Toward More Realistic Languages – Functions, Natural Numbers and Recursion

We began by considering $L\{\text{num, str}\}$

- Numbers, strings, some arbitrary primitive operations.
- Static and dynamic semantics; type safety.

We then considered adding functions to $L\{\text{num, str}\}$

- First order, via function definitions
- Higher order, via function types

Toward More Realistic Languages – Recursive Functions

A more powerful approach to realism is to combine recursive functions and natural numbers, yielding general computational capability.

Harper describes two possibilities:

- One based on primitive recursion: $L\{\text{nat} \rightarrow \}$ or Gödel’s T.
- Another on general recursion: $L\{\text{nat} \rightarrow \}$ or Plotkin’s PCF.

Primitive recursion permits only total functions, termination is guaranteed, but some computations cannot be coded this way (not Turing complete), so not practical for a "realistic" language.

Syntax for $L\{\text{nat} \rightarrow \}$

The expression $\text{rec}_n(c, e_0; x, y, e_1)$ is called primitive recursion. It represents the the $n$-fold iteration of the transformation $x \cdot y \cdot e_1$ starting from $e_0$, where $x$ is bound to the predecessor and $y$ is bound to the result of the $n$-fold iteration.

Sometime iteration, written $\text{iter}_n(c, e_0; y, e_1)$, is considered an alternative to primitive recursion. Similar meaning, except only result of recursive call is bound to $y$ in $e_1$ and there is no binding for the predecessor. Clearly iteration is definable from recursion, by ignoring the predecessor binding. Conversely, can define primitive recursion from iteration and product by pairing simultaneous computation of predecessor with iteration of specified computation.

Static Semantics for $L\{\text{nat} \rightarrow \}$

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{nat}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$x$</td>
<td>$y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{nat}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{arr}(\tau_1, \tau_2)$</td>
<td>$\tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{rec}(e_0; y, e_1)$</td>
<td>$\text{rec}(\lambda x. e_0 , \text{with} , y \Rightarrow e_1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \langle x, e \rangle$</td>
<td>$\lambda (\tau_2, e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ap}(e_1, e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
</tbody>
</table>

We write $\overline{n}$ for the expression $\underbrace{s \ldots s}_{n}(\text{zero})$, in which successor is applied $n \geq 0$ times to zero.
Properties of Typing

Lemma 1 (Substitution)

If \( \Gamma, x : \tau \vdash e' : \tau' \) and \( \Gamma \vdash e : \tau \), then \( \Gamma \vdash [e/x]e' : \tau' \).

The proof is by induction on typing.

Dynamic Semantics for \( L^{\text{nat} \to \text{nat}} \)

We will adopt an eager semantics for successor, so that values of type \( \text{nat} \to \text{nat} \) are numerals, and a call-by-name semantics for function applications. Variables range over computations, which are not necessarily values. These choices are not required, but are natural and convenient since every closed expression in \( L^{\text{nat} \to \text{nat}} \) has a value.

\[
\begin{align*}
\text{val} &\quad \text{val} \\
\text{e} &\quad \text{val} \\
\text{rec}(x, r_0, x, y, e_1) &\quad \text{e} \\
\text{ap} &\quad \text{ap}(e_1, e_2) \Rightarrow \text{ap}(e_1', e_2) \\
\text{ap}(\text{lam}(f)(x), r_2) &\Rightarrow [y/2]e \\
\text{e} &\quad \text{e}' \\
\text{rec}(e, r_0, x, y, e_1) &\Rightarrow \text{rec}(e', r_0, x, y, e_1)
\end{align*}
\]

Note that lazy binding to \( y \) means recursive call on \( e \) will not be executed unless \( y \) is required for evaluation of \( e_1 \).

Definability in \( L^{\text{nat} \to \text{nat}} \)

A function \( f : \mathcal{N} \to \mathcal{N} \) is definable in \( L^{\text{nat} \to \text{nat}} \) iff there exists an expression \( e_f : \text{nat} \to \text{nat} \) such that for every \( n \in \mathcal{N} \):

\[ e_f(n) \equiv \overline{f(n)} : \text{nat} \]

where \( \equiv \) means definitionally equivalent.

For example, the doubling function \( d(n) = 2 \times n \) is definable by the expression

\[ e_d = \lambda x : \text{nat}. \text{rec} x (z \to z \mid s(a) \text{ with } v \Rightarrow s(s(v))) \]
Definability in $L^{\text{nat} \to}$

To see this, observe that $e_f(0) \equiv 0 : \text{nat}$ and that, assuming $e_f(n+1) \equiv s(e_f(n))$, we have:

$$
e_f(0) = 0$$

$$
e_f(0+1) = s(e_f(0)) = s(0) = 1$$

$$
n+1 \equiv n + 1$$

Even the Ackermann function is definable in $L^{\text{nat} \to}$.

Non-Definability in $L^{\text{nat} \to}$

It is impossible to define an infinite loop in $L^{\text{nat} \to}$:

**Theorem 4**

If $e : \tau$, then there exists a value such that $e \Rightarrow^* v$.

But a diagonalization argument can be used to show that there exist functions on the natural numbers that are not definable in $L^{\text{nat} \to}$. (See Harper for details.)

Toward More Realistic Languages – $L^{\text{nat} \to}$

General recursion does not guarantee termination – functions are partial but supports Turing complete computation.

Thus general recursion is a more practical starting point for a “realistic” language – $L^{\text{nat} \to}$ or Plotkin’s PCF.

Syntax for $L^{\text{nat} \to}$

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{nat}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{par}(\tau_1; \tau_2)$</td>
<td>$\tau_1 \to \tau_2$</td>
<td>$\tau_1 \to \tau_2$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$z$</td>
<td>$z$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$s(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{ifz}(e; z; x; e_1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{lam}[x].e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{ap}(e_1; e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{fix}[\tau](x; e)$</td>
</tr>
</tbody>
</table>

The expression $\text{fix}[\tau](x; e)$ is called general recursion.

The expression $\text{ifz}(e; z; x; e_1)$ branches according to whether $e$ evaluates to $z$, binding the predecessor to $x$ in the case that it does not.
Static Semantics for $L^{nat \rightarrow}$

$\Gamma, x : \tau \vdash x : \tau$

$\Gamma \vdash z : \tau \rightarrow \tau$

$\Gamma \vdash \text{nat} : \tau$

$\Gamma \vdash \text{nat} : \tau$

$\Gamma \vdash s : S$  

$\Gamma \vdash e : \text{nat}$

$\Gamma \vdash e : \tau$

$\Gamma, x : \tau \vdash e : \tau$

$\Gamma \vdash \text{ifz}(e_1, e_2, e_3) : \tau$

$\Gamma \vdash \text{fix}(\tau) S x : e : T$  

Deceptively simple, but note that we assume the typing we are trying to establish!
Safety

Theorem 6

1. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

2. If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \mapsto e' \).

Definability for \( L(\text{nat} \mapsto \cdot) \)

We define general recursive functions with syntax:

\[
\text{fun } x : S \ y : m \ \tau \ c \ T \mapsto \tau \ d \ is \ e
\]

where \( x \) stands for the function itself and \( y \) is its argument, and both are bound in \( e \).

Dynamic semantics for general recursive functions:

\[
\text{fun } x : y : T_1 \mapsto S \ e(z) \Rightarrow [\text{fun } x : y : T_1 \mapsto S \ e(z)/x, y] e
\]

Recursively substituting the function itself for \( x \) in its body.

Definability for \( L(\text{nat} \mapsto \cdot) \)

Primitive recursion can also be defined in terms of general recursion.

Specifically, define \( \text{fix} \ e \{ z \mapsto e_0 | s(z) \} \) with \( y \mapsto e \) to be the expression \( e'(e) \) where \( e' \) is the general recursive function

\[
\text{fun } f : n : \text{nat} \mapsto S \ e_0 \ mapsto x : S \ e(x) \Rightarrow [f(x)/y] e_0
\]

It is easy to check that the static and dynamic semantics of primitive recursion functions are derivable in \( L(\text{nat} \mapsto \cdot) \) using this expansion.

Definability for \( L(\text{nat} \mapsto \cdot) \)

Because \( L(\text{nat} \mapsto \cdot) \) admits partial functions, definability is more difficult to characterize here than for \( L(\text{nat} \mapsto \cdot) \).

It turns out that definability in \( L(\text{nat} \mapsto \cdot) \) corresponds to partial recursive functions and that Church’s Law says partial recursive functions are exactly what can be computed with any conceivable programming language.

Therefore, \( L(\text{nat} \mapsto \cdot) \) is as powerful as any other programming language — if perhaps not as user friendly!