Polymorphically-Typed FUN

1 PolyFUN Syntax

$$x \in \mathit{Variable} \qquad n \in \mathbb{N} \qquad b \in \mathit{Bool} \qquad \mathit{name}, \mathit{cons}, \mathit{fld} \in \mathit{Label}$$

$$\begin{aligned}
\mathit{prog} \in \mathit{Program} &::= \mathit{typedef}_1 \dots \mathit{typedef}_n \;\; e \\
\mathit{typedef} \in \mathit{TypeDef} &::= \textbf{type } \mathit{name}[T_1 \dots T_k] = \mathit{cons}_1 : \tau_1 \dots \mathit{cons}_n : \tau_n \\
e \in \mathit{Exp} &::= x \mid n \mid b \mid \textbf{nil} \mid (x_1 : \tau_1 \dots x_n : \tau_n) \Rightarrow e \mid e_f(e_1 \dots e_n) \\
&\mid \textbf{if } e_1 \; e_2 \; e_3 \mid \textbf{let } x = e_1 \textbf{ in } e_2 \mid \textbf{rec } x : \tau = e_1 \textbf{ in } e_2 \mid [\mathit{fld}_1 = e_1 \dots \mathit{fld}_n = e_n] \\
&\mid e.\mathit{fld} \mid \mathit{cons}\langle \tau_1 \dots \tau_k \rangle \; e \mid \textbf{case } e \textbf{ of } \mathit{cons}_1 \; x_1 \Rightarrow e_1 \dots \mathit{cons}_n \; x_n \Rightarrow e_n \\
&\mid [T_1 \dots T_k] \Rightarrow e \mid e\langle \tau_1 \dots \tau_k \rangle
\end{aligned}$$

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: $\textbf{type } \mathit{name}[T_1 \dots T_k] = \mathit{cons}_1 : \tau_1 \dots \mathit{cons}_n : \tau_n$ instead of just $\textbf{type } \mathit{name} = \mathit{cons}_1 : \tau_1 \dots \mathit{cons}_n : \tau_n$, where types $\tau_1 \dots \tau_n$ can now use the type variables $T_1 \dots T_k$. Because of this polymorphism, when we construct a variant we need to pass in type arguments to replace the type variables, i.e., $\mathit{cons}\langle \tau_1 \dots \tau_k \rangle \; e$ instead of just $\mathit{cons} \; e$.
- Add type abstraction and type application to get *parametric polymorphism*. Type abstraction creates a function whose parameters are type variables (i.e., $[T_1 \dots T_k] \Rightarrow e$), and type application calls a type abstraction like a function but passes in types to replace the type variables (i.e., $e\langle \tau_1 \dots \tau_k \rangle$).

2 PolyFUN Type System

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

$$\tau \in \mathit{Type} = \textbf{num} \mid \textbf{bool} \mid \textbf{unit} \mid (\tau_1 \dots \tau_n) \to \tau_r \mid [\mathit{fld}_1 : \tau_1 \dots \mathit{fld}_n : \tau_n] \mid \mathit{name}\langle \tau_1 \dots \tau_k \rangle \mid T \mid [T_1 \dots T_k] \to \tau$$

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, *name* by itself is not a type—it is a type constructor that takes types as arguments and returns a type as a result: $\mathit{name}\langle \tau_1 \dots \tau_k \rangle$.
- We now have type variables. These variables are introduced by the type abstractions ($[T_1 \dots T_k] \Rightarrow e$) and by the variant type declarations ($\textbf{type } \mathit{name}[T_1 \dots T_k] = \mathit{cons}_1 : \tau_1 \dots \mathit{cons}_n : \tau_n$).
- 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 TDI and TDE rules to account for polymorphic variants (recall that the notation $z[x \mapsto y]$ means to create a copy of z where every instance of x has been replaced by y):

$$\frac{\textbf{type } \mathit{name}[T_1 \dots T_k] = \; \dots \mathit{cons} : \tau \dots \; \in \mathit{TypeDef} \qquad \Gamma \vdash e : \tau[T_1 \mapsto \tau_1 \dots T_k \mapsto \tau_k]}{\Gamma \vdash \mathit{cons}\langle \tau_1 \dots \tau_k \rangle \; e : \mathit{name}\langle \tau_1 \dots \tau_k \rangle} \; (\textsc{td}I)$$

$$\frac{\Gamma \vdash e : name\langle \tau_1 \ldots \tau_k \rangle \qquad \textbf{type}\ name[T_1 \ldots T_k] = cons_1 : \tau_{k+1} \ldots cons_n : \tau_{k+n} \in TypeDef \qquad \Gamma, x_1 : \tau_{k+1}[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k] \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_{k+n}[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k] \vdash e_n : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ cons_1\ x_1 \Rightarrow e_1 \ldots cons_n\ x_n \Rightarrow e_n : \tau}\ (\textsc{td}E)$$

And the addition of TABS and TAPP rules to account for parametric polymorphism:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [T_1 \ldots T_k] \Rightarrow e : [T_1 \ldots T_k] \rightarrow \tau}\ (\textsc{tabs}) \qquad \frac{\Gamma \vdash e : [T_1 \ldots T_k] \rightarrow \tau}{\Gamma \vdash e\langle \tau_1 \ldots \tau_k \rangle : \tau[T_1 \mapsto \tau_1 \ldots T_k \mapsto \tau_k]}\ (\textsc{tapp})$$