Elements of Programming Languages
Tutorial 3: Recursion and data structures
Week 5 (October 16–20, 2023)

Exercises marked ⋆ are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. Pairs, variants, and polymorphism in Scala

Scala provides a form of abstraction over types called polymorphism (aka generics in Java). We will discuss polymorphism in more detail in a later lecture, but for the purpose of this tutorial we will just explain that a Scala function definition can take type parameters such as A, B, C which give names to types that can be used in the function definition, and which can be instantiated to different actual types when the function is called. So for example in the following function

```scala
def f[A,B,C](x: T1, ..., xn: Tn): T = {...}
```

the parameters A, B, C can be used in the type annotations T1, ..., Tn, T as well as possibly inside the function definition.

Scala includes built-in pair types (T1,T2), with pairing written (e1,e2) and projection written e._1, e._2. Likewise, Scala’s library includes binary sums Either[T1,T2] with constructors Left(,) and Right(,). Pattern matching can be used to analyze Either[T1,T2]. Using these operations, write Scala functions having the following types, polymorphic in A, B, C:

(a) (A,B) => (B,A)
(b) Either[A,B] => Either[B,A]
(c) ((A,B) => C) => (A => (B => C))
(d) (A => (B => C)) => ((A,B) => C)
(e) (Either[A,B] => C) => (A => C, B => C)
(f) (A => C, B => C) => (Either[A,B] => C)

2. Typing derivations

Construct typing derivations for the following expressions, or argue why they are not well-formed:

(a) \( \lambda x : \text{int} + \text{bool}. \text{case } x \text{ of } \{ \text{left}(y) \Rightarrow y == 0; \text{right}(z) \Rightarrow z \} \)
(b) (*) \( \lambda x : \text{int} \times \text{int}. \text{if } \text{fst } x == \text{snd } x \text{ then left}(\text{fst } x) \text{ else right}(\text{snd } x) \)
3. Lists

We could add built-in lists to \( L_{\text{Data}} \) as follows:

\[
\begin{align*}
\mathit{e} & ::= \cdots \mid \mathit{nil} \mid \mathit{e}_1 :: \mathit{e}_2 \mid \text{\texttt{case1list} } \mathit{e} \text{ of } \{ \mathit{nil} \Rightarrow \mathit{e}_1 ; \mathit{x} :: \mathit{y} \Rightarrow \mathit{e}_2 \} \\
\mathit{v} & ::= \cdots \mid \mathit{nil} \mid \mathit{v}_1 :: \mathit{v}_2 \\
\tau & ::= \cdots \mid \mathit{list} [\tau]
\end{align*}
\]

Define \( L_{\text{List}} \) to be \( L_{\text{Data}} \) extended with the above constructs.

The typing rule for \texttt{case1list} is:

\[
\Gamma \vdash \mathit{e} : \mathit{list} [\tau] \quad \Gamma \vdash \mathit{e}_1 : \tau' \quad \Gamma, \mathit{x} : \tau, \mathit{y} : \mathit{list} [\tau] \vdash \mathit{e}_2 : \tau'
\]

\[
\Gamma \vdash \text{\texttt{case1list} } \mathit{e} \text{ of } \{ \mathit{nil} \Rightarrow \mathit{e}_1 ; \mathit{x} :: \mathit{y} \Rightarrow \mathit{e}_2 \} : \tau'
\]

The basic idea here is: Given a list \( e \), a \texttt{case1list} expression does a case analysis. If \( e \) evaluates to \( \mathit{nil} \), then we evaluate \( e_1 \). Otherwise, \( e \) must evaluate to a non-empty list of the form \( \mathit{v} :: \mathit{v}' \), and we bind \( x \) to the head element \( \mathit{v} \) and \( y \) to the tail \( \mathit{v}' \), and evaluate \( e_2 \).

(a) Write appropriate typing rules for \texttt{nil} and ::.

(b) (⋆) Write appropriate evaluation rules for the above constructs.

4. (⋆) Multiple argument functions and mutual recursion

(a) So far, our function definitions take only one argument. Consider \( L_{\text{Data}} \) with named functions extended with multi-argument function definitions and applications:

\[
\begin{align*}
\mathit{e} & ::= \cdots \mid \mathit{let}\ \text{\texttt{fun} } f (\mathit{x}_1 : \tau_1, \mathit{x}_2 : \tau_2) = \mathit{e}_1 \ \text{in} \ \mathit{f}(\mathit{e}_1, \mathit{e}_2)
\end{align*}
\]

i. Write appropriate typing rules for these constructs.

ii. Show that these constructs can be defined in \( L_{\text{Data}} \).

iii. What about functions of three or more arguments?

(b) In Lecture 5, we considered a simple form of recursion that just defines one recursive function with one argument. Part 4 of this tutorial showed how to accommodate multiple arguments. But what about mutual recursion?

A simple example is

\[
\begin{align*}
\begin{array}{l}
\mathit{let}\ \text{\texttt{rec}} \ \mathit{even}(\mathit{x} : \mathit{int}) : \mathit{bool} = \mathit{if} \ \mathit{x} == 0 \ \mathit{then} \ \mathit{true} \ \mathit{else} \ \mathit{odd}(\mathit{x} - 1) \\
\mathit{and} \ \mathit{odd}(\mathit{x} : \mathit{int}) : \mathit{bool} = \mathit{if} \ \mathit{x} == 0 \ \mathit{then} \ \mathit{false} \ \mathit{else} \ \mathit{even}(\mathit{x} - 1) \\
\mathit{in} \ \mathit{e}
\end{array}
\end{align*}
\]

Show that we can use pairing and \texttt{rec} to define these mutually recursive functions, by filling in the following template with an expression having type \( \text{unit} \rightarrow ((\text{int} \rightarrow \mathit{bool}) \times (\text{int} \rightarrow \mathit{bool})) \) with the desired behavior:

\[
\begin{align*}
\begin{array}{l}
\mathit{let} \ \mathit{p} = \cdots \ \mathit{in} \\
\mathit{let} \ \mathit{pair}(\mathit{even}, \mathit{odd}) = \mathit{p}() \ \mathit{in} \\
\mathit{e}
\end{array}
\end{align*}
\]

Why do we need to define \( \mathit{p} \) as a function taking a unit argument, instead of as a pair?