CMSC 330: Organization of Programming Languages

Functional Programming with OCaml

Review: Interpreter & Compiler

Compiler:

- translates code written in a high-level programming language into a lower-level language
    - like assembly language, byte code, and machine code.
- it converts the code ahead of time before the program runs.
- we run the compiled to code to get the output
- Compiler optimizes the program

Interpreter

- translates the code line-by-line when the program is running
- we get the output when the code completes.

Review: Interpreter & Compiler

Optimization

```
int main(){
  int a = 1+2+3+4;
  return a;
}

% gcc -c a.c -o a.o
% objdump -d a.o
```

```
push    %rbp
mov     %rsp,%rbp
movl    $0xa,-0x4(%rbp)
mov     -0x4(%rbp),%eax
pop     %rbp
ret
```

1+2+3+4 = 10 = 0xa

Review: Interpreter & Compiler

- A simple OCaml Interpreter and Compiler Demo
  - …

- We will learn:
  - Interpreter in CMSC330
  - Compiler in CMSC430

What is a functional language?

A functional language:

- defines computations as **mathematical functions**
- *discourages* use of **mutable state**

State: the information maintained by a computation

`x = x + 1` ?

Functional vs. Imperative

Functional languages

- *Higher* level of abstraction: *What* to compute, not *how*
- *Immutable* state: easier to reason about (meaning)
- *Easier* to develop robust software

Imperative languages

- *Lower* level of abstraction: *How* to compute, not *what*
- *Mutable* state: harder to reason about (behavior)
- *Harder* to develop robust software

Imperative Programming

Commands specify **how** to compute, by destructively changing state:

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

imperative languages save changes to variables

The fantasy of changing state (mutability)

- It's easy to reason about: the machine does this, then this...

The reality?

- Machines are good at complicated manipulation of state
- Humans are not good at understanding it!

Imperative Programming: Reality

Functions/methods may **mutate** state, a **side effect**

```
int cnt = 0;

int f(Node *r) {
      r->data = cnt;
      cnt++;
      return cnt;
}
```

Mutation breaks **referential transparency**: ability to replace an expression with its value without affecting the result

`f(x) + f(x) + f(x)  ≠  3 * f(x)`

Imperative Programming: Reality

Worse: There is no single state

- Programs have many threads, spread across many cores, spread across many processors, spread across many computers...
- each with its own view of memory

So: Can’t look at one piece of code and reason about its behavior

Thread 1 on CPU 1

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

Thread 2 on CPU 2

```
x = x+1;
a[i] = 42;
p.next = p.next.next;
```

Functional programming

Expressions specify **what** to compute

foo(1) + foo(1) === 2 · foo(1)

- Variables never change value
    - Like mathematical variables
- Functions (almost) never have side effects

The reality of immutability:

- No need to think about state
- Can perform local reasoning, assume referential transparency

Easier to build correct programs

ML-style (Functional) Languages

- ML (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
- Standard ML
  - Bell Labs and Princeton, 1990; Yale, AT&T, U. Chicago
- OCaml (Objective CAML)
  - INRIA, 1996
    - French Nat’l Institute for Research in Computer Science
  - O is for “objective”, meaning objects (which we’ll ignore)
- Haskell (1998): *lazy* functional programming
- Scala (2004): functional and OO programming

Key Features of ML

- First-class functions
  - Functions can be parameters to other functions (“higher order”) and return values, and stored as data
- Favor immutability (“assign once”)
- Data types and pattern matching
  - Convenient for certain kinds of data structures
- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports parametric polymorphism
    - *Generics* in Java, *templates* in C++
- Exceptions and garbage collection

Why study functional programming?

Functional languages predict the future:

- Garbage collection
  - LISP [1958], Java [1995], Python 2 [2000], Go [2007]
- Parametric polymorphism (generics)
  - ML [1973], SML [1990], Java 5 [2004], Rust [2010]
- Higher-order functions
  - LISP [1958], Haskell [1998], Python 2 [2000], Swift [2014]
- Type inference
  - ML [1973], C++11 [2011], Java 7 [2011], Rust [2010]
- Pattern matching
  - SML [1990], Scala [2002], Rust [2010], Java 16 [2021]
    - http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html

Why study functional programming?

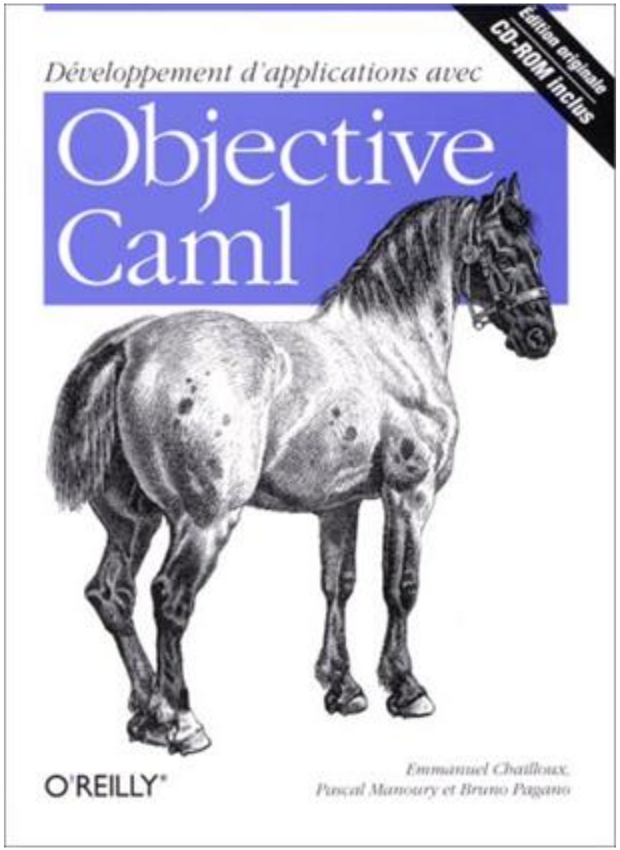
Functional languages in the real world

This slide is old---now there are even more!

- Java 8 ORACLE
- F#, C# 3.0, LINQ Microsoft
- Scala twitter foursquare LinkedIn
- Haskell facebook BARCLAYS at&t
- Erlang facebook amazon T-Mobile
- OCaml facebook Bloomberg CITRIX

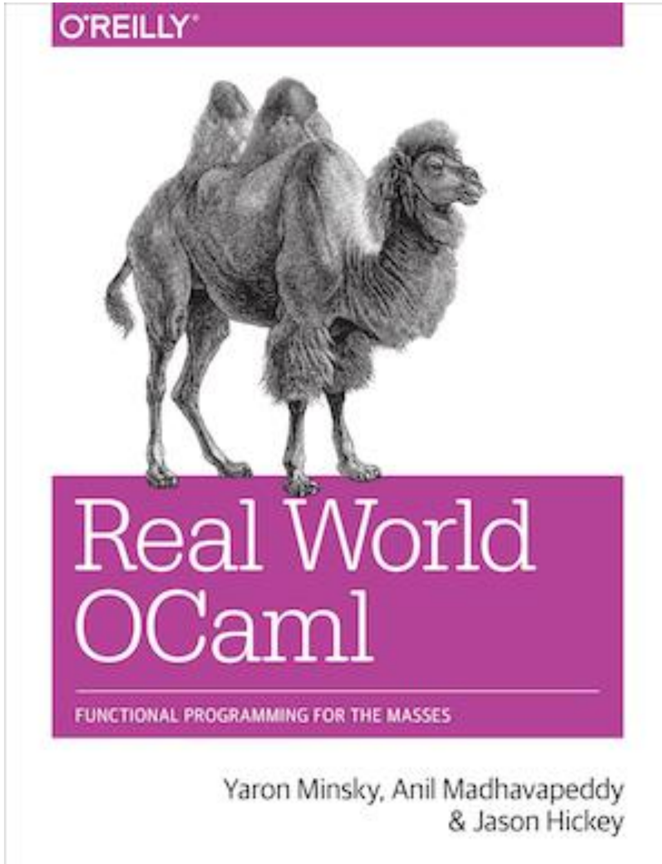
https://ocaml.org/learn/companies.html Jane Street

Useful Information on OCaml



- Translation available on the class webpage
  - *Developing Applications with Objective Caml*
- Webpage also has link to another book
  - *Introduction to the Objective Caml Programming Language*

More Information on OCaml



- Book designed to introduce and advance understanding of OCaml
  - Authors use OCaml in the real world (2nd edition)
  - Introduces new libraries, tools
- Free HTML online
  - realworldocaml.org/

Similar Courses

- CS3110 (Cornell)
- CSE341 (Washington)
- 601.426 (Johns Hopkins)
- COS326 (Princeton)
- CS152 (Harvard)
- CS421 (UIUC)

Other Resources

- Cornell cs3110 book is another course which uses OCaml; it is more focused on programming and less on PL theory than this class is.
- ocaml.org is the home of OCaml for finding downloads, documentation, etc. The tutorials are also very good and there is a page of books.
- OCaml from the very beginning is a free online book.

OCaml Coding Guidelines

- We will not grade on style, but style is important
- Recommended coding guidelines:

- https://ocaml.org/learn/tutorials/guidelines.html

CMSC 330: Organization of Programming Languages

OCaml Expressions, Functions

Lecture Presentation Style

- Our focus: semantics and idioms for Ocaml
  - *Semantics* is what the language does
  - *Idioms* are ways to use the language well

- We will also cover some useful libraries

- Syntax is what you type, not what you mean
  - In one lang: Different syntax for similar concepts
  - Across langs: Same syntax for different concepts
  - Syntax can be a source of fierce disagreement among language designers!

Expressions e

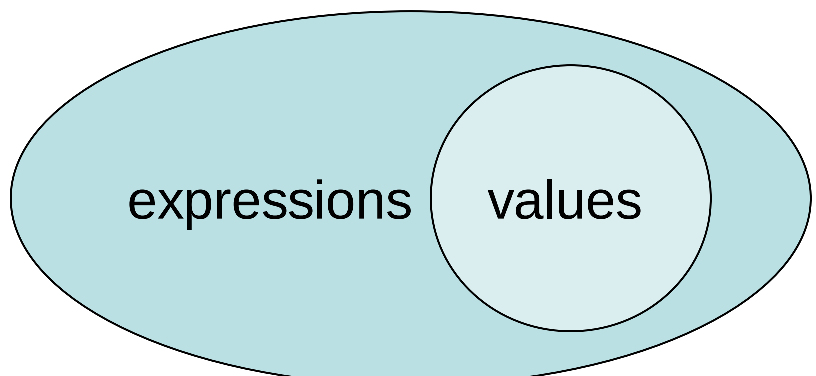
- Expressions are our primary building block
  - Akin to *statements* in imperative languages
- Every kind of expression has
  - Syntax
    - We use metavariable *e* to designate an arbitrary expression
  - Semantics
    - Type checking rules (static semantics): produce a type or fail with an error message
    - Evaluation rules (dynamic semantics): produce a value
      - (or an exception or infinite loop)
      - Used *only* on expressions that type-check

Values

- A value is an expression that is final
  - **34** is a value, **true** is a value
  - **34+17** is an *expression*, but *not* a value

34+17 can be further evaluated.

- Evaluating an expression means running it until it's a value
  - **34+17** *evaluates* to **51**
- We use metavariable *v* to designate an arbitrary value



Types

$\underline{t}$

- Types classify expressions
  - The set of values an expression could evaluate to
  - We use metavariable *t* to designate an arbitrary type
    - Examples include `int`, `bool`, `string`, and more.
- Expression *e* has type *t* if *e* will (always) evaluate to a value of type *t*
  - `0`, `1`, and `-1` are values of type `int` while `true` has type `bool`
  - `34+17` is an expression of type `int`, since it evaluates to `51`, which has type `int`

$e : t$

- Write *e* **:** *t* to say *e* has type *t*
  - Determining that *e* has type *t* is called type checking
    - or simply, typing

If Expressions

- Syntax

return types must match!

`(if` ***e1*** `then` ***e2*** `else` ***e3***`)` : ***t***

: **`bool`** :***t***

(each has the same type ***t****)*

- Type checking
  - Conclude `if` ***e1*** `then` ***e2*** `else` ***e3*** has type ***t*** if
    - ***e1*** has type **`bool`**
    - Both ***e2*** and ***e3*** have type ***t*** (for some ***t***)

If Expressions: Type Checking and Evaluation

```
# if 7 > 42 then "hello" else "goodbye";;
-   : string = "goodbye"

# if true then 3 else 4;;
-   : int = 3

# if false then 3 else 3.0;;
Error: This expression has type float but an expression was
expected of type int
```

int & float!

- Evaluation (happens if type checking succeeds)
  - If ***e1*** ⇒ **`true`**, and ***e2*** ⇒ ***v***, then

    `"if` ***e1*** **`then`** ***e2*** **`else`** ***e3"*** ⇒ ***v***
  - If ***e1*** ⇒ **`false`**, and ***e3*** ⇒ ***v***, then

    `"if` ***e1*** **`then`** ***e2*** **`else`** ***e3"*** ⇒ ***v***

Quiz 1

To what value does this expression evaluate?

```
if 10 < 20 then 2 else 1
```

A. 0

B. 1

C. 2

D. none of the above

Quiz 1

To what value does this expression evaluate?

```
if 10 < 20 then 2 else 1
```

A. 0

B. 1

C. 2

D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22 >10 then 2021 else “home”
```

Has 2 differently typed output

A. 0

B. 1

C. 2

D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22 > 10 then 2021 else "home"
```

A. 0

B. 1

C. 2

D. none of the above: doesn't type check so never gets a chance to be evaluated

Function Definitions

- OCaml functions are like mathematical functions
  - Compute a result from provided arguments

```
(* requires n>= 0
 returns: n! *)
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

function
body

we use "let" to define a function

;; ends an expression in OCaml

;; = "give me the value of this expression"

Type Inference

- As we just saw, a declared variable need not be annotated with its type
  - The type can be inferred

```
(* requires n>=0 *)
(* returns: n! *)
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

n's type is **int**. Why?

n is compared to 0

n is also multiplied

  - Type inference happens *as a part of type checking*
    - Determines a type that satisfies code's constraints

Calling Functions, *aka* Function Application

- Syntax `f e1 … en` `fact (2+1)`
  - Parentheses not required around argument(s)
  - No commas; use spaces instead to separate arguments.
- Evaluation
  - Find the definition of `f`
    - i.e., `let rec f x1 … xn = e`
  - Evaluate arguments `e1 … en` to values `v1 … vn`
  - **Substitute** arguments `v1`, … `vn` for params `x1`, ... `xn` in body `e`
    - Call the resulting expression `e′`
  - Evaluate `e′` to value `v`, which is the final result

Calling Functions: Evaluation

```
let rec fact n =
  if n = 0 then
     1
  else
     n * fact (n-1)
```

Example evaluation

- `fact 2`
  - ➢ `if 2=0 then 1 else 2*fact(2-1)`
  - ➢ `2 * fact 1`
  - ➢ `2 * (if 1=0 then 1 else 1*fact(1-1))`
  - ➢ `2 * 1 * fact 0`
  - ➢ `2 * 1 * (if 0=0 then 1 else 0*fact(0-1))`
  - ➢ `2 * 1 * 1`
  - ➢ `2`

Fun fact: Evaluation order for function call arguments in OCaml is **right to left** (not left to right)

Function Types

- In OCaml, `->` is the function type constructor
  - Type `t1 -> t` is a function with argument or *domain* type `t1` and return or *range* type `t`
  - Type `t1 -> t2 -> t` is a function that takes *two* inputs, of types `t1` and `t2`, and returns a value of type `t`. Etc.
- Examples
  - `not` `(* type bool -> bool *)`
  - `int_of_float` `(* type float -> int *)`
  - `+` `(* type int -> int -> int *)`

Type Checking: Calling Functions

- Syntax ***f* *e1* … *en***
- Type checking
  - If **f : t1 -> … -> tn -> u**
  - and **e1 : t1, …, en : tn**
  - then **f e1 … en : u**

- Example:
  - **not true : bool**
  - since **not : bool -> bool**
  - and **true : bool**

Type Checking: Example

```
let rec fact n =
  if(n = 0)then
     1
  else
    (n * fact(n-1))
```

(n=0): `bool` assuming `n:int`

`(n * fact (n-1)):int`

Function Type Checking: More Examples

- `let next x = x + 1` (* type int -> int *)
- `let fn x = (int_of_float x) * 3` (* type float -> int *)
- `fact` (* type int -> int *)
- `let sum x y = x + y` (* type int -> int -> int *)

- type int -> int

~~int -> float~~ float -> int

int -> int

int -> int -> int

Quiz 3: What is the type of `foo 3 1.5`

```
let rec foo n m =
  if n >= 9 || n > 0 then
    m
  else
    m +. 10.3
```

type int -> float -> float

: float -> float -> float

a) Type Error

b) `int`

c) `float`

d) `int -> int -> int`

Quiz 3: What is the type of `foo 3 1.5`

```
let rec foo n m =
  if n >= 9 || n > 0 then
    m
  else
    m +. 10.3
```

`: float -> float -> float`

a) Type Error

b) **`int`**

c) **`float`**

d) **`int -> int -> int`**

Ocaml compiler infers the types
However, inference is tricky

Type Annotations

- The syntax **(*e* : *t*)** asserts that "*e* has type *t*"
  - This can be added (almost) anywhere you like

manual type annotators, for readability.

```
let (x : int) = 3
let z = (x : int) + 5
```

- Define functions' parameter and return types

```
let fn (x:int):float =
       (float_of_int x) *. 3.14
```

- Checked by compiler: Very useful for debugging

Quiz 4: What is the value of `bar 4`

```
let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else
    bar (n-1) + bar (n-2)
```

n is an int,
bar returns int

type int -> int

a) Syntax Error

b) 4

c) 5

d) 8

Quiz 4: What is the value of `bar 4`

```
let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else
    bar (n-1) + bar (n-2)
```

a) Syntax Error

b) 4

c) 5

d) 8