

CMPSCI 630: Programming Languages
**Toward More Realistic Languages –
 Recursive Types**

Spring 2009
 (with thanks to Robert Harper)

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

Self-Reference and Recursion

In discussing derivability in $\mathcal{L}\{\text{nat} \rightarrow\}$ 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 n {z => s(z) | s(x) => n * fact(x) }
```

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:

```
 $\mu il. \text{unit} + (\text{int} \times il)$ 
```

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

PCF With Lists

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

Category	Item	Concrete
Type	τ	$::= \text{natlist}$
Expr	e	$::= \text{nil}$
		$ \text{cons}(e_1, e_2)$
		$ \text{listcase } e \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \}$

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

PCF With Lists

$$\frac{}{\Gamma \vdash \text{nil} : \text{natlist}}$$

$$\frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{natlist}}{\Gamma \vdash \text{cons}(e_1, e_2) : \text{natlist}}$$

$$\frac{\Gamma \vdash e : \text{natlist} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \text{nat}, y : \text{natlist} \vdash e_2 : \tau}{\Gamma \vdash \text{listcase } e \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \} : \tau}$$

6

PCF With Lists

$$\frac{}{\text{nil val}}$$

$$\frac{v_1 \text{ val} \quad v_2 \text{ val}}{\text{cons}(v_1, v_2) \text{ val}}$$

7

PCF With Lists

$$\frac{e_1 \mapsto e'_1}{\text{cons}(e_1, e_2) \mapsto \text{cons}(e'_1, e_2)} \quad \frac{v_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{cons}(v_1, e_2) \mapsto \text{cons}(v_1, e'_2)}$$

$$\frac{e \mapsto e'}{\text{listcase } e \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \} \mapsto \text{listcase } e' \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \}}$$

8

PCF With Lists

$$\frac{}{\text{listcase nil } \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \} \mapsto e_1}$$

$$\frac{v_1 \text{ val} \quad v_2 \text{ val}}{\text{listcase cons}(v_1, v_2) \{ \text{nil} \Rightarrow e_1 \mid \text{cons}(x, y) \Rightarrow e_2 \} \mapsto [v_1, v_2/x, y]e_2}$$

9

PCF With Lists

```
fun copy (l:natlist):natlist is
  listcase l {
    nil => nil
  | cons (x,y) => cons (x, copy(y))
  }
```

10

Representing Lists

Decompose the constructor Nil into three steps:

- Form a null-tuple.
- Tag it as Nil.
- Allocate and return a "pointer" to it.

11

Representing Lists

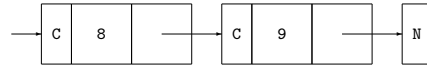
Decompose the constructor $\text{Cons}(n, l)$ into three steps:

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

12

Representing Lists

A picture of $\text{Cons}(8, \text{Cons}(9, \text{Nil}))$ in memory:



13

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.

14

Refining the Representation

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

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

15

Lists as a Recursive Type

We will think of natlist as the **recursive type**

$$\mu l. \text{unit} + (\text{nat} \times \text{nl}).$$

It comes equipped with operations **fold** and **unfold**:

- $\text{fold}(-) : \text{unit} + (\text{nat} \times \text{natlist}) \rightarrow \text{natlist}$.
- $\text{unfold}(-) : \text{natlist} \rightarrow \text{unit} + (\text{nat} \times \text{natlist})$.

16

Recursive Types as Pointers

The values of the recursive type are

- $\text{fold}(\text{in}[1](\langle \rangle))$, corresponding to nil .
- $\text{fold}(\text{in}[r](\langle n, l \rangle))$, corresponding to $\text{cons}(n, l)$.

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

17

Recursive Types and Pointers

The list `nil` decomposes into:

- A **pointer** `fold(...)` to ...
- A **tagged value** `in[1](...)` containing ...
- The **null tuple** $\langle \rangle$.

18

Recursive Types and Pointers

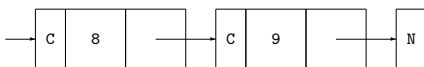
The list `cons(n, l) = fold(in[x](⟨n, l⟩))` decomposes into:

- A **pointer** `fold(...)` to ...
- A **tagged value** `in[x](...)` containing ...
- A **pair** $\langle n, l \rangle$.

19

Recursive Types

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



20

Recursive Types: Syntax

Category	Item	Abstract	Concrete
Type	τ	$::= t$ $\text{rec}(t.\tau)$	t $\mu t.\tau$
Expr	e	$::= \text{fold}[t.\tau](e)$ $\text{unfold}(e)$	$\text{fold}(e)$ $\text{unfold}(e)$

21

Recursive Types: Syntax

- The metavariable t ranges over a class of **type names**
- Think of the introduction form `fold[t.τ](e)` as an abstract pointer.
- Think of the elimination form `unfold(e)` as an abstract de-reference of a pointer.

22

Recursive Types: Static Semantics

Incorporating **type formation judgements** $\Delta \vdash \tau$ type where Δ is a finite set of assumptions of the form t type for some type variable t .

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \text{par}(\tau_1; \tau_2) \text{ type}}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \text{rec}(t.\tau) \text{ type}}$$

23

Recursive Types: Static Semantics

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

$$\frac{\Gamma \vdash e : [\text{rec}(t.\tau)/t]\tau}{\Gamma \vdash \text{fold}[t.\tau](e) : \text{rec}(t.\tau)}$$

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

$$\frac{\Gamma \vdash e : \text{rec}(t.\tau)}{\Gamma \vdash \text{unfold}(e) : [\text{rec}(t.\tau)/t]\tau}$$

Implicitly, τ type and τ_i type for all τ_i used in the hypotheses in Γ

24

Recursive Types: Dynamic Semantics

Allocation of pointers:

$$\frac{\{e \text{ val}\}}{\text{fold}[t.\tau](e) \text{ val}}$$

$$\left\{ \frac{e \mapsto e'}{\text{fold}[t.\tau](e) \mapsto \text{fold}[t.\tau](e')} \right\}$$

Bracketed rule and premise omitted for lazy semantics and included for eager semantics.

25

Recursive Types: Dynamic Semantics

Chasing a pointer:

$$\frac{e \mapsto e'}{\text{unfold}(e) \mapsto \text{unfold}(e')}$$

$$\frac{\text{fold}[t.\tau](e) \text{ val}}{\text{unfold}(\text{fold}[t.\tau](e)) \mapsto e}$$

26

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. ■

27

Encoding Nats

We have previously taken the type of natural numbers as primitive. We may instead treat `nat` as a recursive type, defined as:

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

28

Encoding Nats

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

$$\text{fold}(\text{in}[s](e)).$$

29

Encoding Nats

The conditional branch on zero:

```
ifz e {
  z => e0
| s(x) => e1
}
```

is represented by ...

30

Encoding Nats

```
case unfold(e) -- chase the pointer, analyze tag
{ in[z](_) => e0 -- check for zero
| in[s](x) => -- check for non-zero
  e1 -- predecessor available as x
}
```

31

Encoding Lists

Consider again the `natlist` type.

```
datatype natlist = Nil | Cons of nat * natlist
```

We will think of this as the recursive type

```
 $\mu t.[n : \text{unit}, c : \text{nat} \times t]$ 
```

32

Encoding Lists

The constructor `nil` is represented by the value

```
fold(in[n]⟨⟩).
```

The constructor `cons`(e_1, e_2) is represented by the value

```
fold(in[c]⟨⟨ $e_1, e_2$ ⟩⟩).
```

33

Encoding Lists

The case analysis

```
listcase e {
  nil => e0
| cons (x, y) => e1
}
```

is represented by ...

34

Encoding Lists

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

35

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!