Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Course introduction

- Course topic: what is this course about?
- Examination: how will you be tested?
- Start with course

Introduction

Course topic: functional programming

- Lambda calculus
- From lambda calculus to functional programming
- Functional programming using F$^\sharp$and Haskell

Advantages of functional programming

- Strong mathematical foundations
- Easier to reason about programs
- Parallelism for “free”
- Correctness guarantees through strong typing (optional)

Introduction

Examination

- Theory exam: test understanding of theory
- Practical exam: test ability to apply theory in practice

Theory exam: reduction and typing

- One question on reduction in lambda calculus
- One question on typing in lambda calculus, F$^\sharp$, or Haskell
- **Passing grade** if both questions answered correctly

Introduction

Practical exam: interpreter for a virtual machine

- In a group, build an interpreter for a virtual machine
- According to a specification that will be provided
- Groups may consist of up to 4 students
- Understanding of code tested **individually**

Lecture topics

- Semantics(meaning) of imperative languages
- Lambda calculus, the foundation for functional languages

Semantics of imperative languages

Semantics of imperative languages

Imperative program: sequence of statements

- Statements directly depend on and alter memory
- Meaning of statements may depend on contents of memory
- Any statement may depend on (read) any memory location
- Any statement may alter any memory location

Example: meaning of statement sequence

- Statement s_1 changes the machine state from S_0 to S_1
- Statement s_2 changes the machine state from S_1 to S_2
- Run statement s_1, then run statement s_2: $s_1 s_2$
- Statement $s_1 s_2$ changes the machine state from S_0 to S_2

$$(S_0 \xmapsto{s_1} S_1) \wedge (S_1 \xmapsto{s_2} S_2) \implies S_0 \xmapsto{s_1 s_2} S_2$$

Semantics of imperative languages

Example: meaning of statement sequence

- Statement s_1 changes the machine state from S_0 to S_1
- Statement s_2 changes the machine state from S_1 to S_2
- Run statement s_1, then run statement s_2: $s_1 s_2$
- Statement $s_1 s_2$ changes the machine state from S_0 to S_2

$$(S_0 \xmapsto{s_1} S_1) \land (S_1 \xmapsto{s_2} S_2) \implies S_0 \xmapsto{s_1 s_2} S_2$$

What about $s_2 s_1$?

Semantics of imperative languages

Swap order of $s_1 s_2$: $s_2 s_1$

- Sometimes $s_2 s_1$ has the same meaning as $s_1 s_2 \ldots$
- Sometimes $s_2 s_1$ is completely different from $s_1 s_2$!
- It depends on s_1, s_2, and the relevant machine state S_0
- It depends on implementation details of s_1 and s_2
- Implementation details matter $\Longrightarrow$ leaky abstraction!

Semantics of imperative languages

Swap order of $s_1 s_2$: $s_2 s_1$

- Sometimes $s_2 s_1$ has the same meaning as $s_1 s_2 \ldots$
- Sometimes $s_2 s_1$ is completely different from $s_1 s_2$!
- It depends on s_1, s_2, and the relevant machine state S_0
- It depends on implementation details of s_1 and s_2
- Implementation details matter $\Longrightarrow$ leaky abstraction!

Can we do better?

Semantics of imperative languages

Idea for better abstraction: remove implicit dependencies

- No implicit dependencies $\implies$ all dependencies explicit
- No access to arbitrary machine state
- Only explicitly-mentioned state may be accessed

Semantics of imperative languages

Idea for better abstraction: remove implicit dependencies

- No implicit dependencies $\Longrightarrow$ all dependencies explicit
- No access to arbitrary machine state
- Only explicitly-mentioned state may be accessed

What if s_1 and s_2 only read the same state?

What if $s_1\{x\}$ and $s_2\{x\}$ only read the same state x?

- $s_1\{x\}$ calculates $x + x$, and $s_2\{x\}$ calculates the square x^2

Can we reorder $s_1\{x\}$ and $s_2\{x\}$?

What if $s_1\{x\}$ and $s_2\{x\}$ only read the same state x?

- $s_1\{x\}$ calculates $x+x$, and $s_2\{x\}$ calculates the square x^2

Can we reorder $s_1\{x\}$ and $s_2\{x\}$?

What if $s_1\{x\}$ and $s_2\{x\}$ alter the same state x?

- $s_1\{x\}$ sets x to 1, and $s_2\{x\}$ sets x to 2

Can we reorder $s_1\{x\}$ and $s_2\{x\}$?

Semantics of imperative languages

What if $s_1\{x\}$ and $s_2\{x\}$ only read the same state x?

- $s_1\{x\}$ calculates $x + x$, and $s_2\{x\}$ calculates the square x^2

Can we reorder $s_1\{x\}$ and $s_2\{x\}$?

What if $s_1\{x\}$ and $s_2\{x\}$ alter the same state x?

- $s_1\{x\}$ sets x to 1, and $s_2\{x\}$ sets x to 2

Can we reorder $s_1\{x\}$ and $s_2\{x\}$?

Can we do better?

Semantics of imperative languages

Idea for better abstraction: remove implicit dependencies

- No implicit dependencies $\Longrightarrow$ all dependencies explicit
- No reading if arbitrary machine state
- No mutating of arbitrary machine state
- Only explicitly-mentioned machine state may be read

Idea for better abstraction: remove implicit dependencies

- No implicit dependencies $\Longrightarrow$ all dependencies explicit
- No reading if arbitrary machine state
- No mutating of arbitrary machine state
- Only explicitly-mentioned machine state may be read

NB: No provision at all is made for mutating machine state

Semantics of imperative languages

Wait a minute, this is just like functions

- Not statements, but (mathematical) functions
- Functions depend only on arguments
- Functions do not alter state
- Can calculate function value when all arguments are known
- Can always replace a function call by its value

Referential transparency:

It is always valid to replace a function call by its value

Referential transparency:

It is always valid to replace a function call by its value

Advanced topic:

Allow mutation of state without losing referential transparency

Lambda calculus

Lambda calculus

What is lambda calculus?

- Model of computation based on functions
- Completely different from Turing machines, but equivalent
- Foundation of all functional programming languages
- Truly tiny when compared with its power
- Consists of only (function) abstraction and application

Lambda calculus

Substitution principle

- The (basic) lambda calculus is truly tiny when compared with its power.
- It is based on the substitution principle: calling a function with some parameters returns the function body with the variables replaced.
- There is no memory and no program counter: all we need to know is stored inside the body of the program itself.

Lambda calculus

A lambda calculus term is one of three things:

- a variable (from some arbitrary infinite set of variables)
- an abstraction (a "function of one variable")
- an application (a "function call")

Lambda calculus

Variables (arbitrary infinite set):

$a, b, c, \dots$ $a_0, a_1, \dots$ $b_0, b_1, \dots$

Abstractions:

For any variable x and lambda term T: $\lambda x.T$

Applications:

For any lambda terms F and T: (FT)

Lambda calculus

- Infinite set of variables: $x_0, x_1, \ldots, y_0, y_1, \ldots$, etc.
- Abstractions (function declarations with one parameter): $\lambda x \rightarrow t$ where x is a variable and t is the function body (a program).
- Applications (function calls with one argument): $t\ u$ where t is the function being called (a program) and u is its argument (another program).

Lambda calculus

A simple example would be the identity function, which just returns whatever it gets as input

```
(λx.x)
```

Lambda calculus

We can call this function with a variable as argument, by writing:

```
((λx.x) v)
```

Lambda calculus

A lambda calculus program is computed by replacing lambda abstractions applied to arguments with the body of the lambda abstraction with the argument instead of the lambda parameter:

Lambda calculus

A lambda calculus program is computed by replacing lambda abstractions applied to arguments with the body of the lambda abstraction with the argument instead of the lambda parameter:

$$\frac{}{(\lambda x \rightarrow t)\ u \rightarrow_\beta t[x \mapsto u]}$$

$t[x \mapsto u]$ means that we change variable x with u within t

Lambda calculus


```
((λx.x) v)
```

Lambda calculus

$((\lambda x.x)\ v)$

$((\lambda x.x)\ v)$

Lambda calculus

$((\lambda x.x)\ v)$

$((\lambda x.x)\ v)$

v

Lambda calculus


```
v
```

Lambda calculus

V

V

Multiple applications where the left-side is not a lambda abstraction are solved in a left-to-right fashion:

Lambda calculus

Multiple applications where the left-side is not a lambda abstraction are solved in a left-to-right fashion:

$$\frac{t \rightarrow_\beta t' \quad u \rightarrow_\beta u' \quad (t'u') \rightarrow_\beta v}{(tu) \rightarrow_\beta v}$$

Lambda calculus

Variables cannot be further reduced, that is they stay the same:

HOGESCHOOL ROTTERDAM

Lambda calculus

Variables cannot be further reduced, that is they stay the same:

$$\frac{}{x \rightarrow_\beta x}$$

Lambda calculus

We can encode functions with multiple parameters by nesting lambda abstractions:

```
(λx y.(x y))
```

Lambda calculus

The parameters are then given one at a time:

```
(((λx y.(x y)) A) B)
```

Lambda calculus


```
(((λx y.(x y)) A) B)
```

Lambda calculus


```
(((λx y.(x y)) A) B)
```

```
( ((λx y.(x y)) A) B)
```

Lambda calculus


```
( ((λx y.(x y)) A)  B)
```

Lambda calculus


```
( ((λx y.(x y)) A)  B)
```

```
((λy.( A  y))  B)
```

Lambda calculus


```
((λy.( A  y)) B)
```

Lambda calculus

$((\lambda y.(\boxed{A}\ \ y))\ \ B)$

$((\lambda y.(A\ \ y))\ \ B)$

Lambda calculus

$$((\lambda y.(A\ \ y))\ \ B)$$

Lambda calculus

$((\lambda y.(A\ \ y))\ \ B)$

$\boxed{((\lambda y.(A\ \ y))\ \ B)}$

Lambda calculus

$((\lambda y.(A\ y))\ B)$

Lambda calculus


```
((λy.(A y)) B)
```

```
(A B)
```

Lambda calculus


```
( A  B )
```

Lambda calculus


```
(A  B)
```

```
(A  B)
```

Closing up

Closing up

Example executions of (apparently) nonsensical programs

- We will now exercise with the execution of various lambda programs.
- Try to guess what the result of these programs is, and then we shall see what would have happened.

Closing up

What is the result of this program execution?

$$(((\lambda x\ y.(x\ y))\ (\lambda z.(z\ z)))\ A)$$

Closing up

```
(((λx y.(x y)) (λz.(z z))) A)
```

Closing up

`(((λx y.(x y)) (λz.(z z))) A)`

`( ((λx y.(x y)) (λz.(z z))) A)`

Closing up


```
( ((λx y.(x y)) (λz.(z z))) A )
```

Closing up

$$(\,((\lambda x\ y.(x\ y))\ (\lambda z.(z\ z)))\ \ \mathtt{A})$$

$$((\lambda y.(\,(\lambda z.(z\ z))\ \ y))\ \ \mathtt{A})$$

Closing up

$((\lambda y.(\ (\lambda z.(z\ z))\ \ y))\ \ A)$

Closing up

$$((\lambda y.(\ (\lambda z.(z\ z))\ \ y))\ \ A)$$

$$((\lambda y.((\lambda z.(z\ \ z))\ \ y))\ \ A)$$

Closing up

$$((\lambda y.((\lambda z.(z\ \ z))\ \ y))\ \ A)$$

Closing up

((λy.((λz.(z z)) y)) A)

((λy.((λz.(z z)) y)) A)

Closing up


```
((λy.((λz.(z z)) y)) A)
```

Closing up

$((\lambda y.((\lambda z.(z\ z))\ y))\ A)$

$((\lambda z.(z\ z))\ A)$

Closing up

$((\lambda z.(z\ \ z))\ \ \boxed{A})$

Closing up


```
((λz.(z z)) A )
```

```
((λz.(z z)) A)
```

Closing up

```
((λz.(z z)) A)
```

Closing up


```
((λz.(z z)) A)
```

```
((λz.(z z)) A)
```

Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Closing up

$((\lambda z.(z\ z))\ A)$

Closing up

$((\lambda z.(z\ z))\ A)$

$(A\ \ A)$

Closing up


```
( A   A )
```

Closing up

```
(A  A)
```

```
(A  A)
```

Closing up

What is the result of this program execution? Watch out for the scope of the two "x" variables!

```
(((λx x.(x x)) A) B)
```

Closing up

```
(((λx x.(x x)) A) B)
```

Closing up


```
(((λx x.(x x)) A) B)
```

```
( ((λx x.(x x)) A) B)
```

Closing up


```
( ((λx x.(x x)) A) B)
```

Closing up


```
( ((λx x.(x x)) A) B)
```

```
((λx.(x x)) B)
```

Closing up

```
((λx.(x x)) B)
```

Closing up

$((\lambda x.(x\ \ x))\ \ B)$

$((\lambda x.(x\ \ x))\ \ B)$

Closing up

Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Introduction

Semantics of imperative languages

Lambda calculus

Closing up

```
((λx.(x  x))  B)
```

Closing up

$((\lambda x.(x\ \ x))\ \ B)$

$\boxed{((\lambda x.(x\ \ x))\ \ B)}$

Closing up


```
((λx.(x x)) B)
```

Closing up


```
((λx.(x x)) B)
```

```
(B B)
```

Closing up

(B B)

Closing up

```
(B  B)
```

```
(B  B)
```

Closing up

The first "x" gets replaced with "A", but the second "x" shadows it!

```
(((λx x.(x x)) A) B)
```

Closing up

A better formulation, less ambiguous, would turn:

```
(((λx x.(x x)) A) B)
```

...into:

```
(((λy x.(x x)) A) B)
```

Closing up


```
(((λy x.(x x)) A) B)
```

Closing up


```
(((λy x.(x x)) A) B)
```

```
( ((λy x.(x x)) A) B)
```

Closing up

(((λy x.(x x)) A) B)

Closing up

$(\boxed{((\lambda y\ x.(x\ x))\ A)}\ B)$

$((\lambda x.(x\ x))\ B)$

Closing up


```
((λx.(x  x))  B)
```

Closing up

```
((λx.(x  x))  B)
```


```
((λx.(x  x))  B)
```

Closing up

```
((λx.(x x)) B)
```

Closing up

```
((λx.(x  x))  B)
```

```
((λx.(x  x))  B)
```

Closing up

$((\lambda x.(x\ x))\ B)$

Closing up

$((\lambda x.(x\ \ x))\ \ B)$

$(B\ \ B)$

Closing up

(B B)

Closing up

```
(B  B)
```

```
(B  B)
```

Closing up

What is the result of this program execution? Is there even a result?

```
((λx.(x x)) (λx.(x x)))
```

Closing up

$$\boxed{\texttt{((}\lambda\texttt{x.(x x)) (}\lambda\texttt{x.(x x)))}}$$

Closing up

```
((λx.(x x)) (λx.(x x)))
```

```
((λx.(x x)) (λx.(x x)))
```

Closing up

$((\lambda x.(x\ \ x))\ \ (\lambda x.(x\ \ x)))$

Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Introduction

Semantics of imperative languages

Lambda calculus

Closing up

$$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$

$$(\ (\lambda x.(x\ x))\quad (\lambda x.(x\ x))\ )$$

Closing up

$$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$

Closing up


```
( (λx.(x x)) (λx.(x x)) )
```

```
((λx.(x x)) (λx.(x x)))
```

Closing up

$$((\lambda x.(x\ \ x))\ \ (\lambda x.(x\ \ x)))$$

Closing up


```
((λx.(x  x)) (λx.(x  x)))
```

```
((λx.(x  x)) (λx.(x  x)))
```

Closing up

$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$

Closing up

$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$

$(\ (\lambda x.(x\ x))\quad (\lambda x.(x\ x))\ )$

Closing up

Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Introduction

Semantics of imperative languages

Lambda calculus

Closing up

$$((\lambda x.(x\ \ x))\ \ (\lambda x.(x\ \ x)))$$

Closing up


```
( (λx.(x x)) (λx.(x x)) )
```

```
((λx.(x x)) (λx.(x x)))
```

Closing up

$((\lambda x.(x \;\; x)) \;\; (\lambda x.(x \;\; x)))$

It never ends! Like a `while true: ..`!

Closing up

Ok, I know what you are all thinking: what is this for sick joke? This is no real programming language!

- We have some sort of functions and function calls
- We do not have booleans and if's
- We do not have integers and arithmetic operators
- We do not have a lot of things!

Closing up

Surprise!

With nothing but lambda programs we will show how to build all of these features and more.

Closing up

Stay tuned.

This will be a marvelous voyage.

This is it!

The best of luck, and thanks for the attention!