The simply-typed λ-calculus: sums and products

Alex Kavvos

Reading: PFPL §10.1, 11.1

The language of numbers and strings we have been studying so far has very limited expressivity. We will now proceed to radically expand its capabilities. As a result, it will increasingly resemble a realistic functional programming language. The full language we will study this week is known as the **simply-typed λ-calculus**.

First, we will show how to add facilities that can express the following Haskell data types and programs.

```
("hello", "world") :: (Str, Str)
data EitherNumStr = Left Num | Right Str
```

1 Products

Product types allow the programmer to form tuples. **Binary products** allow us to write functions that return not one, but two values. The **unit type** (or **nullary product**) allows us to write functions that return nothing.[1]

We extend the syntax chart of Lecture 3 by adding the following new types and pre-terms:

$$
\begin{array}{llcll}
\text{types} & \tau & ::= & \ldots & \\
 & & & \tau_1 \times \tau_2 & \text{product type} \\
 & & & \mathbf{1} & \text{unit type} \\
\text{pre-terms} & e & ::= & \ldots & \\
 & & & \langle e_1, e_2 \rangle & \text{pair constructor} \\
 & & & \pi_1(e) & \text{first projection} \\
 & & & \pi_2(e) & \text{second projection}
\end{array}
$$

The **statics** of product types are given by adding the following typing rules.

$$
\textsc{Unit}\ \frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}
\qquad
\textsc{Prod}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}
\qquad
\textsc{Proj-1}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1}
\qquad
\textsc{Proj-2}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}
$$

The **dynamics** of product types are given by adding the following rules.

$$
\textsc{Val-Unit}\ \frac{}{\langle\rangle\ \mathsf{val}}
\qquad
\textsc{Val-Pair}\ \frac{}{\langle e_1, e_2 \rangle\ \mathsf{val}}
\qquad
\textsc{D-Proj-Tuple-1}\ \frac{}{\pi_1(\langle e_1, e_2 \rangle) \longmapsto e_1}
\qquad
\textsc{D-Proj-Tuple-2}\ \frac{}{\pi_1(\langle e_1, e_2 \rangle) \longmapsto e_2}
$$

$$
\textsc{D-Proj-1}\ \frac{e \longmapsto e'}{\pi_1(e) \longmapsto \pi_1(e')}
\qquad
\textsc{D-Proj-2}\ \frac{e \longmapsto e'}{\pi_2(e) \longmapsto \pi_2(e')}
$$

For example, the following typing judgements hold.

$$
\begin{gathered}
\vdash \langle \langle\rangle, \langle \mathsf{str}[\text{‘hello’}], \mathsf{str}[\text{‘world’}] \rangle \rangle : \mathbf{1} \times (\mathsf{Str} \times \mathsf{Str}) \\
\vdash \pi_1(\langle \langle\rangle, \langle \mathsf{str}[\text{‘hello’}], \mathsf{str}[\text{‘world’}] \rangle \rangle) : \mathbf{1} \\
p : (\mathsf{Num} \times \mathsf{Num}) \times \mathsf{Num} \vdash \langle \pi_1(\pi_1(p)), \langle \pi_2(\pi_1(p)), \pi_2(p) \rangle \rangle : \mathsf{Num} \times (\mathsf{Num} \times \mathsf{Num})
\end{gathered}
$$

[1] In Haskell the binary product of two types `a` and `b` is written `(a, b)`. The unit type is written `()`, and has the unique value `()`.

2 Sums

Sum types express choices between values of different types. **Binary sums** allow us to write programs that pattern match on a variable. The **void type** (or **empty type**, or **nullary sum**) offers no choice at all.[2]

We further extend the syntax chart given above by adding the following new types and pre-terms:

$$
\begin{array}{llcll}
\text{types} & \tau & ::= & \ldots & \\
 & & & \tau_1 + \tau_2 & \text{sum type} \\
 & & & \mathbf{0} & \text{void type} \\
\text{pre-terms} & e & ::= & \ldots & \\
 & & & \mathsf{inl}(e) & \text{left injection} \\
 & & & \mathsf{inr}(e) & \text{right injection} \\
 & & & \mathsf{case}(e; x.\, e_1; y.\, e_2) & \text{case analysis}
\end{array}
$$

The **statics** of sums are given by adding the following rules.

$$
\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{abort}(e) : \tau}\ \textsc{Abort}
\qquad
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}(e) : \tau_1 + \tau_2}\ \textsc{Inl}
\qquad
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}(e) : \tau_1 + \tau_2}\ \textsc{Inr}
$$

$$
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}(e; x.\, e_1; y.\, e_2) : \tau}\ \textsc{Case}
$$

The **dynamics** of sums are given by adding the following rules.

$$
\frac{}{\mathsf{inl}(e)\ \mathsf{val}}\ \textsc{Val-Inl}
\qquad
\frac{}{\mathsf{inr}(e)\ \mathsf{val}}\ \textsc{Val-Inr}
\qquad
\frac{e \longmapsto e'}{\mathsf{abort}(e) \longmapsto \mathsf{abort}(e')}\ \textsc{D-Abort-1}
$$

$$
\frac{}{\mathsf{case}(\mathsf{inl}(e); x.\, e_1; y.\, e_2) \longmapsto e_1[e/x]}\ \textsc{D-Case-Inl}
\qquad
\frac{}{\mathsf{case}(\mathsf{inr}(e); x.\, e_1; y.\, e_2) \longmapsto e_2[e/y]}\ \textsc{D-Case-Inr}
$$

$$
\frac{e \longmapsto e'}{\mathsf{case}(e; x.\, e_1; y.\, e_2) \longmapsto \mathsf{case}(e'; x.\, e_1; y.\, e_2)}\ \textsc{D-Case-1}
$$

The definition of substitution is the one in Lecture 4, but extended with the following clauses.

$$
\begin{aligned}
\langle e_1, e_2 \rangle[e/x] &\stackrel{\text{def}}{=} \langle e_1[e/x], e_2[e/x] \rangle & \pi_i(u)[e/x] &\stackrel{\text{def}}{=} \pi_i(u[e/x]) \\
\mathsf{inl}(u)[e/x] &\stackrel{\text{def}}{=} \mathsf{inl}(u[e/x]) & \mathsf{inr}(u)[e/x] &\stackrel{\text{def}}{=} \mathsf{inr}(u[e/x]) \\
\mathsf{case}(u; z.\, e_1; v.\, e_2)[e/x] &\stackrel{\text{def}}{=} \mathsf{case}(u[e/x]; z.\, e_1[e/x]; v.\, e_2[e/x]) & &
\end{aligned}
$$

Notice that z and v are bound in e_1 and e_2 respectively, so the Barendregt convention applies.

For example, the following typing judgements hold.

$$
\begin{aligned}
&\vdash \mathsf{inl}(\mathsf{num}[4]) : \mathsf{Num} + \mathsf{Str} \\
x : \mathsf{Str} + (\mathsf{Str} \times \mathsf{Num}) &\vdash \mathsf{case}(x; y.\, y; z.\, \pi_1(z)) : \mathsf{Str} \\
x : \mathsf{Str} + \mathsf{Num} &\vdash \mathsf{case}(x; y.\, \mathsf{inr}(y); z.\, \mathsf{inl}(z)) : \mathsf{Num} + \mathsf{Str}
\end{aligned}
$$

[2] In Haskell the binary sum of two types is given by the declaration `data Either a b = Left a | Right b`. The void type can be defined by the declaration `data Empty`, but it is less useful.