Fundamental Concepts of Programming Languages

Functional Programming Fundamentals
Lecture 12

conf. dr. ing. Ciprian-Bogdan Chirila

University Politehnica Timisoara
Department of Computing and Information Technology

January 8, 2023

FCPL - 12 - Functional Programming Fundamentals

FCPL - 12 - Functional Programming Fundamentals

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 transparency

- Important characteristic of functional programming
  - There are no side effects !!!
- Pure functional language
  - Assures the referential transparency
- The semantic of a construction and
  the value resulted from the evaluation
  - depend exclusively only on the semantic of its components

Referential transparency 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 transparency

- 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 assignment

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

Variables and assignment

- example:
- 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 assignment

- example:
- 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

FCPL - 12 - Functional Programming Fundamentals

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 (LC)

- from LC we can develop all the other modern programming languages features
- it can be used as a universal code in translating functional languages
    - it is simple, but not necessarily an efficient technique
- it can be easily interpreted
- it 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 expression
- 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
- auto-application 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
  - `(`λ`a.(a a)` λ`x.x)` results λ`x.x`
- If we apply the auto-application function to itself
  - `(`λ`a.(a a)` λ`a.(a a))` results in `(`λ`a.(a a)` λ`a.(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 =` λ`x.x`
  - `def auto-application=` λ`a.(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

Comments

- in λ calculus such functions are expressed through nested functions
- the function $\lambda\texttt{first}.\lambda\texttt{second}.\texttt{first}$ applied to a random argument (arg1) result in a function
- $\lambda\texttt{second}.\texttt{arg1}$
- 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 binding. 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
- $(\lambda\texttt{f.(f }\lambda\texttt{x.x) }\lambda\texttt{a.(a a))}$
- the three involved functions in the expression have `f`, `x` and `a` as bound variables
- $(\lambda\texttt{f.(f }\lambda\texttt{x.x) }\lambda\texttt{a.(a a))} \Rightarrow$
- $(\lambda\texttt{a.(a a) }\lambda\texttt{x.x)} \Rightarrow$
- $(\lambda\texttt{x.x }\lambda\texttt{x.x)} \Rightarrow \lambda\texttt{x.x}$

Variables binding. Free and bound variables

- `(`λ`f.(f `λ`x.x) `λ`a.(a a))`
- expression can be written like:
- `(`λ`f.(f `λ`f.f) `λ`a.(a a))` with the λ`f.f` result after the substitution
- for the first substitution the `f` bound variable is replaced in function λ`f.(f `λ`f.f)` with λ`a.(a a)`
- this implies the replacement of the first `f` from the expression `(f `λ`f.f)`
- it results `(`λ`a.(a a) `λ`f.f)` which can be further reduced

Variables binding. Free and bound variables

- we do not replace `f` from the body of the function λ`f.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
- λ`name.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)`

Bound variable definition

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

More examples

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

Variable domain definition

- given the function
- λ`name.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)`
- we establish the domain of g by analyzing the function body
  `(g λh.(h(g λh.(h λg.(h g)))) g)`
- the appearances of g outside the red marked zone are free

β reduction definition

- given the application `(λname.body argument)`
- we 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

FCPL - 12 - Functional Programming Fundamentals

β reduction strong definition

- given an application (λname.body argument)
- we replace all occurences of `name` from the `body` with the `argument`
- e.g. `(`λ`f.(f `λ`f.f) `λ`a.(a a))`
- the applied function is λ`f.(f `λ`f.f)`
- its body is `(f `λ`f.f)`
- `(`λ`a.(a a) `λ`f.f)`
- `(`λ`f.f `λ`f.f)`
- λ`f.f`

FCPL - 12 - Functional Programming Fundamentals

Name conflicts. Alfa conversions

- applying a β reduction, name conflicts may arrise
- e.g.:

```
def f=λx.λy.(x y)
f x y == (λx.λy.(x y) y z)
=> (λy.(y y) z)
=> z z
```

the result is errorneous

the error may be corrected like:

```
(λx.λy1.(x y1) y z)
=> (λy1.(y y1) z)
=> y z
```

Name conflicts. Alfa conversions

Given a function
λname1.body
the name of the bound variable `name1` and also the free appearances of the `name1` inside the function body may be replaced with a new name, `name2` given the condition that in λname1.body appears no free variable named `name2`
The function λ`y.(x y)` was transformed in function λ`y1.(x y1)`

FCPL - 12 - Functional Programming Fundamentals

Mu reduction

- μ reduction is a transformation that (like β reduction) allows the replacement of a λ expression with an equivalent, simpler one
- given the function
  λ`name.(expression name)`
  it is equivalent to:
  `expression`
- λ`name.(expression name) argument`
  `=> (expression argument)`

FCPL - 12 - Functional Programming Fundamentals

Applied λ calculus

- involves logical values
- involves logical operations
- the C ternary operator
  `condition ? ex1 : ex2`
- we model the logical values with the following functions: `sel_first`, `sel_second`, `build_tuple`

Applied λ calculus

def cond=λe1.λe2.λc.(c e1 e2)

we apply this function succesively to expressions ex1 and ex2:

cond ex1 ex2 ==

λe1.λe2.λc.(c e1 e2) ex1 ex2=>

λe2.λc.(c ex1 e2) ex2=>

λc.(c ex1 ex2)

Applied λ calculus

the `true` and `false` values will be represented by the `sel_first` and `sel_second` functions

```
def true = λp.λs.p
def false = λp.λs.s
```

resulting:

```
cond ex1 ex2 true => ...  =>
λc.(c ex1 ex2) λp.λs.p =>
λp.λs.p ex1 ex2 => ...  => ex1
```

similarly:

```
cond ex1 ex2 false => ...  =>
λc.(c ex1 ex2) λp.λs.s =>
λp.λs.s ex1 ex2 => ...  => ex2
```

FCPL - 12 - Functional Programming Fundamentals

The NOT logical operator

```
def not=λx.(cond false true x)
```

e.g.:

```
not true == λx.(cond false true x) true =>
cond false true true => ...  => false
```

conversely

```
not false == λx.(cond false true x) false =>
cond false true false => ...  => true
```

The AND logical operator

def and=λx.λy.(cond y false x)

e.g.:

we compute true AND false

(and true false) ==

λx.λy.(cond y false x) true false => ... =>

cond false false true => ... => false

we compute false AND true

(and false true) ==

λx.λy.(cond y false x) false true => ... =>

cond true false false => ... => false

The AND logical operator

we compute `NOT false AND true`

```
(and (not false) true) ==
λx.λy.(cond y false x) (λx.(cond false true
x)) true => ... =>
λx.λy.(cond y false x) true true => ... =>
cond true false true => ... => true
```

The OR logical operator

```
def or=λx.λy.(cond true y x)
```

e.g.:

we compute `true OR false`

```
(or true false) ==
λx.λy.(cond true y x) true false => ...  =>
cond true false true => ...  => true
```

FCPL - 12 - Functional Programming Fundamentals

Bibliography

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