# Fundamental Concepts of Programming Languages

Functional Programming Fundamentals

Lecture 12

conf. dr. ing. Ciprian-Bogdan Chirila

# Outline

- Lambda calculus
  - Lambda expressions and functions
  - β reductions
  - Variable binding. Free variables and bound variables
  - Name conflicts. Alfa conversion
  - η reduction
  - Boolean values and conditional expressions
  - Applied λ calculus
  - Evaluation order
  - Church-Rosser Theorems

# Outline

- Lazy evaluation
- High order functions
- Types. Polymorphism
  - Types and type variables
  - Polymorphism
  - Type inference

# Introduction

- Functional programming languages
  - Based on computations with functions
- The execution of a pure functional program
  - The evaluation of expressions that contain function calls
- Functional programs advantages
  - Are wrote fast
  - Are more concise
  - Are high level
  - Good for formal checking
  - Can be executed fast on parallel architectures

# Referential transparence

- Important characteristic of functional programming
  - There are no side effects !!!
- Pure functional language
  - Assures the referential transparence

- The semantic of a construction
- and
- the value resulted from the evaluation
  - depend exclusively only on the semantic of its components

# Referential transparence example

- For the expression (f+g)*(x+y) the semantic and thus the value depend only on:
  - f+g
  - x+y
- For the subexpression f+g the semantic and thus the value depend only on:
  - f and g
  - and it is independent of (x+y)
- For the subexpression x+y the semantic and thus the value depend only on:
  - x and y
  - and it is independent of (f+g)

# Referential transparence

- Allows substitution of expressions with the same semantic
- Thus, we can replace
  - (x+y)*z with x*z+y*z
- The value of the expression does not depend on evaluation order
  - x*z can be replaced with z*x

# Variables and assignments

- make an expression depend on the history of the program execution
- especially global variables
- side effects
- imperative languages
- and
- non pure functional
  - referencial transparency is not enabled

# Variables and assignments

- example giving
- if f and g are functions depending on global variable
  - then the very same expression (f+g)*(x+y)
  - may provide different values on several evaluations
  - depending on the global variable

# Variables and assignments

- example giving
- the expression (x+y)*f will not have the same value with
- x*f+y*f
  - if f is a function which modifies the value of y

# Transparency property

- is very important
- influences the readability of
  - programs
  - analysis
  - automatic formal checking
- it is one of the main property of functional pure languages

# Lambda Calculus

- developed by mathematician Alonzo Church in the 30’s
- Church presents a simple mathematical system that allows formalization of
  - programming laguages
  - programming in general
- the notation may seem unusual
- it can be viewed as a simple functional language

# Lambda Calculus

- from it we can be develop all the other modern programming languages features
- it can be used as a universal code in translating functional languages
  - simple but not necessarily efficient technique
- it can be easily interpreted
- is a mathematical system to manipulate the so called λ  expressions

# a λ expression

- a name
  - string of characters
- a function
- the application of a function

# The function

- λname.body
- name preceded by λ is called the bound variable of the function
  - similar to a formal parameter
- body is a λ expressions
- the function has no name

# The application of a function

- has the form (expression expression)
  - the first expression is a function
  - the second expression is the argument
- represents a concretization of the function
- the name specified as a bound variable in the expression will be replaced with the argument

# Examples

- identity function
- autoapplication function

# Identity function

- **λx.x**
- bound variable
  - first **x**
- body
  - the second **x**
- **(λx.x a)** results in a
- the argument can be a function itself
- **(λx.x λx.x)** results in **λx.x**

# Auto-application function

- **`λ a.(a a)`**
  - **`a`** – is the bound variable
  - **`(a a)`** – is the body
- passing an argument to this function the effect is that the argument is applied to itself
- If we apply auto-application to the identity function
  - **`(λa.(a a) λx.x)`** results **`λx.x`**
- If we apply the auto-application function to itself
  - **`(λa.(a a) λa.(a a))`** results in **`(λa.(a a) λa.(a a))`** …
  - the auto-application never ends

# β reduction

- In order to simplify the writing of λ expressions we will introduce a notation that allows us to associate a name with a function
- `def identity = λx.x`
- `def auto-application= λa.(a a)`
- `(name argument)`
  - the application of the name to the specified argument
- `(name argument)` is similar to `(function argument)`
  - where the name was associated with the function

# β reduction

- is to replace a bound variable with the argument specified in the application
- as many times as it occurs in the function body
- **`(function argument) => expression`**
  - after one β reduction in the application from the left results in the expression from the right
- **`(function argument) => … => expressions`**
  - the expression is obtained after several β reductions

# Examples
# Selecting the first argument

- `def sel_first=λfirst.λsecond.first`
  - `first` – bound variable
  - `λsecond.first` – the body
- `((sel_first arg1)arg2)==`
- `((λfirst.λsecond.first arg1)arg2)=>`
- `(λsecond.arg1 arg2) => arg1`

- applied to a pair of arguments arg1 and arg2
- the function returns the first argument arg1
- the second argument arg2 is ignored

# Comments

- in order to simplify notation we can skip the parentheses
- when there are no ambiguities
- to apply two arguments to sel_first function can be denoted
- **`sel_first arg1 arg2`**
- the notation is of a function with two parameters
- in λ calculus such functions are expressed through nested functions
- the function **`λfirst.λsecond.first`** applied to a random argument (arg1) result in a function
- **`λsecond.arg1`**
- that applied to any other second argument returns arg1

# Examples
# Selecting the second argument

- `def sel_second=λfirst.λsecond.second`
- `sel_second arg1 arg2 ==`
- `λsecond.second arg2 => arg2`

# Examples
# Building a tuple of values

- `def build_tuple arg1 arg2 ==`
- `λfirst.λsecond.λf.(f first second) arg1 arg2 =>`
- `λsecond. λf.(f arg1 second) arg2 =>`
- `λf.(f arg1 arg2)`

- `λf.(f arg1 arg2) sel_first=>`
- `sel_first arg1 arg2 => … =>arg1`

- `λf.(f arg1 arg2) sel_second=>`
- `sel_second arg1 arg2 => … =>arg2`

# Variables bounding
# Free and bound variables

- the issues addressed are similar to variables domain from a programming language
- arguments substitution in the body of a function are well accomplished when bound variables in function expressions are named differently
- `(λf.(f λx.x) λa.(a a))`
- the three involved functions in the expression have `f`, `x` and `a` as bound variables
- `(λf.(f λx.x) λa.(a a)) =>`
- `(λa.(a a) λx.x) =>`
- `(λx.x λx.x) => λx.x`

# Variables bounding
# Free and bound variables

- **`(λf.(f λx.x) λa.(a a))`**
- expression can be written like:
- **`(λf.(f λf.f) λa.(a a))`** with the **`λf.f`** result after the substitution
- at the first substitution the f bound variable is replaced in function **`λf.(f λf.f)`** with **`λa.(a a)`**
- implies the replacement of the first f in the expression **`(f λf.f)`**
- we do not replace **`f`** from the body of the function **`λf.f`**
- in the new function **`f`** is a new bound variable
- accidentally they have the same name

# The domain of the bound variable of a function

- given the function
- `λname.body`
- the domain of the name bound variable is over the function body
- the occurrences of the same name outside the function body does not correspond to the bound variable

# Examples

- considering the expression
- `(λf. λg. λa.(f(g a)) λg.(g g))`
- the domain of the f bound variable is expression
- `λg. λa.(f(g a))`
- the domain of the g bound variable is expression
- `λa.(f(g a))`
- the domain of the g variable is the expression
- `(g g)`

# The domain of the bound variable of a function

- bound variable
- the occurrence of a variable v in an expression E is bound if it is present in an subexpression of E in the form λv.E1
  - v appears in the body of a function with a bound variable called v
- otherwise the occurrence of v is free variable

# More examples

- `v(a b v)`
  - v is free
- `λv.v(x y v)`
  - v is bound
- `v(λv.(y v) y)`
  - v is free in the first occurrence
  - v is bound in the second occurrence

# Variable domain definition

- given the function
- **`λname.body`**
- the domain of the bound variable name extends over the body sequences in which the occurrence of name is free

# Example

- given the expression
- `λg.(g λh.(h(g λh.(h λg.(h g))) g)`
- to analyze the domain of g
- the appearances of g outside the marked zone are free

# β reduction definition

- **`(λname.body argument)`**
- to replace all the free occurrences of name from the body with argument

# Initial example revisited

- `(λf.(f λf.f) λa.(a a))`
- the applied function is
- `λf.(f λf.f)`
- its body is
- `(f λf.f)`
- the first and only the first occurrence of f is free and it will be replaced with the argument specified in the application
- `(λa.(a a) λf.f) => (λf.f λf.f) => λf.f`

# Name conflicts a conversion

# Bibliography


- Horia Ciocarlie – The programming language universe, second edition, Timisoara, 2013
- Carlo  Ghezzi, Mehdi Jarayeri – Programming Languages, John Wiley, 1987.
- Ellis Horrowitz – Fundamentals of programming languages, Computer Science Press, 1984.
- Donald Knuth – The art of computer programming, 2002.