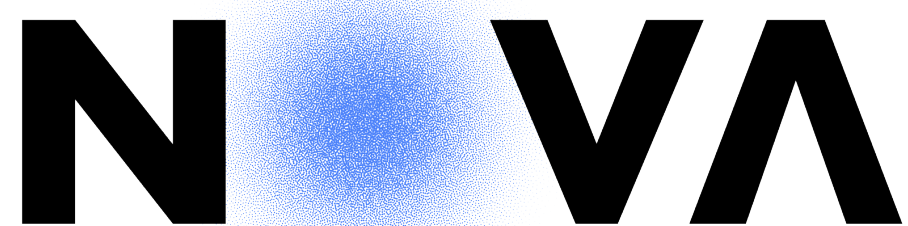


Linguagens e Ambientes de Programação (Aula Teórica 5)

LEI - Licenciatura em Engenharia Informática

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



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Tipo função.
- Polimorfismo.
- Inferência de tipos.

Composição de funções (recap)

```
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
```

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
```

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)

$A \rightarrow B$

- 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

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

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 =
```

```
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 =
```


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 to proof search in intuitionistic logics do not scale up to the standard libraries of modern languages, with 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★: 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

Qual a solução para o problema?

- O tipo de `id`?

```
let id x = x
```

✓ 0.0s

```
id true
```

✓ 0.0s

```
- : bool = true
```

```
id "hello"
```

✓ 0.0s

```
- : string = "hello"
```

```
id (fun x → x + 1)
```

✓ 0.0s

```
- : int → int = <fun>
```

```
id 1
```

✓ 0.0s

```
- : int = 1
```

```
id 'a'
```

✓ 0.0s

```
- : char = 'a'
```

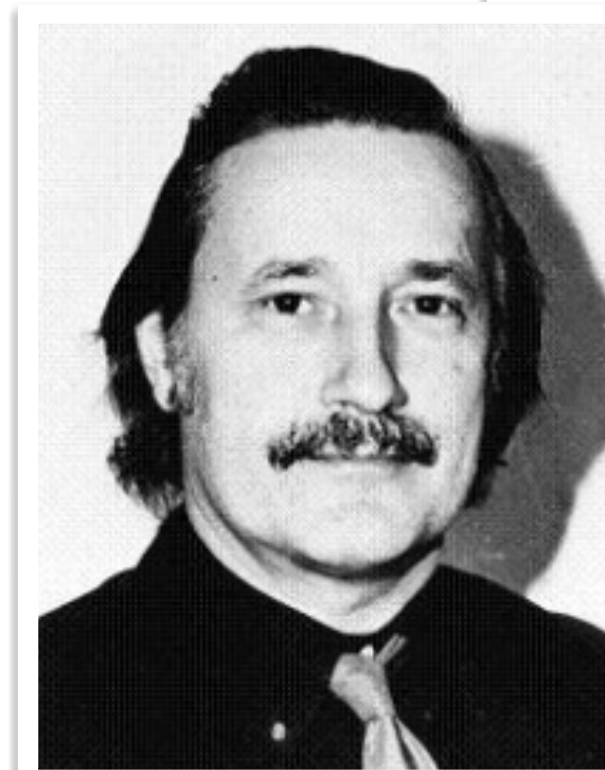
```
id (fun f x y → 1 + f (1+x) (2+y))
```

✓ 0.0s

```
- : (int → int → int) → int → int → int = <fun>
```


Polimorfismo

- Muitas formas: Um símbolo, múltiplos tipos
- Polimorfismo Ad-hoc
(Algol 68 - descrito por Christopher Strachey, 67)
- Polimorfismo Paramétrico
(ML - descrito por Christopher Strachey, 67)
- Polimorfismo por inclusão
(Subtyping) - (Simula, Wegner/Cardelli, 85)



Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY

Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

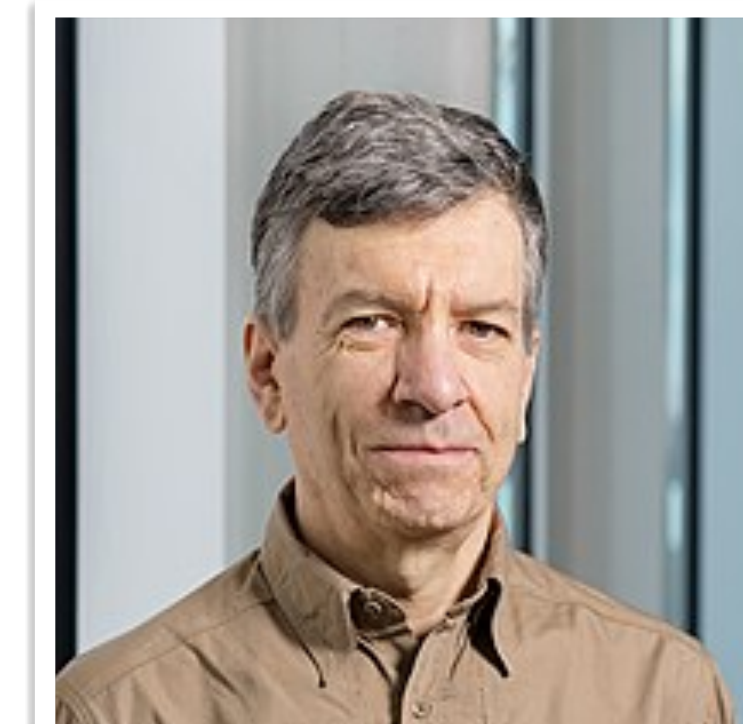
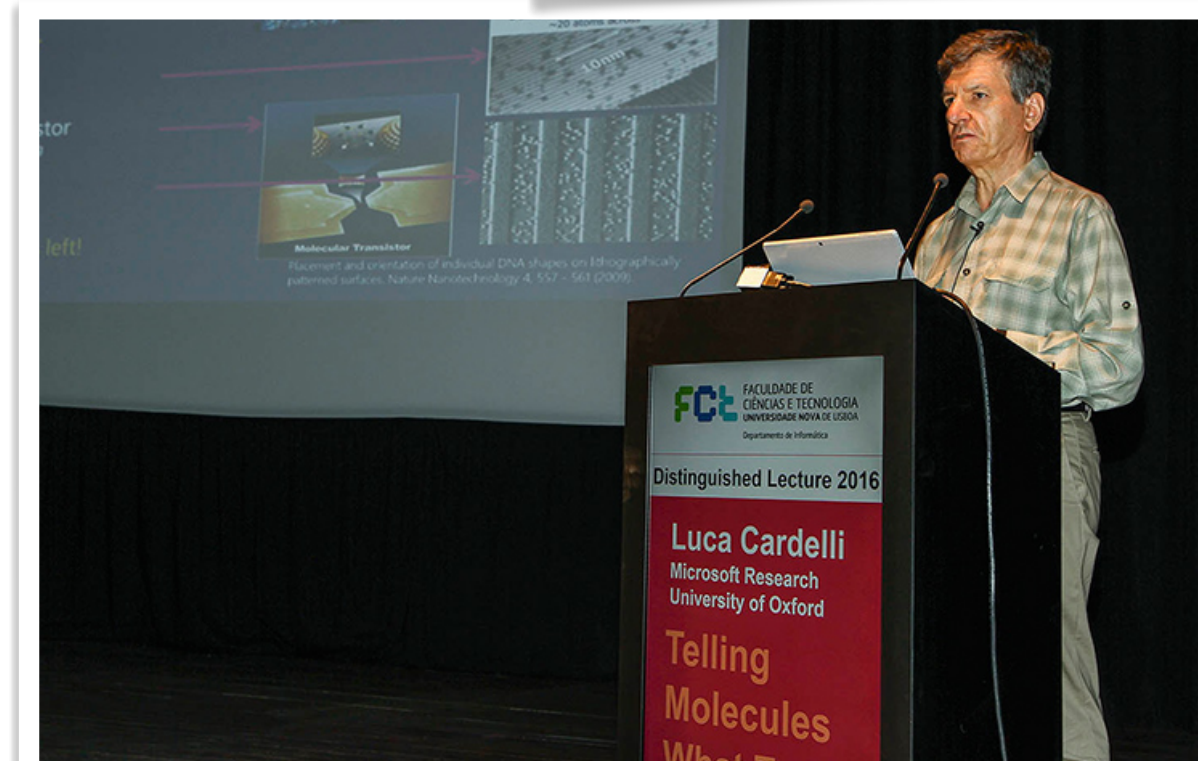
On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli

AT&T Bell Laboratories, Murray Hill, NJ 07974
(current address: DEC SRC, 130 Lytton Ave, Palo Alto CA 94301)

Peter Wegner

Dept. of Computer Science, Brown University
Providence, RI 02912



Polimorfismo Ad-Hoc

- O overloading de nomes é uma forma de polimorfismo.

void	print (boolean b) Prints a boolean value.
void	print (char c) Prints a character.
void	print (char[] s) Prints an array of characters.
void	print (double d) Prints a double-precision floating-point number.
void	print (float f) Prints a floating-point number.
void	print (int i) Prints an integer.
void	print (long l) Prints a long integer.
void	print (Object obj) Prints an object.
void	print (String s) Prints a string.

Polimorfismo paramétrico

- Tipos genéricos, variáveis de tipo
- Abstração do tipo dos tipos de valores a guardar/processar por uma classe/função
- Abstração + Parametrização

```
let id x = x
```

✓ 0.0s

```
val id : 'a → 'a = <fun>
```

```
java.util
```

```
Class Vector<E>
```

```
java.lang.Object
```

```
    java.util.AbstractCollection<E>
```

```
        java.util.AbstractList<E>
```

```
            java.util.Vector<E>
```

```
function identity<Type>(arg: Type): Type {  
    return arg;  
}
```

Polimorfismo de inclusão (ou Universal)

- Ligado à noção de Subtyping
- Inclusão vem da interpretação de tipos como conjuntos.
 - Por nome (Interfaces e classes Java)
 - Estrutural (comparação de objectos em typescript)
- Implementado pela noção de herança que estende a noção de polimorfismo de inclusão com a noção de extensão de código.

Inferência de tipos (II)

Inferência de tipos com variáveis de tipo.

- É preciso inferir o tipo mais genérico que seja uma solução para a tipificação de uma expressão.

```
let equal x y = x = y
✓ 0.0s
val equal : 'a → 'a → bool = <fun>
```

```
let decide x y z = if x then y = z else y < z
✓ 0.0s
val decide : bool → 'a → 'a → bool = <fun>
```

```
let comp f g = fun x → f(g(x))
✓ 0.0s
val comp : ('a → 'b) → ('c → 'a) → 'c → 'b = <fun>
```

Algoritmo de inferência de tipos (intuição)

- Aos literais são dados os seus tipos naturais
- Às funções são dados tipos seta
 - Aos parâmetros de uma função são dadas variáveis de tipo novas
 - Ao resultado das funções são atribuídos os tipos das expressões no seu corpo
- Colecionar as restrições de tipo das várias expressões/operadores (Equações)
- Resolver o sistema de equações resultante
 - 1 solução: encontrou-se um tipo
 - 0 soluções: programa mal tipificado

Inferência de tipos (Exercícios)

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

`fun x y z → if x then y else z`

```
fun w → if w
      then
        fun x y → x = y
      else
        fun x y → x <> y
```