Polymorphically-Typed FUN

1 PolyFUN Syntax

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

$$\begin{aligned}
\mathit{prog} \in \mathit{Program} &::= \overrightarrow{\mathit{typedef}_i}\ e \\
\mathit{typedef} \in \mathit{TypeDef} &::= \textsf{\textbf{type}}\ \mathit{name}[\vec{T_j}] = \overrightarrow{\mathit{cons}_i : \tau_i} \\
e \in \mathit{Exp} &::= x \mid n \mid b \mid \textsf{\textbf{nil}} \mid (\overrightarrow{x_i\!:\!\tau_i}) \Rightarrow e \mid e_f(\vec{e_i}) \\
&\quad\mid \textsf{\textbf{if}}\ e_1\ e_2\ e_3 \mid \textsf{\textbf{let}}\ x = e_1\ \textsf{\textbf{in}}\ e_2 \mid \textsf{\textbf{rec}}\ x\!:\!\tau = e_1\ \textsf{\textbf{in}}\ e_2 \\
&\quad\mid (\!|\overrightarrow{\mathit{fld}_i : e_i}|\!) \mid e.\mathit{fld} \mid \mathit{name}!\mathit{cons}\langle\vec{\tau_i}\rangle\ e \mid \textsf{\textbf{case}}\ e\ \textsf{\textbf{of}}\ \overrightarrow{\mathit{cons}_i\ x_i \Rightarrow e_i} \\
&\quad\mid [\vec{T_i}]e \mid e\langle\vec{\tau_i}\rangle
\end{aligned}$$

The differences from SimpleFUN syntax are: (1) user-defined variant types now include declarations of type variables that can be used in the type definition (the $[\vec{T_j}]$ in **type** $\mathit{name}[\vec{T_j}]$); (2) when we construct a variant we need to pass it the types to fill in for that variant's type variables (the $\langle\vec{\tau_i}\rangle$ in $\mathit{name}!\mathit{cons}\langle\vec{\tau_i}\rangle\ e$); (3) we can create a type abstraction using $[\vec{T_i}]e$; and (4) we can have a type application $e\langle\vec{\tau_i}\rangle$ where e should evaluate to a type abstraction.

2 PolyFUN Type System

$$\tau \in \mathit{Type} = \textsf{\textbf{num}} \mid \textsf{\textbf{bool}} \mid \textsf{\textbf{unit}} \mid (\vec{\tau_i}) \to \tau_r \mid (\!|\overrightarrow{\mathit{fld}_i : \tau_i}|\!) \mid \mathit{name}\langle\vec{\tau_i}\rangle \mid T \mid [\vec{T_j}]\tau$$

The difference from SimpleFUN types are: (1) a user-defined variant type name must now include a list of types to substitute for that variant's type variables (the $\langle\vec{\tau_i}\rangle$ in $\mathit{name}\langle\vec{\tau_i}\rangle$); types can now include type variables (symbolized here with T); and (3) we now have polymorphic types, which are a list of type variables along with a type that uses those type variables.

$$\frac{}{\Gamma, x\!:\!\tau \vdash x : \tau}\ \textsc{var} \qquad \frac{}{\Gamma \vdash n : \textsf{\textbf{num}}}\ n\mathrm{I} \qquad \frac{}{\Gamma \vdash b : \textsf{\textbf{bool}}}\ b\mathrm{I} \qquad \frac{}{\Gamma \vdash \textsf{\textbf{nil}} : \textsf{\textbf{unit}}}\ \textsf{\textbf{nil}}\mathrm{I}$$

$$\frac{\Gamma, \overrightarrow{x_i\!:\!\tau_i} \vdash e : \tau_r}{\Gamma \vdash (\overrightarrow{x_i\!:\!\tau_i}) \Rightarrow e : (\vec{\tau_i}) \to \tau_r}\ {\to}\mathrm{I} \qquad \frac{\Gamma \vdash e_f : (\vec{\tau_i}) \to \tau_r \quad \Gamma \vdash \overrightarrow{e_i : \tau_i}}{\Gamma \vdash e_f(\vec{e_i}) : \tau_r}\ {\to}\mathrm{E} \qquad \frac{\Gamma \vdash e_1 : \textsf{\textbf{bool}} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{\textbf{if}}\ e_1\ e_2\ e_3 : \tau}\ \textsc{if}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x\!:\!\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{\textbf{let}}\ x = e_1\ \textsf{\textbf{in}}\ e_2 : \tau_2}\ \textsc{let} \qquad \frac{\Gamma, x\!:\!\tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x\!:\!\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{\textbf{rec}}\ x\!:\!\tau_1 = e_1\ \textsf{\textbf{in}}\ e_2 : \tau_2}\ \textsc{rec}$$

$$\frac{\Gamma \vdash \overrightarrow{e_i : \tau_i}}{\Gamma \vdash (\!|\overrightarrow{\mathit{fld}_i : e_i}|\!) : (\!|\overrightarrow{\mathit{fld}_i : \tau_i}|\!)}\ \textsc{rcd}I \qquad \frac{\Gamma \vdash e : (\!|\overrightarrow{\mathit{fld}_i : \tau_i}|\!) \quad j \in \vec{i}}{\Gamma \vdash e.\mathit{fld}_j : \tau_j}\ \textsc{rcd}E$$

$$\frac{\textsf{\textbf{type}}\ \mathit{name}[\vec{T_j}] = \overrightarrow{\mathit{cons}_i : \tau_i} \in \mathit{TypeDef} \quad k \in \vec{i} \quad \Gamma \vdash e : \tau_k\Big[\overrightarrow{T_j \mapsto \tau_j}\Big]}{\Gamma \vdash \mathit{name}!\mathit{cons}_k\langle\vec{\tau_j}\rangle\ e : \mathit{name}\langle\vec{\tau_j}\rangle}\ \textsc{td}I \qquad \frac{\begin{array}{c}\Gamma \vdash e : \mathit{name}\langle\vec{\tau_j}\rangle \quad \textsf{\textbf{type}}\ \mathit{name}[\vec{T_j}] = \overrightarrow{\mathit{cons}_i : \tau_i} \in \mathit{TypeDef} \\ \overrightarrow{\Gamma, x_i\!:\!\tau_i\Big[\overrightarrow{T_j \mapsto \tau_j}\Big] \vdash e_i : \tau}\end{array}}{\Gamma \vdash \textsf{\textbf{case}}\ e\ \textsf{\textbf{of}}\ \overrightarrow{\mathit{cons}_i\ x_i \Rightarrow e_i} : \tau}\ \textsc{td}E$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [\vec{T_j}]e : [\vec{T_j}]\tau}\ \textsc{tabs} \qquad\qquad \frac{\Gamma \vdash e : [\vec{T_j}]\tau}{\Gamma \vdash e\langle\vec{\tau_j}\rangle : \tau\left[\overrightarrow{T_j \mapsto \tau_j}\right]}\ \textsc{tapp}$$

The differences from SimpleFUN are in the last four type rules: $\textsc{td}I$, $\textsc{td}E$, $\textsc{tabs}$, and $\textsc{tapp}$. The first two are modifications of the corresponding rules in SimpleFUN to handle polymorphic type operators. The last two are new to PolyFUN to handle generic types. These rules use new notation that looks like this: $\tau\left[\overrightarrow{T_j \mapsto \tau_j}\right]$ (where τ can be any type). This notation means to take τ and replace any instance of a type variable in $\vec{T_j}$ with the corresponding type in $\vec{\tau}_j$. In the type checker code, this functionality is implemented by the `replace` method. This notation is a little awkward in the $\textsc{td}E$ rule; essentially it's saying to do exactly the same thing as the Simple FUN $\textsc{td}E$ rule, except instead of mapping x_i to τ_i we map x_i to a version of τ_i where all of the given type variables have been replaced.