Programming Languages and Environments (Lecture 5)

LEI - Licenciatura em Engenharia Informática

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

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Syllabus

- Function type.
- Polymorphism.
- Type inference.

Function composition (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
```

Recap: Lambda Calculus

$$E ::= \; x \mid \; \lambda x.E \; \mid \; E \; E$$

$$(\lambda x.E) \; E' \longrightarrow E\{E'/x\}$$

$$\frac{E \longrightarrow E''}{E \; E' \longrightarrow E'' \; E'}$$

Encoding of Let expressions

- A demonstration how let expressions can be encoded in pure lambda calculus
- A let expression declares a name, so does a function (the parameter)
- A function call defines the value of a name, so does the let expression

```
let x = 1 in x + 1;;
```

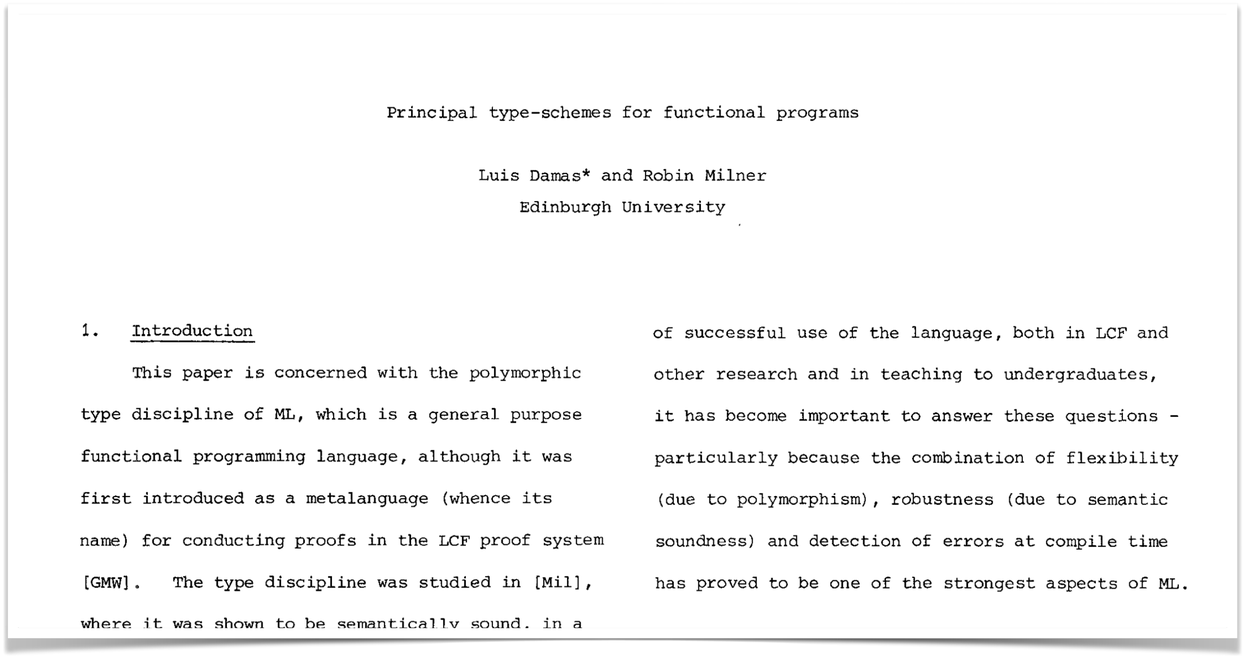
```
(fun x -> x + 1) 1;;
```

- The actual implementation is not the same, but conceptually, they are equivalent

Type Inference (I)

Type inference

- OCaml uses the "Hindley-Milner type inference algorithm" to type its expressions.
- It involves finding the principal (most generic) type for each expression based on its components.
- Luís Damas and Robin Milner defined the algorithm for a language with polymorphic types.



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.

Type inference

- The type system algorithmically determines the type of an expression by analyzing its basic components.
- Examples:
  - `(+)` results in an `int` value and accepts `int` operands.
  - `(+.)` results in an `float` value and accepts `float` operands.
  - `(=)` results in a `bool` values and accepts operands that have the same type.
- In a function `(fun x -> x + 1)` there is no other solution than the type of the function being `int -> int`
- While in a function `(fun x y -> x = y)` there are many possible solutions...

Type inference

- What solutions exist for the type of these expressions?

```
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??) *)
```

Type annotations

- Although OCaml has type inference, one can explicitly state the types of the parameters and return type to guide disambiguation or to debug unexpected type errors.
- We can annotate the types of the arguments

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

- You have to be wary when it comes to annotating types. It can lead to ill-typed programs.

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

Polymorphism

Identity function

- What is the the type of `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>
```

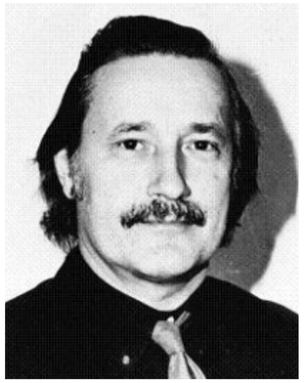
Polymorphism

- Expressiveness: One symbol, many types
  - Ad-hoc Polymorphism
    (Algol 68 - described by Christopher Strachey, 67)
  - Parametric Polymorphism
    (ML - described by Christopher Strachey, 67)
  - Polymorphism by Inclusion
    (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

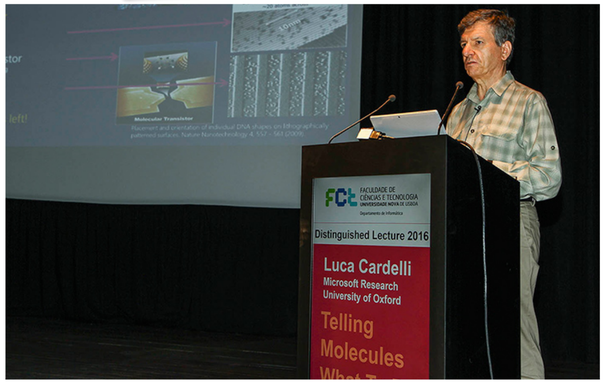
Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the
any of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is
at sort of audience there will be for such lectures—and an associated switching from formal
f presentation which may well be less acceptable in print than it is natural in the lecture room.
) faults, I apologise to the reader.



On Understanding Types, 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



Ad-Hoc polymorphism

- Ad-hoc polymorphism refers to the ability of different functions to have the same name.
- It is achieved through function or operator **overloading**.
- The method or operator is chosen at compile-time

`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.

Parametric polymorphism

- Allows a function to work with any data type.
- Abstraction with the same type as the type of the values to be processed by a class/function.
- Abstraction + Parametrization
- These are the generics in Java or C++.

java.util

Class Vector<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>

```
let id x = x
✓ 0.0s
val id : 'a → 'a = <fun>
```

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

Inclusion polymorphism (Universal)

- Tied to the notion of Subtyping.
- Derives from the interpretation of types as sets.
  - By name (Interfaces and Classes in Java)
  - Structural (Object comparison in Typescript)
- Implemented via inheritance that extends the notion of inclusion polymorphism with the notion of code extension.

Type Inference (II)

Type inference with type variables

- It is common that the type inference algorithm cannot define a concrete type for an expression. In that case the expression is left with an abstract type. These types are read as greek letters:
  - `'a` is read as alpha
  - `'b` is read as beta
  - so on and so forth

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

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

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

Type inference algorithm (Rough idea)

- Literals are given their natural types.
- Functions are given arrow types.
  - Function parameters are given new type variables.
  - The result of functions is assigned the types of the expressions in their body.
- Collect the type constraints from various expressions/operators (Equations).
- Solve the result system of equations:
  - 1 solution → a type has been found.
  - 0 solutions → the program is ill-typed.

Type inference (Exercise)

- How many solutions are there for the types of these expressions?

```
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
```