

CSE 130 : Spring 2011

Programming Languages

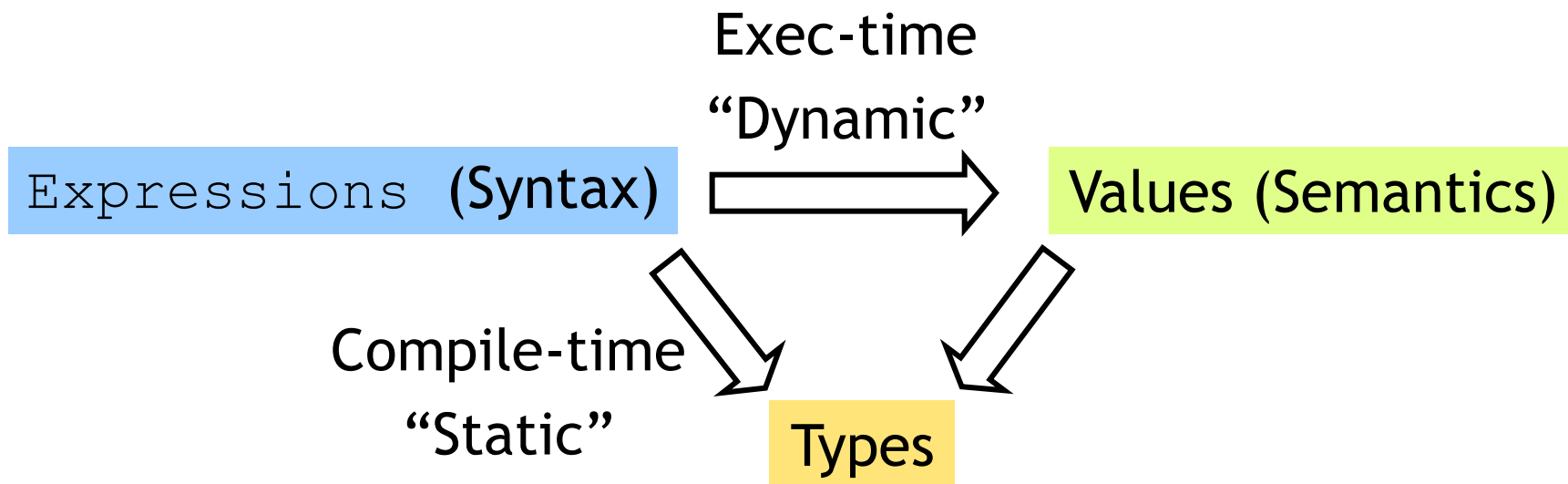
Lecture 5:

Functions and Closures

Ranjit Jhala
UC San Diego



Recap

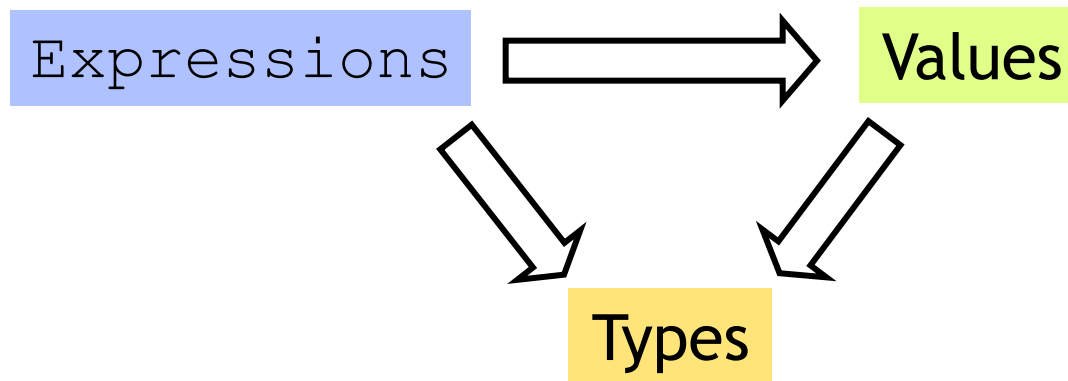


1. Programmer enters expression
2. ML checks if expression is “well-typed”
 - Using a precise set of rules, ML tries to find a unique meaningful type for the expression
3. ML evaluates expression to compute value
 - Of the same “type” found in step 2

Recap

- Variables are **names** for **values**
 - Environment: dictionary/phonebook
 - Most recent binding used
 - Entries never changed, new entries added
- Environment **frozen at fun definition**
 - Re-binding variables cannot change a function
 - Same I/O behavior at every call

Next: Functions



Q: What's a **function expression** ?

Q: What's the **value** of a **function** ?

Q: What's the **type** of a **function** ?

Functions

Expressions

Two ways of writing function expressions:

1. Named functions:

	Parameter (formal)		Body Expr	
let	fname	x	=	e

2. Anonymous functions:

	Parameter (formal)		Body Expr		
let	fname	= fun	x	=>	e

Function Application

Expressions

Application: fancy word for “call”

$(e1\ e2)$

- Function value $e1$
- Argument $e2$
- “apply” argument $e2$ to function value $e1$

Functions

Type

The type of any function is:

- $T1$: the type of the “input”
- $T2$: the type of the “output”

$T1 \rightarrow T2$

let fname = **fun**

$x \rightarrow e$

$T1 \rightarrow T2$

let fname

$x = e$

$T1 \rightarrow T2$

Functions

Type

The type of any function is:

- $T1$: the type of the “input”
- $T2$: the type of the “output”

$T1 \rightarrow T2$

$T1$, $T2$ can be **any** types, including functions!

Function Type Examples

- What is the type of...

```
fun x -> x + 1
```

```
int -> int
```

```
fun f -> f 0 + 1
```

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

Type of function application

Application: fancy word for “call”

$(e1\ e2)$

- “apply” argument $e2$ to function value $e1$

Type of function application

Application: fancy word for “call”

$(e1\ e2)$

- “apply” argument $e2$ to function value $e1$
- if $e1$ has type $T1 \rightarrow T2$ and
 $e2$ has type $T1$ then
 $(e1\ e2)$ has type $T2$

Functions

Values

Two questions about function values:

What is the **value**:

1. ... of a function ?

2. ... of a function “application” (call) ?

(***e1*** ***e2***)

Values of functions: Closures

- “Body” expression **not evaluated** until application
 - but type-checking takes place at compile time
 - i.e. when function is defined
- Function value =
 - <code + environment at definition>
 - “closure”

```
# let x = 2+2;;  
val x : int = 4  
# let f = fun y -> x + y;;  
val f : int -> int = fn
```

Environment at f's definition:

x	4 : int
---	---------

Value of f (**closure**):

f fn <fun y -> x + y, >

Values of functions: Closures

- “Body” expression **not evaluated** until application
 - but type-checking takes place at compile time
 - i.e. when function is defined
- Function value =
 - <code + environment at definition>
 - “closure”

```
# let x = 2+2;;  
val x : int = 4  
# let f = fun y -> x + y;;  
val f : int -> int = fn  
# let x = x + x ;;  
val x : int = 8  
# f 0;;  
val it : int = 4
```

Binding used to eval (f ...)

x	4 : int
f	fn <code, 
x	8 : int

Binding for subsequent **x**

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;;
```

Diagram illustrating variable binding in the code snippet above:

- A dotted box around the `a` in `let a = 20;;` is labeled "Binding".
- A dotted box around the `x` in `let f x =` is labeled "Binding".

Inside a function:

A “bound” occurrence:

1. Formal variable
 2. Variable bound in `let-in`
- `x`, `y`, `z` are “bound” inside `f`

A “free” occurrence:

- Not bound occurrence
- `a` is “free” inside `f`

Environment at definition, frozen inside “closure”, is used for values of free variables

Nested function bindings

```
let a = 20;;
```

```
let f x =
```

```
  let a = 1 in
```

```
  let g z = a + z in
```

```
    a + (g x)
```

```
;;
```

```
f 0;
```

Inside a function:

A “bound” occurrence:

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

x, **a**, **z** are “bound” inside **f**

A “free” occurrence:

Not bound occurrence

nothing is “free” inside **f**

Environment at definition, frozen
inside “closure”, is used for values
of free variables

Nested function bindings

```
let a = 20;;
```

```
let f x =
```

```
  let a = 1+1 in
```

```
  let g z = a + z in
```

```
    a + (g x)
```

```
;;
```

```
f 0;
```

Q: Where do **values** of **bound variables** come from ?

Bound variable values determined when fun evaluated (“executed”)

- From arguments
- Local variable bindings
 - Obtained from **evaluation**

Values of function application

Application: fancy word for “call”

$(e1\ e2)$

- “apply” the argument $e2$ to the (function) $e1$

Application Value:

1. Evaluate $e1$ in **current** env to get (function) $v1$
 - $v1$ is **code** + env
 - code is (**formal x** + **body e**) , env is E
2. Evaluate $e2$ in **current** env to get (argument) $v2$
3. Evaluate body e in env E **extended** by binding x to $v2$

Example 1

```
let x = 1;;  
let f y = x + y;;  
let x = 2;;  
let y = 3;;  
f (x + y);;
```

Example 2

```
let x = 1;;  
let f y =  
  let x = 2 in  
  fun z -> x + y + z  
;;
```

```
let x = 100;;  
let g = (f 4) ;;  
let y = 100;;  
(g 1) ;;
```

Example 3

```
let f g =  
  let x = 0 in  
    g 2  
;;  
  
let x = 100;;  
  
let h y = x + y;;  
  
f h;;
```

Example 4

```
let a = 20;;  
let f x =  
  let y = 10 in  
  let g z = y + z in  
    a + (g x)  
  ;;  
f 0;
```

Function/Application Values and Types

```
fun x -> x / 0
```

```
int -> int
```

```
(fun x -> x / 0) 10
```

```
int
```

This expression has no value!

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

Dynamic Scoping

```
let x = 1;;  
let f y =  
  let x = 2 in  
  fun z -> x + y + z  
;;
```

```
let x = 100;;  
let g = (f 4) ;;  
let y = 100;;  
(g 1) ;;
```

Dynamic Scoping

```
let x = 0;;  
let f () = x;;  
let g () =  
    let x = 10 in f ()
```

Static vs. Dynamic Scoping

- Any opinions?
- Analogies to other language features?
- Know any languages that use dynamic?
- What does Python do?