1 might provide several database-specific modules such that all implementation details are hidden behind a common signature. A programmer then chooses at *compile-time* which database should be used. In contrast, interface types allows this choice to be deferred until *runtime*. Providing this kind of flexibility to users of Standard ML's module system would require first-class structures [17].

7 Conclusion and Future Work

An interface type can be understood as an existential type representing an unknown instance of some type class. We demonstrated the usefulness of interface types through a case study from the real world and formalized a type system with support for interface types. Based on this type system, we implemented a prototype of a type inference algorithm that can be included easily in a production Haskell compiler.

Here are some items for future work:

- How about higher-order polymorphism? For higher-order classes such as `Monad`, the interface type would be parameterized and encapsulate a particular implementation of `Monad`.
- How about multi-parameter type classes? To support interface types for multi-parameter type classes, we would need explicit pack and unpack operations that coerce multiple values to/from an interface type.
- What if a type is coerced multiple times to an interface type? Each coercion results in the application of a wrapper, so that there might be a stack of wrappers. There is not much that can be done about it at the source level and it is not clear if it would have a significant performance impact on a realistic program. However, the implementation could be instrumented to have the application of a wrapper constructor check dynamically if it is applied to another wrapper and thus avoid the piling up of wrappers.

References

1. K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In M. Aksit and S. Matsuoka, editors, *11th ECOOP*, number 1241 in LNCS, pages 104–127, Jyväskylä, Finland, June 1997. Springer.
2. I. S. Diatchki and M. P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In A. Löh, editor, *Proceedings of the 2006 ACM SIGPLAN Haskell Workshop*, pages 72–83, Portland, Oregon, USA, Sept. 2006.
3. GHC. The Glasgow Haskell compiler. `http://www.haskell.org/ghc/`, 2008.
4. J. Goerzen. Haskell database connectivity. `http://software.complete.org/software/projects/show/hdbc`, 2008.
5. Hugs 98. `http://www.haskell.org/hugs/`, 2003.
6. M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.
7. M. P. Jones. First-class polymorphism with type inference. In N. Jones, editor, *Proc. 1997 ACM Symp. POPL*, pages 483–496, Paris, France, Jan. 1997. ACM Press.
8. D. Le Botlan and D. Rémy. MLF: raising ML to the power of System F. In O. Shivers, editor, *Proc. ICFP 2003*, pages 27–38, Uppsala, Sweden, Aug. 2003. ACM Press, New York.
9. D. Leijen. HMF: Simple type inference for first-class polymorphism. In *ICFP*, pages 283–293. ACM Press, 2008.
10. D. Leijen and A. Löh. Qualified types for MLF. In Pierce [15], pages 144–155.
11. K. Läufer. Type classes with existential types. *J. Funct. Program.*, 6(3):485–517, May 1996.
12. M. Odersky and K. Läufer. Putting type annotations to work. In *Proc. 1996 ACM Symp. POPL*, pages 54–67, St. Petersburg, FL, USA, Jan. 1996. ACM Press.
13. B. Oliveira and M. Sulzmann. Objects to unify type classes and GADTs. `http://www.cs.mu.oz.au/~sulzmann/manuscript/objects-unify-type-classes-gadts.ps`, Apr. 2008.
14. S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.
15. B. C. Pierce, editor. *ICFP*, Tallinn, Estonia, Sept. 2005. ACM Press, New York.
16. PostgreSQL, the most advanced Open Source database system in the world. `http://www.postgresql.org`, 2008.
17. C. V. Russo. First-class structures for Standard ML. In G. Smolka, editor, *Proc. 9th ESOP*, number 1782 in LNCS, pages 336–350, Berlin, Germany, Mar. 2000. Springer.
18. D. Rémy. Simple, partial type-inference for System F based on type-containment. In Pierce [15], pages 130–143.
19. M. Tofte. Essentials of Standard ML Modules. In *Advanced Functional Programming*, pages 208–238. Springer-Verlag, 1996.
20. D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In J. Lawall, editor, *Proc. ICFP 2006*, pages 251–262, Portland, Oregon, USA, Sept. 2006. ACM Press, New York.
21. D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH: First-class polymorphism for Haskell. In *ICFP*, 2008. To appear, `http://www.cis.upenn.edu/~dimitriv/fph/`.
22. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 347–372, Berlin, Germany, July 2007. Springer.