Parametric Polymorphism

Swap the elements of a pair:

\[
\text{swap} \ (a, b) = (b, a):
\]

In MinHs, we have to write a function for each possible type combination:

```haskell
let fun swapIntBool:: (Int,Bool)->(Bool,Int) pair =
    (snd(pair, fst (pair))
end

let fun swapBoolInt:: (Bool,Int)->(Int,Bool) pair =
    (snd(pair, fst (pair))
end
```
Swap the elements of a pair:

\[
\text{swap} \ (a, b) = (b, a):
\]

In MinHs, we have to write a function for each possible type combination:

\[
\text{letfun swapIntBool:: (Int,Bool)->(Bool,Int) pair = (snd(pair, fst (pair)) end}
\]

\[
\text{letfun swapBoolInt:: (Bool,Int)->(Int,Bool) pair = (snd(pair, fst (pair)) end}
\]

What about C, Java, C++?
We extend MinHs to support polymorphic functions:

```haskell
letfun swap :: (a, b) -> (b, a) pair ......
```

Whenever a polymorphic function is applied to a concrete value, the type variables are instantiated:

```haskell
swap (1, True)
```

instantiates type variable `a` to `Int`, `b` to `Bool`. 
We extend MinHs to support **polymorphic functions**:  

```haskell
letfun swap:: (a,b) -> (b,a) pair .......
```

Whenever a polymorphic function is applied to a concrete value, the type variables are instantiated:

```haskell
swap (1, True)
```

instantiate type variable `a` to `Int`, `b` to `Bool`.

We make the type instantiation step explicit, and view polymorphic functions as functions which require a type as argument.
Fun a in
  Fun b in
    letfun swap ::(a,b) -> (b,a) pair =
      (snd (pair), fst (pair))

Instantiating a polymorphic function with a type:

    inst(Fun t in e, τ)

instantiates \( t \) to \( \tau \) everywhere in \( e \)
inst(Fun a in
    inst (Fun b in
        letfun swap::(a,b) -> (b,a) pair =
            (snd (pair), fst (pair)),
        Int), Bool)
evaluates to

    fun swap:: (Bool, Int) -> (Int, Bool) pair = ..........
We write the type of

\[
\begin{align*}
\text{Fun } a \text{ in} \\
\text{Fun } b \text{ in} \\
\text{letfun } \text{swap} :: \ (a,b) \rightarrow (b,a) \text{ pair } = \\
\quad (\text{snd } (\text{pair}), \text{fst } (\text{pair}))
\end{align*}
\]

as

\[
\forall a. \forall b. (a, b) \rightarrow (b, a)
\]
Polymorphic MinHs

Concrete Syntax:

Polytypes \( \sigma \) ::= \( \tau \) | \( \forall t. \sigma \)

Monotypes \( \tau \) ::= \( t \) | \( \ldots \)

Expressions \( e \) ::= Fun \( t \) in \( e \) | inst(\( e, \tau \)) | \( \ldots \)

Values \( v \) ::= Fun \( t \) in \( e \) | \( \ldots \)

Abstract Syntax:

\( \rightarrow \) Fun(\( t, e \))

\( \rightarrow \) inst(\( e, \tau \))
Valid Types

Valid types may not contain any **free** type variables:

- $\forall a. \forall b. a \rightarrow b$ is ok
- $\forall a. a \rightarrow b$ is not ok, since $b$ occurs freely
**Valid Types**

Valid types may not contain any *free* type variables:

- $\forall a. \forall b. a \rightarrow b$ is ok
- $\forall a. a \rightarrow b$ is not ok, since $b$ occurs freely

We need to keep track of the type variables currently bound
Let $\Delta$ be the set of currently defined type variables:

\[
\begin{align*}
\Delta &\vdash \text{Bool ok} \\
\Delta &\vdash \text{Int ok} \\
\Delta &\vdash \tau_1 \text{ ok} \quad \Delta &\vdash \tau_2 \text{ ok} \\
\hline
\end{align*}
\]

\[
\Delta \vdash \forall t.\sigma \text{ ok}
\]

\[
\Delta \vdash t \text{ ok}
\]

All other rules stay the same (apart from the fact that $\Delta$ is part of each judgement).
Let $\Delta$ be the set of currently defined type variables:

$$
\begin{align*}
\Delta \vdash \text{Bool } \text{ok} & \quad \Delta \vdash \text{Int } \text{ok} \\
\Delta \vdash \tau_1 \text{ ok} & \quad \Delta \vdash \tau_2 \text{ ok} \\
\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok} & \\
\Delta \cup \{t\} \vdash \sigma \text{ ok} & \quad t \not\in \Delta \\
\Delta \vdash \forall t. \sigma \text{ ok} \\
\Delta \vdash t \text{ ok}
\end{align*}
$$

All other rules stay the same (apart from the fact that $\Delta$ is part of each judgement).
Let $\Delta$ be the set of currently defined type variables:

$$
\begin{align*}
\Delta \vdash \mathbf{Bool} & \quad \text{ok} \quad \quad \\
\Delta \vdash \mathbf{Int} & \quad \text{ok} \\
\Delta \vdash \tau_1 & \quad \text{ok} \quad \Delta \vdash \tau_2 & \quad \text{ok} \\
\Delta \vdash \tau_1 \rightarrow \tau_2 & \quad \text{ok}
\end{align*}
$$

$$
\begin{align*}
\Delta \cup \{t\} & \vdash \sigma & \quad t \notin \Delta \\
\Delta & \vdash \forall t. \sigma & \quad \text{ok}
\end{align*}
$$

$$
\begin{align*}
t & \in \Delta \\
\Delta & \vdash t & \quad \text{ok}
\end{align*}
$$

All other rules stay the same (apart from the fact that $\Delta$ is part of each judgement).
Typing Rules

Two additional rules:

\[
\Delta \cup \{t\}, \Gamma \vdash e : \sigma \quad t \notin \Delta \\
\Delta, \Gamma \vdash \text{Fun}(t.e) : \forall t.\sigma
\]

\[
\Delta, \Gamma \vdash e : \forall t.\sigma \quad \Delta \vdash \tau \text{ ok} \\
\Delta, \Gamma \vdash \text{inst}(e, \tau) : \{\tau/t\}\sigma
\]
**Dynamic Semantics**

Instantiation of type variables:

\[
\text{inst}(\text{Fun}(t.e), \tau) \rightarrow_\text{M} \{\tau/t\}e
\]

\[
e \rightarrow_\text{M} e'
\]

\[
\text{inst}(e, \tau) \rightarrow_\text{M} \text{inst}(e', \tau)
\]
Progress and Preservation

- **Preservation:** If $e : \sigma$ and $e \xrightarrow{M} e'$, then $e' : \sigma$

- **Progress:** If $e : \sigma$, then $e$ is either a value, or there exists an $e'$ such that $e \xrightarrow{M} e'$.

Both progress and preservation can be proven using induction over the typing rules of polymorphic MinHs.
ARE POLYMORPHIC FUNCTIONS FIRST-CLASS CITIZENS?

→ Are the types $\forall a. (a \to a)$ and $(\forall a. a) \to (\forall a. a)$ the same?
Are the types $\forall a. (a \rightarrow a)$ and $((\forall a \cdot a) \rightarrow (\forall a \cdot a))$ the same?

Can both be expressed in MinHs?
ARE POLYMORPHIC FUNCTIONS FIRST-CLASS CITIZENS?

→ Are the types $\forall a. (a \rightarrow a)$ and $(\forall a. a) \rightarrow (\forall a. a)$ the same?
→ Can both be expressed in MinHs?
→ Is it possible to write a function which returns a polymorphic function?