Recitation 4:
Products, Sums, and Pattern Matching

15-312: Principles of Programming Languages

Wednesday, February 5, 2014

In class we have been looking at how to add more power to our programming languages through function definitions. In particular, the language PCF $\mathcal{L}(\texttt{nat} \rightharpoonup)$ allows us to define recursion and partial functions. In today's recitation and homework 2, we will look at how finite data types allow us to structure data.

1 Product Types

The binary product of two types $\tau_1 \times \tau_2$ consists of ordered pair of values from each type. The eliminatory forms for this type are the projections which select the first or the second component of the pair. One can also consider the type of the nullary product `unit` which has no eliminatory form and contains no interesting value.

$$\begin{array}{llll}
\mathsf{Type} & \tau & ::= & \texttt{unit} & \texttt{unit} \\
 & & & \texttt{prod}(\tau_1;\tau_2) & \tau_1 \times \tau_2 \\
\mathsf{Exp} & e & ::= & \texttt{triv} & \langle\rangle \\
 & & & \texttt{pair}(e_1;e_2) & \langle e_1, e_2\rangle \\
 & & & \texttt{prl}(e) & e.l \\
 & & & \texttt{prr}(e) & e.r
\end{array}$$

The statics for products is then defined as follows

$$\frac{}{\Gamma \vdash \langle\rangle : \texttt{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_2 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.l : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.r : \tau_2}$$

It is possible to formulate the dynamics for product types either eagerly or lazily. We'll go with eager in this class.

$$\frac{}{\langle\rangle\ \mathsf{val}} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}} \qquad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle} \qquad \frac{e \mapsto e'}{e.l \mapsto e'.l} \qquad \frac{e \mapsto e'}{e.r \mapsto e'.r}$$

Task Are we missing any rules here? Write those down

2 Sum Types

Most data structures involve alternatives such the distinction between a leaf and a interior node in a tree. `datatype` declarations in ML usually involve a labelled sum of different types. A well-known mistake in programming language design is that of C-style pointers. A pointer is either NULL or a 'address' to a value. The erasure of this distinction is the source of many a bug.

The simplest sum type, the binary sum is a choice between two types. The elimination form of a sum type is a case analysis of a value.

$$\begin{array}{rrlll}
\textsf{Type} & \tau & ::= & \texttt{sum}(\tau_1;\tau_2) & \tau_1+\tau_2 \\
\textsf{Exp} & e & ::= & \texttt{inl}[\tau_1;\tau_2](e) & \texttt{inl}(e) \\
 & & & \texttt{inr}[\tau_1;\tau_2](e) & \texttt{inr}(e) \\
 & & & \texttt{case}(e;x_1.e_1;x_2.e2) & \texttt{case}\ e\ \{\texttt{inl}(x_1)\Rightarrow e_1|\texttt{inr}(x_2)\Rightarrow e_2\}
\end{array}$$

For the sake of readability, we drop the type annotation from `inl` and `inr`, but we need the annotation to retain the other type in the term. The statics of binary sums are defined as follows

$$\frac{\Gamma\vdash e:\tau_1}{\Gamma\vdash\texttt{inl}(e):\tau_1+\tau_2}\qquad\frac{\Gamma\vdash e:\tau_2}{\Gamma\vdash\texttt{inr}(e):\tau_1+\tau_2}$$

$$\frac{\Gamma\vdash e:\tau_1+\tau_2\qquad\Gamma,x_1:\tau_1\vdash e_1:\tau\qquad\Gamma,x_2:\tau_2\vdash e_2:\tau}{\Gamma\vdash\texttt{case}(e;x_1.e_1;x_2.e_2):\tau}$$

As before, it is possible to define the lazy and eager dynamics for binary sums

$$\frac{e\ \textsf{val}}{\texttt{inl}(e)\ \textsf{val}}\qquad\frac{e\ \textsf{val}}{\texttt{inr}(e)\ \textsf{val}}\qquad\frac{e\mapsto e'}{\texttt{inl}(e)\mapsto\texttt{inl}(e')}\qquad\frac{e\mapsto e'}{\texttt{inr}(e)\mapsto\texttt{inr}(e')}$$

$$\frac{e\mapsto e'}{\texttt{case}(e;x_1.e_1;x_2.e_2)\mapsto\texttt{case}(e';x_1.e_1;x_2.e_2)}\qquad\frac{e\ \textsf{val}}{\texttt{case}(\texttt{inl}(e);x_1.e_1;x_2.e_2)\mapsto[e/x_1]e_1}$$

$$\frac{e\ \textsf{val}}{\texttt{case}(\texttt{inr}(e);x_1.e_1;x_2.e_2)\mapsto[e/x_2]e_2}$$

It is possible to translate a number of types using sums and products. For example, booleans can be defined as

$$\texttt{bool}\triangleq unit+unit$$

Task Define `true` and `false` for this definition of `bool`.

3 Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms of most finite datatypes. So, instead of writing

```
let x = e in  + x.r end
```

we, could alternatively write

```
 case e of {(x,y) => x + y}
```

For homework 2, we will implement most of the language of patterns as described in Chapter 13 of PFPL. The language extends the PCF with sums and products with the following constructs.

$$\begin{array}{llcll}
\mathsf{Exp} & e & ::= & \mathtt{match}[p_1 \dots p_n](\vec{x_1}.e_2; \dots ; \vec{x_n}.e_n) & \mathtt{match}\, e\ \{p_1 \Rightarrow e_1 | \dots | p_n \Rightarrow e_n\} \\
\mathsf{Pattern} & p & ::= & \mathtt{wild} & \_ \\
 & & & x & x \\
 & & & \mathtt{z} & \mathtt{z} \\
 & & & \mathtt{s}(p) & \mathtt{s}(p) \\
 & & & \mathtt{triv} & \langle\rangle \\
 & & & \mathtt{pair}(p_1; p_2) & \langle p_1, p_2 \rangle \\
 & & & \mathtt{inl}(p) & \mathtt{inl}(p) \\
 & & & \mathtt{inr}(p) & \mathtt{inr}(p)
\end{array}$$

Here, each $p \Rightarrow e$ is called a rule and match takes a list of rules of the form $\{p_1 \Rightarrow e_1 | \dots p_n \Rightarrow e_n\}$

3.1 Statics

The statics of this language uses a few auxiliary judgements to typecheck patterns and rules. The first judgement is written as

$$x_1 : \tau_1, \dots, x_n : \tau_n \Vdash p : \tau$$

This judgement is similar to the $\Gamma \vdash e : \tau$ judgement with the subtle difference that we cannot add arbitrary variables to the context. So, each well-formed pattern of a certain type has a fixed Λ. Which means that this judgement can be thought of as a function that takes p and τ as input and outputs a Λ. This judgement is inductively defined as follows.

$$\frac{}{x : \tau \Vdash x : \tau} \qquad \frac{}{\emptyset \Vdash \_ : \tau} \qquad \frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}} \qquad \frac{}{\emptyset \Vdash \mathtt{z} : \mathtt{nat}} \qquad \frac{\Lambda \Vdash p : \mathtt{nat}}{\Lambda \Vdash \mathtt{s}(p) : \mathtt{nat}}$$

$$\frac{\Lambda \Vdash p : \tau_1}{\Lambda \Vdash \mathtt{inl}(p) : \tau_1 + \tau_2} \qquad \frac{\Lambda \Vdash p : \tau_2}{\Lambda \Vdash \mathtt{inr}(p) : \tau_1 + \tau_2}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \qquad \Lambda_2 \Vdash p_2 : \tau_1 \qquad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2}$$

The typing judgement for rules and sequences of rules can then be written as

$$p \Rightarrow e : \tau \rightsquigarrow \tau'$$

This judgement is read as "The rule $p \Rightarrow e$ transforms type τ to τ'".

This judgement is defined by the rules

$$\frac{\Lambda \Vdash p : \tau \qquad \Gamma\ \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau \rightsquigarrow \tau'} \qquad \frac{\Gamma \vdash p_1 \Rightarrow e_1 : \tau \rightsquigarrow \tau' \qquad \dots \qquad \Gamma \vdash p_n \Rightarrow e_n : \tau \rightsquigarrow \tau'}{\Gamma \vdash p_1 \Rightarrow e_1\ | \dots |\ p_n \Rightarrow e_n : \tau \rightsquigarrow \tau'}$$

Finally, the typing rule for the match expression is as follows

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash rs : \tau \rightsquigarrow \tau'}{\Gamma \vdash \mathtt{match}\, e\ \{rs\} : \tau'}$$

Here, rs is a sequence of zero or more rules of the form $p \Rightarrow e$.

4 Takeaways

Takeaway 1 Every type has introduction and elimination forms. Introduction forms specify ways to create terms of that type. On the other hand, elimination forms, specify ways to use terms of that type. In some cases, introduction or elimination cases might be absent. For example, the nullary product `unit` has no elimination form as there is no way to use $\langle\rangle$.

Takeaway 2 Every type has a canonical form which is the set of 'final' terms for that type. A term reaches a canonical form when it has been fully evaluated.