

Linguagens e Ambientes de Programação

(Aula Teórica 4)

LEI - Licenciatura em Engenharia Informática

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

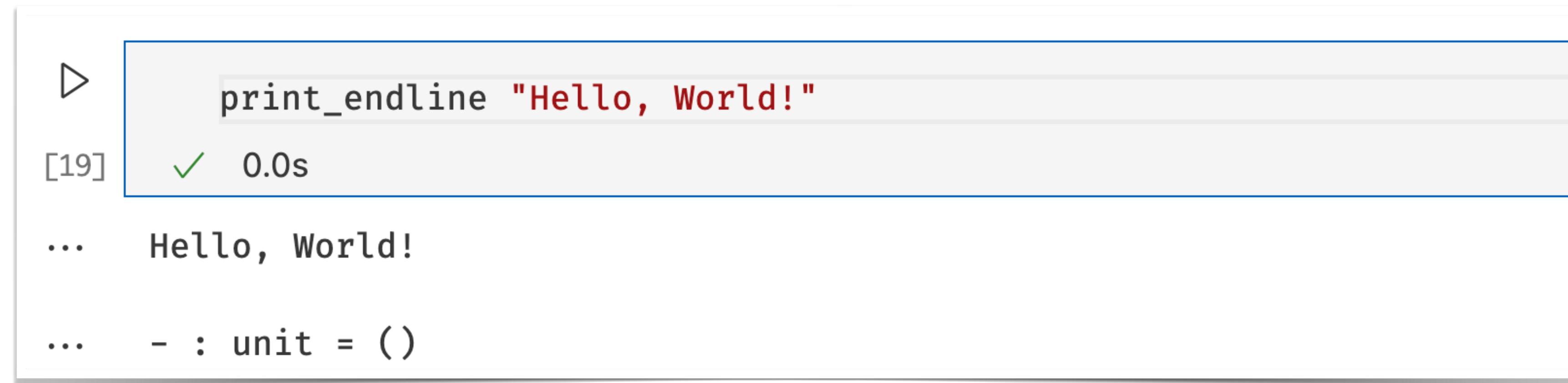
Agenda

- Input/output básico
- Unit, Sequências, como ignorar valores intermédios
- Documentação
- Pré- e pós-condições.
- Correção de programas.
- Testes unitários.
- Funções como valores.
- Tipo função.
- Composição.
- Polimorfismo.
- Inferência de tipos.

Input/Output & Unit

Input/Output Básico

- O tipo unit só faz sentido em expressões que têm efeitos laterais, a impressão é um exemplo típico disso
- `print_endline`
- `print_string`
- `print_char`
- `print_int`
- `print_float`



A screenshot of a code editor showing a successful execution of a `print_endline` expression. The code is `print_endline "Hello, World!"`. The output shows the string "Hello, World!" printed to the console, along with the type annotation "- : unit = ()". The status bar indicates the operation took 0.0s.

```
▶ [19] print_endline "Hello, World!" ✓ 0.0s
... Hello, World!
... - : unit = ()
```

Declaração como operador de sequência.

- declarações que ignoram resultados

```
▶      let () = print_string "hello, " in print_endline "world!"  
[14]   ✓  0.0s
```

```
...  hello, world!
```

```
...  - : unit = ()
```

```
▶      let _ = uma_funcao_com_efeitos_laterais () in 4  
[13]   ✓  0.0s
```

```
...  - : int = 4
```

```
▶      print_string "hello, "; print_endline "world!"
```

```
[15]   ✓  0.0s
```

```
...  hello, world!
```

```
...  - : unit = ()
```

Declaração como operador de sequência.

- declarações que ignoram resultados

```
1; 2  
[18] ✓ 0.0s
```

```
... File "[18]", line 1, characters 0-1:  
1 | 1; 2  
  ^
```

Warning 10 [non-unit-statement]: this expression should have type unit.

```
File "[18]", line 1, characters 0-1:  
1 | 1; 2  
  ^
```

Warning 10 [non-unit-statement]: this expression should have type unit.

```
... - : int = 2
```

Declaração como operador de sequência.

- declarações que ignoram resultados

```
▶      let () = print_string "hello, " in print_endline "world!"  
[14] ✓ 0.0s  
... hello, world!  
  
... ▶ (ignore 1); 2  
[21] ✓ 0.0s  
... - : int = 2  
  
... ✓ print_string "hello, "; print_endline "world!"  
[15] ✓ 0.0s  
... hello, world!  
... - : unit = ()
```

Documentação

OCamldoc - documentação real em ocaml

- A documentação do código serve para ajudar a ler o código de uma função, mas também para perceber a funcionalidade de um módulo inteiro.

```
(** 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 *)  
  
(** [fact n] is the factorial of [n]  
|   requires: [n >= 0] *)  
let rec fact x = if x = 0 then 1 else x * fact(x-1)  
  
(** [even x] is true if [x] is even, false otherwise  
|   requires: [x >= 0] *)  
let rec even x = if x = 0 then true else if x = 1 then fal  
  
(** [odd x] is true if [x] is odd, false otherwise  
|   requires: [x >= 0] *)  
and odd x = if x = 0 then false else if x = 1 then true el
```

```
jcs@joaos-imac lap2024 % ocamldoc -html lap.ml
```

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

val sum : 'a → 'b
sum lst is the sum of the elements of lst.

OCamldoc - documentação real em ocaml

- Tags que se podem usar:

2.5 Documentation tags (@-tags)

Predefined tags

The following table gives the list of predefined @-tags, with their syntax and meaning.

<code>@author string</code>	The author of the element. One author per <code>@author</code> tag. There may be several <code>@author</code> tags for the same element.
<code>@deprecated text</code>	The <code>text</code> should describe when the element was deprecated, what to use as a replacement, and possibly the reason for deprecation.
<code>@param id text</code>	Associate the given description (<code>text</code>) to the given parameter name <code>id</code> . This tag is used for functions, methods, classes and functors.
<code>@raise Exc text</code>	Explain that the element may raise the exception <code>Exc</code> .
<code>@return text</code>	Describe the return value and its possible values. This tag is used for functions and methods.
<code>@see < URL > text</code>	Add a reference to the <code>URL</code> with the given <code>text</code> as comment.
<code>@see 'filename' text</code>	Add a reference to the given file name (written between single quotes), with the given <code>text</code> as comment.
<code>@see "document-name" text</code>	Add a reference to the given document name (written between double quotes), with the given <code>text</code> as comment.
<code>@since string</code>	Indicate when the element was introduced.
<code>@before version text</code>	Associate the given description (<code>text</code>) to the given <code>version</code> in order to document compatibility issues.
<code>@version string</code>	The version number for the element.

OCamldoc - preconditions and post-conditions

- A documentação de uma função também pode/deve indicar as suas pré-condições e pós-condições. Estas condições são informais.

```
(** 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 *)

(** [fact n] is the factorial of [n]
   | requires: [n >= 0] *)
let rec fact x = if x = 0 then 1 else x * fact(x-1)

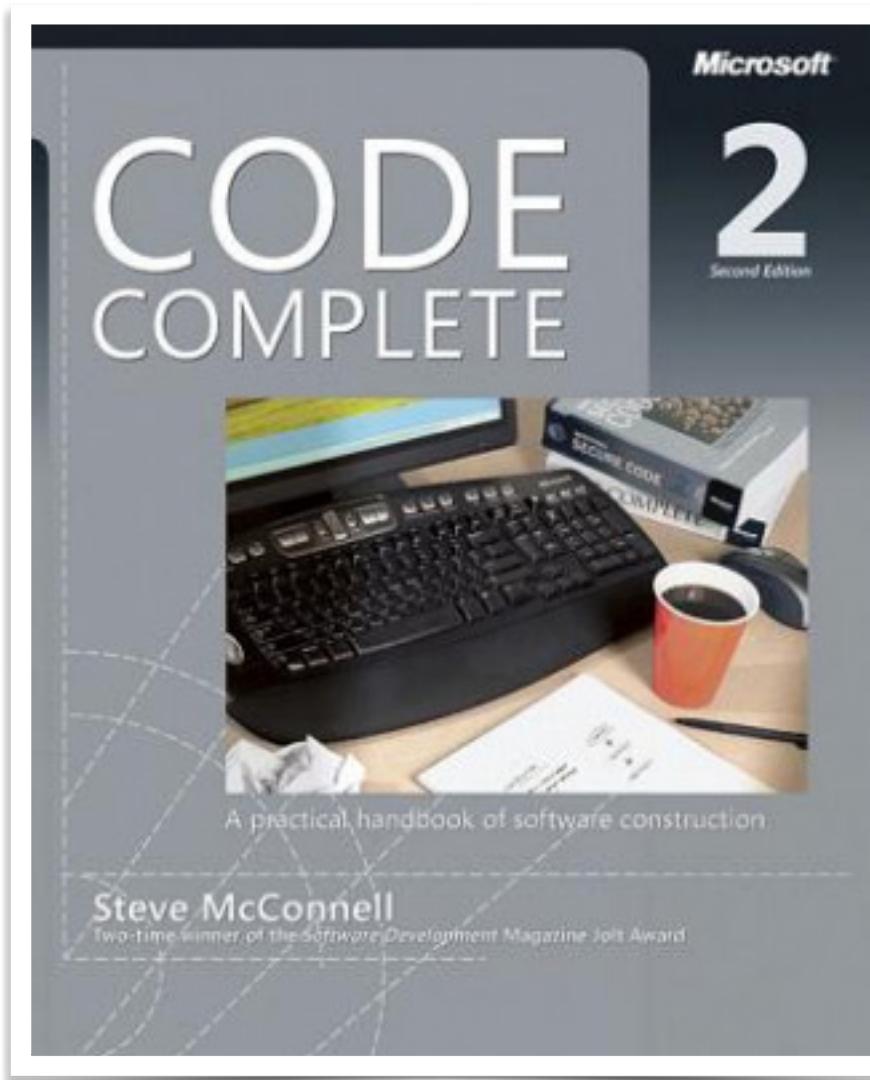
(** [even x] is true if [x] is even, false otherwise
   | requires: [x >= 0] *)
let rec even x = if x = 0 then true else if x = 1 then false else odd(x-1)

(** [odd x] is true if [x] is odd, false otherwise
   | requires: [x >= 0] *)
and odd x = if x = 0 then false else if x = 1 then true else even(x-1)
```

Correção

Um slide sobre correção

- Programação por contrato (Design by Contract - Bertrand Meyer, 1986)
 - Verifica que todas as chamadas cumprem as pré-condições (se não, pára), garante as pós-condições.
- Programação com pré-condições e pós-condições (Lógica de Hoare, 1969)
 - Não assume o cumprimento das pré-condições, mas...
 - dá garantias de cumprimento das pós-condições no caso de cumprimento das pré-condições e em caso de terminação.
 - Há ferramentas que garantem a correção dos programas (os contratos) em tempo de compilação.
- Programação defensiva
 - Não assume nada sobre o input e testa todas as suas “pré-condições” em tempo de execução. Tem outputs/excepções para todas as entradas possíveis. (A verdadeira pré-condição é “true”).
 - Pode declarar pós-condições mas normalmente não testa os seus resultados.


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


Mais um slide sobre correção (exemplos)

- Pré e pós-condições informais vs.

```
(** [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)
```

- Uma linguagem que verifica formalmente a especificação de uma função/método (Dafny).
- Existem mais ferramentas de verificação: 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)  
{  
  ...  
}
```

Testes unitários

Testes unitários

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

```
val sum : 'a -> 'b
```

sum *lst* is the sum of the elements of *lst*.

```
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))
```

Testes unitários

```
module Lap: sig ...
  The first special case is the whole module. Then ...
  val fact : int -> int
    fact n is the factorial of n
  val even : int -> bool
    even x is true if x is even
  val odd : int -> bool
    odd x is true if x is odd
  val sum : 'a -> 'a list -> 'a
    sum lst is the sum of all elements in lst
```

```
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 57));
  "fifty seven is not even" >:: (fun _ -> assert_equal false (odd 57));
  "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
```

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

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

Funções (outra vez)

Funções como valores

- As funções são valores da linguagem, podem ser como qualquer outro valor.
- As funções podem ser usadas como parâmetros para outras funções.

The screenshot shows a functional programming environment with three code blocks and their corresponding results:

- Top Block:**

```
let f x y = x+y
✓ 0.2s
```

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

Result on the right:
h f
✓ 0.0s
- : int = 3
- Middle Block:**

```
let g x y = x * y
✓ 0.0s
```

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

Result on the right:
h g
✓ 0.0s
- : int = 2
- Bottom Block:**

```
let h i = 1 + i 1 1
✓ 0.0s
```

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

Result on the right:
h (fun x y → x - y)
✓ 0.0s
- : int = 1

Funções como valores

```
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>
```

- As funções podem ser usadas como resultados de outras funções (já tínhamos visto).
- As funções podem “capturar” nomes do contexto de definição, chamam-se “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
```

Funções como valores

```
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>
```

- As funções podem ser usadas como resultados de outras funções (já tínhamos visto).
- As funções podem “capturar” nomes do contexto de definição, chamam-se “closures”.

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

✓ 0.0s

```
- : int = 5
```

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

✓ 0.0s

```
- : int = 6
```

Funções como valores

- A aplicação de funções aplica-se da esquerda para a direita, logo não é preciso usar parâmetros.

```
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
```

Composição de funções

```
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
```

Tipo Seta (Função)

O tipo função em ocaml

- O tipo função corresponde à implicação na correspondência (Curry-Howard).
- Avaliação em lambda calculus corresponde à dedução natural.
- Corresponde à definição de contradomínio e domínio (conjunto resultado) na correspondência entre tipos e conjuntos.
- Declara o tipo dos valores aceites como argumento (tipo do parâmetro no corpo da função) e o tipo do resultado da função (tipo da expressão corpo da função)

A -> B

- O tipo função primitivo admite apenas um parâmetro e um resultado. Para termos funções que aceitam vários parâmetros e devolvem vários resultados usamos tipos compostos.

Vários parâmetros numa função

- O tipo seta é associativo à direita
- Vários parâmetros A, B, C e resultado D

$$A \rightarrow B \rightarrow C \rightarrow D$$

- (A mesma coisa) Um parâmetro e uma função como resultado

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

- (Outra coisa) Uma função como parâmetro e um resultado

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

Anotações de tipos

- A linguagem OCaml tem inferência de tipos, e é essa a forma natural de escrever programas. No entanto, é possível declarar tipos e anotar as expressões com tipos para que possamos a complexidade de alguns programas.

```
let f (x:int) (y:int) = x + y
```

Dedução natural: exercício

- Considere os predicados representados pelos tipos: “a”, “b”, “c”, “d”
- Considere os componentes com anotações de tipo: “x”, “f”, “g”, “h”, “i”
(note que a definição dos nomes não é importante para este exercício)
- É capaz de produzir uma expressão do tipo “d” ?

```
(* Predicates *)
type a = int
type b = string
type c = float
type d = char

(* 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 =
```

Dedução natural: exercício

- Considere os predicados representados pelos tipos: “a”, “b”, “c”, “d”
- Considere os componentes com anotações de tipo: “x”, “f”, “g”, “h”, “i”
(note que a definição dos nomes não é importante para este exercício)
- É capaz de produzir uma expressão do tipo “d” ?

```
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 =
```

Síntese de programação baseada em “componentes” e tipos

- O sistema de tipos é uma peça fundamental em características avançadas dos ambientes de programação modernos, como a síntese de programas.

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) components and a query *type*, the goal is to synthesize a *term* that inhabits the query. Classical approaches based on proof search in intuitionistic logics do not scale up to the standard libraries of modern languages, which contain hundreds or thousands of components. Recent graph reachability based methods proposed for languages like Java do scale, but only apply to monomorphic data and components: polymorphic data and components

Hoogle \star : 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.

Inferência de tipos (I)

Inferência de tipos

- A linguagem OCaml usa o “Hindley–Milner type inference algorithm” para tipificar as suas expressões.
- Trata-se de encontrar o tipo principal (mais genérico) para cada expressão com base nos seus componentes.
- Luis Damas e Robin Milner definiram o algoritmo para uma linguagem com tipos polimórficos.

Principal type-schemes for functional programs

Luis Damas* and Robin Milner
Edinburgh University

1. Introduction

This paper is concerned with the polymorphic type discipline of ML, which is a general purpose functional programming language, although it was first introduced as a metalanguage (whence its name) for conducting proofs in the LCF proof system [GMW]. The type discipline was studied in [Mil], where it was shown to be semantically sound. In a

of successful use of the language, both in LCF and other research and in teaching to undergraduates, it has become important to answer these questions – particularly because the combination of flexibility (due to polymorphism), robustness (due to semantic soundness) and detection of errors at compile time has proved to be one of the strongest aspects of ML.

Inferência de tipos

- O sistema de tipos determina algorítmicamente qual o tipo de uma expressão através da análise das suas componentes básicas
- Exemplos:
 - (+) tem como resultado um valor int e aceita operandos int
 - (.) tem como resultado um valor float e aceita operandos float
 - (=) tem como resultado um valor bool e aceita operandos que tenham o mesmo tipo (ou mais que um???)
- Numa função (`fun x → x+1`) não há outra solução do que o tipo da função ser `int → int`
- Já numa função (`fun x y → x = y`) há muitas soluções possíveis...

Inferência de tipos

- Exercícios, que soluções existem para o tipo destas expressões?

`fun x → if x = 1 then "hello" else "bye"`

`let x = "world" in "Hello, " ^ x`

`fun x y → if x = "Hello" then int_of_string y else "World"`

`fun x y → if x = "Hello" then int_of_string y else 0`

`fun x y z → if x then y else z (what now??)`

Polimorfismo

Inferência de tipos (II)