CSE 130 [Winter 2014] Programming Languages

Environments & Closures
(Continued)

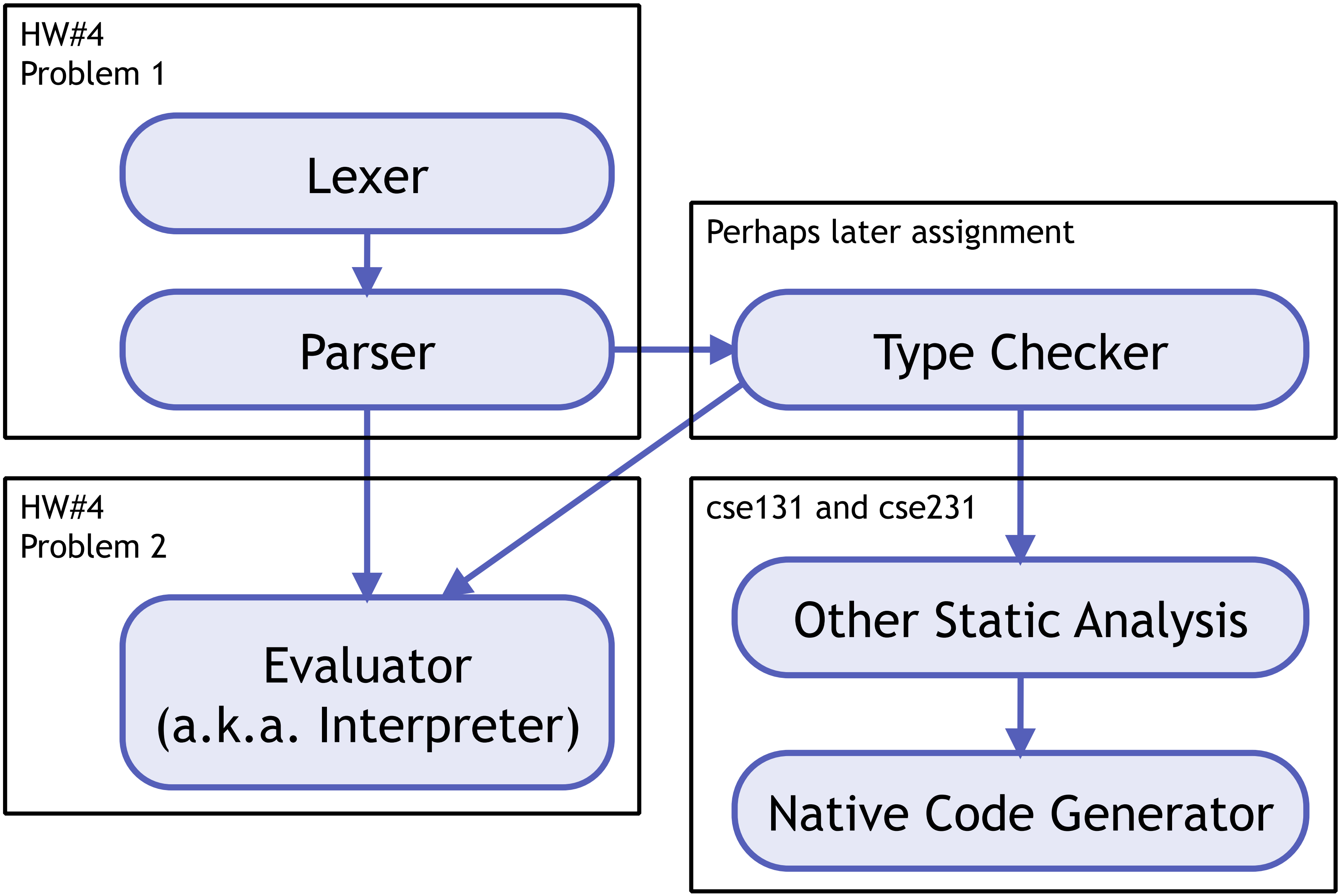
Ravi Chugh
UCSDCSE
Computer Science and Engineering

Feb 04

Announcements

- Midterm next Thursday (Feb 13) in class
  - Includes this week's topics:
    Polymorphism and Type Inference
  - Practice questions at bottom of Schedule webpage
- HW#4 due Friday Feb 21
  - Will be posted later today
  - Problem 1: Lexer/Parser for NanoML
  - Problem 2: Evaluator for NanoML

Structure of a PL Implementation



Announcements

- Midterm next Thursday (Feb 13) in class
  - Includes this week's topics:
    Polymorphism and Type Inference
  - Practice questions at bottom of Schedule webpage
- HW#4 due Friday Feb 21
  - Will be posted later today
  - Problem 1: Lexer/Parser for NanoML
  - Problem 2: Evaluator for NanoML
- Work on Problem 2 before midterm
  - Will help clarify environments and closures

Review Session?

(a) Sunday evening

(b) Monday evening

(c) Tuesday evening

(d) Tuesday in class (instead of lexing/parsing for HW#4)

(e) Never

Values of Functions = “Closures”

Two questions:

What is the value:

1. … of a function ? `fun x -> e`

Closure =

Code of Fun. (formal **x** + body ***e***)

\+ Environment at Fun. Definition

Functions

Values

Two questions:

What is the value:

1. … of a function ? `fun x -> e`

2. … of a function “application” (call) ? `(e1 e2)`

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Environment frozen inside function definition

Used to evaluate function application `(e1 e2)`

Which vars are needed from frozen env?

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Inside a function,

A "bound" occurrence:

1. Formal variable
2. Variable bound in **`let-in`**

`x`, **`y`**, **`g`** are "bound" inside **`f`**

A "free" occurrence:

- An unbound variable

`a` is "free" inside **`f`**

Frozen Environment

needed for values of free vars

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Bound values determined when function is evaluated (“called”)

- Arguments
- Local variable bindings

Values of Function Application

Value of a function "application" (call) `(e1 e2)`

1. Eval **`e1`** in current-env E_0 to a **closure**
   = **code** (formal **`x`** + body **`e`**) **+ closure-env** E

2. Eval **`e2`** in current-env E_0 to get (argument) $v2$

3. Evaluate body **`e`** in env E extended with **`x`** := $v2$

Q: What is the value of `res` ?

```
let f g =
   let x = 0 in
   g 2
;;

let x = 100;;

let h y = x + y;;

let res = f h;;
```

(a) Syntax Error

(b) `102`

(c) Type Error

(d) `2`

(e) `100`

Immutability: The Colbert Principle

"A function behaves the same way on Wednesday as it behaved on Monday, *no matter what happened on Tuesday!*"

Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment
- Static/Lexical: Determined from the program text
  - Without executing the program
- Very useful for readability, debugging:
  - Don't have to figure out "where" a variable got assigned
  - Unique, statically known definition for each occurrence

CSE 130 [Winter 2014] Programming Languages

Polymorphism

Ravi Chugh

UCSDCSE

Computer Science and Engineering

What type to assign to … ?

```
# let id x = x ;;
```

int → int
bool → bool
*int * bool → int * bool*
(int → bool) → (int → bool)
…

These types are too specific

What type to assign to … ?

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

$'a \rightarrow 'a$

Read “**For every** type $'a$,
`id` has type $'a \rightarrow 'a$ ”

Type variables prefixed with single quote

What type to assign to … ?

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

$'a \to 'a \quad \approx \quad \forall 'a.\ 'a \to 'a$

Implicit quantification over
every type $'a$

What type to assign to … ?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>
# let snd x y = y ;;
```

$'a \to 'b \to 'a$

$\approx \quad \forall 'a, 'b.\ \ 'a \to 'b \to 'a$

Implicit quantification over **every type** *'a* and *'b*

What type to assign to … ?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>
# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>
```

Polymorphism

- Poly = *many*, morph = *kind*
- Also called “forall types”
- Also called “parametric polymorphism”
- Type vars can be instantiated with any type
  - ML infers instantiations automatically!

```
# id 5 ;;
- : int = 5
```

≈

```
id[int] 5
```

```
# id (fun x -> x+1) ;;
- : int -> int = <fun>
```

≈

```
id[int->int] …
```

Polymorphism

`id ::`	$\forall$'*a*. '*a* → '*a*
`id[int] ::`	*int* → *int*
`id[int->int] ::`	*(int* → *int)* → *(int* → *int)*

Instantiation is **always implicit**,
OCaml doesn't allow explicit type arguments

```
# id 5 ;;
- : int = 5
```

≈ `id[int] 5`

```
# id (fun x -> x+1) ;;
- : int -> int = <fun>
```

≈ `id[int->int] …`

What does `res1` evaluate to?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>

# let res1 = fst 17 ;;
???
```

(a) Type Error

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

(c) 'b -> int = <fun>

(d) 'b -> 'b = <fun>

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

What does `res1` evaluate to?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>

# let res1 = fst 17 ;;
???
```

(a) Type Error

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

(c) 'b -> int = <fun>

(d) 'b -> 'b = <fun>

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

What does `res2` evaluate to?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>
# let res2 = snd 17 ;;
???
```

(a) Type Error

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

(c) 'b -> int = <fun>

(d) 'b -> 'b = <fun>

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

What does `res2` evaluate to?

```
# let fst x y = x ;;
val fst : 'a -> 'b -> 'a = <fun>

# let snd x y = y ;;
val snd : 'a -> 'b -> 'b = <fun>
# let res2 = snd 17 ;;
???
```

(a) Type Error

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

(c) 'b -> int = <fun>

(d) 'b -> 'b = <fun>

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

Catch #1

- What's with underscores in type vars ???
- We won't go into it in this course…

  If you're really curious: `http://mlton.org/ValueRestriction`

- For our purposes, assume: ’_*a* ≈ ’*a*

```
# let res1 = fst 17 ;;
val res1 : '_a -> int = <fun>
# let res2 = snd 17 ;;
val res2 : '_a -> '_a = <fun>
```

Catch #2

- Where did the ’b variables go ???
- Forall types are equivalent up to renaming

$$'a \to int \;\approx\; 'b \to int \;\approx\; 'c \to int$$

$$'a \to 'a \;\approx\; 'b \to 'b \;\approx\; 'c \to 'c$$

```
# let res1 = fst 17 ;;
val res1 : 'a -> int = <fun>
# let res2 = snd 17 ;;
val res2 : 'a -> 'a = <fun>
```

Catch #2

- Where did the ’b variables go ???
- Forall types are equivalent up to renaming

$$'a \to int \;\approx\; 'b \to int \;\approx\; 'c \to int$$

$$'a \to 'a \;\approx\; 'b \to 'b \;\approx\; 'c \to 'c$$

$$'a \to 'b \to 'a \;\approx\; 'b \to 'a \to 'b$$

- ML infers type variable names from left to right, starting with ’a, ’b, ’c, …

Polymorphism Enables Reuse

rev	:	*'a list → 'a list*
len	:	*'a list → int*
swap	:	*'a * 'b → 'b * 'a*
sort	:	*('a → 'a → bool) → 'a list → 'a list*
filter	:	*('a → bool) → 'a list → 'a list*
map	:	*('a → 'b) → 'a list → 'b list*
foldl	:	*('a → 'b → 'a) → 'a → 'b list → 'a*
foldr	:	*('a → 'b → 'b) → 'a list → 'b → 'b*

- If function (algorithm) is independent of type, can reuse generic code for all types !

Polymorphic Data Types

- Data types are also polymorphic!

```
type 'a list =
   | Nil
   | Cons of 'a * 'a list
```

- Type variable is instantiated for each use:

`Cons(1,Cons(2,Nil))`	*: int list*
`Cons("a",Cons("b",Nil))`	*: string list*
`Cons((1,2),Cons((3,4),Nil))`	*: (int*int) list*
`Nil`	*: 'a list*

Option Types

- Consider the following type:

```
type 'a option =
   | None
   | Some of 'a
```

- How could this be useful?

Option Types

- Remember this function?

```
val assoc: 'b -> 'a -> ('a*'b) list -> 'b
```

- We had to pass in a "default" value (Yuck!)

- Instead, we could return an option
  - **None** denotes "failure"
  - **Some** denotes "success"

```
val assoc: 'a -> ('a*'b) list -> 'b option
```

Exercise: Simple Calculator

```
type expr =
   | Num of int
   | Add of expr * expr
   | Div of expr * expr
```

Let’s write a function

```
val eval : expr -> int
```

Exercise: Simple Calculator

```
type expr =
   | Num of int
   | Add of expr * expr
   | Div of expr * expr
```

Let's write a function

```
val safeEval : expr -> int option
```

… that returns **None** if a div-by-zero occurs

Syntax for datatype with multiple type variables

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

Q: What is the type of x ?

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

let x = Node ("alice", 5, Leaf, Leaf)
```

(a) `(int, string) tree`

(b) `('a,'b) tree`

(c) `int tree`

(d) `string tree`

(e) `(string, int) tree`

Multiple Type Variables

- Data types can have multiple type vars

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

- Type variables instantiated for each use:

`Node("alice",2,Leaf,Leaf)`	: *(string, int) tree*
`Node("charlie",3,Leaf,Leaf)`	: *(string, int) tree*

`Node("bob",13,`
`Node("alice",2,Leaf,Leaf),`
`Node("charlie",3,Leaf,Leaf))`
	: *(string, int) tree*

Multiple Type Variables

- Data types can have multiple type vars

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

- Type variables instantiated for each use:

`Node("alice",2,Leaf,Leaf)`	*: (string, int) tree*
`Node("charlie",3,Leaf,Leaf)`	*: (string, int) tree*
`Node("bob",13, Node("alice",2,Leaf,Leaf), Node(3,"charlie",Leaf,Leaf))`	TYPE ERROR

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

Binary Search Trees

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

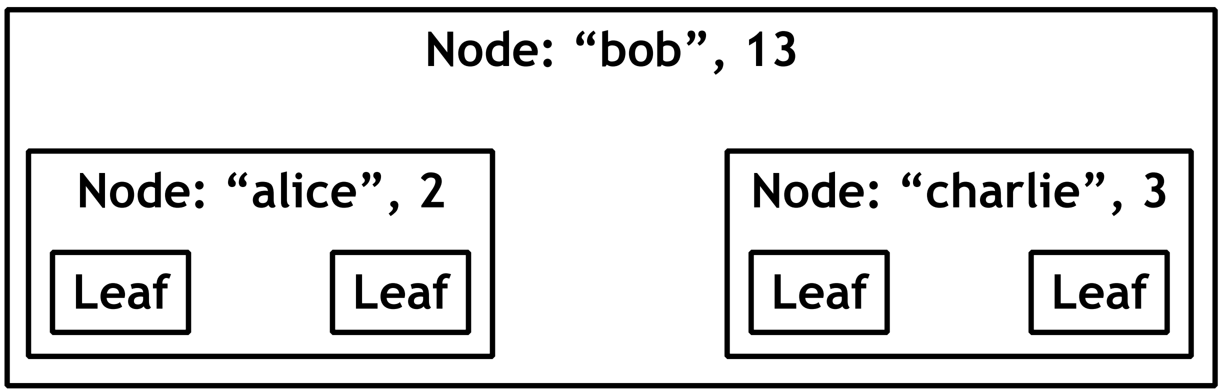
Node *(key, value, left, right)*

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))
```

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
```

Clicker Question

```
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 ::` $\forall$*'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!

failwith

- OCaml has a more general exception handling mechanism
  - `raise e`
  - `try e with pat1 -> e1 | … | patn -> en`

- But we won't study this any further…

What does bar evaluate to?

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

(a) Type Error

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

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

(d) 'a -> ’b = <fun>

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
```

$\approx$ $\boxed{\forall \text{'a}, \text{'b}.}$ *('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