Introduction to functional programming and lambda calculus

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Course introduction
- Exam and practicum
- Semantics of traditional programming languages
- Basic lambda calculus

Course introduction

Course introduction

Course topics

- We will discuss a completely new paradigm for expressing programs
- This paradigm, functional programming, is based on different premises on computation
- It gives guarantees of correctness in complex places, like parallelism or separation of concerns
- It requires a radical conceptual shift in the way you think about programming

Course introduction

Course topics

- We will begin with a short discussion on traditional programming language **semantics**
- We will then show the **lambda calculus**, which is the foundation for functional languages
-
- We will then bridge the gap between theory and practice
-
  - We will translate the lambda calculus into two mainstream functional languages: F# and Haskell
  - This will cover a huge chunk of possible applications in countless other languages and libraries, from C# LINQ to Java streams, to Scala, Scheme, Closure, etc.

Examination

Examination

Exam structure

- There is a theoretical exam, where you show understanding of the basic principles
- There is a practical exam, where you show understanding of their concrete applications

Examination

Theoretical exam

- One question on a lambda calculus program execution
- One question on the type system of a lambda calculus program, F# program, or Haskell program
- Both questions must be answered correctly to get a **voldoende**

Examination

Build, in groups of max four, any of the following applications in either Haskell or F#:

- A 2D simulation of a supermarket with customers, cash registers, and various aisles
- A 2D simulation of a supply chain with trucks, containers, and ships
- An interpreter for a Python-like language (with a parser for an extra challenge)
- An interpreter for the lambda calculus (with a parser for an extra challenge)

We will get together at the end of the course, and the teacher(s) will ask you to **individually** perform some activities on the code to prove understanding and familiarity.

Semantics of traditional programming languages

Semantics of traditional programming languages

- Traditional, imperative programming languages are based on sharing memory through instructions
- This means that subsequent instructions are not independent from each other
- Any function call makes use of the available memory

Semantics of traditional programming languages

For example, consider the semantic rules that describe the working of “;”

First we run s_1 with the initial memory, then we run s_2 with the modified memory.

Semantics of traditional programming languages

For example, consider the semantic rules that describe the working of ";"

First we run s_1 with the initial memory, then we run s_2 with the modified memory.

$$\frac{\langle s_1, S, H\rangle \rightarrow \langle S_1, H_1\rangle \wedge \ \langle s_2, S_1, H_1\rangle \rightarrow \langle S_2, H_2\rangle}{\langle (s_1; s_2), S, H\rangle \rightarrow \langle S_2, H_2\rangle}$$

What does "*first we run* s_1 *with the initial memory, then we run* s_2 *with the modified memory*" imply?.

What does "*first we run* s_1 *with the initial memory, then we run* s_2 *with the modified memory*" imply?.

- The same instructions, executed at different moments, will produce **different results**.
- Change the order of some method calls, and some weird dependence might cause bugs or break things.

Semantics of traditional programming languages

Goals

- Our goal is to ensure that behaviour of code is consistent.
- Change the order of some method calls, and the results remain the same.
- This makes it easier to test, parallelize, and in general ensure correctness.

How do we achieve this?

Semantics of traditional programming languages

How do we achieve this?

We give (shared) memory up: every piece of code is a function which output only depends on input.

This very important property is called **referential transparency**.

Basic lambda calculus

Basic lambda calculus

Introduction

- The (basic) lambda calculus is an alternative mechanism to Turing Machines and the Von Neumann architecture.
- It is very different, but has equivalent expressive power.
- It is the foundation of all functional programming languages.

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.

Basic lambda calculus

A lambda calculus program (just *program* from now on) is made up of three syntactic elements:

- Variables: x, y, ...
- 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).

Basic lambda calculus

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

$$(\lambda x \rightarrow x)$$

Basic lambda calculus

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

$$((\lambda x \rightarrow x)\ \ v)$$

Basic 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:

Basic 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

Basic lambda calculus

```
((λx→x) v)
```

Basic lambda calculus


```
((λx→x) v)
```

$\underline{\texttt{((}\lambda\texttt{x}\rightarrow\texttt{x) v)}}$

Basic lambda calculus

$\underline{((\lambda \mathtt{x} \rightarrow \mathtt{x})\ \ \mathtt{v})}$

Basic lambda calculus

$\underline{\texttt{((}\lambda\texttt{x}\rightarrow\texttt{x)~~v)}}$

$\texttt{v}$

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

Basic lambda calculus

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

$$\frac{t \rightarrow_\beta t' \wedge \; u \rightarrow_\beta u' \wedge \; t' \; u' \rightarrow_\beta v}{t \; u \rightarrow_\beta v}$$

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

Basic lambda calculus

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

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

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

$$(\lambda x\ \ y \rightarrow (x\ \ y))$$

Basic lambda calculus

The parameters are then given one at a time:

```
(((λx y→(x y)) A) B)
```

Basic lambda calculus


```
(((λx y→(x y)) A) B)
```

Basic lambda calculus

$$(((\lambda x\ y \rightarrow (x\ y))\ A)\ B)$$

$$(\underline{((\lambda x\ y \rightarrow (x\ y))\ A)}\ B)$$

Basic lambda calculus

$(\underline{((\lambda x\ y \rightarrow (x\ y))\ A)}\ B)$

Basic lambda calculus

$(\underline{((\lambda x\ y \rightarrow (x\ y))\ A)}\ B)$

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

Basic lambda calculus


```
((λy→(A  y))  B)
```

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

$$\underline{((\lambda y \rightarrow (A\ y))\ B)}$$

Basic lambda calculus

$$\underline{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{(A y)) B)}}$$

Basic lambda calculus

$\underline{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{(A y)) B)}}$

`(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?

```
(((λx y→(x y)) (λz→(z z))) A)
```

Closing up

$$\texttt{(((}\lambda\texttt{x y}\rightarrow\texttt{(x y)) (}\lambda\texttt{z}\rightarrow\texttt{(z z))) A)}$$

Closing up


```
(((λx y→(x y)) (λz→(z z))) A)
```

$$(\underline{((\lambda x\ y \to (x\ y))\ (\lambda z \to (z\ z)))}\ A)$$

Closing up

$$\texttt{(}\underline{\texttt{((}\lambda\texttt{x y}\rightarrow\texttt{(x y)) (}\lambda\texttt{z}\rightarrow\texttt{(z z)))}}\texttt{ A)}$$

Closing up

$(\underline{((\lambda x\ y \rightarrow (x\ y))\ (\lambda z \rightarrow (z\ z)))}\ A)$

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

Closing up

$$\boxed{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{((}\lambda\texttt{z}\rightarrow\texttt{(z z)) y)) A)}}$$

Closing up

$$\boxed{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{((}\lambda\texttt{z}\rightarrow\texttt{(z z)) y)) A)}}$$

$$\boxed{\underline{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{((}\lambda\texttt{z}\rightarrow\texttt{(z z)) y)) A)}}}$$

Closing up

$\underline{\texttt{((}\lambda\texttt{y}\rightarrow\texttt{((}\lambda\texttt{z}\rightarrow\texttt{(z z)) y)) A)}}$

Closing up

$\underline{((\lambda y \rightarrow ((\lambda z \rightarrow (z\ z))\ y))\ A)}$

$((\lambda z \rightarrow (z\ z))\ A)$

Closing up


```
((λz→(z z)) A)
```

Closing up

```
((λz→(z z)) A)
```

```
((λz→(z z)) A)
```

Closing up

$\underline{((\lambda z \rightarrow (z\ z))\ A)}$

Closing up

$\underline{((\lambda z \rightarrow (z\ z))\ A)}$

$(A\ A)$

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

$\texttt{(((}\lambda\texttt{x x}\rightarrow\texttt{(x x)) A) B)}$

$\texttt{(}\underline{\texttt{((}\lambda\texttt{x x}\rightarrow\texttt{(x x)) A)}}\texttt{ B)}$

Closing up

$$\texttt{(}\underline{\texttt{((}\lambda\texttt{x x}\rightarrow\texttt{(x x)) A)}}\texttt{ B)}$$

Closing up

$$\texttt{(}\underline{\texttt{((}\lambda\texttt{x x}\rightarrow\texttt{(x x)) A)}}\texttt{ B)}$$

$$\texttt{((}\lambda\texttt{x}\rightarrow\texttt{(x x)) B)}$$

Closing up


```
((λx→(x x)) B)
```

Closing up

$$((\lambda x \rightarrow (x\ \ x))\ \ B)$$

$$\underline{((\lambda x \rightarrow (x\ \ x))\ \ B)}$$

Closing up

$$\boxed{\underline{((\lambda x \rightarrow (x\ \ x))\ \ B)}}$$

Closing up

$$\underline{((\lambda x \rightarrow (x\ \ x))\ \ B)}$$

$$(B\ \ B)$$

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

$(\underline{((\lambda y\ x \rightarrow (x\ x))\ A)}\ B)$

Closing up

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

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

Closing up


```
((λx→(x x)) B)
```

Closing up

$$\texttt{((}\lambda\texttt{x}\rightarrow\texttt{(x x)) B)}$$

$$\underline{\texttt{((}\lambda\texttt{x}\rightarrow\texttt{(x x)) B)}}$$

Closing up

$\underline{((\lambda x \rightarrow (x\ \ x))\ \ B)}$

Closing up

$\underline{((\lambda x\rightarrow(x\ \ x))\ \ B)}$

(B B)

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

```
((λx→(x x)) (λx→(x x)))
```

Closing up

$$((\lambda x \rightarrow (x \;\; x)) \;\; (\lambda x \rightarrow (x \;\; x)))$$

Closing up

$$\boxed{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}$$

$$\boxed{\underline{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}}$$

Closing up

$$\boxed{\underline{((\lambda x \rightarrow (x\ x))\ (\lambda x \rightarrow (x\ x)))}}$$

$\underline{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}$

$((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))$

Closing up

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

Closing up

$$\boxed{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}$$

$$\boxed{\underline{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}}$$

Closing up

$$\boxed{\underline{((\lambda x \rightarrow (x\ \ x))\ \ (\lambda x \rightarrow (x\ \ x)))}}$$

Closing up

$\underline{\texttt{((}\lambda\texttt{x}\rightarrow\texttt{(x x)) (}\lambda\texttt{x}\rightarrow\texttt{(x x)))}}$

$\texttt{((}\lambda\texttt{x}\rightarrow\texttt{(x x)) (}\lambda\texttt{x}\rightarrow\texttt{(x x)))}$

Closing up

```
((λx→(x x)) (λx→(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!