If we want to define a MinHs function which takes a pair and returns a new pair with the same elements but in reverse order, we would have to define a new function of each possible combination of argument types:

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

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

Since the code is exactly the same for each function, and we have an infinite number of possible combinations, this is clearly unsatisfactory. We therefore define a new variant of MinHs, \textit{Polymorphic MinHs}, which allows us to write a single function for all possible type combinations.

Note that this is different from the case where, for example, arithmetic operations like addition are applied to a number of different types. In most languages, + represents addition of integer values as well as floating point values. However, it gets translated into two different operations: since the machine representation of the two types is very different, the actual operations are different, and we just \textit{overload} the symbol + to represent both in the source language. Although overloading is also often referred to as polymorphism, we will only use it when the operation is exactly the same, independent of the argument type, as in the \textit{swap} example. We will take about overloading later in the course.

1 Polymorphic MinHs

In Polymorphic MinHs we can implement a \textit{swap} function which has type \( \forall a. \forall b. (a, b) \rightarrow (b, a) \). If this function is applied to a value, its formal parameter is instantiated to this value. Moreover, the type variables \( a \) and \( b \) are instantiated to the type of the first and second component of the value. So, polymorphic functions are not only parametrized over the values, but also over their concrete type. In MinHs, in contrast to “real” polymorphic languages, we make the parametrisation and instantiation explicit. We introduce to new language constructs:

- \textit{Fun a in e} binds the type variable \( a \) in the expression \( e \) (abstract syntax: \texttt{Fun (a,t)})
- \textit{inst(e, t)} instantiates the (polymorphic) type of \( e \) to the type \( t \) — to be well defined, \( e \) has to evaluate to a term of the form \texttt{Fun a in e'} (abstract syntax: \texttt{inst(e, t)})

Now, we can use the notation to define a polymorphic version of the function \textit{swap}:

\[
\text{Fun a in} \\
\quad \text{Fun b in} \\
\quad \text{letfun swap ::(a,b) -> (b,a) pair =} \\
\quad \quad \text{(snd (pair), fst (pair))}
\]
Before applying this definition to a concrete value, we apply it to the types:

```
inst(Fun a in
    inst (Fun b in
        letfun swap::(a,b) -> (b,a) pair =
            (snd (pair), fst (pair)),
        Int), Bool)
```

This is, of course, pretty awkward, but keep in mind that MinHs is not designed to be a real life language, but to help us study certain language features. Note that many compiler for existing polymorphic languages internally produce code similar to Polymorphic MinHs, making type instantiation and abstraction explicit.

### 1.1 Well-formed Types

A well-formed type in Polymorphic MinHs may contain type variables, as long as they are bound. To check the well-formedness of a type, we therefore need some means in the inference rules which keeps track of the set of variables bound in the current scope.

Let $\Delta$ be the set of currently defined type variables, a type $t$ is well defined, if $\Delta \vdash t \text{ ok}$ is derivable using the following inference rules:

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

The other inference rules stay the same apart from the fact the $\Delta$ has to be added to each judgement.

### 1.2 Static Semantics

The static semantics of Polymorphic MinHs is the same as that of regular MinHs. We only have to add two new rules, one for each of the newly introduced constructs.

\[
\begin{align*}
& \Delta \cup \{t\}, \Gamma \vdash e : \sigma & t \not\in \Delta \\
& \Delta, \Gamma \vdash \text{Fun}(t,e) : \forall \forall. \sigma
\end{align*}
\]

\[
\begin{align*}
& \Delta, \Gamma \vdash e : \forall \tau. \sigma & \Delta \vdash \tau \text{ ok} \\
& \Delta, \Gamma \vdash \text{inst}(e, \tau) : \{\tau/t\} \sigma
\end{align*}
\]

Note the similarities of the dynamic semantics rule for application and the static semantics rule of type instantiation.

### 1.3 Dynamic Semantics

Since the extension mainly concerns the type system, there is nothing much to do here. Evaluating the instantiation of a type abstraction works as described in the following two rules:

\[
\begin{align*}
& \text{inst}(\text{Fun}(t,e), \tau) \equiv_{\text{M}} \{\tau/t\} e \\
& e \equiv_{\text{M}} e' \\
& \text{inst}(e, \tau) \equiv_{\text{M}} \text{inst}(e', \tau)
\end{align*}
\]
1.4 Are Polymorphic Functions First-Class Citizens?

Note that although functions are first class citizens, meaning that they can occur anywhere in the program where other types of values can occur, polymorphic functions are not. For example, it is not possible to define a terminating function with the following type

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

which, given a polymorphic function, will return a polymorphic function. Such a function is different from one with type

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

which, given a function of any type \(a, \ a \rightarrow a\) (for example a function of type \(Int \rightarrow Int\), will return a function of type \(a \rightarrow a\).