Programming Languages and Environments (Lecture 4)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Syllabus

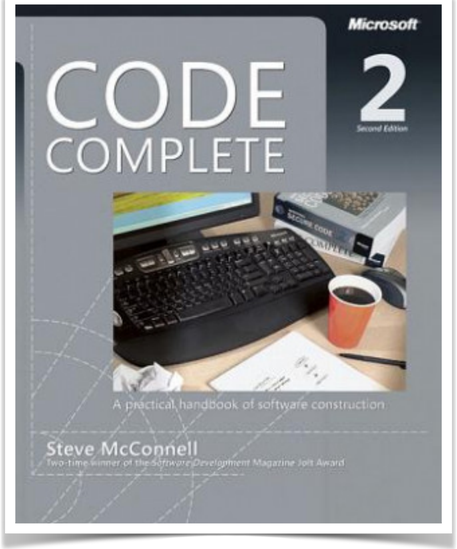
- Unit testing.
- Functions as values.
- Composition.
- Function Types

Correctness

Notes on correction

- Design by Contract (Bertrand Meyer 1986)
  - Verifies that all calls meet the preconditions (if not, it stops) and ensures the postconditions.
- Formal systems (Hoare Logic 1969)
  - Does not guarantee the validity of the pre-conditions.
  - Assumes the validity of the pre-conditions and guarantees the validity of the post-conditions, if it terminates (partial correctness).
  - Plethora of tools that guarantee the correctness of annotated programs at compile time.
- Defensive programming
  - Does not assume the input and tests all pre-conditions at runtime. Output/ exceptions for all possible inputs (Precondition is `true`).
  - You can declare preconditions but typically, this approach does not test its results.

$$\{ A \} P \{ B \}$$



Notes on correctness (examples)

- Informal pre and post-conditions.

```
(** [fib n] is the [n]th fibonnaci number
    requires n > 0 *)
let rec fib n = if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

- Programming language that verifies the specification (via contracts) of a function/ method. (Dafny)
- Other Formal Verification Tools: Verifast, Why3, Infer, Cameleer, …

```
let rec fib n =
  if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
(*@ requires n > 0 *)
```

```
function fib(n: nat): nat
{
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  ...
}
```

Unit Testing

Unit testing

Module Lap

```
module Lap: sig .. end
```

The first special comment of the file is the comment associated with the whole module. This is module LAP with sample code for LAP 2024

```
val fact : int -> int
```

`fact n` is the factorial of `n` Requires: `n >= 0`

```
val even : int -> bool
```

`even x` is true if `x` is even, false otherwise Requires: `x >= 0`

```
val odd : int -> bool
```

`odd x` is true if `x` is odd, false otherwise Requires: `x >= 0`

```
let _ = assert (true = even 2)
let _ = assert (false = odd 2)
let _ = assert (true = odd 57)
let _ = assert (false = even 57)

let _ = assert (1 = fact 0)
let _ = assert (720 = fact 6)

let _ = assert (1 = fib 1)
let _ = assert (1 = fib 2)
let _ = assert (2 = fib 3)
let _ = assert (3 = fib 4)
let _ = assert (5 = fib 5)
let _ = assert (8 = fib 6)
```

```
"two is even" >:: (fun _ -> assert_equal true (even 2))
```

Unit testing

```
module Lap: sig ..
      The first special c
      whole module. Th

val fact : int ->
      fact n is the fac

val even : int ->
      even x is true if

val odd : int -> b
      odd x is true if x

val sum : 'a -> 'b
      sum lst is the s
```

```
let tests =
 [
  "two is even" >:: (fun _ -> assert_equal true (even 2));
  "two is not odd" >:: (fun _ -> assert_equal false (odd 2));
  "fifty seven is odd" >:: (fun _ -> assert_equal true (even 2));
  "fifty seven is not even" >:: (fun _ -> assert_equal false (odd 2));
  "factorial of 0 is 1" >:: (fun _ -> assert_equal 1 (fact 0));
  "factorial of 6 is 720" >:: (fun _ -> assert_equal 720 (fact 6));
  "fibonacci 1 is 1" >:: (fun _ -> assert_equal 1 (fib 1));
  "fibonacci 2 is 1" >:: (fun _ -> assert_equal 1 (fib 2));
  "fibonacci 3 is 2" >:: (fun _ -> assert_equal 2 (fib 3));
  "fibonacci 4 is 3" >:: (fun _ -> assert_equal 3 (fib 4));
  "fibonacci 5 is 5" >:: (fun _ -> assert_equal 5 (fib 5));
  "fibonacci 6 is 8" >:: (fun _ -> assert_equal 8 (fib 6))
]

let test_suit = "test suite for sum" >::: tests

let _ = run_test_tt_main test_suit
```

```
= even 2)
= odd 2)
= odd 57)
= even 57)

fact 0)
= fact 6)

fib 1)
fib 2)
fib 3)
fib 4)
fib 5)
fib 6)
```

```
jcs@joaos-imac lap2024 % dune exec ./test.exe
............
Ran: 12 tests in: 0.11 seconds.
OK
```

https://cs3110.github.io/textbook/chapters/data/ounit.html

Functions++

Functions as values

- Fundamentally, functions are values.
- Functions can be parametrized by other functions and we call these higher-order functions.
- Higher-order functions are also defined as functions that return functions as their result.

```
let f x y = x+y
✓ 0.2s
val f : int → int → int = <fun>
```

```
let g x y = x * y
✓ 0.0s
val g : int → int → int = <fun>
```

```
let h i = 1 + i 1 1
✓ 0.0s
val h : (int → int → int) → int = <fun>
```

```
h f
✓ 0.0s
- : int = 3
```

```
h g
✓ 0.0s
- : int = 2
```

```
h (fun x y → x - y)
✓ 0.0s
- : int = 1
```

Functions as values

```
let f x = if x = 1 then fun x y → x + y else fun x y → x * y
```

✓ 0.0s

```
val f : int → int → int → int = <fun>
```

- We've seen before that functions can also return other functions.
- And functions can also capture the context of the definition, we call these closures.

```
let g = f 1 in g 2 3
```

✓ 0.0s

```
- : int = 5
```

```
let g = f 2 in g 2 3
```

✓ 0.0s

```
- : int = 6
```

Functions as values

```
let f x = if x = 1 then fun x y → x + y else fun x y → x * y
```

✓ 0.0s

```
val f : int → int → int → int = <fun>
```

- We've seen before that functions can also return other functions.
- And functions can also capture the context of the definition, we call these closures.

```
(f 1) 2 3
```

✓ 0.0s

```
- : int = 5
```

```
(f 2) 2 3
```

✓ 0.0s

```
- : int = 6
```

Functions as values

- In OCaml, function application follows the direction from left-to-right, so there is no need to use parameters.

```
f 3 2 1
✓ 0.0s
- : int = 2
```

```
f 3 2 1 = (f 3) 2 1 && (f 3) 2 1 = ((f 3) 2) 1
✓ 0.0s
- : bool = true
```

Function composition

```
let comp (f:int → int) (g:int->int) = fun x → f (g (x))

let dup = fun x → x + x

let quad = comp dup dup

let x = quad 2
```

✓ 0.0s

```
val comp : (int → int) → (int → int) → int → int = <fun>

val dup : int → int = <fun>

val quad : int → int = <fun>

val x : int = 8
```

Arrow type

Function Type in OCaml

- In the Curry-Howard correspondence, the function type is the **implication**.
- And the evaluation, in lambda calculus, corresponds to natural deduction.
- This corresponds to the definition of **codomain** and **domain** (result set) in the mapping between types and sets.
- It declares the type of values accepted as arguments (parameter type in the function body) and the type of the function's output (type of the expression in the function body).

$$A \to B$$

- The primitive function type only allows for one parameter and one result. To create functions that accept multiple parameters and return multiple results, we need to use composite types.

Multiple parameters

- The arrow type is right-associative.
- Consecutive parameters A, B and C with a result D.

$$A \to B \to C \to D$$

- (Or) A parameter and a function as a result.

$$A \to (B \to (C \to D))$$

- (This is different) A function as a parameter with a result D.

$$((A \to B) \to C) \to D$$

Natural deduction: exercise

- Consider the predicates represented as types: `a`, `b`, `c` and `d`
- Consider, also, the components with type annotations: `x`, `f`, `g`, `h` and `i`.
- Is it possible to produce an expression of type `d`?

```
(* Predicates *)
type a =
type b =
type c =
type d =

(* Components *)
a
let x : a =

a -> a -> b
let f : a → a → b =

(a -> b) -> c
let g : (a → b) → c =

(c -> c) -> d
let h : (c → c) → d =

c -> c -> c
let i : c → (c → c) =

d
let question : d =
```

Natural deduction: exercise

- Consider the predicates represented as types: `a`, `b`, `c` and `d`
- Consider, also, the components with type annotations: `x`, `f`, `g`, `h` and `i`.
- Is it possible to produce an expression of type `d`? Yes!

```
let question : d = h (i ( g ( f x )));;
```

```
(* Predicates *)
type a =
type b =
type c =
type d =

(* Components *)
a
let x : a =

a -> a -> b
let f : a → a → b =

(a -> b) -> c
let g : (a → b) → c =

(c -> c) -> d
let h : (c → c) → d =

c -> c -> c
let i : c → (c → c) =

d
let question : d =
```

Program synthesis based on components and types

- The type system is a fundamental component in advanced features of modern programming environments, such as program synthesis.

Program Synthesis by Type-Guided Abstraction Refinement

ZHENG GUO, UC San Diego, USA
MICHAEL JAMES, UC San Diego, USA
DAVID JUSTO, UC San Diego, USA
JIAXIAO ZHOU, UC San Diego, USA
ZITENG WANG, UC San Diego, USA
RANJIT JHALA, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

We consider the problem of type-directed component based synthesis where, given a set of (typed) co
and a query *type*, the goal is to synthesize a *term* that inhabits the query. Classical approaches
proof search in intuitionistic logics do not scale up to the standard libraries of modern languages, w
hundreds or thousands of components. Recent graph reachability based methods proposed for l
like Java do scale, but only apply to monomorphic data and components: polymorphic data and co

Hoogle⋆: Constants and λ-abstractions in Petri-net-based Synthesis using Symbolic Execution

Henrique Botelho Guerra
INESC-ID and IST, University of Lisbon, Portugal
João F. Ferreira
INESC-ID and IST, University of Lisbon, Portugal
João Costa Seco
NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

Abstract

Type-directed component-based program synthesis is the task of automatically building a function with applications of available components and whose type matches a given goal type. Existing approaches to component-based synthesis, based on classical proof search, cannot deal with large sets of components. Recently, HOOGLE+, a component-based synthesizer for Haskell, overcomes this issue by reducing the search problem to a Petri-net reachability problem. However, HOOGLE+ cannot synthesize constants nor λ-abstractions, which limits the problems that it can solve.

Syllabus

- Unit testing.
- Functions as values.
- Composition.
- Function Types