Swap the elements of a pair:

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

In Mini-Hs, we have to write a function for each possible type combination:

\[
\begin{align*}
\text{let fun swapIntBool:: (Int,Bool)\rightarrow(Bool,Int) } & \text{ pair } = \\
& (\text{snd}(\text{pair}), \text{fst}(\text{pair})) \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{let fun swapBoolInt:: (Bool,Int)\rightarrow(Int,Bool) } & \text{ pair } = \\
& (\text{snd}(\text{pair}), \text{fst}(\text{pair})) \\
\text{end}
\end{align*}
\]

What about C, Java, C++?

We extend Mini-Hs to support polymorphic functions:

\[
\begin{align*}
\text{let fun swap:: (a,b)\rightarrow (b,a) pair } & \text{ ......}
\end{align*}
\]

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

\[
\begin{align*}
\text{swap } (1, \text{True}) \\
\text{instantiates type variable a to Int, b to Bool.}
\end{align*}
\]

We make the type instantiation step explicit, and view polymorphic functions as functions which require a type as argument.
We write the type of

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

as

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

POLYMORPHIC MINHS

Concrete Syntax:

- Polytypes: \( \sigma ::= \tau \mid \forall \tau.\sigma \)
- Monotypes: \( \tau ::= t \mid \ldots \)

Expressions: \( e ::= \text{Fun}\ t\ in\ e \mid \text{inst}(e,\tau) \mid \ldots \)

Values: \( v ::= \text{Fun}\ t\ in\ e \mid \ldots \)

Abstract Syntax:

- \( \text{Fun}(t, e) \)
- \( \text{inst}(e,\tau) \)

VALID TYPES

Valid types may not contain any free type variables:

Slide 7

\[ \forall a.\forall b. a \rightarrow b \text{ is ok} \]
\[ \forall a. a \rightarrow b \text{ is not ok, since } b \text{ occurs freely} \]

We need to keep track of the type variables currently bound

\[ \Delta \]

Let \( \Delta \) be the set of currently defined type variables:

Slide 8

\[
\begin{align*}
\Delta \vdash \text{Bool} & \quad \Delta \vdash \text{Int} & \quad \Delta \vdash \tau_1 & \quad \Delta \vdash \tau_2 \\
\Delta \vdash \forall \tau.\sigma & \quad \Delta \vdash \tau_1 \rightarrow \tau_2 \\
\Delta \cup \{t\} \vdash \sigma & \quad t \notin \Delta \\
\Delta \vdash \forall \tau.\sigma & \quad t \in \Delta
\end{align*}
\]

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

Two additional rules:

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

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

Dynamic Semantics

Instantiation of type variables:

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

\[ e \rightarrow_{\mu} e' \]
\[ \frac{}{\text{inst}(e, \tau) \rightarrow_{\mu} \text{inst}(e', \tau)} \]

Progress and Preservation

- Preservation: If \( e : \sigma \) and \( e \rightarrow_{\mu} e' \), then \( e' : \sigma \)
- Progress: If \( e : \sigma \), then \( e \) is either a value, or there exists a \( e' \) such that \( e \rightarrow_{\mu} e' \).

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 \( \forall a . (a \rightarrow a) \) 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?