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
```

What about C, Java, C++?
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
```

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)
```

instantiates 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.
letfun swap :: (a, b) -> (b, a) pair = (snd (pair), fst (pair))

Instantiating a polymorphic function with a type:

```
inst(Fun t in e, \tau)
```

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

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

\[
\text{Fun } a \in \text{Fun } b \in \\
\text{letfun swap:: (a,b) \to (b,a) pair =} \\
(\text{snd (pair)}, \text{fst (pair)})
\]

as

\[
\forall a.\forall b. (a, b) \to (b, a)
\]
Polytypic MinHs

Concrete Syntax:

Polytypes \( \sigma \) ::\( \tau \mid \forall t. \sigma \)
Monotypes \( \tau \) ::\( t \mid \ldots \)
Expressions \( e \) ::\( \text{Fun } t \text{ in } e \mid \text{inst}(e, \tau) \mid \ldots \)
Values \( v \) ::\( \text{Fun } t \text{ in } e \mid \ldots \)

Abstract Syntax:

\( \Rightarrow \text{Fun}(t.e) \)
\( \Rightarrow \text{inst}(e, \tau) \)
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:

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

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

\[
\Delta \vdash t \quad \text{ok}
\]
Let $\Delta$ be the set of currently defined type variables:

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

\[
\begin{align*}
\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2 & \quad \Delta \vdash \tau_1 \rightarrow \tau_2 \\
\Delta \vdash \forall t. \sigma & \quad t \notin \Delta \\
\Delta \vdash \sigma & \\
\Delta \vdash t & \quad \Delta \vdash t \quad \Delta \vdash \alpha \quad \Delta \vdash \beta
\end{align*}
\]

VAlID TYPES
Let $\Delta$ be the set of currently defined type variables:

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

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

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

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

Two additional rules:

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

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

Instantiation of type variables:

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

\[
e \xrightarrow{M} e'
\]

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

- **Preservation:** If $e : \sigma$ and $e \rightarrow^* \_ e'$, then $e' : \sigma$

- **Progress:** If $e : \sigma$, then $e$ is either a value, or there exists a $e'$ such that $e \rightarrow^* \_ e'$.

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 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?
ARE POLYMORPHIC FUNCTIONS FIRST-CLASS CITIZENS?

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

Can both be expressed in MinHs?

Is it possible to write a function which returns a polymorphic function?