510: Programming Languages

Recursive Types

David Walker

Fall, 2002

Recursive Datatypes

Datatype's 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 MinML?

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(0) = 1
  | fact(n) = n * fact(n-1)
```

Self-Reference and Recursion

MinML has a **self-referential function expression** to model recursion:

```
fun fact(n:int):int is
    if n=0 then 1 else n * fact(n-1)
```

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

Self-Reference and Recursion

Similarly, we will include a **self-referential type expression** in MinML to model recursive types:

$$\texttt{rec}\, il \,\texttt{is}\, \texttt{unit+}(\texttt{int*}il)$$

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

MinML With Lists

As a warm-up, let's extend MinML with a primitive type of integer lists.

$$\begin{array}{lll} \textit{Types} & \tau & ::= \ \texttt{ilist} \\ \\ \textit{Exp's} & e & ::= \ \texttt{nil} \mid \texttt{cons}(e_1, e_2) \mid \\ & & \texttt{lcase}\, e \,\texttt{of}\, \texttt{nil} \,\texttt{=>}\, e_1 \,\texttt{|}\, \texttt{cons}(x, y) \,\texttt{=>}\, e_2 \,\texttt{end} \end{array}$$

The variables x and y are bound in e_2 in a `listcase` expression.

MinML With Lists

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{ilist}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{ilist}}{\Gamma \vdash \texttt{cons}(e_1, e_2) : \texttt{ilist}}$$

$$\frac{\Gamma \vdash e : \texttt{ilist} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x{:}\texttt{int}, y{:}\texttt{ilist} \vdash e_2 : \tau}{\Gamma \vdash \texttt{lcase}\, e\, \textsf{of}\, \texttt{nil} \texttt{=>} e_1 \mid \textsf{cons}(x, y) \texttt{=>} e_2\, \textsf{end} : \tau}$$

MinML With Lists

$$\frac{}{\texttt{nil}\ \textsf{value}}$$

$$\frac{v_1\ \textsf{value} \qquad v_2\ \textsf{value}}{\texttt{cons}(v_1, v_2)\ \textsf{value}}$$

MinML With Lists

$$\frac{e_1 \mapsto e_1'}{\texttt{cons}(e_1, e_2) \mapsto \texttt{cons}(e_1', e_2)}$$

$$\frac{v_1 \ \text{value} \qquad e_2 \mapsto e_2'}{\texttt{cons}(v_1, e_2) \mapsto \texttt{cons}(v_1, e_2')}$$

$$\frac{e \mapsto e'}{\begin{array}{c}\texttt{lcase}\, e \,\texttt{of}\, \texttt{nil} \,\texttt{=>}\, e_1 \mid \texttt{cons}(x, y) \,\texttt{=>}\, e_2 \,\texttt{end} \\ \mapsto \\ \texttt{lcase}\, e' \,\texttt{of}\, \texttt{nil} \,\texttt{=>}\, e_1 \mid \texttt{cons}(x, y) \,\texttt{=>}\, e_2 \,\texttt{end}\end{array}}$$

MinML With Lists

$$\frac{}{\texttt{lcase}\,\texttt{nil}\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e_1 \mid \texttt{cons}\,\texttt{(}x, y\texttt{)}\,\texttt{=>}\,e_2\,\texttt{end} \mapsto e_1}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{lcase}\,\texttt{cons}\,\texttt{(}v_1, v_2\texttt{)}\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e_1 \mid \texttt{cons}\,\texttt{(}x, y\texttt{)}\,\texttt{=>}\,e_2\,\texttt{end} \mapsto \{v_1, v_2/x, y\}e_2}$$

MinML With Lists

```
fun copy (l:ilist):ilist is
    listcase l of
      nil => nil
    | cons (x,y) => cons (x, copy y)
end
```

Representing Lists

Decompose the constructor `Nil` into three steps:

- Form a null-tuple.
- Tag it as `nil`.
- Allocate and return a “pointer” to it.

(In practice we can optimize this using numerous tricks.)

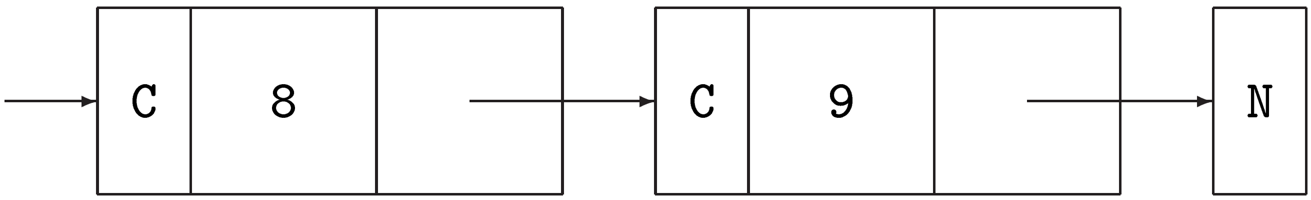
Representing Lists

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.

Representing Lists

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



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.

Refinining the Representation

Idea: separate **allocation** from **tagging** and **tupling**.

- **Allocation**: recursive types.
- **Tagging**: sums types.
- **Tupling**: products types.

Lists as a Recursive Type

We will think of `ilist` as the **recursive type**

$$\texttt{rec}\,il\,\texttt{is}\,\texttt{unit+}(\texttt{int*}il).$$

It comes equipped with operations **roll** and **unroll**:

- $\texttt{roll}(-) : \texttt{unit+}(\texttt{int*ilist}) \to \texttt{ilist}$.

- $\texttt{unroll}(-) : \texttt{ilist} \to \texttt{unit+}(\texttt{int*ilist})$.

Recursive Types as Pointers

The values of the recursive type are

- `roll(inl(()))`, corresponding to `nil`.

- `roll(inr((`n`,`l`)))`, corresponding to `cons(`n, l`)`.

This abstract representation corresponds **directly** to its concrete implementation!

Recursive Types and Pointers

The list `nil` decomposes into:

- A **pointer** `roll(...)` to ...
- A **tagged value** `inl(...)` containing ...
- The **null tuple** `()`.

(Optimized representations are possible.)

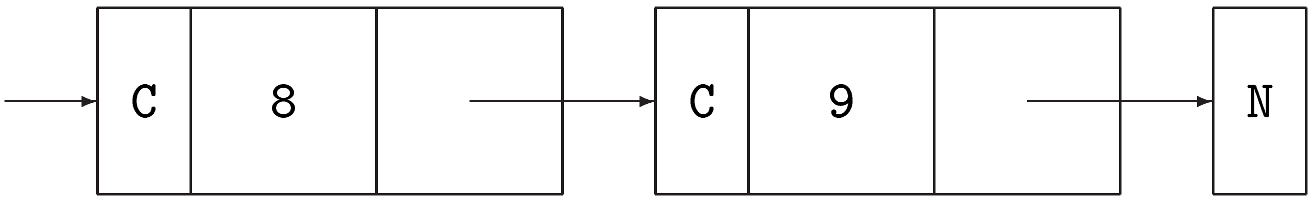
Recursive Types and Pointers

The list $\texttt{cons}(n, l) = \texttt{roll(inr((}n\texttt{,}l\texttt{)))}$ decomposes into:

- A **pointer** $\texttt{roll}(\ldots)$ to ...
- A **tagged value** $\texttt{inr}(\ldots)$ containing ...
- A **pair** $\texttt{(}n\texttt{,}l\texttt{)}$.

Recursive Types

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



Recursive Types: Abstract Syntax

Abstract syntax:

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & t \mid \mathtt{rec}\, t\, \mathtt{is}\, \tau \\
\textit{Expressions} & e & ::= & \mathtt{roll}(e) \mid \mathtt{unroll}(e) \\
\textit{Values} & v & ::= & \mathtt{roll}(v)
\end{array}
$$

The variable t is bound in τ in $\mathtt{rec}\, t\, \mathtt{is}\, \tau$.

- Think of $\mathtt{roll}(v)$ as an abstract pointer.

- Think of $\mathtt{unroll}(v)$ as an abstract de-reference of a pointer.

Recursive Types: Static Semantics

"Roll" the recursive type (allocate a pointer):

$$\frac{\Gamma \vdash e : \{\mathtt{rec}\, t\, \mathtt{is}\, \tau / t\}\tau}{\Gamma \vdash \mathtt{roll}(e) : \mathtt{rec}\, t\, \mathtt{is}\, \tau}$$

"Unroll" the recursive type (chase a pointer):

$$\frac{\Gamma \vdash e : \mathtt{rec}\, t\, \mathtt{is}\, \tau}{\Gamma \vdash \mathtt{unroll}(e) : \{\mathtt{rec}\, t\, \mathtt{is}\, \tau / t\}\tau}$$

Recursive Types: Dynamic Semantics

Chasing a pointer:

$$\frac{}{\texttt{unroll(roll(}v\texttt{))} \mapsto v}$$

$$\frac{e \mapsto e'}{\texttt{unroll(}e\texttt{)} \mapsto \texttt{unroll(}e'\texttt{)}}$$

Allocation of pointers:

$$\frac{e \mapsto e'}{\texttt{roll(}e\texttt{)} \mapsto \texttt{roll(}e'\texttt{)}}$$

Encoding Lists

Consider again the `ilist` type.

```
datatype ilist = Nil | Cons of int * ilist
```

We will think of this as the recursive type

$$\mathtt{rec}\, il\, \mathtt{is}\, \mathtt{unit}\mathtt{+}(\mathtt{int}\mathtt{*}il)$$

Encoding Lists

The constructor $\texttt{nil}$ is represented by the value

$$\texttt{roll(inl(()))}.$$

The constructor $\texttt{cons(}n\texttt{,}l\texttt{)}$ is represented by the value

$$\texttt{roll(inr((}n\texttt{,}l\texttt{)))}.$$

Encoding Lists

The case analysis

$$\begin{array}{l}\texttt{listcase}\ e\\ \quad\texttt{of nil =>}\ e_1\\ \quad\ \ \texttt{| cons (}n\texttt{,}\ l\texttt{) =>}\ e_2\end{array}$$

is represented by . . .

Encoding Lists

...the MinML code

```
case unroll(e)  (chase pointer, analyze tag)
  of inl(x:unit) =>        (check for nil)
        e1                 (do nil comp.)
   | inr(y:int*ilist) =>  (check for cons)
     split y as (n,l)      (get head & tail)
       in e2               (do cons comp.)
```

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!