Polymorphically-Typed FUN

1 PolyFUN Syntax

\[ \begin{align*}
 x \in \text{Variable} & \quad n \in \mathbb{N} & \quad b \in \text{Bool} & \quad \text{name, cons, fld} \in \text{Label} \\
\end{align*} \]

\[ \begin{align*}
 \text{prog} \in \text{Program} & \quad ::= \quad \text{typedef}_1 \ldots \text{typedef}_n \ldots \text{e} \\
\text{typedef} \in \text{TypeDef} & \quad ::= \quad \text{type} \text{name}[T_1 \ldots T_k] = \text{cons}_1 : \tau_1 \ldots \text{cons}_n : \tau_n \\
\text{e} \in \text{Exp} & \quad ::= \quad x \mid n \mid b \mid \text{nil} \mid (x_1 : \tau_1 \ldots x_n : \tau_n) \Rightarrow e \mid e(f_1 \ldots e_n) \\
& \quad \mid \text{if} \ e_1 \ e_2 \ e_3 \mid \text{let} \ x = e_1 \ \text{in} \ e_2 \mid \text{rec} \ x : \tau = e_1 \ \text{in} \ e_2 \mid [\text{fld}_1 = e_1 \ldots \text{fld}_n = e_n] \\
& \quad \mid e.\text{fld} \mid \text{cons}(\tau_1 \ldots \tau_k) \ e \mid \text{case} \ e \ \text{of} \ \text{cons}_1 \ x_1 \Rightarrow e_1 \ldots \text{cons}_n \ x_n \Rightarrow e_n \\
& \quad \mid [T_1 \ldots T_k] \Rightarrow e \mid e(\tau_1 \ldots \tau_k) \\
\end{align*} \]

Compared to the SimpleFUN language in handout 4, we have performed the following changes to get the PolyFUN language above:

- Add type polymorphism to variants, which now act like generics. User-defined variant types now include declarations of type variables which can be used in the constructor types: \text{type} \text{name}[T_1 \ldots T_k] = \text{cons}_1 : \tau_1 \ldots \text{cons}_n : \tau_n instead of just \text{type} \text{name} = \text{cons}_1 : \tau_1 \ldots \text{cons}_n : \tau_n, where types \tau_1 \ldots \tau_n can now use the type variables \( T_1 \ldots T_k \). Because of this polymorphism, when we construct a variant we need to pass in type arguments to replace the type variables, i.e., \text{cons}(\tau_1 \ldots \tau_k) \ e instead of just \text{cons} e.

- Add type abstraction and type application to get parametric polymorphism. Type abstraction creates a function whose parameters are type variables (i.e., \([T_1 \ldots T_k] \Rightarrow e\)), and type application calls a type abstraction like a function but passes in types to replace the type variables (i.e., \(e(\tau_1 \ldots \tau_k)\)).

2 PolyFUN Type System

The PolyFUN types are similar to SimpleFUN types with a few changes:

\[ \tau \in \text{Type} = \text{num} \mid \text{bool} \mid \text{unit} \mid (\tau_1 \ldots \tau_n) \rightarrow \tau \mid [\text{fld}_1 : \tau_1 \ldots \text{fld}_n : \tau_n] \mid \text{name}\langle \tau_1 \ldots \tau_k \rangle \mid T \mid [T_1 \ldots T_k] \rightarrow \tau \]

The first five types haven’t changed; the last three are different:

- User-defined variant names are now type constructors rather than types themselves. In other words, \text{name} by itself is not a type—it is a type constructor that takes types as arguments and returns a type as a result: \text{name}\langle \tau_1 \ldots \tau_k \rangle.

- We now have type variables. These variables are introduced by the type abstractions ([\( T_1 \ldots T_k \Rightarrow e \)]) and by the variant type declarations (\text{type} \text{name}[T_1 \ldots T_k] = \text{cons}_1 : \tau_1 \ldots \text{cons}_n : \tau_n).

- Finally, type abstractions yield a polymorphic type, i.e., a type where the type variables can be replaced with any given type to yield a new type.

The type rules for PolyFUN are exactly like the type rules for SimpleFUN except (1) changes to the τI and τE rules to account for polymorphic variants (recall that the notation ε[x \mapsto y] means to create a copy of ε where every instance of x has been replaced by y):

\[ \begin{align*}
\text{type} \text{name}\langle T_1 \ldots T_k \rangle = \ldots \text{cons} : \tau \ldots \in \text{TypeDef} \quad \Gamma \vdash e : \tau \langle T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k \rangle \\
\Gamma \vdash \text{cons}(\tau_1 \ldots \tau_k) \ e : \text{name}(\tau_1 \ldots \tau_k) & \quad (\text{τD}) \\
\end{align*} \]
\[ \Gamma \vdash e : \text{name}(\tau_1 \ldots \tau_k) \quad \text{type name}[T_1 \ldots T_k] = \text{cons}_1 : \tau_k+1 \ldots \text{cons}_n : \tau_k+n \in \text{TypeDef} \]

\[
\begin{align*}
\Gamma &\vdash x_1 : \tau_{k+1}[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k] \vdash e_1 : \tau \ldots \Gamma, x_n : \tau_{k+n}[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k] &\vdash e_n : \tau \\
\Gamma \vdash \text{case } e \text{ of } \text{cons}_1 x_1 \Rightarrow e_1 \ldots \text{cons}_n x_n \Rightarrow e_n : \tau
\end{align*}
\]

(\text{TD}E)

And the addition of \text{TABS} and \text{TAPP} rules to account for parametric polymorphism:

\[
\begin{align*}
\Gamma &\vdash e : \tau \quad \Gamma \vdash e : [T_1 \ldots T_k] \Rightarrow e : [T_1 \ldots T_k] \rightarrow \tau \quad \text{(TABS)} \\
\Gamma &\vdash e : [T_1 \ldots T_k] \rightarrow \tau \\
\Gamma &\vdash e(\tau_1 \ldots \tau_k) : \tau[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k] \quad \text{(TAPP)}
\end{align*}
\]