Aggregate Data Structures: Products

Programming languages usually offer a way to bundle values of (possibly) different types together to a new type:

- two floating point values represent a point in a two dimensional space
- ...
Haskell:

→ n-tuples:
  • declaration (not required): type Point = (Float, Float)
  • construction and access: let x = (1.75, 1.21) in fst x

→ named fields:
  • declaration: data Point = Point {x::Float, y::Float}
  • construction:
    - let p = Point 5.0 10.0 in ... 
    - let p = Point {x=5.0 y=10.0} in ...
  • construction:
    - access functions x,y:: Point -> Float automatically generated
    - x p returns value of x-field of p
C:

→ declaration:

```c
struct point {
    float x;
    float y;
};
```

→ usage

```c
struct point p;
p.x = 10.0;
```
Java: “Degenerate’ classes which consist only of data fields and contain no methods closest to C structs:

class Point {
    public float x;
    public float y;
}

Java: “Degenerate’ classes which consist only of data fields and contain no methods closest to C structs:

class Point {
    public float x;
    public float y;
}

→ Considered poor style by many OO-programmer
→ Alternative: protect data fields and provide methods to create, access and alter date structure
class Point {
    private float x;
    private float y;

    public Point (float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX () {return x;}
    public float getY () {return y;}

    public void setX (float x) {this.x = x;}
    public void setY (float y) {this.y = y;}
}
Products (or tuples) in MinHs:

To keep things as simple as possible, we add:

- no type declarations
- no named fields
- only pairs \((a_1, a_2)\), and
- nullary tuples () (we will see later what they are good for)

to MinHs
Products (or tuples) in MinHs:

To keep things as simple as possible, we add

- no type declarations
- no named fields
- only pairs \((a_1, a_2)\), and
- nullary tuples \(()\) (we will see later what they are good for)

to MinHs

**New MinHs types:**

- Unit: type of a nullary product
- \(\tau_1 \times \tau_2\): binary product with element types \(\tau_1\) and \(\tau_2\)
Concrete and abstract syntax:

- Constructors:

\[(e_1, e_2) \quad \text{pair}(e_1, e_2)\]
\[
() \quad \text{unitel}
\]

- Destructors:

\[\text{fst}(e) \quad \text{fst}(e)\]
\[\text{snd}(e) \quad \text{snd}(e)\]

- Types:

\[\tau_1 \times \tau_2 \quad \text{Cross}(\tau_1, \tau_2)\]
\[\text{Unit} \quad \text{Unit}\]
Example:

\[
\text{div} 
: (\text{Int}, \text{Int}) \rightarrow \text{Int} \\
\text{letfun } \text{div} \ (\text{args}) = \\
\quad \text{if } (\text{fst} \ (\text{args}) > \text{snd} \ (\text{args})) \ \text{then} \\
\quad \quad 0 \\
\quad \text{else } \text{div} \ (\text{fst} \ (\text{args}) - \text{snd} \ (\text{args}), \ \text{snd} \ (\text{args}))
\]
**Static Semantics: Typing Rules**

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

\[ \Gamma \vdash \text{pair}(e_1, e_2) : \text{cross}(\tau_1, \tau_2) \]

\[ \Gamma \vdash e : \text{cross}(\tau_1, \tau_2) \]

\[ \Gamma \vdash \text{fst}(e) : \tau_1 \]

\[ \Gamma \vdash \text{snd}(e) : \tau_2 \]

\[ \Gamma \vdash \text{unitel} : \text{Unit} \]
Dynamic Semantics: evaluation rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 \mapsto_M e'_1$</td>
<td>$\text{pair}(e_1, e_2) \mapsto_M \text{pair}(e'_1, e_2)$</td>
<td></td>
</tr>
<tr>
<td>$e_2 \mapsto_M e'_2$</td>
<td>$\text{pair}(v_1, e_2) \mapsto_M \text{pair}(v_1, e'_2)$</td>
<td></td>
</tr>
<tr>
<td>$e \mapsto_M e'$</td>
<td>$\text{fst}(e) \mapsto_M \text{fst}(e')$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{snd}(e) \mapsto_M \text{snd}(e')$</td>
<td></td>
</tr>
<tr>
<td>$\text{fst}(\text{pair}(v_1, v_2)) \mapsto_M v_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{snd}(\text{pair}(v_1, v_2)) \mapsto_M v_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SUM TYPES

Model types which can contain elements of either one type or another

Haskell:

Elements of type differ only in name (enumeration types):

```haskell
data Colour = Red | Green | Blue
```

Access: pattern matching or case-statement:

```haskell
toString :: Colour -> String
toString Red = "The colour red"
toString Blue = "The colour blue"
...
toString col = case col of
    Red   -> "The colour red"
    Blue  -> "The colour blue"
    Green -> "The colour green"
```
Different Content

```haskell
data Expr = Add Expr Expr
          | IntLit Int
          | FloatLit Float

data List a = Cons a (List a)
          | Nil

data Maybe a = Just a
               | Nothing
```
Sumtypes in C:

- Enumeration type
  
  ```c
  typedef enum {Red, Green, Blue} colour_t;
  ```

- "Maybe" type:
Sumtypes in C:

- Enumeration type
  
  ```c
  typedef enum {Red, Green, Blue} colour_t;
  ```

- “Maybe” type: just a pointer
  
  ```c
  int *ref;
  ```
Sumtypes in C:

- Enumeration type
  ```
  typedef enum {Red, Green, Blue} colour_t;
  ```

- “Maybe” type: just a pointer
  ```
  int *ref;
  ```

- Recursive types:
  ```
  typedef struct int_list {
      int elem;
      struct int_list *next;
  } int_list_t;
  ```
Different content

union {
    float f;
    int i;
} unsafe;

unsafe.f = 1.23421;
printf("%d ", unsafe.i);
Different content

union {
    float f;
    int   i;
} unsafe;

unsafe.f = 1.23421;
printf("%d ", unsafe.i);

In C, unions are often used with labels, maintained and checked only by programmer
typedef enum {AddExpr, IntLit, FloatLit} exprTag_t;

typedef struct {
    expr_t expr1;
    expr_t expr2;
} addExpr_t;

typedef struct {
    exprTag_t tag;
    union {
        int intLit;
        float floatLit;
        addExpr_t addExpr;
    }
} expr_t;
expr_t *expr;
...
switch (expr->tag) {
    case IntLit: ...
    case FloatLit: ...
    case AddExpr: ...
}
expr_t *expr;
...
switch (expr->tag) {
    case IntLit: ...
    case FloatLit: ...
    case AddExpr: ...
}

Java does not support unions or enumeration types directly

➡️ why?

➡️ how could a type like Colour or Expr be modelled in Java then?
abstract class Expr {
    abstract Value eval();
}

class AddExpr extends Expr {
    Expr expr1;
    Expr expr2;

    AddExpr (Expr expr1, Expr expr2) { this.expr1 ... Value eval () { ... 
    }
we only look at binary sums: either \( \tau_1 \) or \( \tau_2 \)

n-ary sums can be expressed by nesting binary sums
Concrete and abstract syntax:

→ Types:

\[ \tau_1 + \tau_2 \quad \text{sum}(\tau_1, \tau_2) \]

→ Constructors:

\[
\text{inl}(e_1) \quad \text{inl}(\tau_1, \tau_2, e_1) \\
\text{inr}(e_2) \quad \text{inr}(\tau_1, \tau_2, e_2)
\]

→ Operations:

\[
\text{case } e \text{ of } \text{inl}(x) \rightarrow e_1 \\
\text{inr}(y) \rightarrow e_2 \quad \text{case}(\tau_1, \tau_2, e, x.e_1, y.e_2)
\]
\textbf{Static Semantics: typing rules}

\[
\begin{align*}
\Gamma & \vdash e : \text{sum}(\tau_1, \tau_2) \quad \Gamma \cup \{x : \tau_1\} & \vdash e_1 : \tau \quad \Gamma \cup \{y : \tau_2\} & \vdash e_2 : \tau \\
\Gamma & \vdash \text{case}(\tau_1, \tau_2, e, x.e_1, y.e_2) : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_1 : \tau_1 \\
\Gamma & \vdash \text{inl}(\tau_1, \tau_2, e_1) : \text{sum}(\tau_1, \tau_2)
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_2 : \tau_2 \\
\Gamma & \vdash \text{inr}(\tau_1, \tau_2, e_2) : \text{sum}(\tau_1, \tau_2)
\end{align*}
\]
**Dynamic Semantics: Evaluation Rules**

(we omit the type annotations here)

\[
\begin{align*}
\text{inl}(e) & \mapsto_\text{M} \text{inl}(e') \\
\text{inr}(e) & \mapsto_\text{M} \text{inr}(e')
\end{align*}
\]

\[
\begin{align*}
\text{case}(e, x.e_1, y.e_2) & \mapsto_\text{M} \text{case}(e', x.e_1, y.e_2)
\end{align*}
\]

\[
\begin{align*}
\text{case}(\text{inl}(v), x.e_1, y.e_2) & \mapsto_\text{M} \{v/x\}e_1
\end{align*}
\]

\[
\begin{align*}
\text{case}(\text{inr}(v), x.e_1, y.e_2) & \mapsto_\text{M} \{v/y\}e_2
\end{align*}
\]
Which MinHs types correspond to the following Haskell types:

\[
\text{data Colors} = \text{Red} \mid \text{Green} \mid \text{Blue}
\]
ISOMORPHIC TYPES

Which MinHs types correspond to the following Haskell types:

```haskell
data Colors = Red | Green | Blue
```

We cannot define the same type in MinHs, but we can define an isomorphic type.

A type $\tau_1$ is isomorphic to a type $\tau_2$ iff there is a bijection between $\tau_1$ and $\tau_2$
ISOMORPHIC TYPES

Which MinHs types correspond to the following Haskell types:

```haskell
data Colors = Red | Green | Blue
```

We cannot define the same type in MinHs, but we can define an isomorphic type.

A type $\tau_1$ is isomorphic to a type $\tau_2$ iff there is a bijection between $\tau_1$ and $\tau_2$

Is isomorphic to

- $\text{unit + (unit + unit)}$
- $(\text{unit + unit}) + \text{unit}$
- all three types have exactly three elements
  - $\text{inl()}, \text{inr(inl ())}, \text{inr (inr ())}$
  - $\text{inl (inl ())}, \text{inl (inr ()), inr ()}$
  - Red, Green, Blue
**Recursive Types**

But there is no way to express a type isomorphic to this Haskell type in MinHs:

```haskell
data IntList = Nil | IList Int IntList
```

since we can’t express recursion so far!
We add a

→ type constructor:

\[
\text{rec } <\text{typeName}> \text{ is } <\text{type}>
\]

where \text{typeName} may occur anywhere in \text{type}

→ constructor: roll

→ destructor: unroll
We add a

→ type constructor:

\[ \text{rec } \langle \text{typeName} \rangle \text{ is } \langle \text{type} \rangle \]

where \( \text{typeName} \) may occur anywhere in \( \text{type} \)

→ constructor: roll

→ destructor: unroll

Example: Lists of integer values

→ type:

\[ \text{rec } \text{List is (Unit } + (\text{Int } \ast \text{ List})) \]

→ terms:

\[
\begin{align*}
\text{roll (inl () )} & \quad [] \\
\text{roll (inr (1, roll (inl ()) ) )} & \quad [1] \\
\text{roll (inr (2, roll (inr (1, roll (inl ()) ) ) ) )} & \quad [2,1]
\end{align*}
\]
Concrete and abstract syntax:

⇒ Types:

\[ \text{rec } t \text{ is } \tau \quad \text{rec}(t.\tau) \]

⇒ Constructor:

\[ \text{roll}(e) \]

⇒ Destructor:

\[ \text{unroll}(e) \]
**Static Semantics: Typing Rules**

\[
\begin{align*}
\Gamma \vdash e : \{\text{rec}(t.\tau)/t\}\tau \\
\Gamma \vdash \text{roll}(e) : \text{rec}(t.\tau)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \text{rec}(t.\tau) \\
\Gamma \vdash \text{unroll}(e) : \{\text{rec}(t.\tau)/t\}\tau
\end{align*}
\]
**Dynamic Semantics: Evaluation Rules**

\[
\begin{align*}
\frac{e \leftrightarrow^M e'}{	ext{roll}(e) \leftrightarrow^M \text{roll}(e')} & \quad \frac{e \leftrightarrow^M e'}{	ext{unroll}(e) \leftrightarrow^M \text{unroll}(e')} \\
\text{unroll}(\text{roll}(e)) \leftrightarrow^M e
\end{align*}
\]