# Simply-Typed FUN

## 1 SimpleFUN Syntax

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

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

Compared to the extended version of the simply-typed lambda calculus from handout 3, we have performed the following changes to get the SimpleFUN language above:

- Add a primitive value **nil**, distinct from any integer or boolean value.
- Allow functions to have an arbitrary number of parameters $(x_1 : \tau_1 \ldots x_n : \tau_n) \Rightarrow e$, and hence also allow function applications to have an arbitrary number of arguments $e_f(e_1 \ldots e_n)$. The type system will ensure that the number of arguments always matches the number of parameters.
- Add the conditional **if** expression, the variable definition **let** expression, and the recursive variable definition **rec** expression.
- Generalize the *pair* data structure $(e_1, e_2)$, which contains exactly two elements, to a *record* data structure $[\mathit{fld}_1 = e_1 \ldots \mathit{fld}_n = e_n]$ that contains an arbitrary number of elements. The elements of the record are given user-defined names $\mathit{fld}_1 \ldots \mathit{fld}_n$ to access them using dot notation (i.e., $e.\mathit{fld}$) rather than being accessed using built-in operators **fst**$(e)$ and **snd**$(e)$.
- Generalize the *union* data structure, with two built-in constructors **inl** and **inr**, to a *variant* data structure that has an arbitrary number of user-defined constructors. We use the **type** declaration to enable programmers to define these variant data structures and what constructors they allow (along with the type of value each constructor contains). We use the abstract syntax *cons e* to call a constructor with a particular value and hence create a variant. We extend the **case** expression to pattern-match against the user-defined constructors. **Note:** we assume for convenience that every constructor name is unique among all constructor names used in **type** declarations.

It is important to remember that these additions are merely conveniences—they do not add any expressive power to the language itself. There are no arithmetic, relational, or logical operators given in the syntax; for simplicity we'll assume that they are all given as built-in functions with the appropriate types: $\{+, -, \times, \div\} : (\textbf{num}, \textbf{num}) \to \textbf{num}$; $\{<, =\} : (\textbf{num}, \textbf{num}) \to \textbf{bool}$; $\{\wedge, \vee\} : (\textbf{bool}, \textbf{bool}) \to \textbf{bool}$; and $\neg : \textbf{bool} \to \textbf{bool}$.

## 2 SimpleFUN Type System

We extend the type system from handout 3 to handle the new language as described above.

$$\tau \in \mathit{Type} = \textbf{num} \mid \textbf{bool} \mid \textbf{unit} \mid (\tau_1 \ldots \tau_n) \to \tau_r \mid [\mathit{fld}_1 : \tau_1 \ldots \mathit{fld}_n : \tau_n] \mid \mathit{name}$$

A type is **num** for numbers; **bool** for booleans; **unit** for the single unit value **nil**; an arrow type for a function with an arbitrary number of parameters; a record type with the set of field names and their associated types; or a user-defined variant type name. The primitive types **num** and **bool** are the same as before. We've added the primitive type **unit** to account for the new primitive value **nil**. We've modified the arrow type $\tau_1 \to \tau_2$ from handout 3 to allow an arbitrary number of parameters $(\tau_1 \ldots \tau_n) \to \tau_r$. The record type $[\mathit{fld}_1 : \tau_1 \ldots \mathit{fld}_n : \tau_n]$ takes the place of the pair type $\tau_1 \times \tau_2$ from handout 3. The user-defined variant type *name* takes the place of the union type $\tau_1 + \tau_2$ from handout 3.

## 2.1 Nominal vs Structural Types

A *nominal* type is one where type equality is determined by the name of the type (hence 'nominal'). A *structural* type is one where type equality is determined by the structure of the type, i.e., its consituent parts. Consider the following example in C:

```
typedef struct { int a; int b; } foo;
typedef struct { int x; int y; } bar;

foo foo_var;
bar bar_var;
```

The question is, do `foo_var` and `bar_var` have the same type? If we declared a function that had a `foo` as a parameter, could we pass it `bar_var` instead of `foo_var`? One could argue that the types are structurally identical (being structs with two integer fields) and hence it makes sense to treat them the same. On the other hand, one could argue that the types have been given different names and hence should be treated as semantically different from each other. The former view would mean they are structural types, the latter view would mean they are nominal types.

There are various tradeoffs between these opposing views in terms of safety, expressiveness, and complexity. To gain familiarity with both views, in SimpleFUN we employ both nominal types (for variants) and structural types (for records). These choices are made manifest in the type system rules given below.

## 2.2 The SimpleFUN Type Rules

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash n : \textbf{num}}\ (n\text{I}) \qquad \frac{}{\Gamma \vdash b : \textbf{bool}}\ (b\text{I}) \qquad \frac{}{\Gamma \vdash \textbf{nil} : \textbf{unit}}\ (\textbf{nil}\text{I})$$

$$\frac{\Gamma, x_1:\tau_1 \ldots x_n:\tau_n \vdash e : \tau_r}{\Gamma \vdash (x_1:\tau_1 \ldots x_n:\tau_n) \Rightarrow e : (\tau_1 \ldots \tau_n) \to \tau_r}\ (\to\text{I}) \qquad \frac{\Gamma \vdash e_f : (\tau_1 \ldots \tau_n) \to \tau_r \quad \Gamma \vdash e_1:\tau_1 \ \ldots\ \Gamma \vdash e_n:\tau_n}{\Gamma \vdash e_f(e_1 \ldots e_n) : \tau_r}\ (\to\text{E})$$

$$\frac{\Gamma \vdash e_1:\textbf{bool} \quad \Gamma \vdash e_2:\tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ e_2\ e_3 : \tau}\ (\textsc{if}) \qquad \frac{\Gamma \vdash e_1:\tau_1 \quad \Gamma, x:\tau_1 \vdash e_2:\tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ (\textsc{let}) \qquad \frac{\Gamma, x:\tau_1 \vdash e_1:\tau_1 \quad \Gamma, x:\tau_1 \vdash e_2:\tau_2}{\Gamma \vdash \textbf{rec}\ x:\tau_1 = e_1\ \textbf{in}\ e_2 : \tau_2}\ (\textsc{rec})$$

$$\frac{\Gamma \vdash e_1:\tau_1 \ \ldots\ \Gamma \vdash e_n:\tau_n}{\Gamma \vdash [fld_1 = e_1 \ldots fld_n = e_n] : [fld_1:\tau_1 \ldots fld_n:\tau_n]}\ (\textsc{rcd}I) \qquad \frac{\Gamma \vdash e : [\ldots fld:\tau \ldots]}{\Gamma \vdash e.fld : \tau}\ (\textsc{rcd}E)$$

$$\frac{\textbf{type}\ name = \ldots cons:\tau \ldots \in TypeDef \qquad \Gamma \vdash e:\tau}{\Gamma \vdash cons\ e : name}\ (\textsc{td}I)$$

$$\frac{\Gamma \vdash e : name \qquad \textbf{type}\ name = cons_1:\tau_1 \ldots cons_n:\tau_n \in TypeDef \qquad \Gamma, x_1:\tau_1 \vdash e_1:\tau\ \ldots\ \Gamma, x_n:\tau_n \vdash e_n:\tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ cons_1\ x_1 \Rightarrow e_1 \ldots cons_n\ x_n \Rightarrow e_n : \tau}\ (\textsc{td}E)$$