Recursive Datatypes

Datatypes get interesting when they are recursive.

\[
\text{datatype \textit{ilist} = } \text{Nil} \mid \text{Cons of int} \times \text{\textit{ilist}}
\]

\[
\text{datatype \textit{itree} = } \text{Empty} \mid \text{Node of \textit{itree} \times int} \times \text{\textit{itree}}
\]

\[
\text{datatype \textit{expr} = } \text{Num of int} \mid \text{Plus of \textit{expr} \times \textit{expr}} \mid \text{Times of \textit{expr} \times \textit{expr}}
\]

How can we account for these in MinML?

Self-Reference and Recursion

Recursive datatypes in ML are \textit{self-referential}:

\[
\text{fun fact(0)} : \textit{int} = 1
\]

\[
\mid \text{fact(n)} : \textit{int} = n \times \text{fact(n-1)}
\]

Similarly, recursive functions in ML are \textit{self-referential}:

MinML With Lists

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

\[
\text{rec \textit{il} is unit+(int*\textit{il})}
\]

Think of this as the \textit{type} bound to \textit{ilist} by the recursive datatype declaration.

\[
\text{Types } \tau ::= \ldots | \text{\textit{ilist}}
\]

\[
\text{Expressions } e ::= \ldots | \text{\textit{nil}} | \text{\textit{cons}(e_1,e_2)} | \text{\textit{listcase e of \textit{nil} }=} e_1 | \text{\textit{cons}(x,y) }=} e_2 \text{ end}
\]

The variables \textit{x} and \textit{y} are bound in \textit{e_2} in a \textit{listcase} expression.
MinML With Lists

Γ ⊢ nil : ilist
Γ ⊢ e₁ : int Γ ⊢ e₂ : ilist
Γ ⊢ e : ilist Γ ⊢ e₁ : τ Γ, x : int, y : ilist ⊢ e₂ : τ
Γ ⊢ listcase e of nil => e₁ | cons (x, y) => e₂ end : τ

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

• Form the pair $(n,l)$.
• Tag the result as a $\text{Cons}$.
• 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: $\text{Nil}$ or $\text{Cons}$.
• For $\text{Nil}$, pass to the empty case.
• For $\text{Cons}$, split the pair and pass to the non-empty case.

Lists as a Recursive Type

We will think of $\text{ilist}$ as the recursive type

$$\text{rec } \text{il} = \text{unit} + (\text{int} \times \text{ilist}).$$

It comes equipped with operations $\text{roll}$ and $\text{unroll}$:

• $\text{roll}(\cdot) : \text{unit} + (\text{int} \times \text{ilist}) \rightarrow \text{ilist}$.
• $\text{unroll}(\cdot) : \text{ilist} \rightarrow \text{unit} + (\text{int} \times \text{ilist})$.

Recursive Types as Pointers

The values of the recursive type are

• $\text{roll}(\text{inl}())$, corresponding to $\text{nil}$.
• $\text{roll}(\text{inr}((n,l)))$, 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 `roll(...)` to ...
- A tagged value `inl(...)` containing ...
- The null tuple `()`.

(Optimized representations are possible.)
Encoding Lists

Consider again the ilist type.

datatype ilist = Nil | Cons of int * ilist

We will think of this as the recursive type

typedef rec il is unit + "int*ila

The constructor nil is represented by the value

roll(inl()).

The constructor cons(n, l) is represented by the value

roll(inr((n, l))).

The case analysis

listcase e

of nil => e₁

| cons (n, l) => e₂

end

is represented by...

...the MinML code

case unroll(e) (chase the pointer, analyze tag)

of inl(x:unit) => e₁ (check for nil)

| inr(y:int*ilist) => (check for cons)

split y as (n, l) in e₂ end (get head and tail)

end

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!