Overview

The ML datatype mechanism combines

- sum, or disjoint union, types;

- recursive types;

- abstract types

into a single mechanism.
Overview

Datatype values are built using **constructors**.

- *e.g.*, $3::\text{nil}$.
- *e.g.*, $\text{node}(\text{empty}, 1, \text{empty})$

Datatype values are decomposed using **pattern matching**.

```plaintext
fun depth (node (t1, _, t2)) = 
  1 + \text{max}(\text{depth } t1, \text{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**: \{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.

Types \( \tau ::= \) unit | \( \tau_1 \times \tau_2 \)

Exp's \( e ::= () | \text{check } e_1 \text{ is } (\) in \( e_2 \) end | \( (e_1, e_2) | \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \) end

Values \( v ::= () | (v_1,v_2) \)

The variables \( x \) and \( y \) are bound within \( e_2 \) in the expression \( \text{split } e_1 \text{ as } (x,y) \text{ in } e_2 \) end.
Product Types: Static Semantics

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

\[ \Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash \text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end } : \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \ast \tau_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau \]

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

\[
\text{check}() \text{ is } () \text{ in } e \text{ end } \mapsto e
\]

\[
e_1 \mapsto e_1'
\]

\[
\text{check } e_1 \text{ is } () \text{ in } e_2 \text{ end } \mapsto \text{check } e_1' \text{ is } () \text{ in } e_2 \text{ end}
\]
Product Types: Dynamic Semantics

\[ e_1 \mapsto e'_1 \]
\[ (e_1, e_2) \mapsto (e'_1, e_2) \]

\[ e_2 \mapsto e'_2 \]
\[ (v_1, e_2) \mapsto (v_1, e'_2) \]

split \( (v_1, v_2) \) as \((x, y)\) in \( e \) end \mapsto \{ v_1, v_2/x, y \} e

\[ e_1 \mapsto e'_1 \]\n
split \( e_1 \) as \((x, y)\) in \( e_2 \) end \mapsto \text{split } e'_1 \text{ as } (x, y) \text{ in } e_2 \text{ end}
Product Types: Example

ML code:

\[
\text{fun ifact (0, a) = a} \\
| \text{ifact (n, a) = ifact (n-1, n*a)}
\]

MinML code:

\[
\text{fun ifact (p:int*int) is} \\
\quad \text{split p as (n, a) in} \\
\quad \quad \text{if n=0 then a else ifact (-(n,1), *(n, a)) fi}
\]
Product Types: Example

The \texttt{split} construct provides a \textit{single layer} of pattern matching.

- No nested tuples.
- No possibility of failure.
Product Types: Safety

Preservation:

• By induction on evaluation.

• Using substitution lemma for \textit{split}.

Progress:

• Canonical forms of product type are pairs.

• Can always split a pair of the right type.
Sum Types

Sum, or disjoint union, types give you choices.

- **Nullary**: \texttt{void}, with no elements.

- **Binary**: $\tau_1 + \tau_2$. Values are either a value of type $\tau_1$ tagged \texttt{inl}, or a value of type $\tau_2$ tagged \texttt{inr}.

- **n-ary**: $\tau_1 + \cdots + \tau_n$.

- **Labelled**: [present:string, absent:unit].

We'll consider nullary and binary sums.
Sum Types: Abstract Syntax

Types  \( \tau \) ::= void \mid \tau_1 + \tau_2

Exp's  \( e ::= \text{inl}_{\tau_1 + \tau_2}(e_1) \mid \text{inr}_{\tau_1 + \tau_2}(e_2) \mid \text{case}_\tau e_0 \text{ of inl}(x:\tau_1) => e_1 \mid \text{inr}(y:\tau_2) => e_2 \text{ end} \)

Val's  \( v ::= \text{inl}_{\tau_1 + \tau_2}(v_1) \mid \text{inr}_{\tau_1 + \tau_2}(v_2) \)

In the expression
\[
\text{case}_\tau e_0 \text{ of inl}(x:\tau_1) => e_1 \mid \text{inr}(y:\tau_2) => e_2 \text{ end},
\]
the variable \( x \) is bound in \( e_1 \) and the variable \( y \) is bound in \( e_2 \).
Sums: 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 \texttt{inl} or \texttt{inr} to indicate where they came from.

Thus $\texttt{int} + \texttt{int}$ is quite different from $\texttt{int}$!

- Elements are $\texttt{inl}(n)$ and $\texttt{inr}(n)$.

- Disjoint union is different from ordinary set union!
Sums: Informal Description

The case construct provides non-nested, exhaustive pattern matching over a sum type:

```plaintext
case e:int+int
  of inl(x:int) => +(x,1)
  | inr(y:int) => -(y,1)
```
Sums: Static Semantics

\[
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash \text{inl}_{\tau_1+\tau_2}(e_1) : \tau_1+\tau_2
\]

\[
\Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{inr}_{\tau_1+\tau_2}(e_2) : \tau_1+\tau_2
\]

\[
\Gamma \vdash e_0 : \tau_1+\tau_2 \\
\Gamma, x_1:\tau_1 \vdash e_1 : \tau \\
\Gamma, x_2:\tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case}_\tau e_0 \text{ of } \text{inl}(x_1:\tau_1) =\rangle e_1 | \text{inr}(x_2:\tau_2) =\rangle e_2 \text{ end} : \tau
\]
Sums: Dynamic Semantics

\[
\frac{e \mapsto e'}{\text{inl}_{\tau_1 + \tau_2}(e) \mapsto \text{inl}_{\tau_1 + \tau_2}(e')}
\]

\[
\frac{e \mapsto e'}{\text{inr}_{\tau_1 + \tau_2}(e) \mapsto \text{inr}_{\tau_1 + \tau_2}(e')}
\]

\[
case_{\tau_1 + \tau_2}(v) \text{ of } \text{inl}(x_1 : \tau_1) \Rightarrow e_1 \mid \text{inr}(x_2 : \tau_2) \Rightarrow e_2 \text{ end} \\
\mapsto \{v/x_1\}e_1
\]

\[
case_{\tau_1 + \tau_2}(v) \text{ of } \text{inl}(x_1 : \tau_1) \Rightarrow e_1 \mid \text{inr}(x_2 : \tau_2) \Rightarrow e_2 \text{ end} \\
\mapsto \{v/x_2\}e_2
\]
Booleans are **definable** from sums!

- bool = unit + unit.

- true = inl(), false = inr().

- if \( e \) then \( e_1 \) else \( e_2 \) fi =
  \[
  \text{case } e \text{ of inl}(x_1: \text{unit}) \Rightarrow e_1 | \text{inr}(x_2: \text{unit}) \Rightarrow e_2 \text{ end}
  \]
Programming with Sums

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

```
datatype T = A | B | C of int
```

- \( T = \text{unit} + (\text{unit} + \text{int}) \).
- \( A = \text{inl}() \).
- \( B = \text{inr}(\text{inl}()) \).
- \( C(n) = \text{inr}(\text{inr}(n)) \).
Pattern matching corresponds to case analysis:

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

Corresponding MinML code:

```ml
case e
  of inl(w:unit) => a
  | inr(x:unit+int) =>
      case x
        of inl(y:unit) => b
        | inr(z:int) => c
```

21
Sums: Safety

Preservation: by induction on evaluation.

Progress: by induction on typing.

- Canonical forms of type $\tau_1 + \tau_2$: $\text{inl}_{\tau_1 + \tau_2}(v_1)$ or $\text{inr}_{\tau_1 + \tau_2}(v_2)$.

- Proof by induction on typing.

The exhaustiveness of case is crucial for progress!
Unit and Void

The type \texttt{unit} has \textbf{one} element, (\texttt{)}. The type \texttt{void} has \textbf{no} elements! Consequently,

- If a function has type \texttt{int}→\texttt{void}, it \textbf{must not terminate} for any argument.

- If a function has type \texttt{int}→\texttt{unit}, it \textbf{might return}, but the result has to be (\texttt{}).

(Some languages use \texttt{void} when they mean \texttt{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

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

But these problems never arise in ML! Why?

• Absence of "pointer mentality" — value-oriented programming.

• Without pointers there are no null pointers!

Why are there no null pointers in ML?

• Sum types obviate the need for them!

• SML: datatype ’a option = NONE | SOME of ’a
The Null Pointer

In ML there is a type distinction between

- A **genuine** value of type $\tau$, and
- An **optional** value of type $\tau$ option.

The key to this is the presence of **sum types**.

- Case analysis **changes the type** from $\tau$ option to $\tau$.

- The type system tracks whether a value is present or not! There is no need for a **NONE** check!
The Null Pointer

Skeletal ML code for working with options:

\[
\text{fun dispatch (x : } \tau \text{ option)} = \\
\text{case x} \\
\text{of NONE => } e_0 \\
\text{| SOME (x' : } \tau \text{) => } e_1
\]

Within \(e_1\) the variable \(x'\) is \textbf{known} not to be “null”!
The Null Pointer

Skeletal Java code for working with null pointers:

```java
if (x == null)
    s1
else
    s2
```

Within $s_2$ the type of $x$ is still `Object` and might still be `null`!
The Null Pointer

A harder case:

```java
if (MyMethod(x))
    s_1
else
    s_2
```

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.