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:
  - struct point {
    float x;
    float y;
  };
- usage
  - 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;
}
- Considered poor style by many OO-programmer
- Alternative: protect data fields and provide methods to create, access and alter data 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;}
}
**Static Semantics: Typing Rules**

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{pair}(e_1, e_2) : \text{cross}(\tau_1, \tau_2)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \text{cross}(\tau_1, \tau_2) & \quad \Gamma \vdash e : \text{cross}(\tau_1, \tau_2) \\
\Gamma \vdash \text{fst}(e) : \tau_1 & \quad \Gamma \vdash \text{snd}(e) : \tau_2
\end{align*}
\]

\[
\Gamma \vdash \text{unitel} : \text{Unit}
\]

**Dynamic Semantics: Evaluation Rules**

\[
\begin{align*}
e_1 \longrightarrow_u e'_1 & \quad e_2 \longrightarrow_u e'_2 \\
\text{pair}(e_1, e_2) \longrightarrow_u \text{pair}(e'_1, e'_2) & \quad \text{pair}(v_1, v_2) \longrightarrow_u \text{pair}(v'_1, v'_2)
\end{align*}
\]

\[
\begin{align*}
e \longrightarrow_u e' & \quad e \longrightarrow_u e' \\
\text{fst}(e) \longrightarrow_u \text{fst}(e') & \quad \text{snd}(e) \longrightarrow_u \text{snd}(e')
\end{align*}
\]

\[
\begin{align*}
\text{fst}(\text{pair}(v_1, v_2)) \longrightarrow_u v_1 & \quad \text{snd}(\text{pair}(v_1, v_2)) \longrightarrow_u v_2
\end{align*}
\]

**Sum Types**

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

Haskell:

- Elements of type differ only in name (enumeration types):
  
  data Colour = Red | Green | Blue

  Access: pattern matching or case-statement:
  
  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
  
  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: just a pointer
  ```c
  int *ref;
  ```
- Recursive types:
  ```c
  typedef struct {int elem; struct int_list *next; } int_list_t;
  ```

➜ Different content

```c
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

Slide 13

Slide 14

Slide 15

```c
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;
```

```c
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 () { ... }
}

Concrete and abstract syntax:

- Types:
  \[ \tau_1 + \tau_2 \rightarrow \text{sum(}\tau_1, \tau_2\text{)} \]

- Constructors:
  \( \text{inl} (e_1) \rightarrow \text{inl(} \tau_1, \tau_2, e_1 \text{)} \)
  \( \text{inr} (e_2) \rightarrow \text{inr(} \tau_1, \tau_2, e_2 \text{)} \)

- Operations:
  \( \text{case } r \text{ of } \text{inl}(x) \rightarrow e_1 \)
  \( \text{inr}(y) \rightarrow e_2 \text{ case}(\tau_1, \tau_2, e, x.e_1, y.e_2) \)

SUM TYPES

Slide 18 ➔ we only look at binary sums: either \( \tau_1 \) or \( \tau_2 \)
➔ n-ary sums can be expressed by nesting binary sums

Static Semantics: Typing Rules

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

Dynamic Semantics: Evaluation Rules

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

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

(we omit the type annotations here)

\[
\begin{align*}
& e \rightarrow_M e' \\
& \text{inl}(e) \rightarrow_M \text{inl}(e') \\
& \text{inr}(e) \rightarrow_M \text{inr}(e') \\
& e \rightarrow_M e' \\
& \text{case}(e, x, e_1, y, e_2) \rightarrow_M \text{case}(e', x, e_1, y, e_2) \\
& \text{case}(\text{inl}(v), x, e_1, y, e_2) \rightarrow_M [v/x]e_1 \\
& \text{case}(\text{inr}(v), x, e_1, y, e_2) \rightarrow_M [v/y]e_2
\end{align*}
\]

**Recursive Types**

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

\[
\text{data IntList} = \text{Nil} | \text{IList} \text{ Int} \text{ IntList}
\]

since we can’t express recursion so far!

**Isomorphic Types**

Which MinHs types correspond to the following Haskell types:

\[
\text{data Colors} = \text{Red} | \text{Green} | \text{Blue}
\]

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:

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

**Recursive Types**

We add a

- type constructor:
  \[
  \text{rec } <\text{name}> \text{ is } <\text{type}>
  \]
  where \text{name} may occur anywhere in \text{type}
  - constructor: \text{roll}
  - destructor: \text{unroll}

**Example: Lists of integer values**

- type:
  \[
  \text{rec List is (Unit + (Int \ast List))}
  \]
  - terms:
    \[
    \begin{array}{ll}
    \text{roll(inl)} & [] \\
    \text{roll (inr } 1, \text{roll(inl) ))) & [1] \\
    \text{roll (inr } 2, \text{roll (inr } 1, \text{roll(inl) ))) & [2, 1]
    \end{array}
    \]
Concrete and abstract syntax:

- **Types:**
  
  ```
  rec t is τ rec(t, τ)
  ```

- **Constructor:**
  
  ```
  roll(e)
  ```

- **Destructor:**
  
  ```
  unroll(e)
  ```

---

**Static Semantics: typing rules**

- Slide 26

  ```
  Γ ⊢ e : [rec(t, τ)/t]τ
  Γ ⊢ roll(e) : rec(t, τ)
  Γ ⊢ e : rec(t, τ)
  Γ ⊢ unroll(e) : [rec(t, τ)/t]τ
  ```

---

**Dynamic Semantics: evaluation rules**

- Slide 27

  ```
  e ⇒ e' roll(e) ⇒ roll(e')
  unroll(e) ⇒ unroll(e')
  unroll(roll(e)) ⇒ e
  ```