Recitation 6:
Dynamic and Hybrid Typing

15-312: Principles of Programming Languages

Wednesday, February 19, 2014

It is fashionable today to extol the virtues of “dynamically-typed” languages, such as Python, Ruby, Javascript, LISP and so on. In these languages, all values are internally tagged with a \textit{class} when they are created, and all operations must check the class of their operands each time they are used. As you will see in the next assignment, it is possible to translate Dynamic PCF to Hybrid PCF, vastly reducing the number of judgements needed in the dynamics.

In this recitation, we will work through another translation from Hybrid PCF to a slightly modified version of statically-typed PCF which includes product, sum, boolean, and recursive types as well as pattern matching.

1 Hybrid PCF

In Dynamic PCF, all the tagging and tag checking was done implicitly. The programmer has no control over when dynamically classified values are used, because these are the only kinds of values there are. We now consider a language where tagging and tag checking is controlled by the programmer, by defining an extension of statically-typed PCF with a new type, \texttt{dyn}, for dynamically classified values.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract Form</th>
<th>Concrete Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau ) ::=</td>
<td>( \texttt{parr}(\texttt{dyn}; \texttt{dyn}) )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{nat} )</td>
<td>( \texttt{nat} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{bool} )</td>
<td>( \texttt{bool} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{prod}(\texttt{dyn}; \texttt{dyn}) )</td>
<td>( \texttt{dyn} \times \texttt{dyn} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{unit} )</td>
<td>( \texttt{unit} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{dyn} )</td>
<td>( \texttt{dyn} )</td>
</tr>
<tr>
<td>Label</td>
<td>( \texttt{lab} ::= )</td>
<td>( \texttt{fun} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{num} )</td>
<td>( \texttt{num} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{cons} )</td>
<td>( \texttt{cons} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{nil} )</td>
<td>( \texttt{nil} )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{cond} )</td>
<td>( \texttt{cond} )</td>
</tr>
<tr>
<td>Exp</td>
<td>( e ::= )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{query}{\texttt{lab}}(e) )</td>
<td>( \texttt{lab}\texttt{?}e )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{new}{\texttt{lab}}(e) )</td>
<td>( \texttt{lab}e )</td>
</tr>
<tr>
<td></td>
<td>( \texttt{cast}{\texttt{lab}}(e) )</td>
<td>( e@\texttt{lab} )</td>
</tr>
</tbody>
</table>
The statics for HPCF are as you would expect for most cases, with the addition of the rules for introducing and eliminating dynamically classified values.

\[
\begin{align*}
\Gamma \vdash e : \text{dyn} & \rightarrow \text{dyn} \quad \text{(dyn-I_1)} \\
\Gamma \vdash \text{fun!} e : \text{dyn} & \quad \text{(dyn-I_2)} \\
\Gamma \vdash \text{nil!} e : \text{dyn} & \quad \text{(dyn-I_3)} \\
\Gamma \vdash e : \text{nat} & \quad \text{(dyn-I_4)} \\
\Gamma \vdash e : \text{bool} & \quad \text{(dyn-I_5)} \\
\Gamma \vdash e : \text{dyn} & \rightarrow \text{dyn} \quad \text{(dyn-E_1)} \\
\Gamma \vdash e \oplus \text{fun} : \text{dyn} & \rightarrow \text{dyn} \quad \text{(dyn-E_2)} \\
\Gamma \vdash e : \text{dyn} & \rightarrow \text{dyn} \quad \text{(dyn-E_3)} \\
\Gamma \vdash e \oplus \text{num} : \text{nat} & \quad \text{(dyn-E_4)} \\
\Gamma \vdash e \oplus \text{cond} : \text{bool} & \quad \text{(dyn-E_5)} \\
\Gamma \vdash e : \text{dyn} & \quad \text{(dyn-Q)}
\end{align*}
\]

The dynamics say that \text{new} is a canonical form of \text{dyn} and \text{cast} is post-inverse to \text{new} if the classes match, or an error otherwise (error propagation rules are omitted below.)

\[
\begin{align*}
e \text{val} & \quad \text{lab!} e \text{ val} \\
\text{lab!} e \rightarrow \text{lab!} e' & \quad \text{lab!} e \rightarrow \text{lab!} e' \oplus \text{lab} \\
e \rightarrow e' & \quad \text{e@lab} \rightarrow e' \oplus \text{lab} \\
\text{lab} \neq \text{lab}' & \quad \text{lab!} e \rightarrow \text{lab!} e' \oplus \text{lab}
\end{align*}
\]

\[
\begin{align*}
\text{(lab! e)} \oplus \text{lab} & \rightarrow e \\
\text{(lab! e)} \oplus \text{lab}' & \rightarrow \text{err} \\
\text{lab!} e \rightarrow \text{lab!} e' & \quad \text{lab!} (\text{lab!} e) \rightarrow \text{tt} \\
\text{lab!} (\text{lab!} e) & \rightarrow \text{ff}
\end{align*}
\]

2 Translation

For our translation, we would like to have the property that if two expressions have the same type in HPCF, then their translation preserves equality of types in the resulting language. In order to guide our translation of expressions, we will first define a translation of types. If \(\tau\) is a type in HPCF, we will use \(|\tau|\) to denote the resulting type after translation. The types in our target language are what you would expect (products, sums, bools, functions, nats, and recursive types).
Task 1  Define the following recursively:

\[ |\text{dyn} \rightarrow \text{dyn}| = \]
\[ |\text{nat}| = \]
\[ |\text{bool}| = \]
\[ |\text{dyn} \times \text{dyn}| = \]
\[ |\text{unit}| = \]
\[ |\text{dyn}| = \]

In translating expressions, we want to make sure that every case preserves the types through translation as we outlined in the last task. If \( e \) is an expression in HPCF, we will use \( |e| \) to denote the resulting expression after translation to our target language. The forms of expressions are what you would expect given the types (with pattern matching).

Task 2  Define the following recursively:

\[ |\text{num}?>e| = \]
\[ |\text{cons}!e| = \]
\[ |e@\text{fun}| = \]