CS103 Daily Doggie Bag 21

The most important topics and details of the day.

April 17, 2007

**Topic 1. Language extensions: the language DP.** As demonstrated in examples for hw102, it is possible to encode pairs in D with functional trickery, but it is more efficient and appealing from a programming perspective to include them directly in the language. We therefore extend our language with operations for constructing and deconstructing pairs. In general, pairs are very useful datatypes, since they can be used to define other datatypes such as \( n \)-tuples and lists.

\[
v ::= \cdots \mid (v,v)
\]

\[
e ::= \cdots \mid (e,e) \mid \text{left}(e) \mid \text{right}(e)
\]

The operational rules are straightforward; note that our definition of values and the PAIR rule specifies an eager evaluation of pairs (this will show up in an example, below):

<table>
<thead>
<tr>
<th>PAIR</th>
<th>LEFT</th>
<th>RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 \Rightarrow v_1 ) ( e_2 \Rightarrow v_2 ) ( (e_1,e_2) \Rightarrow (v_1,v_2) )</td>
<td>( e \Rightarrow (v,v') ) ( \text{left}(e) \Rightarrow v )</td>
<td>( e \Rightarrow (v',v) ) ( \text{right}(e) \Rightarrow v )</td>
</tr>
</tbody>
</table>

Example:

\[
\text{add} \triangleq (\text{Function } x \rightarrow \text{left}(x) + \text{right}(x))
\]

\[
\begin{align*}
1 & \Rightarrow 1 \\
2 & \Rightarrow 2 \\
3 & \Rightarrow 3 \\
2 + 3 & \Rightarrow 5 \\
(1,2+3) & \Rightarrow (1,5)
\end{align*}
\]

\[
\text{left}(1,5) \Rightarrow 1 \\
\text{right}(1,5) \Rightarrow 5 \\
\text{left}(1,5) + \text{right}(1,5) \Rightarrow 6
\]

\[
\begin{align*}
\text{add} & \Rightarrow \text{add} \\
(1,2+3) & \Rightarrow (1,5) \\
\text{left}(1,5) + \text{right}(1,5) & \Rightarrow 6
\end{align*}
\]

To implement pairs, we follow the familiar pattern; we extend our abstract syntax definition to include pairs and projection:

\[
\text{type expr} =
\]

\[
\cdots \\
| \text{Pr of expr} * \text{expr} \\
| \text{Left of expr} \\
| \text{Right of expr}
\]

We extend our definition of concrete-to-abstract expression transformations:

\[
\begin{align*}
[[e_1,e_2]] & = \text{Pr}([[e_1],[e_2]]) \\
[[\text{left}(e)]] & = \text{Left}([[e]]) \\
[[\text{right}(e)]] & = \text{Right}([[e]])
\end{align*}
\]

We then extend \texttt{eval} to implement the operational semantics for these language forms.
let rec eval e =
  match e with
  ...
  | Pr(e1, e2) -> Pr(eval e1, eval e2)
  | Left(e) -> (match eval e with
      Pr(e1, e2) -> e1
      | _    -> raise TypeMismatch)
  | Right(e) -> (match eval e with
      Pr(e1, e2) -> e2
      | _    -> raise TypeMismatch)

Topic 2. Type system extensions: the language IDR. The distance between type checking and type inference is large, but that between the ID feature set and a slightly extended one is often not so. Adding datastructures to the language is often a fairly simple problem. Usually, all that is required is to come up with the appropriate new type syntax, and derivation rules for typing values and operations that conserve type preservation and safety. For example, it is fairly straightforward to add records to the language.

Type syntax We extend the type syntax with a record type forms, similar to OCaml record types, that represent the type of field values:

\[ \tau ::= t \mid \text{Int} \mid \text{Bool} \mid \tau \rightarrow \tau \mid \{ l_1 : \tau_1 ; \ldots ; l_n : \tau_n \} \]  

Type judgements Type judgements are extended with rules for typing record construction and destruction:

\[ \text{RECORD} \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \] 
\[ \Gamma \vdash \{ l_1 = e_1 ; \ldots ; l_n = e_n \} : \{ l_1 : \tau_1 ; \ldots ; l_n : \tau_n \} \] 

\[ \text{SELECT} \quad \Gamma \vdash e : \{ \ldots ; l : \tau ; \ldots \} \] 
\[ \Gamma \vdash e . l : \tau \]  

Proving type safety for IDR is also easy on the basis of ID. Essentially, all that is necessary is to add RECORD and SELECT cases to the ID type preservation Lemma. Extending the type reconstruction algorithm is also relatively straightforward, since records do not introduce new variables the derivation rules above are essentially deterministic.