CSE 130 [Winter 2014] Programming Languages

Polymorphism

(Continued)

Ravi Chugh

UCSDCSE
Computer Science and Engineering

Feb 06

Announcements

- Midterm next Thurs (Feb 13) in class
  - Closed-book, Closed-notes
  - Practice questions at bottom of Schedule webpage

- Midterm Review Tues (Feb 11) in class
  - Come prepared with specific questions

Binary Search Trees

```
type ('a,'b) tree =
  | Leaf
  | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
```

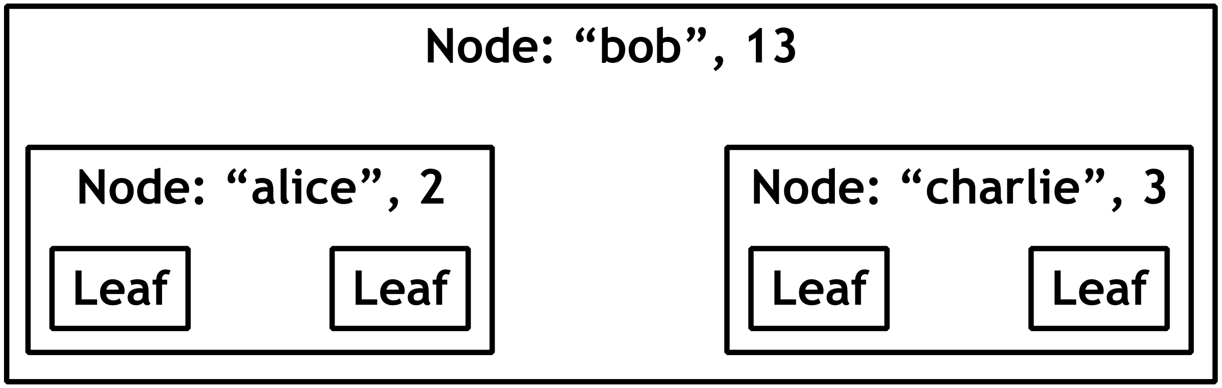
Node *(key, value, left, right)*

BST Property:

keys in left < key < keys in right

Binary Search Trees

BST Property: *keys in left < key < keys in right*



Node: “bob”, 13

Node: “alice”, 2

Leaf

Leaf

Node: “charlie”, 3

Leaf

Leaf

```
Node(“bob”, 13,
     Node(“alice”,2,Leaf,Leaf),
     Node(“charlie”,3,Leaf,Leaf))
```

Exercise: BST Lookup

BST Property: *keys in left < key < keys in right*

```
type ('a,'b) tree =
   | Leaf
   | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
```

Write a function to lookup keys…

```
val lookup: 'a -> ('a,'b) tree -> 'b option
```

Try this at home!

Try this at home!

```
type ('a, 'b) wierdlist =
    Nil
  | Cons 'a * ('b, 'a) wierdlist
```

Pick a well-typed expression from:

(a) `Cons(1, Cons("a", Cons(3.14, Nil)))`
(b) `Cons(1, Cons("a", Cons(1, Nil)))`
(c) `Cons(1, Cons("a", Cons("a", Nil)))`
(d) `Cons(1, Cons(1, Cons("a", Nil)))`
(e) `Cons(1, Cons(1, Cons(1, Nil)))`

Polymorphic Data Structures

- Container data structures independent of type !
- Appropriate type is instantiated at each use

’a list

(’a , ’b) tree

(’a , ’b) hashtbl

…

- Static type checking catches errors early
  - Cannot add int key to string hashtable
- Generics: in Java, C#, VB (borrowed from ML)

What does foo evaluate to?

```
# let foo x =
    if x > 0 then
      x * x
    else
      failwith “x is negative” ;;
???
```

(a) Type Error

(b) int -> string = <fun>

(c) int -> int = <fun>

(d) int -> 'a = <fun>

(e) 'a -> 'b = <fun>

`failwith`

`failwith :` *∀'a. string → 'a*

- Call to `failwith` can be instantiated to produce any type

```
# let foo x =
    if x > 0 then
      x * x
    else
      failwith “x is negative” ;;

val foo : int -> int = <fun>
```

- Can be used to “escape with fatal error”
- Very useful for building up skeleton code
  - Can use `failwith “…”` for sub-expressions not yet written, but can still type check overall function!

Recall our trusty friend, `fold`

“fold-right”
a.k.a. `List.fold_right`

```
let rec foldr f acc xs =
  match xs with
  | []     -> acc
  | x::xs’ -> f x (foldr f acc xs’)
```

(’a → ’b → ’b) → ’a list → ’b → ’b

“fold-left”
a.k.a. `List.fold_left`

```
let foldl f initAcc xs =
  let rec helper acc xs = match xs
  | []      -> acc
  | x::xs’  -> helper (f acc x) xs’
in helper initAcc xs
```

≈ $\boxed{\forall 'a,'b.}$ *(’a → ’b → ’b) → ’a list → ’b → ’b*

Recall our trusty friend, `fold`

“fold-right”
a.k.a. `List.fold_right`

```
let rec foldr f acc xs =
  match xs with
  | []       -> acc
  | x::xs’  -> f x (foldr f acc xs’)
```

(’a → ’b → ’b) → ’a list → ’b → ’b

“fold-left”
a.k.a. `List.fold_left`

```
let foldl f initAcc xs =
  let rec helper acc xs = match xs
  | []       -> acc
  | x::xs’  -> helper (f acc x) xs’
in helper initAcc xs
```

(’a → ’b → ’a) → ’a → ’b list → ’a

- A bit annoying that the interfaces differ…
- Let’s write a version of fold-left of type:

(’a → ’b → ’b) → ’a list → ’b → ’b

Recall our trusty friend, `fold`

“fold-right”
a.k.a. `List.fold_right`

```
let rec foldr f acc xs =
  match xs with
  | []       -> acc
  | x::xs’  -> f x (foldr f acc xs’)
```

(’a → ’b → ’b) → ’a list → ’b → ’b

“fold-left”
a.k.a. `List.fold_left`

```
let foldl f initAcc xs =
  let rec helper acc xs = match xs
  | []       -> acc
  | x::xs’  -> helper (f acc x) xs’
in helper initAcc xs
```

(’a → ’b → ’a) → ’a → ’b list → ’a

Try this at home!

- Let’s write a version of fold-left of type:

(’a → ’b → ’b) → ’a list → ’b → ’b

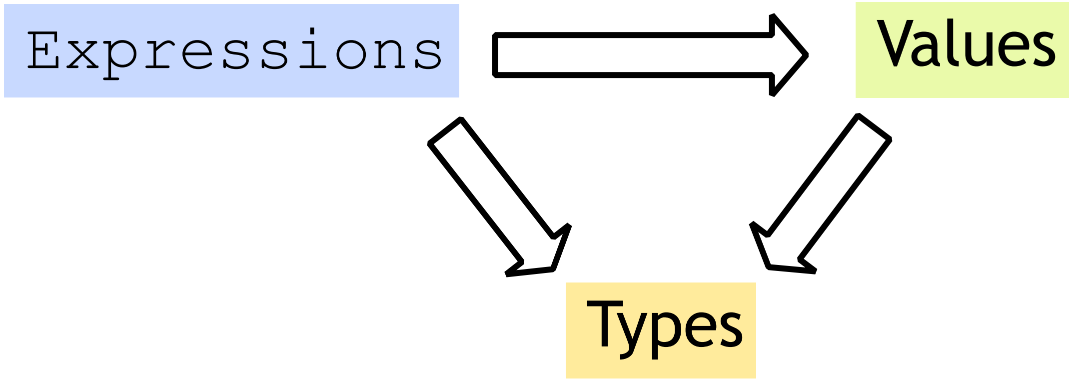
CSE 130 [Winter 2014]
Programming Languages

Type Inference

Ravi Chugh

UCSDCSE
Computer Science and Engineering

Holy Trinity



`Expressions`

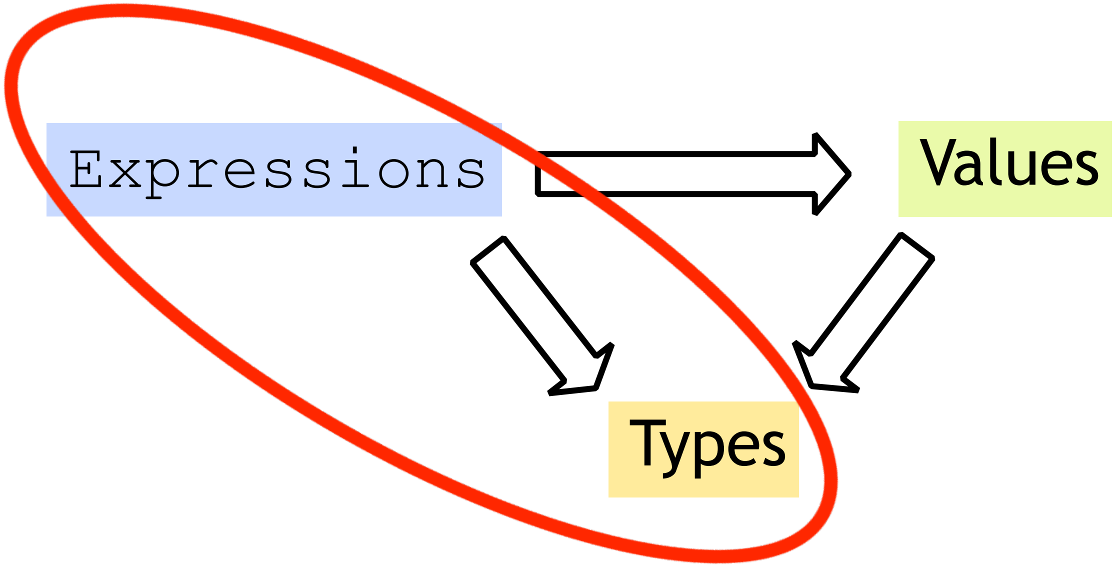
Values

Types

- Static types are an abstract description of, or a prediction about, run-time values

- Polymorphic function and data types are particularly useful

Holy Trinity



How DOES OCaml figure out all the types ?

Especially with polymorphism ?!

Type Inference Strategy

Process each binding in order

```
let x = e1 in e2
```

- Infer a type T_1 for `e1` (or TYPE ERROR)
- Remember that `x` : T_1 and
  Infer a type T_2 for `e2`

Type Inference Strategy

To infer the type of an expression `e`:

1. Recursively traverse `e` and collect “clues”
   - Clues are **equality constraints** between types
   - Introduce constraint variables (K_1, K_2, K_3, …) for unknown types, which will be computed

2. Solve constraints
   - If no solution is possible, then TYPE ERROR
   - If a constraint variable K is unconstrained, then **generalize** to polymorphic type variable (’a, ’b, ’c, …)

1. Collecting Clues

Simple expressions are simple:

`1` : *int*

`true` : *bool*

`(1, true)` : *(int * bool)*

1. Collecting Clues

More complex expressions:

If

$T_1 = T_2 = int$

Then

`e1 + e2` $: int$

1. Collecting Clues

More complex expressions:

If

$T_1 = bool$

$T_2 = T_3$

Then

`if e1 then e2 else e3` $: T_2$

1. Collecting Clues

More complex expressions:

If

T_0 = “the type of all the patterns”

$T_1 = T_2 = \ldots = T_n$

Then

```
match e0 with
  | pattern1 -> e1
  | pattern1 -> e2
  | ...       -> ...
```

$: T_1$

1. Collecting Clues

More complex expressions:

If

T_1 is of the form $T_{11} \rightarrow T_{12}$

$T_2 = T_{11}$

Then

`e1 e2` : T_{12}

1. Collecting Clues

Generate constraint variables for (possibly) polymorphic expressions:

Generate new K_{in} and K_{out}

Assume `x` : K_{in}

If

T₁ = “the solution for K_{out}”

Then

`fun x -> e1` : $K_{in} \rightarrow K_{out}$

1. Collecting Clues

Generate constraint variables for (possibly) polymorphic expressions:

Generate new *K*

Then

`[]` : *K list*

2. Solving Constraints

Chase down all equalities (a.k.a. “unification”)

`foo` $: K_1 \rightarrow K_2$

Constraints	Solution	Inferred Type
$K_1 = bool$ $K_2 = int$	$K_1 := bool$ $K_2 := int$	`foo` $: bool \rightarrow int$
$K_1 = bool$	$K_1 := bool$	`foo` $: bool \rightarrow {}'a$
$K_1 = bool$ $K_2 = int$ $K_1 = K_2$	*NONE !*	`foo` TYPE ERROR

Type Inference Strategy

- 2 steps:
  - Generate constraints while traversing expression
  - Solve constraints
- In practice, both steps done simultaneously
  - On paper, as well as in implementation
- Let’s try some examples!

```
# let x = 2 + 3;;
```

x : *int*

```
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
```

x *: int*

y *: string*

```
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
# let inc z = x + y ;;
```

`x` : *int*

`y` : *string*

`inc` : $K_z \rightarrow K_1$

Constraints

int = *int*

string = *int*

K_1 = *int*

Solution

NONE !

Inferred Type

TYPE ERROR

```
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
# let inc y = x + y ;;
```

- Notice the shadowing
- Rename to keep types/constraints separate

```
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
# let inc y' = x + y' ;;
```

`x` : *int*

`y` : *string*

`inc` : $K_{y'} \rightarrow K_1$

Constraints

$$int = int$$
$$K_{y'} = int$$
$$K_1 = int$$

Solution

$$K_{y'} := int$$
$$K_1 := int$$

Inferred Type

`inc` : *int* → *int*

```
# let choose b x y =
    if b then x else y;;
```

`choose` $: K_b \to K_x \to K_y \to K_1$

Constraints

$$K_b = bool$$
$$K_x = K_y$$
$$K_x = K_1$$

Solution

$K_b := bool$

$K_x := ANY$

$K_y := K_x$

$K_1 := K_x$

Inferred Type

`choose` $: bool \to 'a \to 'a \to 'a$

What’s the type of `foo`?

```
# let foo b x y =
    if x = y then 0 else 1;;
```

(a) `int`

(b) `bool -> ’a -> ’a -> int`

(c) `’a -> bool -> bool -> int`

(d) `’a -> ’b -> ’b -> int`

(e) Type Error

What's the type of `foo`?

```
# let foo b x y =
    if x = y then 0 else 1;;
```

`foo` : $K_b \to K_x \to K_y \to K_1$

Constraints

$$K_x = K_y$$
$$K_1 = int$$

Solution

$K_x := ANY$ $K_b := ANY$

$K_y := K_x$ $K_1 := int$

Inferred Type

`foo` : $'a \to 'b \to 'b \to int$

What’s the type of `bar`?

```
# let bar x =
    let (z, y) = x in
    z - y;;
```

(a) `(int -> int) * int`

(b) `int -> int * int`

(c) `int * int -> int`

(d) `int -> int -> int`

(e) Type Error

What's the type of `bar`?

```
# let bar x =
    let (z, y) = x in
    z - y;;
```

`bar` : $K_x \rightarrow K_1$

Constraints

$K_x = K_z * K_y$

- Introduce new constraint vars K_z and K_y for unknown tuple type

What's the type of `bar`?

```
# let bar x =
    let (z, y) = x in
    z - y;;
```

`bar` $: K_x \rightarrow K_1$

Constraints

$K_x = K_z * K_y$

$K_z = int$

$K_y = int$

$K_1 = int$

Solution

$K_1 := int$ $K_y := int$

$K_z := int$ $K_x := int * int$

Inferred Type

`foo` $: int * int \rightarrow int$

Recursive Bindings

```
# let rec cat xs =
   match xs with
     | []      -> “”
     | x::xs’  -> x^(cat xs’);;
```

cat $: K_{xs} \rightarrow K_1$

Constraints

$K_{xs} = K_2$ *list*

- Must be some kind of list
- Introduce new constraint var K_2

Recursive Bindings

```
# let rec cat xs =
    match xs with
      | []      -> “”
      | x::xs’  -> x^(cat xs’);;
```

cat $: K_{xs} \rightarrow K_1$

Constraints

$K_{xs} = K_2\ list$

$K_1 = string$

Recursive Bindings

```
# let rec cat xs =
   match xs with
     | []       -> ""
     | x::xs'   -> x^(cat xs');;
```

`cat` $: K_{xs} \rightarrow K_1$

Constraints

$K_{xs} = K_2\ list$ $K_{xs'} = K_2\ list$

$K_1 = string$

$K_x = K_2$

Recursive Bindings

```
# let rec cat xs =
   match xs with
     | []       -> “”
     | x::xs’  -> x^(cat xs’);;
```

`cat` $: K_{xs} \rightarrow K_1$

Constraints

$$K_{xs} = K_2\ list \qquad K_{xs'} = K_2\ list$$

$$K_1 = string \qquad K_x = string$$

$$K_x = K_2 \qquad K_1 = string$$

Recursive Bindings

```
# let rec cat xs =
    match xs with
      | []      -> “”
      | x::xs’  -> x^(cat xs’);;
```

`cat` $: K_{xs} \rightarrow K_1$

Constraints

$K_{xs} = K_2\ list$ $\quad$ $K_{xs'} = K_2\ list$

$K_1 = string$ $\quad$ $K_x = string$

$K_x = K_2$ $\quad$ $K_1 = string$

Solution

$K_1 := string$

$K_{xs} := string\ list$

…

`cat` $: string\ list \rightarrow string$

```
# let rec map f xs =
    match xs with
      | []       -> []
      | x::xs'  -> (f x)^(map f xs');;
```

`map` $: K_f \rightarrow K_{xs} \rightarrow K_1$

Constraints

$K_{xs} = K_2\ list$

$K_1 = K_3\ list$

$K_x = K_2$

$K_{xs'} = K_2\ list$

$K_1 = string$

$K_f = K_4 \rightarrow K_5$

$K_x = K_4$

$K_5 = string$

$K_f = K_f$

$K_{xs} = K_{xs'}$

Solution

TYPE ERROR!

string
not compatible with
$K_3\ list$

```
# let pipe x f = f x;;
```

`pipe` $: K_x \rightarrow K_f \rightarrow K_1$

Inferred Type `pipe` $: \text{'}a \rightarrow (\text{'}a \rightarrow \text{'}b) \rightarrow \text{'}b$

Constraints

$$K_f = K_2 \rightarrow K_3$$
$$K_x = K_2$$
$$K_3 = K_1$$

Solution

$$K_x := ANY$$
$$K_2 := K_x$$
$$K_3 := ANY$$
$$K_1 := K_3$$
$$K_f := K_x \rightarrow K_3$$

Some More Practice

```
# let compose f g = g (f x);;
```

```
# let foo1 f g x =
    if f x
    then x
    else g x;;
```

```
# let foo2 f g x =
    if f x
    then x
    else foo2 f g (g x);;
```

What does bar evaluate to?

```
# let bar x = failwith “!!!” ;;
???
```

(a) Type Error

(b) 'a -> string = <fun>

(c) 'a -> 'a   = <fun>

(d) 'a -> ’b   = <fun>