510: Programming Languages

Product and Sum Types

David Walker

Fall, 2002

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::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: $\tau_1 * \tau_2$. Values are ordered pairs.

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

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{unit} \mid \tau_1\texttt{*}\tau_2 \\
\\
\textit{Exp's} & e & ::= & \texttt{()} \mid \texttt{check}\, e_1 \,\texttt{is ()}\, \texttt{in}\, e_2 \,\texttt{end} \mid \\
& & & \texttt{(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{split}\, e_1 \,\texttt{as (}x\texttt{,}y\texttt{)}\, \texttt{in}\, e_2 \,\texttt{end} \\
\\
\textit{Values} & v & ::= & \texttt{()} \mid \texttt{(}v_1\texttt{,}v_2\texttt{)}
\end{array}
$$

The variables x and y are bound within e_2 in the expression $\texttt{split}\, e_1 \,\texttt{as (}x\texttt{,}y\texttt{)}\, \texttt{in}\, e_2 \,\texttt{end}$.

Product Types: Static Semantics

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{check}\, e_1\, \texttt{is ()}\, \texttt{in}\, e_2\, \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} : \tau_1\texttt{*}\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\texttt{*}\tau_2 \qquad \Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{split}\, e_1\, \texttt{as (}x\texttt{,}y\texttt{)}\, \texttt{in}\, e_2\, \texttt{end} : \tau}$$

Product Types: Dynamic Semantics

$$\frac{}{\texttt{check}\,()\,\texttt{is}\,()\,\texttt{in}\,e\,\texttt{end} \mapsto e}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{check}\,e_1\,\texttt{is}\,()\,\texttt{in}\,e_2\,\texttt{end} \mapsto \texttt{check}\,e_1'\,\texttt{is}\,()\,\texttt{in}\,e_2\,\texttt{end}}$$

Product Types: Dynamic Semantics

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

$$\frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')}$$

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

$$\frac{e_1 \mapsto e_1'}{\texttt{split}\,e_1\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2\,\texttt{end} \mapsto \texttt{split}\,e_1'\,\texttt{as}\,(x, y)\,\texttt{in}\,e_2\,\texttt{end}}$$

Product Types: Example

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)) f
```

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.

Sum Types

Sum, or **disjoint union**, types give you choices.

- Nullary: `void`, with **no** elements.
- Binary: τ_1+τ_2. Values are **either** a value of type τ_1 tagged `inl`, **or** a value of type τ_2 tagged `inr`.
- n-ary: τ_1+$\cdots$+τ_n.
- Labelled: `[present:string, absent:unit]`.

We'll consider nullary and binary sums.

Sum Types: Abstract Syntax

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{void} \mid \tau_1 \texttt{+} \tau_2 \\ \textit{Exp's} & e & ::= & \texttt{inl}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e_1\texttt{)} \mid \texttt{inr}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e_2\texttt{)} \mid \\ & & & \texttt{case}_\tau\, e_0\, \texttt{of inl(}x\texttt{:}\tau_1\texttt{) =>}\, e_1 \texttt{ | inr(}y\texttt{:}\tau_2\texttt{) =>}\, e_2\, \texttt{end} \\ \textit{Val's} & v & ::= & \texttt{inl}_{\tau_1 \texttt{+} \tau_2}\texttt{(}v_1\texttt{)} \mid \texttt{inr}_{\tau_1 \texttt{+} \tau_2}\texttt{(}v_2\texttt{)} \end{array}$$

In the expression

$$\texttt{case}_\tau\, e_0\, \texttt{of inl(}x\texttt{:}\tau_1\texttt{) =>}\, e_1 \texttt{ | inr(}y\texttt{:}\tau_2\texttt{) =>}\, e_2\, \texttt{end},$$

the variable x is bound in e_1 and the variable y is bound in e_2.

Sums: Informal Description

The type τ_1+τ_2 is the **disjoint union** of τ_1 and τ_2.

- Values of each type τ_1 and τ_2 are included within it.
- Elements are **tagged** with `inl` or `inr` to indicate where they came from.

Thus `int+int` is quite different from `int`!

- Elements are `inl`(n) and `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:

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

Sums: Static Semantics

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1\texttt{+}\tau_2}\texttt{(}e_1\texttt{)} : \tau_1\texttt{+}\tau_2}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1\texttt{+}\tau_2}\texttt{(}e_2\texttt{)} : \tau_1\texttt{+}\tau_2}$$

$$\frac{\Gamma \vdash e_0 : \tau_1\texttt{+}\tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}_\tau\ e_0\ \texttt{of inl(}x_1\texttt{:}\tau_1\texttt{) =>}\ e_1\ \texttt{| inr(}x_2\texttt{:}\tau_2\texttt{) =>}\ e_2\ \texttt{end} : \tau}$$

Sums: Dynamic Semantics

$$\frac{e \mapsto e'}{\texttt{inl}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e\texttt{)} \mapsto \texttt{inl}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e'\texttt{)}}$$

$$\frac{e \mapsto e'}{\texttt{inr}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e\texttt{)} \mapsto \texttt{inr}_{\tau_1 \texttt{+} \tau_2}\texttt{(}e'\texttt{)}}$$

$$\frac{}{\begin{array}{c}\texttt{case}_\tau\ \texttt{inl}_{\tau_1 \texttt{+} \tau_2}\texttt{(}v\texttt{)}\ \texttt{of}\ \texttt{inl(}x_1\texttt{:}\tau_1\texttt{)}\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{inr(}x_2\texttt{:}\tau_2\texttt{)}\ \texttt{=>}\ e_2\ \texttt{end} \\ \mapsto \{v/x_1\}e_1\end{array}}$$

$$\frac{}{\begin{array}{c}\texttt{case}_\tau\ \texttt{inr}_{\tau_1 \texttt{+} \tau_2}\texttt{(}v\texttt{)}\ \texttt{of}\ \texttt{inl(}x_1\texttt{:}\tau_1\texttt{)}\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{inr(}x_2\texttt{:}\tau_2\texttt{)}\ \texttt{=>}\ e_2\ \texttt{end} \\ \mapsto \{v/x_2\}e_2\end{array}}$$

Programming with Sums

Booleans are **definable** from sums!

- `bool` $=$ `unit+unit`.

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

- `if` e `then` e_1 `else` e_2 `fi` $=$
  `case` e `of` `inl(`x_1`:unit)` `=>` e_1 | `inr(`x_2`:unit)` `=>` e_2 `end`.

Programming with Sums

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

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

- $\texttt{T} = \texttt{unit+(unit+int)}$.
- $\texttt{A} = \texttt{inl(())}$.
- $\texttt{B} = \texttt{inr(inl(()))}$.
- $\texttt{C}(n) = \texttt{inr(inr(}n\texttt{))}$.

Programming with Sums

Pattern matching corresponds to case analysis:

```
case e
  of A => a
   | B => b
   | C(z) => c
```

Programming with Sums

Corresponding MinML code:

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

Sums: Safety

Preservation: by induction on evaluation.

Progress: by induction on typing.

- Canonical forms of type τ_1+τ_2: $\mathtt{inl}_{\tau_1+\tau_2}(v_1)$ or $\mathtt{inr}_{\tau_1+\tau_2}(v_2)$.

- Proof by induction on typing.

The exhaustiveness of `case` is crucial for progress!

Unit and Void

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

- If a function has type `int`$\rightarrow$`void`, it **must not terminate** for any argument.

- If a function has type `int`$\rightarrow$`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

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 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!

The Null Pointer

Skeletal ML code for working with options:

```
fun dispatch (x : τ option) =
    case x
      of NONE => e0
       | SOME (x' : τ) => e1
```

Within e_1 the variable x' is **known** not to be "null"!

The Null Pointer

Skeletal Java code for working with null pointers:

```
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:

```
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.