Interfaces Types for Haskell

Peter Thiemann Stefan Wehr

University of Freiburg, Germany

APLAS; December 11, 2008; Bangalore, India

Motivation

What's the outcome of the following Haskell program?

```
Prelude> map show [1, True, "APLAS"]
```

Motivation

What's the outcome of the following Haskell program?

```
Prelude> map show [1, True, "APLAS"]
```

```
ERROR: Couldn't match expected type `Bool' against
           inferred type `[Char]'
```

Motivation

What's the outcome of the following Haskell program?

```
Prelude> map show [1, True, "APLAS"]
```

```
  ERROR: Couldn't match expected type 'Bool' against
           inferred type '[Char]'
```

With interface types, we can have

```
Prelude> map show ([1, True, "APLAS"] :: [Show])
["1","True","\"APLAS\""]
```

- `Show` is an **interface type**: it stands for some instance of the builtin type class `Show`
- Type annotation forces the list elements to have type `Show`

A real world case study

- Requirements for a database access library:
  - Support for common database operations
  - Support for “special features” of a particular database
  - Public interface independent from a particular database, except for
    - opening new connections
    - use of special features

A real world case study

- Requirements for a database access library:
  - Support for common database operations
  - Support for “special features” of a particular database
  - Public interface independent from a particular database, except for
    - opening new connections
    - use of special features
- HDBC (Haskell Database Connectivity)
  - Library for accessing SQL databases from Haskell
  - Drivers for different databases:
    ODBC, PostgreSQL, Sqlite v3

API of HDBC

- Representing connections through type classes

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

- Extending connections through sub classes

```haskell
class IConnection c => IConnectionAsync c where
  listen :: c -> String -> IO ()
  notify :: c -> String -> IO ()
```

- Sample driver (for Sqlite)

```haskell
module Database.HDBC.Sqlite (ConSqlite(), connectSqlite) where
data ConSqlite = ConSqlite { sqliteQuery :: String -> IO [[String]] }
instance IConnection ConSqlite where
  dbQuery = sqliteQuery
openConnection :: FilePath -> IO ConSqlite
```

API of HDBC

- Representing connections through type classes

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

- Extending connections through sub classes

```
class IConnection c => IConnectionAsync c where
  listen :: c -> String -> IO ()
  notify :: c -> String -> IO ()
```

- Sample driver (for Sqlite)

```
module Database.HDBC.Sqlite (ConSqlite(), connectSqlite) where
data ConSqlite = ConSqlite { sqliteQuery :: String -> IO [[String]] }
instance IConnection ConSqlite where
  dbQuery = sqliteQuery
openConnection :: FilePath -> IO ConSqlite
```

Problem: Difficult to hide the concrete connection type

What is a good type for opening a connection?

- Use database-specific datatype

  ```
  openConnection :: FilePath -> IO ConSqlite
  ```

  Drawback: reveals concrete connection type

What is a good type for opening a connection?

- Use database-specific datatype

```
openConnection :: FilePath -> IO ConSqlite
```

Drawback: reveals concrete connection type

- Use continuations and rank-2 types

```
withSqliteConnection :: FilePath
                     -> (forall c . Connection c => c -> IO a)
                     -> IO a
```

Drawback: program must be written in CPS

What is a good type for opening a connection?

- Use database-specific datatype

```
openConnection :: FilePath -> IO ConSqlite
```

Drawback: reveals concrete connection type

- Use continuations and rank-2 types

```
withSqliteConnection :: FilePath
                      -> (forall c . Connection c => c -> IO a)
                      -> IO a
```

Drawback: program must be written in CPS

- Use algebraic data types with existential quantification

```
data ExIConnection = forall c . Connection c
                                => ExIConnection c
instance IConnection ExIConnection where
    dbQuery (ExIConnection c) = dbQuery c
openConnection :: FilePath -> IO ExIConnection
openConnection fp = do con <- internOpen fp
                       return (ExIConnection con)
```

Drawback: boilerplate code, explicit pack/unpack operations

Doing it the OO way

- Here is how you would program the example in Java:

```
public interface IConnection { String[][] dbQuery(String query); }
public class SqliteDB {
    public IConnection openConnection(File f) { ... }
}
```

Doing it the OO way

- Here is how you would program the example in Java:

```
public interface IConnection { String[][] dbQuery(String query); }
public class SqliteDB {
    public IConnection openConnection(File f) { ... }
}
```

- Translated to Haskell:

```
-- type class IConnection as before
module Database.HDBC.Sqlite (connectSqlite) where
data ConSqlite = ConSqlite { sqliteQuery :: String -> IO [[String]] }
instance IConnection ConSqlite where dbQuery = sqliteQuery
internConnectSqlite :: FilePath -> IO ConSqlite
connectSqlite :: FilePath -> IO IConnection
connectSqlite = internConnectSqlite
```

- **The interface type** `IConnection` abstracts over some instance of `IConnection`
- No boilerplate code
- Connection type remains abstract

The fine print

- For a type class $\mathbf{I}$, the **interface type** $\mathbf{I}$ is equivalent to the bounded existential type $\exists\alpha.\mathbf{I}\ \alpha \Rightarrow \alpha$.
- **Introduction** of interface types through type annotations: $(e :: \mathbf{I})$ results in wrapping e in a constructor $K_{\mathbf{I}}$
- **Elimination** of interface types: automatically because $K_{\mathbf{I}}$ is an instance of $\mathbf{I}$
- **Subtyping** on interface types: induced by subclassing on type classes, e.g.

  `IConnectionAsync` is a subtype of `IConnection`.

- Some type classes cannot be used as interface types:

```
class Show a          where show    :: a -> String        OK
class IConnection c   where dbQuery :: c -> [[String]]    OK
class Eq a            where (==)    :: a -> a -> Bool     Error
class Read a          where read    :: String -> a        Error
```

λ^I, a calculus for interface types

- Based on
  - Jones' system for qualified types [Jon94]
  - Odersky and Läufer's system [OL96] for type annotations
  - Peyton Jones and colleagues' system for higher-rank polymorphism [PVWS07]
- Sound and complete type inference
- Unclear whether principal types do exist
- Translation to System F
- Prototype implementation available

Syntax and subtyping

- Syntax

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

- Subtyping

$$
\frac{\mathbf{I} \text{ subclass of } \mathbf{J}}{\mathbf{I} \leq \mathbf{J}} \qquad \frac{m \text{ instance of } \mathbf{J}}{m \leq \mathbf{J}}
$$

$$
\frac{\bar{s} \leq \bar{t}}{T\,\bar{s} \leq T\,\bar{t}} \qquad \frac{t_1 \leq s_1 \quad s_2 \leq t_2}{s_1 \to s_2 \leq t_1 \to t_2} \qquad \frac{s \leq t}{\forall \overline{a}.Q \Rightarrow s \leq \forall \overline{a}.Q \Rightarrow t}
$$

$$
t \leq t \qquad \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}
$$

Entailment and subsumption

- Entailment $P \Vdash \mathbf{I}\, m$
  Extends Haskell's entailment relation with the following rule:

$$\frac{\mathbf{I} \text{ subclass of } \mathbf{J}}{P \Vdash \mathbf{J}\, \mathbf{I}}$$

- Subsumption relation $P \vdash^{dsk} s \preceq t \quad P \vdash^{dsk^*} s \preceq t$
  Extends Peyton Jones and colleagues' subsumption relation with support for qualified types and the following two rules:

$$\frac{T \text{ covariant} \quad P \vdash^{dsk^*} \overline{s} \preceq \overline{r}}{P \vdash^{dsk^*} T\,\overline{s} \preceq T\,\overline{r}} \qquad \frac{P \Vdash \mathbf{I}\, m}{P \vdash^{dsk^*} m \preceq \mathbf{I}}$$

Entailment and subsumption

- Entailment $P \Vdash \mathbf{I}\, m$
  Extends Haskell's entailment relation with the following rule:

$$\frac{\mathbf{I} \text{ subclass of } \mathbf{J}}{P \Vdash \mathbf{J}\,\mathbf{I}}$$

- Subsumption relation $P \vdash^{dsk} s \preceq t \quad P \vdash^{dsk*} s \preceq t$
  Extends Peyton Jones and colleagues' subsumption relation with support for qualified types and the following two rules:

$$\frac{T \text{ covariant} \quad P \vdash^{dsk*} \overline{s} \preceq \overline{r}}{P \vdash^{dsk*} T\,\overline{s} \preceq T\,\overline{r}} \qquad \frac{P \Vdash \mathbf{I}\, m}{P \vdash^{dsk*} m \preceq \mathbf{I}}$$

Lemma (Subtyping implies subsumption)

If $s \leq t$ then $P \vdash^{dsk} s \preceq t$.

Expression typing

- Declarative typing judgment $P \mid \Gamma \vdash e : s$
  - Support for qualified types (Jones [Jon94])
  - Support for higher-rank types introduced through type annotations (Odersky and Läufer [OL96])
- Bidirectional inference judgment $P \mid \Gamma \vdash_{\delta}^{poly} e : s$
  - Checking mode: $\delta = \Downarrow$
  - Inference mode: $\delta = \Uparrow$
  - Slight variation of Peyton Jones and colleagues' system [PVWS07]

Expression typing

- Declarative typing judgment $P \mid \Gamma \vdash e : s$
  - Support for qualified types (Jones [Jon94])
  - Support for higher-rank types introduced through type annotations (Odersky and Läufer [OL96])
- Bidirectional inference judgment $P \mid \Gamma \vdash^{poly}_{\delta} e : s$
  - Checking mode: $\delta = \Downarrow$
  - Inference mode: $\delta = \Uparrow$
  - Slight variation of Peyton Jones and colleagues' system [PVWS07]

Lemma (Completeness)

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

Expression typing

- Declarative typing judgment $P \mid \Gamma \vdash e : s$
  - Support for qualified types (Jones [Jon94])
  - Support for higher-rank types introduced through type annotations (Odersky and Läufer [OL96])
- Bidirectional inference judgment $P \mid \Gamma \vdash^{poly}_{\delta} e : s$
  - Checking mode: $\delta = \Downarrow$
  - Inference mode: $\delta = \Uparrow$
  - Slight variation of Peyton Jones and colleagues' system [PVWS07]

Lemma (Completeness)

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

Lemma (Soundness)

Suppose $P \mid \Gamma \vdash^{poly}_{\delta} e : s$. 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.

Translation to System F

- Type-directed translation from λ^I to System F
- Evidence for predicates passed as *dictionaries*:
  $E_{\mathbf{I}}\{\tau\}$ is the type of evidence values for class $\mathbf{I}$ at instance τ
- Subsumption to an interface type $\mathbf{I}$ introduces a wrapper constructor $K_{\mathbf{I}} : \forall\alpha.E_{\mathbf{I}}\{\alpha\} \to \alpha \to W_{\mathbf{I}}$

Lemma (Type preservation)

The translation from λ^I to System F preserves types.

Summary

- Idea behind interface types:

  Use the name of a type class as a type!

- Allows for heterogeneous lists and type abstraction
- Reduces boilerplate code
- Formalization close to the type checking algorithm implemented in GHC
- Prototype implementation available

References

[Jon94] Mark P. Jones.
Qualified Types: Theory and Practice.
Cambridge University Press, Cambridge, UK, 1994.

[OL96] Martin Odersky and Konstantin Läufer.
Putting type annotations to work.
In *Proc. 1996 ACM Symp. POPL*, pages 54–67, St. Petersburg, FL, USA, January 1996. ACM Press.

[PVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields.
Practical type inference for arbitrary-rank types.
J. Funct. Program., 17(1):1–82, 2007.

Representing connections as records

- Implementation chosen by HDBC up to version 1.0.1.2
- The `Connection` datatype

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

- Concrete implementation of a database driver

```
module Database.HDBC.Sqlite (connectSqlite) where
connectSqlite :: FilePath -> IO Connection
```

- Client usage

```
openConnection :: IO Connection
openConnection = connectSqlite "/var/sqlite/app.db"
allCustomers :: Connection -> IO [[String]]
allCustomers con = dbQuery con "SELECT * FROM customers"
main = do con <- openConnection
          customers <- allCustomers con
          putStrLn (show customers)
```

- **Problem:** Extending the `Connection` type with new operations requires a new, incompatible datatype