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?

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.
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

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

- Implementation of abstract data type

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

- Extending the abstract data type

```haskell
class Connection conn =
  BetterConnection conn where
  notify :: conn -> String -> IO ()
```
Example: HDBC interface

- Signature of abstract data type

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

- Implementation of abstract data type

```haskell
module PostgreSQLDB where
import HDBC
instance Connection PostgreSQLConnection where
  exec = pgsqlExec
```
Example: HDBC interface

- Signature of abstract data type

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

- Implementation of abstract data type

```haskell
module PostgreSQLDB where
import HDBC
instance Connection PostgreSQLConnection where
  exec = psqlExec
```

- Extending the abstract data type

```haskell
class Connection conn => BetterConnection conn where
  notify :: conn -> String -> IO ()
```
Why want an abstract type?

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

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

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

```
connect :: URL -> IO Connection
where
  Connection behaves like a Java interface type?
```
Why want an abstract type?

▶ **Encapsulation**
What is a good type for connect?

▶ Can do with

\[
\text{connectWith} :: \text{URL} \rightarrow (\text{forall c. Connection c} \rightarrow c \rightarrow \text{IO a}) \rightarrow \text{IO a}
\]

▶ 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 $\textbf{exists}$ $c$. $(I \ c) \Rightarrow c$

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

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

▶ Wrapper

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

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

▶ Worker

```haskell
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
There is no subtyping in Haskell!
There is no subtyping in Haskell!

But, there is the generic instance relation:

\[ \forall c. \text{BetterConnection} c \Rightarrow c \rightarrow T \leq \forall c. \text{Connection} c \Rightarrow c \rightarrow T \]
Collecting the Pieces
Subtyping in Haskell

- There is no subtyping in Haskell!
- But, there is the generic instance relation:
  \[
  \forall c. \text{BetterConnection} \implies c \rightarrow T <\forall c. \text{Connection} \implies c \rightarrow T
  \]
- And there is the double negation equivalence:
  \[
  \exists a. P \implies T = (\forall a. P \implies T \rightarrow x) \rightarrow x
  \]
There is no subtyping in Haskell!

But, there is the generic instance relation:

\[ \forall c. \text{BetterConnection } c \Rightarrow c \rightarrow T \]
\[ \prec \]
\[ \forall c. \text{Connection } c \Rightarrow c \rightarrow T \]

And there is the double negation equivalence:

\[ \exists a. P \Rightarrow T = (\forall a. P \Rightarrow T \rightarrow x) \rightarrow x \]

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 −> \textbf{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

\[ s, t ::= \cdots | \mathbf{I} \]

Typing rules

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

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

(S-refl) \( t \leq t \)  

(S-trans) \[
\frac{
\begin{align*}
    & t_1 \leq t_2 \\
    & t_2 \leq t_3 \\
\end{align*}
}{
    t_1 \leq t_3
}\]

(S-subclass) \[
\frac{
\begin{align*}
    & I \Rightarrow c J \\
\end{align*}
}{
    I \leq J
}\]

(S-instance) \[
\frac{
\begin{align*}
    & m \in I J \\
\end{align*}
}{
    m \leq J
}\]

(S-tycon) \[
\frac{
\begin{align*}
    & \overline{s} \leq \overline{t} \\
\end{align*}
}{
    T \overline{s} \leq T \overline{t}
}\]

(S-fun) \[
\frac{
\begin{align*}
    & t_1 \leq s_1 \\
    & s_2 \leq t_2 \\
\end{align*}
}{
    s_1 \rightarrow s_2 \leq t_1 \rightarrow t_2
}\]

(S-qual) \[
\frac{
\begin{align*}
    & s \leq t \\
\end{align*}
}{
    \forall a. Q \Rightarrow s \leq \forall a. Q \Rightarrow t
}\]
Restricted Subtyping

\[ t \leq t \]

\[ \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3} \]

\[ \frac{\overline{s} \leq \overline{t}}{T \overline{s} \leq T \overline{t}} \]

\[ \frac{t_1 \leq s_1 \quad s_2 \leq t_2}{s_1 \rightarrow s_2 \leq t_1 \rightarrow t_2} \]
Restricted Subtyping

\[
\begin{align*}
t &\leq' t \\
t_1 &\leq' t_2 \quad t_2 &\leq' t_3 \\
\implies t_1 &\leq' t_3 \\
\bar{s} &\leq' \bar{t} \\
\implies Ts &\leq' T\bar{t}
\end{align*}
\]

Restricted subtyping vs generic instance

**Lemma**

If \( s \leq' t \) and \( t \leadsto s' \) and \( t \leadsto t' \) then \( \text{true} \vdash s' \leq t' \).
Translation of Types

\[ a \sim' \Box/a \]

\[ T \bar{t} \sim' \text{map} \left( \lambda x. C_i[x] \right) \Box / T \bar{t}' \]

\[ t_1 \sim' \pi_1 \# t'_1 \quad t_2 \sim' C_2/t'_2 \]

\[ t_1 \rightarrow t_2 \sim' \lambda x. C_2[\Box x] / \pi_1(t'_1 \rightarrow t'_2) \]

\[ \text{I} \sim' K_1 \Box / W_1 \]

\[ t \sim' C'/t' \]

\[ \forall a. P \Rightarrow t \sim' C'/\forall a. P \Rightarrow t' \]

\[ a \sim \emptyset \# a \]

\[ T \bar{t} \sim' \pi \# T \bar{t}' \]

\[ t_1 \sim' \pi_1 \# t'_1 \quad t_2 \sim' \pi_2 \# t'_2 \]

\[ t_1 \rightarrow t_2 \sim' \pi_2 \# \pi_1(t'_1 \rightarrow t'_2) \]

\[ \text{I} \sim' \forall c. \text{I} c \Rightarrow \# c \]

\[ \forall a. Q \Rightarrow t \sim' \pi \# \forall a. Q \Rightarrow t' \]
Translation of Terms

\[
\begin{array}{c}
x \mapsto x
\end{array}
\]

\[
\begin{array}{c}
e \mapsto e' \\
\lambda x. e \mapsto \lambda x. e'
\end{array}
\]

\[
\begin{array}{c}
e \mapsto e' \\
\lambda (x :: s). e \mapsto \lambda (x :: s'). e'
\end{array}
\]

\[
\begin{array}{c}
e \mapsto e' \\
\forall c. Q \# s' \\
s \mapsto' C' / s''
\end{array}
\]

\[
\begin{array}{c}
\lambda (x :: s). e \mapsto \land \overline{c}(Q). \lambda (y :: s'). (\lambda (x :: s''). e') (C'[y])
\end{array}
\]

\[
\begin{array}{c}
f \mapsto f' \\
e \mapsto e'
\end{array}
\]

\[
\begin{array}{c}
f e \mapsto f' e'
\end{array}
\]

\[
\begin{array}{c}
e \mapsto e' \\
f \mapsto f'
\end{array}
\]

\[
\begin{array}{c}
\text{let } x = e \text{ in } f \mapsto \text{let } x = e' \text{ in } f'
\end{array}
\]

\[
\begin{array}{c}
e \mapsto e' \\
s \mapsto' C' / s'
\end{array}
\]

\[
\begin{array}{c}
(e :: s) \mapsto (C'[e'] :: s')
\end{array}
\]
Let $P \mid \Gamma' \vdash e' : s'$ be the typing judgment for Haskell with higher-rank qualified polymorphism.

If $P \mid \Gamma \vdash e : s$, $s \sim' s'$, $\Gamma \sim' \Gamma'$, and $e \leftrightarrow e'$, 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? ⇒ 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.