Utrecht University

Concepts of Programming Language Design

Composite and Algebraic Data Types

Gabriele Keller
Tom Smeding

Overview

higher & first-order syntax

inference rules, induction

tools to talk about languages

abstract machines

big step and small step operational semantics

value & type environments

composite types/
algebraic data types

control stacks

higher-order functions/
partial application/function closures

semantic features

functional

exception handling

static & dynamic scoping

static & dynamic typing

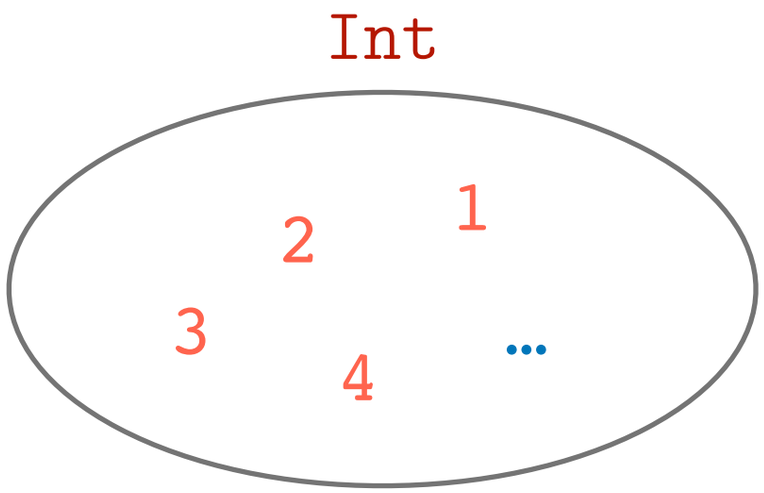
language concepts

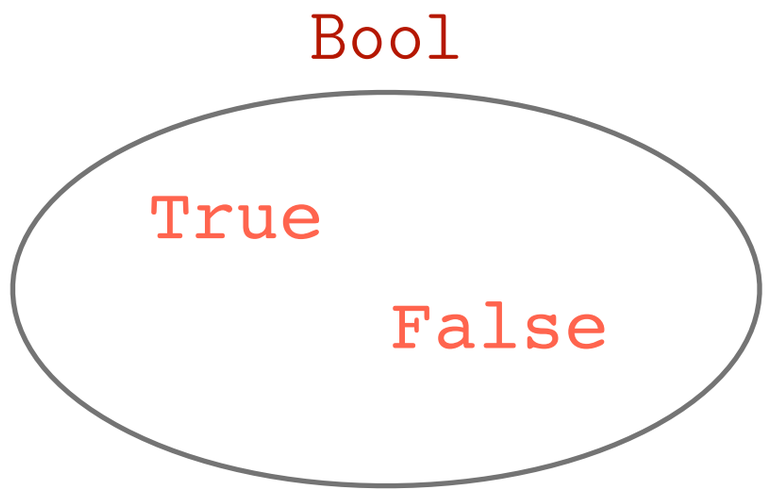
explicit typing

procedural/imperative

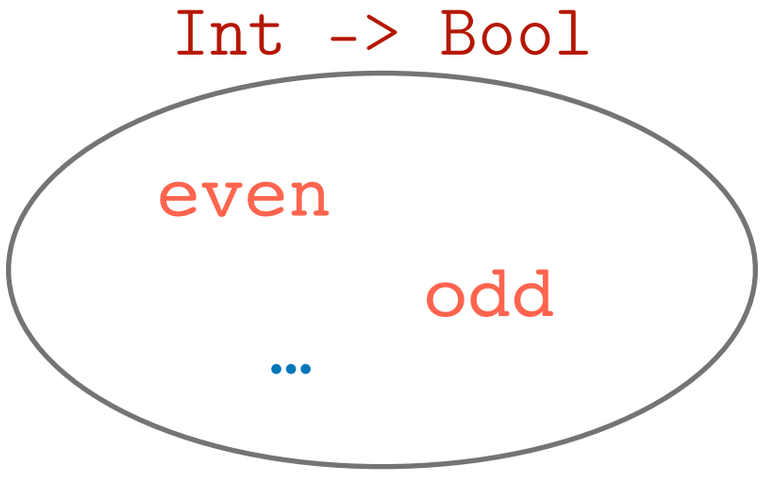
Composite types

- What are types?
  - Sets of values which share applicable operations
    - We've looked at some basic types, such as `Int`, `Bool`





    - and one *type operator*, `->`:



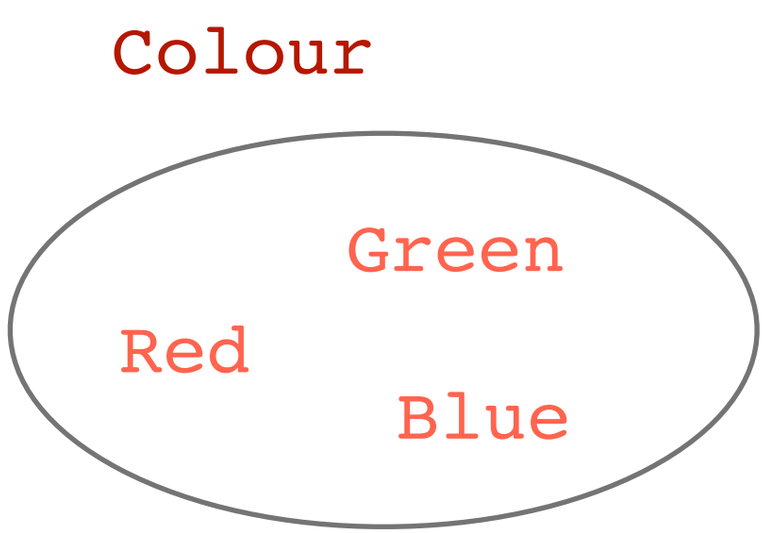
Composite types

- How can we define own types from scratch?
- What about other type (set) operators?
  - product of sets: A x B
  - union of sets: A ∪ B
  - power sets: $\mathcal{P}$(A)

- How does it work in different programming languages?
- Three main ways:
  - machine oriented (i.e., close to the actual representation)
  - object/data oriented
  - operation (functionality) centred

Defining our own type ‘from scratch’

- Enumeration types:
  - a new type with a finite number of elements
- Example: defining a new type to model colours



Defining our own type ‘from scratch’

- Many languages offer enumeration types as syntactic sugar over existing types (with various levels of static checks, different operations allowed):
  - C

```
typedef enum {Red = 1, Green = 2, Blue = 3} colour;
```

  - C#

```
enum Colour : byte {Red, Green, Blue};
```

Defining our own type ‘from scratch’

- In functional languages, like Haskell, it’s a regular algebraic data type, with pattern matching (other operations possible by deriving type class membership)

  - Haskell

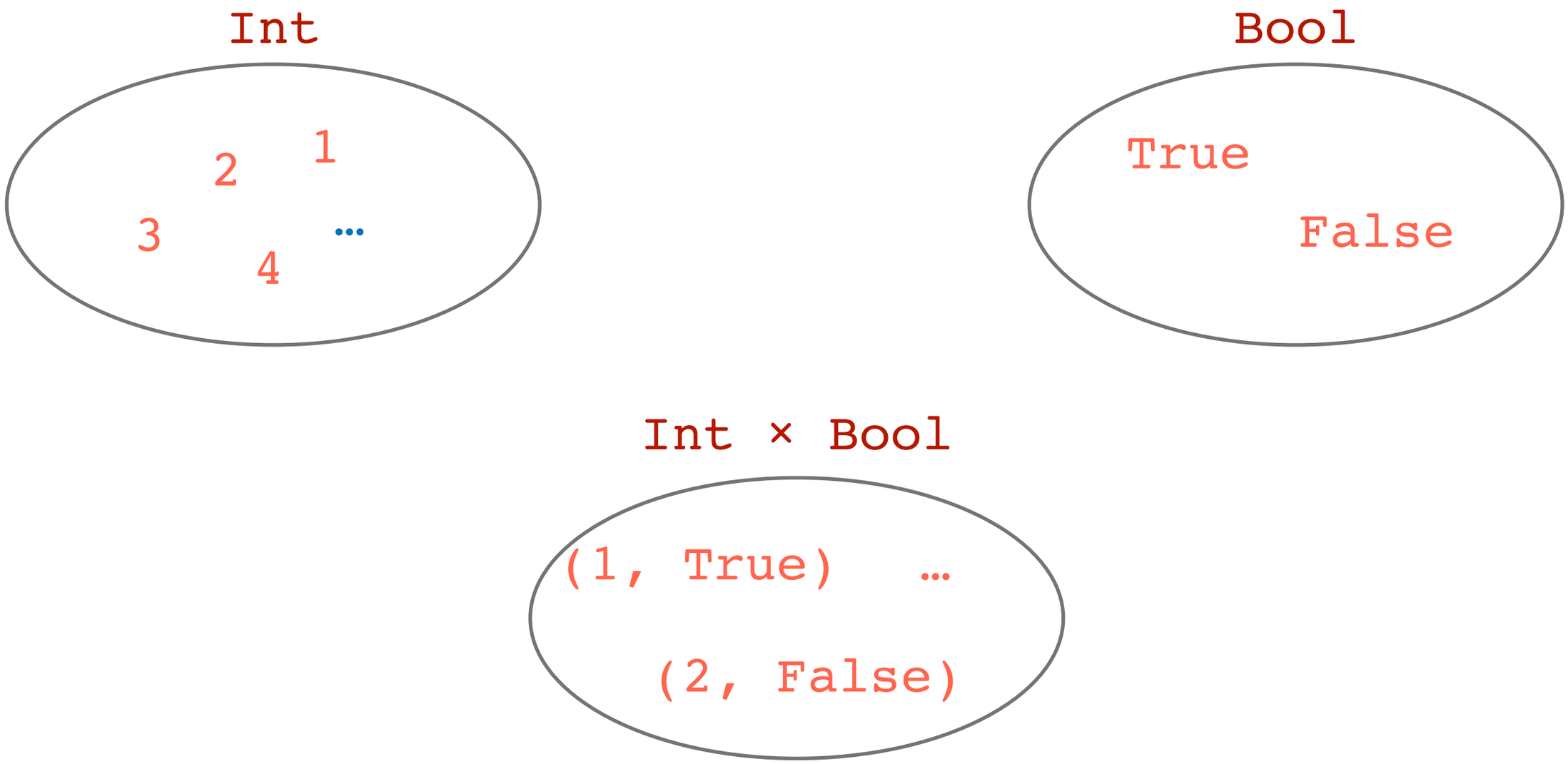
```haskell
data Colour = Red | Green | Blue
    deriving (Eq)
```

- Rust also allows pattern matching, choice of representation type, and associated methods

```rust
enum Colour { Red(i32),Green(i32),Blue(i23)};
```

Product types

- Defining a new type by combining values of existing types:



Tuples example: modelling a point in a 2D space

- Structs in C:

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

struct point middlePoint (
   struct point p1,
   struct point p2) {
   struct point mid;
   mid.x = (p1.x + p2.x)/2.0;
   mid.y = (p1.y + p2.y)/2.0;
   return mid;
}
```

Tuples example: modelling a point in a 2D space

- In C#

```
public struct Point {
  public float X {get; set;}
  public float Y {get; set;}

  public Point(float x, float y) {
              X = x;
              Y = y;
          }
  …
  }
```

Tuples example: modelling a point in a 2D space

- In Java
  - using degenerate classes in Java:

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

Point  middlePoint (Point p1, Point p2) {
  Point mid;
  mid.x = (p1.x + p2.x)/2.0;
  mid.y = (p1.y + p2.y)/2.0;
  return mid;
}
```

Tuples example: modelling a point in a 2D space

- In Haskell:
  - using simple pairs (tuples are built-in type constructors):

```
type Point = (Float, Float) - not necessary to define a type synonym

middlePoint:: Point -> Point -> Point
middlePoint (x1, y1) (x2, y2) =
  ((x1+x2)/2, (y1+y2)/2)

middlePoint' p1 p2 =
  ((fst p1 + fst p2)/2, (snd p1 + snd p2)/2)
```

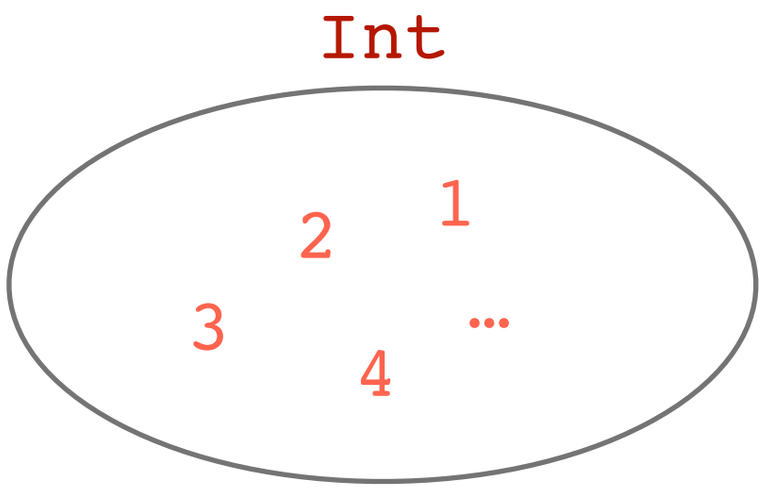
  - using algebraic data types (with unnamed and named fields):

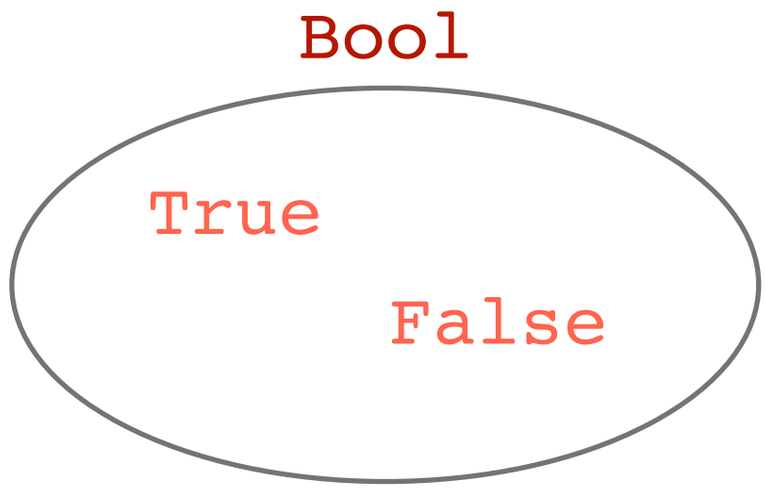
```
data Point = Point Float Float

middlePoint (Point x1 y1) (Point x2 y2) =
  Point ((x1+x2)/2) ((y1+y2)/2)
```

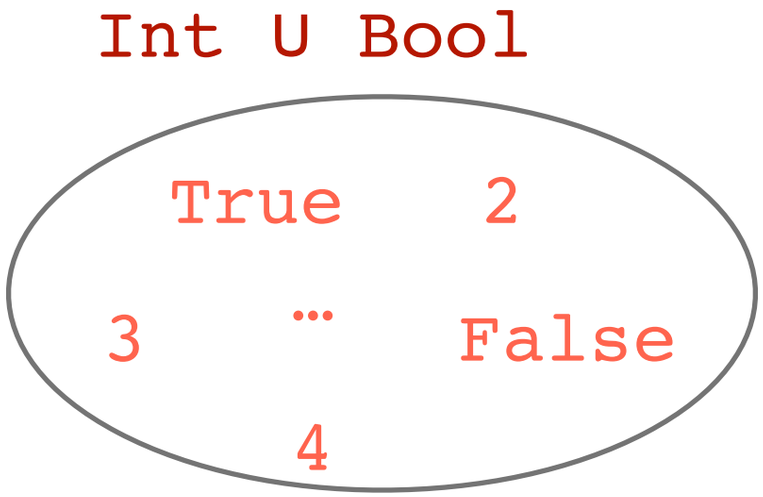
```
data Point = Point {x:: Float, y:: Float}

middlePoint (Point x1 y1) (Point x2 y2) =
  Point ((x1+x2)/2) ((y1+y2)/2)

middlePoint' p1 p2 = Point { x = (x p1 + x p2)/2
                           y = (x p2 + y p2)/2}
```

Composite types

- Composite types that offer alternatives of existing types



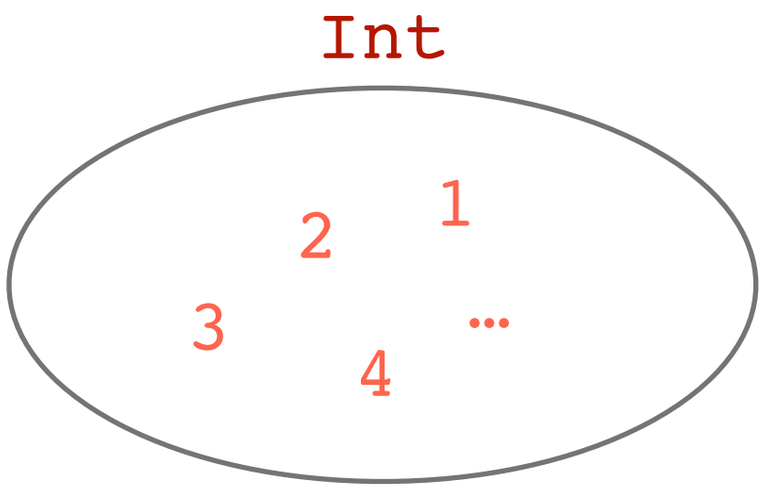


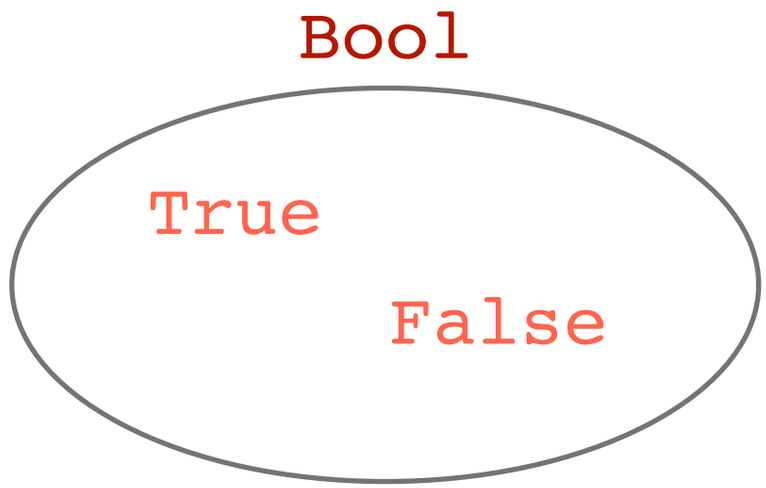


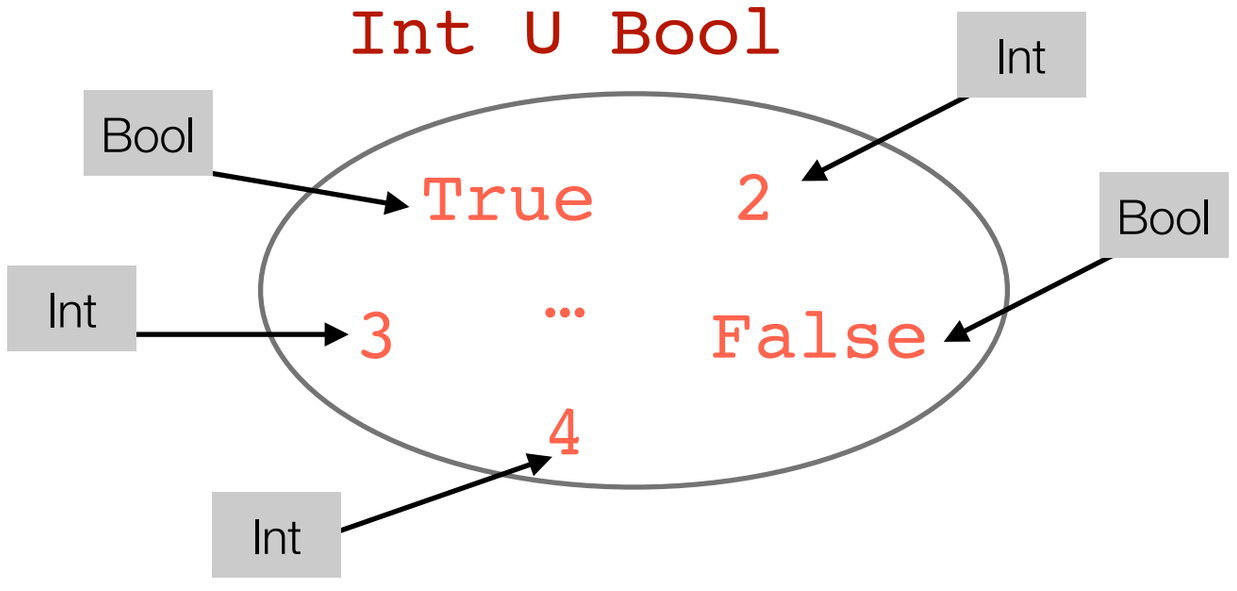
we need to
'label' the values for
type safety

Composite types

- Composite types that offer alternatives of existing types







we need to 'label' the values for type safety

Composite types

- Alternatives with varying component types in C:

```
union {
  int i;
  float f;
} unsafe;

unsafe.f = 1.23456;
printf (“the value is: %d”, unsafe.i);
```

```
the value is: 1067320848
```

Composite types

- Alternatives with varying component types in C:

```
typedef enum {I, F} valueTag;

typedef struct {
 valueTag tag;
 union {
    int   intLit;
    float floatLit; } val;
 } value_t;
```

```
value_t * val;
…
switch (val->tag) {
  case I: … val->IntLit…
  case F: … val->floatLit…
```

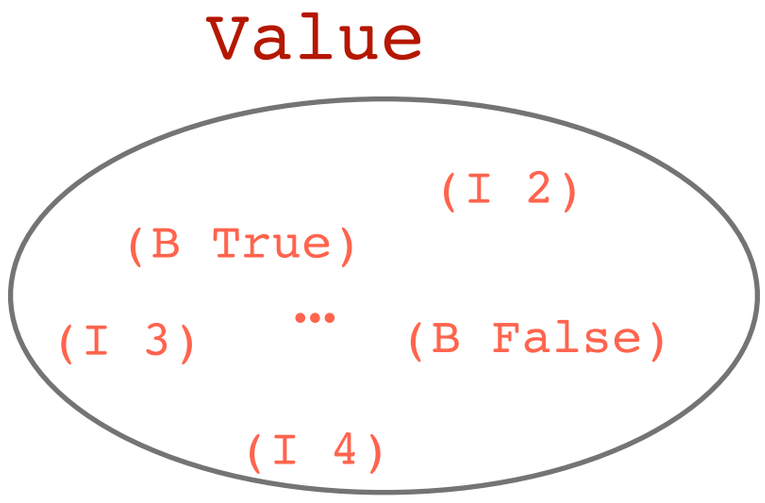
C makes things explicit which more abstract languages handle for you behind the scenes

more control for the programmer
but also
more ways to introduce bugs/undefined behaviour

Composite types

- In Haskell

```
data Value
  =  I Integer
  |  B Bool
```



Composite types

- Alternatives with varying component types in object oriented languages:

```
public abstract class Value {
    private.Value() {}
}

public class I: Value {
  public int V;
  public I(int v) {V = v;}
}

public class B: Value {
  public bool V;
  public B(bool v) {V = v;}
}
```

Collections

- Often, structured collections are needed
  - lists, trees, …
  - mappings from a key to a value
    - arrays
    - vectors
    - tables
- For lists and trees and such, we need a way to express recursion on the type level!

Recursive types

- Abstract view on a list of integers
  - an empty list is a list
  - if x is an integer, and xs a list of integers, then we can build a list with the head x and the tail xs
- C
  - space for recursive structures cannot be allocated statically
  - necessary to store the address of the (possibly empty) tail of a list

```
typedef struct list_node {
  int                 elem;
  struct list_node * next;
} int_list_t;
```


Recursive types

- C#

```
class ListNode {
    int      data;
    ListNode next;
    public ListNode(int d) {
        data = d;
        next = null;
    }}
```

- In Haskell (again, via algebraic data types)

```
data IntList
  =  ICons   Int IntList
  |  Nil
```

Utrecht University

Observation

- In an OO approach, we associate the operations directly with the new type
  - class declaration contains the methods
  - easy to extend the type by adding new subclasses
  - cumbersome to add new methods - we have to change each subclass
- In a functional approach, the operations can be defined anywhere (if the constructors are exported)
  - easy to add new functionality - just add the function anywhere in the program
  - cumbersome to add new variants - we have add a case to each function definition

Extending MinHs with support for composite types

- We add algebraic data types to MinHs
  - product types
  - sum types
  - recursive types
- no support for letting the user give new names to types
  - could be easily added

Algebraic Data Types

- Algebraic data types (ADTs) in combination with pattern matching are a convenient way to construct and decompose composite types
  - traditionally, used in functional languages (Haskell, ML, …)
  - also part now of many modern languages: Scala, Swift, Rust, C#
- Matching on simple constants:

Algebraic Data Types for MinHs: Products

- Products aka pairs in MinHs
  - minimal extension, only restricted case-pattern matching
  - no type declaration
  - no named fields
  - only pairs `(`e_1`,` e_2`)` and
  - nullary tuples/unit `()`
- New MinHs types:
  - `Unit`: singleton type with element `()`
  - τ_1 `*` τ_2`:` binary product type with element type τ_1 and τ_2
- Operations on these types:
  - `fst` and `snd` to extract the first/second component

Products in MinHs: Concrete and Abstract Syntax

- Constructors

`(`e_1`,` e_2`)`	`(Pair` e_1 e_2`)`
`()`	`()`

- Destructors

`fst` e	`(Fst` e`)`
`snd` e	`(Snd` e`)`

- Types

τ_1 `*` τ_2	τ_1 `*` τ_2
`Unit`	`Unit`

Products in MinHs

- Example:

```
recfun div :: ((Int * Int) -> Int) args =
  if (fst args < snd args)
    then 0
    else 1+ div (fst args - snd args, snd args)
```

- Side note:

  - what is the difference between these two (Haskell) functions:

```
average1 :: (Float, Float) -> Float
average1 (x, y) = (x+y)/2

average2 :: Float -> Float -> Float
average2 x y = (x+y)/2
```

Products in MinHs: Static Semantics

- Typing rules:

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

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{(Fst } e\texttt{)} : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{(Snd } e\texttt{)} : \tau_2}$$

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

Products in MinHs: Dynamic Semantics

- Evaluation rules (M-machine)
  - we add `(Pair` v_1 v_2`)` and `()` to the set of values/final states

$$\frac{e_1 \mapsto_M e_1'}{\texttt{(Pair } e_1 \ e_2\texttt{)} \mapsto_M \texttt{(Pair } e_1' \ e_2\texttt{)}} \qquad \frac{e_2 \mapsto_M e_2'}{\texttt{(Pair } v_1 \ e_2\texttt{)} \mapsto_M \texttt{(Pair } v_1 \ e_2'\texttt{)}}$$

$$\frac{e \mapsto_M e'}{\texttt{(Fst } e\texttt{)} \mapsto_M \texttt{(Fst } e'\texttt{)}} \qquad \frac{e \mapsto_M e'}{\texttt{(Snd } e\texttt{)} \mapsto_M \texttt{(Snd } e'\texttt{)}}$$

$$\frac{}{\texttt{(Fst (Pair } v_1 \ v_2\texttt{))} \mapsto_M v_1} \qquad \frac{}{\texttt{(Snd(Pair } v_1 \ v_2\texttt{))} \mapsto_M v_2}$$

Sum-types

- Sum-types to express alternatives in MinHs
  - we use binary sums:
    - $\tau_1 + \tau_2$: either τ_1 or τ_2 (products: both τ_1 and τ_2)
  - n-ary sums can be expressed by nesting
  - similarities to the Haskell type `Either`:

```haskell
data Either a b = Left a
                | Right b
```

Sum-types

- Types

$\tau_1 + \tau_2$ $\qquad\qquad$ $\tau_1 + \tau_2$

- Constructors

`Inl` e	`(Inl` τ_1 τ_2 e`)`
`Inr` e	`(Inr` τ_1 τ_2 e`)`

- Destructors (a very restricted form of pattern matching):

```
case e of
    Inl x -> e1
    Inr y -> e2
```

```
(Case τ1 τ2 e (x. e1) (y. e2))
```

Sums in MinHs: Static Semantics

- Typing rules:

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma \cup \{x : \tau_1\} \vdash e_1 : \tau \qquad \Gamma \cup \{y : \tau_2\} \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2)) : \tau}$$

Sums in MinHs: Dynamic Semantics

- Evaluation rules (M-machine), omitting the types for brevity
  - we add `(Inl` v`)` and `(Inr` v`)` to the set of final states/values

$$\frac{e \mapsto_M e'}{\texttt{(Inl } e\texttt{)} \mapsto_M \texttt{(Inl } e'\texttt{)}} \qquad \frac{e \mapsto_M e'}{\texttt{(Inr } e\texttt{)} \mapsto_M \texttt{(Inr } e'\texttt{)}}$$

$$\frac{e \mapsto_M e'}{\texttt{(Case } e\ (x.e_1)\ (y.e_2)\texttt{)} \mapsto_M \texttt{(Case } e'\ (x.e_1)\ (y.e_2)\texttt{)}}$$

$$\frac{}{\texttt{Case(Inl } v\texttt{) } (x.e_1)(y.e_2) \mapsto_M e_1[x:=v]}$$

$$\frac{}{\texttt{Case(Inr } v\texttt{) } (x.e_1)(y.e_2) \mapsto_M e_1[y:=v]}$$

Recursive Types

- What about the list type?

```
data IntList = Nil | ICons Int IntList
```

```
Unit + (Int *    )
```

we need a way to recursively refer to a type!

```
Rec t. Unit + (Int * t  )
```

we use notation like in higher-order abstract syntax to express that 't' is bound

Recursive types

- Types

`Rec` t `.` τ (Rec (t.τ))

- Constructor

`Roll` e (Roll e)

- Destructor

`unroll` e (Unroll e)

Examples

- List of integer values:
  - Type

```
Rec List. (Unit + (Int * List))
```

  - Terms

```
                                    Roll (Inl ())       []
                 Roll(Inr  (1,  (Roll (Inl ()))         [1]
Roll (Inr (1, Roll(Inr  (2,  (Roll (Inl ()))))         [1,2]
```

```
        ()  ::              Unit
      Inl ()  ::              Unit + (Int * Rec List. (Unit + (Int * List)))
Roll (Inl ()) :: Rec List. (Unit + (Int * List))

Inl () =  unroll (Roll (Inl ()) :: Unit + (Int * (Rec List. (Unit + (Int * List)))
```

Examples

```
recfun head
 :: ((Rec L = Unit + (Int * L)) -> Int) xs
 = case unroll xs of
     Inl unit -> 0
     Inr cons -> fst cons
```

```
recfun tail
  ::((Rec L = Unit + (Int * L)) -> (Rec L = Unit + (Int * L))) xs
  = case unroll xs of
      Inl unit -> Roll (Inl ())
      Inr cons -> snd cons
```

Recursive Types in MinHs: Static Semantics

- Typing rules:

$$\frac{\Gamma \vdash e : \quad \tau\ [t := \texttt{Rec}(t.\tau)]}{\Gamma \vdash (\texttt{Roll}\ e) : \quad \texttt{Rec}(t.\tau)}$$

$$\frac{\Gamma \vdash e : \texttt{Rec}(t.\tau)}{\Gamma \vdash (\texttt{Unroll}\ e) : \tau[t := \texttt{Rec}(t.\tau)]}$$

Sums in MinHs: Dynamic Semantics

- Evaluation rules (M-machine)
  - we add `(Roll` v`)` to the set of values/final states

$$\frac{e \mapsto_M e'}{\texttt{(Roll } e\texttt{)} \mapsto_M \texttt{(Roll } e'\texttt{)}}$$

$$\frac{e \mapsto_M e'}{\texttt{(Unroll } e\texttt{)} \mapsto_M \texttt{(Unroll } e'\texttt{)}}$$

$$\texttt{(Unroll (Roll } v\texttt{))} \mapsto_M v$$

A lazy interpretations of type constructors

- With our new algebraic data types, we added terms of the form

  - `(Pair` v_1 v_2`), (Inl` v`), (Inr` v`), (Roll` v`)`

  to the set of final states F, if $v, v_i \in F$, resulting in a strict interpretation

- Lazy interpretation: final states can have the form

  - `(Pair` e_1 e_2`), (Inl` e`), (Inr` e`), (Roll` e`)`

  for $e, e_i \in S$ (any legal state)

  Terms of this form are also said to be in *Weak Head Normal Form (WHNF)*

Products in MinHs: Dynamic Semantics

- Evaluation rules (M-machine), lazy
  - no rule for terms of the form `(Pair` e_1 e_2`)`

$$\frac{e \mapsto_M e'}{\texttt{(Fst } e\texttt{)} \mapsto_M \texttt{(Fst } e'\texttt{)}} \qquad \frac{e \mapsto_M e'}{\texttt{(Snd } e\texttt{)} \mapsto_M \texttt{(Snd } e'\texttt{)}}$$

$$\frac{}{\texttt{(Fst (Pair } e_1 \; e_2\texttt{))} \mapsto_M e_1} \qquad \frac{}{\texttt{(Snd(Pair } e_1 \; e_2\texttt{))} \mapsto_M e_2}$$

Recursive types in MinHs: Dynamic Semantics

- Evaluation rules (M-machine), lazy sum types

$$\frac{}{\texttt{Case(Inl}\ e\texttt{)}\ \texttt{(}x\texttt{.}e_1\texttt{)(}y\texttt{.}e_2\texttt{)} \mapsto_M e_1[x:=e]}$$

$$\frac{}{\texttt{Case(Inr}\ e\texttt{)}\ \texttt{(}x\texttt{.}e_1\texttt{)(}y\texttt{.}e_2\texttt{)} \mapsto_M e_2[y:=e]}$$

$$\frac{e \mapsto_M e'}{\texttt{(Case}\ e\ \texttt{(}x\texttt{.}e_1\texttt{)}\ \texttt{(}y\texttt{.}e_2\texttt{))} \mapsto_M \texttt{(Case}\ e'\ \texttt{(}x\texttt{.}e_1\texttt{)}\ \texttt{(}y\texttt{.}e_2\texttt{))}}$$

Recursive types in MinHs: Dynamic Semantics

- Evaluation rules (M-machine), lazy

$$\frac{}{\texttt{(Unroll (Roll } e\texttt{))} \mapsto_M e}$$

$$\frac{e \mapsto_M e'}{\texttt{(Unroll } e\texttt{)} \mapsto_M \texttt{(Unroll } e'\texttt{)}}$$

Isomorphic Types

- Type correspondence: which MinHs type corresponds to the following Haskell

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

- `Colour` is isomorphic to

  - `Unit + (Unit + Unit)` and also to

  - `(Unit + Unit) + Unit`

since all three types have exactly three elements

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

  - a type τ_1 is isomorphic to a type τ_2 iff there exists a bijection between τ_1 and τ_2

- Recursive types:

```
data Tree = Node Int Tree Tree | Leaf
```

  - `Rec tree . ((Int, (tree, tree)) + Unit)`

Isomorphic Types

- In actual programming languages, we want to have **named** user defined types which are distinguished by the type checker:

```
data Direction = North | South | East | West

data Colour    = Red   | Black | Blue | Yellow


data Vector = Vector Float Float

data Point  = Point Float Float
```

Isomorphic Types

- Type generic programming exploits the fact that all compound types are built from unit, products, sums, and recursion

  - think about writing a *show* function that works on any user-defined type in Haskell

  - covered in more detail in Advanced Functional Programming course