CSCI-GA.3033.003
Scripting Languages

12/02/2013

OCaml

Acknowledgement

The material on these slides is based on notes provided by Dexter Kozen.

About OCaml

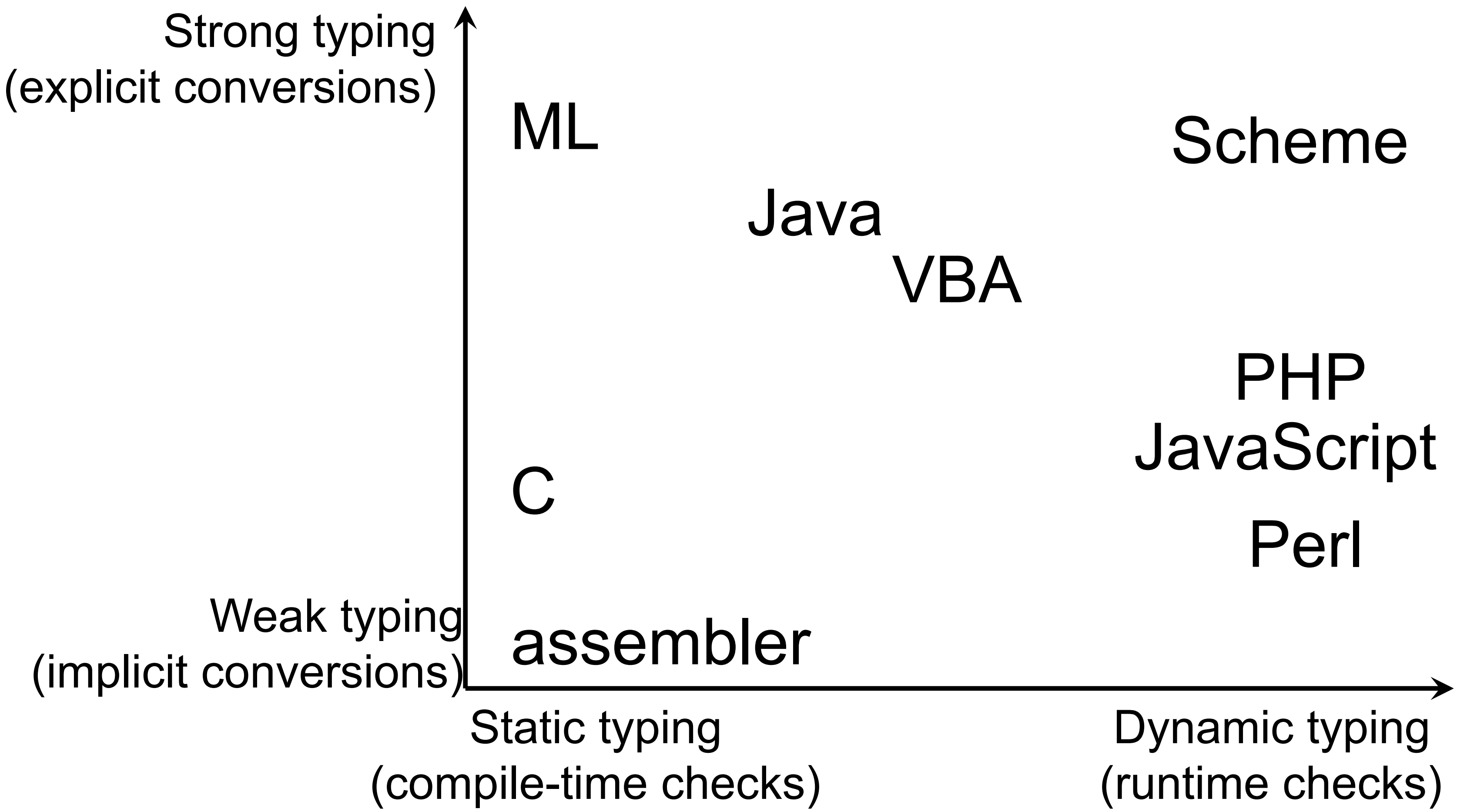
- A functional programming language
  - All computation is done with functions
  - No state (mutable variables), simple and clean
  - Functions are first class: you can pass them, return them, etc.
  - Support for object-oriented programming (the O in OCaml)
- Statically typed, type-safe language
  - You can't apply the wrong operations to the wrong data (e.g., can't try to divide two strings)
  - Avoids many "silly errors" and provides security (e.g., no buffer overflows)

OCaml

Ocaml Features

- **Garbage collection**: automatic memory management
- **Type inference**: You don't have to write the type information down. The compiler will figure it out.
- **Parametric polymorphism**: Write functions and data structures that can be used with any type. Note that Java provides *subtype polymorphism*.
- **Algebraic data types**: Can build data structures easily. *Pattern matching* makes working with them convenient.
- **Advanced modules**: Encapsulate implementations behind interfaces. *Functors* (functions that manipulate modules) add functionality beyond most languages' module systems.

Weak/Strong, Static/Dynamic Typing



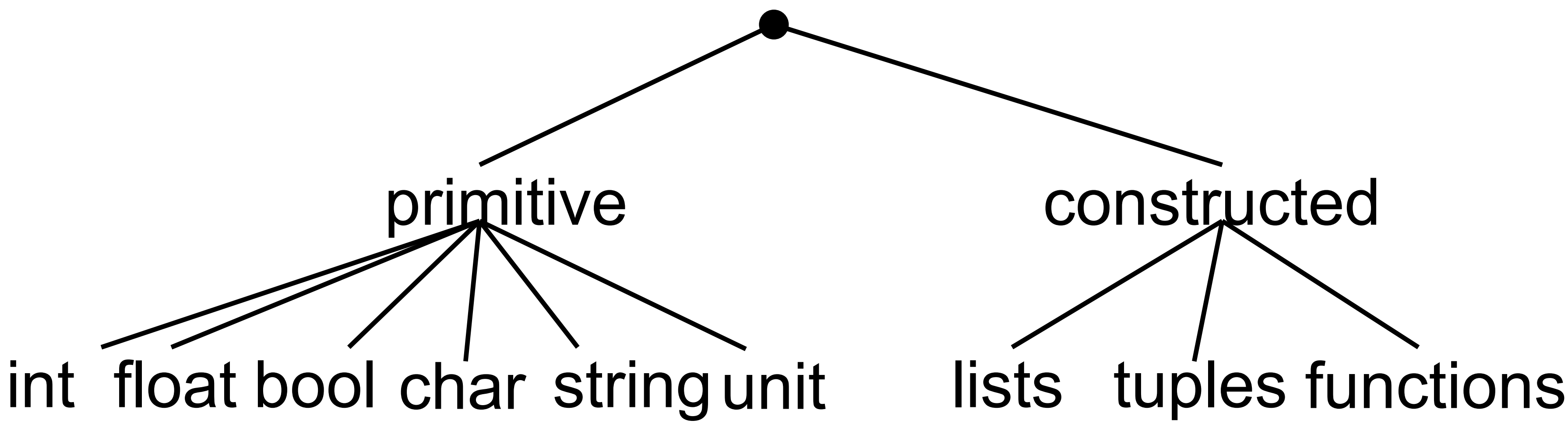
History

- Robin Milner and others at the University of Edinburgh were working on theorem provers
- Problem: the provers would sometimes put incorrect “proofs” together, and claim they were valid
- Designed ML (Meta Language) as a language to let you construct valid proofs
- In the early ‘80s, there was a schism in the ML community
  - French developed Caml and later Objective Caml (OCaml)
  - British and Americans developed Standard ML (SML)

How to Write + Run Code

- **ocaml**
  - REPL
- **ocaml** *file*.**ml**
  - Run the interpreter
- **ocamlc** *file*.**ml; ./a.out**
  - Run the compiler

Types


primitive
constructed
int
float
bool
char
string
unit
lists
tuples
functions

OCaml

Type Inference

- The compiler deduces the type of an expression

y = x + 1

- Compiler knows that + takes two integers and returns an integer, so x and y should be integers.

z = x + 1.0

- This should produce an error, since x is being used as both an int and a float
- Compiler infers types by aggregating type information, and reducing expressions to implicitly typed values

OCaml

Type Inference

- Statically determining whether a program will have a type-error is impossible
- All statically typed languages are conservative, and may reject some programs that are perfectly okay
- Milner formulated the type-inference system for ML, and proved its soundness
- Note that Milner also worked on concurrent programming languages (CCS, CSP, pi-Calculus), and won the Touring award – in large part to his work on ML

OCaml

Syntax (subset)

Syntactic class	Grammar rule	Example
identifiers	*x,f*	a, x,y,x_y, z100….
constants	*c*	1, 01, 1.0, true, “hello” ‘A’
unary operator	*u*	-, not
binary operator	*b*	+,*,-, <,>, ^, !=,…
terms	e::= *x* | *c* | *u e* | *e b e*| if *e* then *e* else *e* | let *d* and … *d* in *e* | *e* (*e*, …,*e*)	not b, 2 + 2, foo
declarations	*t ::= x = e* | *f (x,…,x) : t = e*	one = 1

OCaml

Program Errors

- An expression can have a legal syntax, but may be wrong. The expression must be *well-typed*.
- Ways that a program can be wrong:
  - Syntax errors: let 0 x =
  - Type errors: “x” + 3
  - Semantic errors: 1/0
  - More general errors: computes the wrong output

OCaml

Example

```
let abs (x : int) : int =
  if x < 0 then -x else x
```

```
let abs : int -> int =
  function x -> if x < 0 then -x else x
```

```
let abs = fun x -> if x < 0 then -x else x
```

---

val abs : int -> int = <fun>

- Every expression and declaration has both a *type* and a *value*.
- Here, we have bound the name abs to a function whose type is int -> int.

OCaml

Type Checking

```
$ ocaml
# let abs = fun x -> if x < 0 then -x else x ;;
val abs : int -> int = <fun>

# abs 3;;
- : int = 3

# abs "foo";;
Error: This expression has type string but an expression was
expected of type
         int
```

OCaml

Scope

- Variable declarations **bind** variables within a **scope**

let x = e1 in e2

- The scope of x is the expression e2.
- Functions also bind variables

let f x = e1 in e2

The scope of the formal parameter x is the expression in e1. The scope of the variable f (which is bound to a function value) is the body of the let, e2.

Scope

- A let expression can introduce multiple variables

```
let x = 2 and y = 3 in x + y
```

- To declare a recursive function, the function must be in scope in its own body. In Ocaml, you use let rec instead of let for this.

```
let rec even x = x = 0 || odd (x-1)
   and odd x = not (x = 0 || not (even (x-1)))
in
  odd 3110
```

OCaml

Curried functions

- A function with multiple parameters is really just syntactic sugar for a function passed a tuple as an argument

```
let plus (x, y) = x + y;;
val plus : int * int -> int = <fun>
plus (2, 3)
```

- Ocaml has another way to declare functions with multiple arguments. In **curried** form:

```
let plus x y = x + y
```

- Notice there are no commas between the parameters

Curried functions

- There are also no commas when calling the function:

plus 2 3

- Functions *really* only have one argument. The plus function is being passed one argument, 2, and it returns a function that takes one argument. (plus 2) (3)

plus 2 3

= ((function (x : int) -> function (y : int) -> x + y) 2) 3

= (function (y : int) -> 2 + y) 3

= 2 + 3

= 5

Lists

- Lists are **immutable.** You cannot change the elements of a list.
- Lists are **homogenous.** The can only contain one type.
- Examples: [1;2;3], 1::[2;3], [1;2]@[3;4]

Pattern Matching

- Use *match expressions* to extract elements from a list

```
match lst with
  |  [] -> 0
  | [x] -> 1
  | _ -> 2
```

- Returns 0 if the list is empty, 1 if the list has 1 element, and 2 if it has 2 or more elements

OCaml

Pattern Matching

- Use recursive functions to do something to every element in a list

```
let rec length (lst : string list) : int =
  match lst with
    [] -> 0
  | h :: t -> 1 + length t
```

- This function computes the length of the list lst.

OCaml

Variant Types

- Allows you to have variables that contain more than one kind of value
- Similar to an enum in java, or a tagged union in C

```
type answer = Yes | No | Maybe;;
```

- Type keyword declared name for the type.
- Declared with a set of **constructors**

```
type eitherPoint = TwoD of float * float
                 | ThreeD of float * float * float
```

OCaml

Pattern Matching

- Ocaml patterns are very rich
- Can match on variant types, tuples, records, and can pull the values apart

```
let answer_to_string (a : answer) : string =
  match a with
  | Yes -> “yes”
  | No -> “no”
```

Polymorphism

- Suppose we want to write a function that swaps positions of a pair:

```
let swapInt ((x : int), (y : int)) : int * int = (y, x)
and swapReal ((x : float), (y : float)) : float * float = (y, x)
and swapString ((x : string), (y : string)) : string * string = (y, x)
```

- Better way:

```
let swap ((x : 'a), (y : 'b)) : 'b * 'a = (y, x);;
```

- Or without type declarations:

```
# let swap (x, y) = (y, x);;
val swap : 'a * 'b -> 'b * 'a = <fun>
```

Mapping

- Apply a function to every element in a list, and return a new list:

```
List.map f [a; b; c] = [f a; f b; f c]
```

- Copy a list (using an anonymous function)

```
let copy = map (fun x -> x)
```

OCaml

Folding

- Apply a function to every element in a list, accumulates a result:

```
fold_right f [a; b; c] r = f a (f b (f c r))
fold_left f r [a; b; c] = f (f (f r a) b) c
```

- Sum all the elements in a list:

```
let sum_right_to_left il = fold_right (+) il 0
let sum_left_to_right = fold_left (+) 0
```

OCaml

Folding

- Fold is very powerful!
- Can write many other list functions in terms of fold

```
type intlist = Nil | Cons of (int * intlist)
let mapp f lst = fold_right (fun x y -> Cons (f x, y)) lst Nil
```

- Note that fold_left would give the result in reverse order

```
let length = fold_left (fun x _ -> 1 + x) 0
```

- Select a subset of the list

```
let filter f lst =
  fold_right (fun x y -> if f x then Cons (x, y) else y) lst Nil
```

Administrative

Last Slide

- Next lecture: Review!