Towards Interface Types for Haskell

Work in Progress

Peter Thiemann

Joint work with Stefan Wehr
Universität Freiburg, Germany

WG 2.8 Meeting, Nesjavellir, Island, 17.07.2007

What is a type class?

What is a type class?

- A type class is a signature of an abstract data type.

What is a type class?

- A type class is a signature of an abstract data type.
- But where is the abstract type?

Example: HDBC interface

- Signature of abstract data type

```
module HDBC where
class Connection conn where
  exec :: conn -> String -> IO QueryResult
```

Example: HDBC interface

- Signature of abstract data type

```
module HDBC where
class Connection conn where
  exec :: conn -> String -> IO QueryResult
```

- Implementation of abstract data type

```
module PostgreSQLDB where
import HDBC
instance Connection PostgreSQLConnection where
  exec = pgsqlExec
```

Example: HDBC interface

- Signature of abstract data type

```
module HDBC where
class Connection conn where
  exec :: conn -> String -> IO QueryResult
```

- Implementation of abstract data type

```
module PostgreSQLDB where
import HDBC
instance Connection PostgreSQLConnection where
  exec = pgsqlExec
```

- Extending the abstract data type

```
class Connection conn => BetterConnection conn where
  notify :: conn -> String -> IO ()
```

Why want an abstract type?

- **Encapsulation**
  What is a good type for `connect`?

Why want an abstract type?

- **Encapsulation**
  What is a good type for connect?
- Can do with

```
connectWith :: URL -> (forall c. Connection c => c -> IO a) -> IO a
```

  - but: requires user code in continuation
  - no "connection value" that can be stored
  - not possible as member of class Connection

Why want an abstract type?

- **Encapsulation**
  What is a good type for connect?
- Can do with

```
connectWith :: URL -> (forall c. Connection c => c -> IO a) -> IO a
```

  - but: requires user code in continuation
  - no "connection value" that can be stored
  - not possible as member of class Connection
- How about

```
connect :: URL -> IO Connection
```

  where Connection behaves like a Java interface type?

Interfaces for Haskell

A Design Proposal

- Type class **I** $\Rightarrow$ interface type **I**
- Type **I** is **exists** c. (I c) => c

Interfaces for Haskell

A Design Proposal

- Type class **I** $\Rightarrow$ interface type **I**
- Type **I** is **exists** c. (I c) => c
- Subtyping for interface types
  if **I** is a subclass of **J**,
  then **I** $\leq$ **J**
- Subtyping for instance types
  if t is an instance type of `J`,
  then $t \leq$ **J**

- Type class **I** $\Rightarrow$ interface type **I**
- Type **I** is **exists** c. (I c) => c
- Subtyping for interface types
  if **I** is a subclass of **J**,
  then **I** $\leq$ **J**
- Subtyping for instance types
  if *t* is an instance type of `J`,
  then *t* $\leq$ **J**
- Introduction by type annotation
  $\Rightarrow$ no new syntax

Example Patterns of Use

- Create a connection

```
betterConnect :: URL -> IO BetterConnection
betterConnect url =
  do c <- pgconnect url
     -- c :: PGSQLConnection
     return (c :: BetterConnection)
```

- Wrapper

```
dbwrapper :: URL -> (URL -> IO Connection) -> IO Result
dbwrapper url connect =
  do c <- connect url
     do_something c

... dbwrapper url betterConnect ...
```

- Worker

```
worker :: Connection -> IO Result
withBetterConnection :: (BetterConnection -> IO a) -> IO a

... withBetterConnection worker ...
```

Collecting the Pieces

Surprise!

- Everything needed is (almost) there

Collecting the Pieces

Existential Types in Haskell

```
data T_Connection where
  T_Connection :: forall conn.
    Connection conn => conn -> T_Connection
data T_BetterConnection where
  T_BetterConnection :: forall conn.
    BetterConnection conn => conn -> T_BetterConnection

instance T_Connection Connection where ...
instance T_Connection BetterConnection where ...
instance T_BetterConnection BetterConnection where ...
```

- Tagged existentials
- Need pattern match to unpack

Collecting the Pieces

Subtyping in Haskell

- There is no subtyping in Haskell!

Collecting the Pieces

Subtyping in Haskell

- There is no subtyping in Haskell!
- But, there is the generic instance relation:
  forall c. BetterConnection c => c −> T
  $\preceq$
  forall c. Connection c => c −> T

Collecting the Pieces

Subtyping in Haskell

- There is no subtyping in Haskell!
- But, there is the generic instance relation:
  forall c. BetterConnection c => c -> T
  $\preceq$
  forall c. Connection c => c -> T
- And there is the double negation equivalence:

exists a. P => T = (**forall** a. P => T -> x) -> x

Collecting the Pieces

Subtyping in Haskell

- There is no subtyping in Haskell!
- But, there is the generic instance relation:
  forall c. BetterConnection c => c −> T
  $\preceq$
  forall c. Connection c => c −> T
- And there is the double negation equivalence:

  exists a. P => T = (**forall** a. P => T −> x) −> x

- Approach: Translate existential types to (higher-rank) polymorphism where possible

Example Translation

Create a Connection

```
betterConnect :: URL -> IO BetterConnection
betterConnect url =
  do c <- pgconnect url
     -- c :: PGSQLConnection
     return (c :: BetterConnection)
```

translates to

```
betterConnect' :: URL -> IO T_BetterConnection
betterConnect' url =
  do c <- pgconnect url
     return (T_BetterConnection c)
```

Example Translation
Wrapper

```
dbwrapper :: URL -> (URL -> IO Connection) -> IO Result
dbwrapper url connect =
  do c <- connect url
     do_something c

... dbwrapper url betterConnect ...
```

translates to

```
dbwrapper' :: URL -> forall c. Connection c => (URL -> IO c) -> IO Result
dbwrapper' url connect =
  do c <- connect url
     do_something c

betterConnect' :: URL -> IO T_BetterConnection
... dbwrapper' url betterConnect' ...
```

Example Translation

Worker

```
worker :: Connection -> IO Result
withBetterConnection :: (BetterConnection -> IO a) -> IO a

... withBetterConnection worker ...
```

translates to

```
worker' :: forall c . Connection c => c -> IO Result
withBetterConnection' :: (forall c. BetterConnection c => c -> IO a) -> IO a

... withBetterConnection' worker' ...
```

Interfaces for Haskell

Translational Approach

- Starting point: Haskell with higher-rank polymorphism (as in current implementations)
- Extensions:
  Extended syntax of types

$$s, t ::= \cdots \mid \mathbf{I}$$

  Typing rules

$$\text{(E-ann')}\ \frac{P \mid \Gamma \vdash' e : s \quad s \leq t}{P \mid \Gamma \vdash' (e :: t) : t}$$

$$\text{(E-sub')}\ \frac{P \mid \Gamma \vdash' e : s \quad s \leq' t}{P \mid \Gamma \vdash' e : t}$$

Subtyping

$$\textit{(S-refl)}\; t \le t$$

$$\textit{(S-trans)}\; \frac{t_1 \le t_2 \quad t_2 \le t_3}{t_1 \le t_3}$$

$$\textit{(S-subclass)}\; \frac{\mathbf{I} \Rightarrow_C \mathbf{J}}{\mathbf{I} \le \mathbf{J}}$$

$$\textit{(S-instance)}\; \frac{m \in_I \mathbf{J}}{m \le \mathbf{J}}$$

$$\textit{(S-tycon)}\; \frac{\overline{s} \le \overline{t}}{T\,\overline{s} \le T\,\overline{t}}$$

$$\textit{(S-fun)}\; \frac{t_1 \le s_1 \quad s_2 \le t_2}{s_1 \to s_2 \le t_1 \to t_2}$$

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

Restricted Subtyping

$$t \leq' t \qquad \frac{t_1 \leq' t_2 \quad t_2 \leq' t_3}{t_1 \leq' t_3} \qquad \frac{\overline{s} \leq' \overline{t}}{T\,\overline{s} \leq' T\,\overline{t}}$$

$$\frac{t_1 \leq s_1 \quad s_2 \leq' t_2}{s_1 \longrightarrow s_2 \leq' t_1 \longrightarrow t_2}$$

Restricted Subtyping

$$t \leq' t \qquad \frac{t_1 \leq' t_2 \quad t_2 \leq' t_3}{t_1 \leq' t_3} \qquad \frac{\overline{s} \leq' \overline{t}}{T\,\overline{s} \leq' T\,\overline{t}}$$

$$\frac{t_1 \leq s_1 \quad s_2 \leq' t_2}{s_1 \to s_2 \leq' t_1 \to t_2}$$

Restricted subtyping vs generic instance

Lemma

If $s \leq' t$ and $s \rightsquigarrow' s'$ and $t \rightsquigarrow' t'$ then $\texttt{true} \vdash s' \preceq t'$.

Translation of Types

$$a \leadsto' \Box / a \qquad \frac{\overline{t_i \leadsto' C'_i / t'_i}}{T\,\overline{t} \leadsto' \mathrm{mapT}\ \overline{(\lambda x. C'_i[x])}\ \Box / T\,\overline{t'}}$$

$$\frac{t_1 \leadsto \pi_1 \sharp t'_1 \quad t_2 \leadsto' C_2 / t'_2}{t_1 \rightarrow t_2 \leadsto' \lambda x. C_2[\Box\, x] / \pi_1(t'_1 \rightarrow t'_2)} \qquad \mathbf{I} \leadsto' K_{\mathbf{I}}\,\Box / W_{\mathbf{I}}$$

$$\frac{t \leadsto' C' / t'}{\forall a.P \Rightarrow t \leadsto' C' / \forall a.P \Rightarrow t'}$$

$$a \leadsto \emptyset \sharp a \qquad \frac{\overline{t} \leadsto \overline{\pi} \sharp \overline{t'}}{T\,\overline{t} \leadsto \overline{\pi} \sharp T\,\overline{t'}} \qquad \frac{t_1 \leadsto \pi_1 \sharp t'_1 \quad t_2 \leadsto \pi_2 \sharp t'_2}{t_1 \rightarrow t_2 \leadsto \pi_2 \sharp \pi_1(t'_1 \rightarrow t'_2)}$$

$$\mathbf{I} \leadsto \forall c. \mathbf{I}\, c \Rightarrow \sharp c \qquad \frac{t \leadsto \pi \sharp t'}{\forall a.Q \Rightarrow t \leadsto \pi \sharp \forall a.Q \Rightarrow t'}$$

Translation of Terms

$$x \hookrightarrow x \qquad \frac{e \hookrightarrow e'}{\lambda x.e \hookrightarrow \lambda x.e'} \qquad \frac{e \hookrightarrow e' \quad s \leadsto \emptyset \sharp s'}{\lambda(x :: s).e \hookrightarrow \lambda(x :: s').e'}$$

$$\frac{e \hookrightarrow e' \quad s \leadsto \forall \overline{c}.Q \sharp s' \quad s \leadsto' C'/s''}{\lambda(x :: s).e \hookrightarrow \Lambda\overline{c}(Q).\lambda(y :: s').(\lambda(x :: s'').e')\,(C'[y])}$$

$$\frac{f \hookrightarrow f' \quad e \hookrightarrow e'}{f\, e \hookrightarrow f'\, e'} \qquad \frac{e \hookrightarrow e' \quad f \hookrightarrow f'}{\texttt{let}\, x = e\, \texttt{in}\, f \hookrightarrow \texttt{let}\, x = e'\, \texttt{in}\, f'}$$

$$\frac{e \hookrightarrow e' \quad s \leadsto' C'/s'}{(e :: s) \hookrightarrow (C'[e'] :: s')}$$

Results

- Let $P \mid \Gamma' \vdash e' : s'$ be the typing judgment for Haskell with higher-rank qualified polymorphism.
- If $P \mid \Gamma \vdash' e : s$, $s \leadsto' s'$, $\Gamma \leadsto' \Gamma'$, and $e \hookrightarrow e'$, then $P \mid \Gamma' \vdash e' : s'$.

Conclusions

- Type translation maps subtyping to generic instantiation
- Term translation is typing preserving
- Both are purely syntactic
- Q: Is the term translation meaning preserving?
- Q: Is the translated term amenable to type inference?
- Q: Can we do direct inference and translation to F2?
- If Java interface types make sense for Haskell, then how about type classes for Java?

Conclusions

- Type translation maps subtyping to generic instantiation
- Term translation is typing preserving
- Both are purely syntactic
- Q: Is the term translation meaning preserving?
- Q: Is the translated term amenable to type inference?
- Q: Can we do direct inference and translation to F2?
- If Java interface types make sense for Haskell, then how about type classes for Java? $\Rightarrow$ **JavaGI @ECOOP'07**

Digression: The ML way

```
signature CONNECTION =
sig type connection
    val exec : connection -> string -> queryresult
end

signature BETTERCONNECTION =
sig type connection
    val exec : connection -> string -> queryresult
    val notify : connection -> string -> unit
end

structure PostgreSQL : BETTERCONNECTION =
struct type connection = postgreSQLConnection
        val exec = ...
        val notify = ...
end
```

- Encapsulation and Extensibility:
  BETTERCONNECTION <: CONNECTION
- But: application code as a functor taking a connection.