Recursive Datatypes

Datatypes get interesting when they are recursive.

datatype ilist = Nil | Cons of int * ilist

datatype itree = Empty | Node of itree * int * itree

datatype expr =
  Num of int |
  Plus of expr * expr |
  Times of expr * expr

How can we account for these in formal terms?

Self-Reference and Recursion

Recursive datatypes in ML are self-referential:

datatype ilist = Nil | Cons of int * ilist

Similarly, recursive functions in ML are self-referential:

fun fact v0 x = 1
  | fact vn x = n y fact vn-1 x

Self-Reference and Recursion

In discussing derivability in $L_{\text{nat} \rightarrow}$ we introduced the notion of the general recursive function. Using the same concrete syntax and assuming integers and their arithmetic, we can write:

fun fact(n:int):int is
  ifz z n {z => svzx | svxx => n y factvxx}

Think of this as the value bound to fact by the recursive function declaration.

Self-Reference and Recursion

Similarly, we will introduce a self-referential type expression to model recursive types:

$\mu \text{ilist} (\text{nat} \times \text{ilist})$

Think of this as the type bound to ilist by the recursive datatype declaration.

PCF With Lists

As a warm-up, let’s extend $L_{\text{nat} \rightarrow}$ with a primitive type of nat lists.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>natlist</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>nnil</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cons(x1,x2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>listcase e [nil=&gt;e1</td>
</tr>
</tbody>
</table>

The variables $x$ and $y$ are bound in $e_2$ in a listcase expression.
PCF With Lists

\[ \Gamma \vdash \text{nil} : \text{natlist} \]

\[ \Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat} \]

\[ \Gamma \vdash e : \text{nat} \quad \Gamma, x : \text{nat}, y : \text{natlist} \vdash er : \text{r} \]

\[ \Gamma \vdash \text{listcase} (\text{nil} \mapsto e_1 | \text{cons}(x,y) \mapsto e_2) : r \]

---

Decompose the constructor \text{Nil} into three steps:

- Form a null-tuple.
- Tag it as \text{Nil}.
- Allocate and return a “pointer” to it.
Representing Lists

Decompose the constructor Cons(n,l) into three steps:

- Form the pair (n,l).
- Tag the result as a Cons.
- Allocate and return a “pointer” to it.

Recursive Types

Pattern matching reverses the steps:

- Dereference the “pointer” to recover a tagged value.
- Dispatch on the tag: Nil or Cons.
- For Nil, pass to the empty case.
- For Cons, split the pair and pass to the non-empty case.

Lists as a Recursive Type

We will think of natlist as the recursive type

\[ \mu nl. \text{unit} \times (\text{nat} \times nl) \] 

It comes equipped with operations fold and unfold:

- fold(-) : unit+(nat x natlist) -> natlist.

Refining the Representation

Idea: separate allocation from tagging and tupling.

- Allocation: recursive types.
- Tagging: sum types.
- Tupling: product types.

Recursive Types as Pointers

The values of the recursive type are

- fold(\text{inl}(\text{[]})), corresponding to nil.
- fold(\text{inr}(\text{[n,l]})), corresponding to \text{Cons}(n,l).

This abstract representation corresponds directly to its concrete implementation!
Recursive Types and Pointers

The list nil decomposes into:

- A **pointer** `fold(...)` to ...
- A **tagged value** `ta[1](...)` containing ...
- The null tuple `()`. 

Recursive Types

A picture of Cons(8, Cons(9, Nil)) in memory:

```
  C 8 —— C 9 —— R
```

Recursive Types: Syntax

- The metavariable `t` ranges over a class of **type names**
- Think of the introduction form `fold(l.r)v(e)` as an abstract pointer.
- Think of the elimination form `unfold(e)` as an abstract dereference of a pointer.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td><code>t</code></td>
<td><code>r</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td>Expr</td>
<td><code>e</code></td>
<td><code>fold(l.r)v(e)</code></td>
<td><code>fold(e)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>unfold(e)</code></td>
<td><code>unfold(e)</code></td>
</tr>
</tbody>
</table>

Recursive Types: Static Semantics

Incorporating **type formation judgements** \(\Delta \vdash t \text{ type}\) where \(\Delta\) is a finite set of assumptions of the form `t` type for some type variable `t`.

\[
\begin{align*}
\Delta \vdash \text{type} & \vdash \text{type} \\
\Delta \vdash r_1 \text{ type} & \Delta \vdash r_2 \text{ type} \\
\Delta & \vdash \text{pair}(r_1, r_2) \text{ type} \\
\Delta, t \text{ type} & \vdash \text{rec}(l.r) \text{ type} \\
\end{align*}
\]
Recursive Types: Static Semantics

“Roll” the recursive type (allocate a pointer):

\[ \Gamma \vdash e : [\text{rec}(\tau)]/\tau \]

“Unroll” the recursive type (chase a pointer):

\[ \Gamma \vdash e : \text{rec}(\tau) \]

\[ \Gamma \vdash \text{unrec}(\tau) : [\text{rec}(\tau)]/\tau \]

Implicitly, \( \tau \) type and \( \tau_i \) type for all \( \tau_i \) used in the hypotheses in \( \Gamma \)

Recursive Types: Dynamic Semantics

Chasing a pointer:

\[ e \mapsto e' \]

\[ \text{unrec}(e) \mapsto \text{unrec}(e') \]

\[ \text{fold}(\tau)(e) \mapsto \text{fold}(\tau)(e') \]

Safety

Theorem 1 (Preservation)

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

Proof: The proof proceeds by induction on evaluation.

Theorem 2 (Progress)

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

Proof: The proof is by induction on typing.

Encoding Nats

We have previously taken the type of natural numbers as primitive. We may instead treat \( \text{nat} \) as a recursive type, defined as:

\[ \mu t. [z : \text{unit}, s : t] \]

The natural zero, \( z \), is represented by the value

\[ \text{fold}(\text{in}(z))(\langle \rangle)) \]

The successor, \( s(e) \), is represented by the value

\[ \text{fold}(\text{in}(z))(e)) \]
Encoding Nats

The conditional branch on zero:

```plaintext
ifz e {
  z ⇒ e0
  s(z) ⇒ e1
}
```

is represented by ...

Encoding Nats

```plaintext
case unfold(v) -- chase the pointer, analyze tag
  { in[z](z) ⇒ e0 -- check for zero
    | in[z](s) ⇒ -- check for non-zero
    e1 -- predecessor available as z
  }
```

Encoding Lists

Consider again the natlist type.

```plaintext
datatype natlist = Nil | Cons of nat × natlist
```

We will think of this as the recursive type

μt. {n : unit, c : nat × t}

Encoding Lists

The case analysis

```plaintext
listcase e {
  nil ⇒ e0
  | cons (x, y) ⇒ e1
}
```

is represented by ...

Encoding Lists

```plaintext
case unfold(v) -- chase the pointer, analyze tag
  { in[n](z) ⇒ e0 -- check for nil
    | in[c](⟨x, y⟩) ⇒ -- check for cons
    e1 -- get head as x and tail as y
  }
```

Encoding Lists

The constructor all is represented by the value

```
fold(in[n]⟨⟩).
```

The constructor cons(e1, e2) is represented by the value

```
fold(in[c]⟨e1, e2⟩).
```
Summary

ML datatypes are a combination of product, sum, and recursive types.

- Recursive types for self-reference and allocation;
- Sum types for distinguishing cases;
- Product types for supporting multiple fields.

The correspondence is faithful to the typical implementation!