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

Introduction

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

Introduction

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

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

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

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

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

What about $s_2 s_1$?

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 $\implies$ leaky abstraction!

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 access 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 $s_2\{x\}$?

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 $s_2\{x\}$?

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 of arbitrary machine state
- No mutating of arbitrary machine state
- Only explicitly-mentioned machine state may be read

Semantics of imperative languages

Idea for better abstraction: remove implicit dependencies

- No implicit dependencies $\implies$ all dependencies explicit
- No reading of 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

Wait a minute, this is just like functions

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

Wait a minute, this is just like functions

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

NB: Imperative “functions” need not be functions!

Non-function “functions” are more properly called procedures

Semantics of imperative languages

Referential transparency:

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

Semantics of imperative languages

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

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):

$a, b, c, \ldots \qquad a_0, a_1, \ldots \qquad b_0, b_1, \ldots$

Abstractions:

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

Applications:

For any lambda terms F and T: (FT)

Lambda calculus

A simple example: the identity function (just returns its input)

```
(λx.x)
```

Lambda calculus

A simple example: call the identity function on a variable

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

Lambda calculus

β-reduction

- Redex: application of an abstraction to an argument
- Result: in abstraction body replace parameter by argument

$$((\lambda x.B)A) \rightarrow_\beta B[x \mapsto A]$$

Lambda calculus

Multiple parameters via nested abstractions:

$(\lambda x.(\lambda y.(xy)))$

The parameters are then given one at a time:

```
(((λx y.(x y)) A) B)
```

Lambda calculus

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

Lambda calculus


```
(((λx y.(x y)) A) B)
```

```
( ((λx y.(x y)) A)  B)
```

Lambda calculus


```
( ((λx y.(x y)) A)  B)
```

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

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

Lambda calculus


```
((λy.(A  y))  B)
```

Lambda calculus

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

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

Lambda calculus

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

Lambda calculus


```
((λy.(A y)) B)
```

```
(A B)
```

Closing up

Closing up

Example executions of (apparently) nonsensical programs

- Manual execution of various lambda programs.
- Try to work out the result of these programs.

Closing up

What is the result of this program?

```
(((λx y.(x y)) (λz.(z z))) A)
```

Closing up


```
(((λx y.(x y)) (λz.(z z))) A)
```

Closing up

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

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

Closing up


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

Closing up

(((λx y.(x y)) (λz.(z z))) A)

((λy.((λz.(z z)) y)) 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

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

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

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

Closing up

```
((λz.(z z)) A)
```

Closing up


```
((λz.(z z)) A)
```

```
((λz.(z z)) A)
```

Closing up


```
((λz.(z z)) A)
```

Closing up

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

$(A\ \ A)$

What is this program's result? Hint: scope!

```
(((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

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

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

Closing up

```
((λx.(x  x))  B)
```

```
((λx.(x  x))  B)
```

```
((λx.(x  x))  B)
```

Closing up

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

Closing up


```
((λx.(x x)) B)
```

```
(B B)
```

Closing up

Outer x is shadowed by inner x!

```
(((λx x.(x x)) A) B)
```

Closing up

To disambiguate, 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

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

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

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

Closing up


```
((λx.(x  x))  B)
```

Closing up


```
((λx.(x  x))  B)
```

```
((λx.(x  x))  B)
```

Closing up


```
((λx.(x x)) B)
```

Closing up

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

$(B\ \ B)$

Closing up

What is this program's result? Is there even one?

```
((λx.(x  x)) (λx.(x  x)))
```

Closing up

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

Closing up

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

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

Closing up

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

Closing up

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

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

Closing up

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

Closing up

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

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

Closing up

```
((λx.(x x)) (λx.(x x)))
```

Closing up

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

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

Closing up

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

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

Closing up

What you are all thinking:

This is no real programming language!

Is this a joke?

- 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

What you are all thinking:

This is no real programming language!

Is this a joke?

- 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!

Surprise!

All these things are in there awaiting discovery!

Closing up

What you are all thinking:

This is no real programming language!

Is this a joke?

- 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!

Surprise!

All these things are in there awaiting discovery!

Stay tuned

This will be a marvelous voyage!

This is it!

The best of luck, and thanks for the attention!