Overview

Datatype values are built using constructors.

- e.g., 3::nil.
- e.g., node(empty,1,empty)

Datatype values are decomposed using pattern matching.

- fun depth (node (t1, _, t2)) = 1 + max(depth t1, depth t2)

Overview

To analyze these features of ML, we’ll start with these types:

- Product, or tuple, types.
- Sum, or disjoint union, types.

Then we’ll add recursive and, later, abstract types.

Product Types

Product, or tuple, types give you structured data.

- Nullary products: unit. Sole value is ()
- Binary products: $\tau_1 \times \tau_2$. Values are ordered pairs.
- $n$-ary products: $\tau_1 \times \cdots \times \tau_n$. Values are ordered $n$-tuples.
- Labelled products, or records: $\{\text{name: string, salary: float}\}$. Elements are labelled tuples.

We’ll formalize binary and nullary products.

Product Types: Abstract Syntax

Adding product types to MinML is easy.

$$
\begin{align*}
\text{Types} \quad \tau & ::= \ldots | \text{unit} | \tau_1 \times \tau_2 \\
\text{Expressions} \quad e & ::= \ldots | () | \text{check}_{c_1} \text{is} () \text{in} c_2 \mid (c_1, c_2) \\
\text{Values} \quad v & ::= \ldots | () \mid (v_1, v_2)
\end{align*}
$$

The variables $x$ and $y$ are bound within $c_2$ in the expression $\text{split}_{c_1} (x,y) \text{in} c_2 \text{end}$.
Product Types: Static Semantics

\[ \Gamma \vdash () : \text{unit} \]

\[ \Gamma \vdash e_1 : \text{unit} \Rightarrow \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash \text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end } : \tau \]

\[ \Gamma \vdash e_1 : \tau \times \tau \Rightarrow \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash \text{split } e_1 \text{ as } (x,y) \text{ in } e_2 \text{ end } : \tau \]

Product Types: Dynamic Semantics

\[ e_1 \Rightarrow e'_1 \]

\[ (e_1, e_2) \Rightarrow (e'_1, e'_2) \]

\[ e_2 \Rightarrow e'_2 \]

\[ (v_1, v_2) \Rightarrow (v'_1, v'_2) \]

\[ \text{split } e_1 \text{ as } (x,y) \text{ in } e_2 \text{ end } \Rightarrow \text{split } e'_1 \text{ as } (x,y) \text{ in } e_2 \text{ end} \]

\[ \text{split } (v_1,v_2) \text{ as } (x,y) \text{ in } e \text{ end } \Rightarrow [v_1,v_2/x,y] e \]

Product Types: Example

The split construct provides a single layer of pattern matching.

- No nested tuples.
- No possibility of failure.

Product Types: Safety

Preservation:

- By induction on evaluation.
- Using substitution lemma for split.

Progress:

- Canonical forms of product type are pairs.
- Can always split a pair of the right type.

ML code:

fun ifact (0, a) = a
| ifact (n, a) = ifact (n-1, n*a)

MinML code:

fun ifact (p:int*int) is
split p as (n, a) in
if n=0 then a else ifact (-(n,1), *(n, a)) fi
end
Sum Types

Sum, or disjoint union, types give you choices.

- Nullary: void, with no elements.
- Binary: \( \tau_1 + \tau_2 \). Values are either a value of type \( \tau_1 \) tagged inl, or a value of type \( \tau_2 \) tagged inr.
- n-ary: \( \tau_1 + \cdots + \tau_n \).
- Labelled: [present:string, absent:unit].

We'll consider nullary and binary sums.

Sum Types: Informal Description

The type \( \tau_1 + \tau_2 \) is the disjoint union of \( \tau_1 \) and \( \tau_2 \).

- Values of each type \( \tau_1 \) and \( \tau_2 \) are included within it.
- Elements are tagged with inl or inr to indicate where they came from.

Thus \( \text{int} + \text{int} \) is quite different from \( \text{int} \)!

- Elements are inl(n) and inr(n).
- Disjoint union is different from ordinary set union!

Sums: Static Semantics

\[
\begin{align*}
\Gamma, \tau_1 : \tau_1 & \vdash e_1 : \tau_1 \\
\Gamma, \tau_1 + \tau_2 (e_1) : \tau_1 + \tau_2 & \vdash e_2 : \tau_2 \\
\Gamma, \tau_1 + \tau_2 (e_2) : \tau_1 + \tau_2 & \vdash e_3 : \tau_3 \\
\Gamma, \text{case } v \text{ of } \text{inl}(x:\tau_1) & \Rightarrow e_1 | \text{inr}(y:\tau_2) \Rightarrow e_2 \text{ end } & \vdash e : \tau
\end{align*}
\]

Sums: Dynamic Semantics

\[
\begin{align*}
\text{inl}_1 + \text{inr}_2 (e) & \mapsto \text{inl}_1(e) \\
\text{inr}_1 + \text{inr}_2 (e) & \mapsto \text{inr}_1(e) \\
\text{case } v \text{ of } \text{inl}(x:\tau_1) & \Rightarrow e_1 | \text{inr}(x:\tau_2) \Rightarrow e_2 \text{ end } & \mapsto (v/x_1)e_1 \\
\text{case } v \text{ of } \text{inr}(x:\tau_1) & \Rightarrow e_1 | \text{inr}(x:\tau_2) \Rightarrow e_2 \text{ end } & \mapsto (v/x_2)e_2
\end{align*}
\]
Programming with Sums

Booleans are **definable** from sums!

- bool = unit+unit.
- true = inl(()), false = inr(()).
- if\(\epsilon\) then\(e_1\) else\(e_2\)fi =
  \[ case e of inl(x:unit) \Rightarrow e_1 | inr(x:unit) \Rightarrow e_2 \end{case} \]

\[ \]

Programming with Sums

Pattern matching corresponds to case analysis:

\[ \text{case } e \]
\[ \text{of } A \Rightarrow a \]
\[ | B \Rightarrow b \]
\[ | C(z) \Rightarrow c \]

\[ \]

Programming with Sums

In fact any **non-recursive** data type is similarly definable.

\[ \text{datatype } T = A | B | C \text{ of } \text{int} \]
\[ \]
\[ \text{• } T = \text{unit+unit+int} \]
\[ \text{• } A = \text{inl}(()), \text{• } B = \text{inr}((())) \]
\[ \text{• } C(n) = \text{inr}((n)) \]

\[ \]

Programming with Sums

Corresponding MinML code:

\[ \text{case } e \]
\[ \text{of } \text{inl}(x:unit) \Rightarrow a \]
\[ | \text{inr}(x:unit+int) \Rightarrow \]
\[ | \text{case } x \]
\[ \text{of } \text{inl}(y:unit) \Rightarrow b \]
\[ | \text{inr}(z:int) \Rightarrow c \]
\[ \text{end} \]
\[ \text{end} \]

\[ \]

Sums: Safety

**Preservation:** by induction on evaluation.

**Progress:** by induction on typing.

- Canonical forms of type \(\tau_1+\tau_2\): \(\text{inl}_{\tau_1+\tau_2}(\tau_1)\) or \(\text{inr}_{\tau_1+\tau_2}(\tau_2)\).
- Proof by induction on typing.

The exhaustiveness of case is crucial for progress!

Unit and Void

The type unit has **one** element, (). The type void has **no** elements! Consequently,

- If a function has type \(\text{int} \rightarrow \text{void}\), it **must not terminate** for any argument.
- If a function has type \(\text{int} \rightarrow \text{unit}\), it **might return**, but the result has to be ().

(Some languages use void when they mean unit . . . )
The Null Pointer

Many languages have a so-called null pointer or null object.

- The value null in Java.
- The cast (T*)0 in C.

The “null pointer” is used to model the absence of a value.
- Often as a default initial value for variables.
- As a “base case” for complex data structures.

The null pointer is a standard source of bugs.
- Null pointer exception in Java.
- Bus error in C.

Standard languages have no ability to track whether a pointer is null.
- Must check for null on each access.
- Explicit null checks do not change the type.

The Null Pointer

In ML there is a type distinction between
- A genuine value of type τ, and
- An optional value of type τ option.

The key to this is the presence of sum types.
- Case analysis changes the type from τ option to τ.
- The type system tracks whether a value is present or not! There is no need for a NONE check!

Skeletal ML code for working with options:

```ml
fun dispatch (x : τ option) =
    case x
    of NONE => ed
    | SOME (x' : τ) => ee

Within ee the variable x′ is known not to be “null”!
```

Skeletal Java code for working with null pointers:

```java
if (x == null)
    sf
else
    sf

Within sf the type of x is still Object and might still be null!
```
The Null Pointer

A harder case:

```java
if (MyMethod(x))
  s1
else
  s2
```

The compiler cannot (in general) track that MyMethod returning
false implies that x is non-null!

Summary

Products support structured data.

- Similar to struct's in C, but with automatic allocation and
  no "pointers".

Sums support alternative data.

- Choice of two distinguishable alternatives.
- Case analysis propagates type change.