equational programming

2019 10 28

lecture 1

overview

- practical issues
- introductory remarks
- lambda terms
- material

overview

- practical issues
- introductory remarks
- lambda terms
- material

who and when

lectures:

Mondays 11.00-12.45 and Thursday 13.30-15.15

exercise classes:

Tuesdays 09.00-10.45 and Fridays ??

Haskell labs:

Group A: Tuesdays 11.00-12.45 and Fridays 15.30-17.15

Group B: Tuesdays 13.30-15.15 and Fridays 13.30-15.15

Geoffrey Frankhuizen and George Karlos

material via canvas

theory page:

course notes lambda calculus

course notes equational specifications

slides and exercise sheets

practical work page:

Haskell assignments

and: some additional material via links in slides

exam and grade

exam in week 8 of the course

resit of the exam in January

3 sets of Haskell exercises (obligatory)

4 sets of theory exercises (not obligatory)

mimimun 5,5 both for Haskell and for exam

final grade 25% Haskell exercises, 75% written exam

bonus of at most 0.5 on the exam grade for theory exercises

contact

Femke van Raamsdonk, T464

email at f.van.raamsdonk at vu.nl

refer to the course in the subject

no email via canvas

overview

- practical issues
- introductory remarks
- lambda terms
- material

equational programming

foundations of functional programming

functional programming

a functional program is an expression,

and is executed by evaluating the expression

(use definitions from left to right)

focus on what and not so much on how

the functions are pure (or, mathematical)

an input always gives the same output

example functional programming style

in Haskell: applying functions to arguments

```
sum [1 .. 100]
```

in Java: changing stored values

```
total = 0;
for (i = 1; i <= 100; ++i)
  total = total + i;
```

taste of Haskell

definition of sum:

```
sum [] = 0
sum (n:ns) = n + sum ns
```

type of sum:

```
Num a => [a] -> a
```

that is:
for any type a of numbers, sum maps a list of elements of a to a

use of sum: application of the function sum to the argument [1,2,3]

```
sum [1,2,3]
```

evaluation by equational reasoning

definition: `double x = x + x`

evaluation:

```
double 2
= { unfold definition double }
2 + 2
=  { applying + }
4
```

```
double (double 2)
= { unfold definition inner double }
double (2 + 2)
= {unfold definition double }
(2 + 2) + (2 + 2)
= {apply first +}
4 + (2+2)
= {apply last +}
4 + 4
= {apply +}
```

functional programming: properties

high level of abstraction

concise programs

more confidence in correctness

(read, check, prove correct)

higher-order functions

foundations: equational reasoning and λ-calculus

Haskell: properties

lazy evaluation strategy

powerful type system

functional programming: some history

Lisp John McCarthy (1927–2011), Turing Award 1971

FP John Backus (1924–2007), Turing Award 1977

ML Robin Milner (1934-2010), Turing Award 1991, et al

Miranda David Turner (born 1946)

Haskell

Haskell
a group containing ao Philip Wadler and Simon Peyton Jones

functional programming languages

	typed	untyped
strict	ML	Lisp
lazy	Haskell	

See also F# (Microsoft), Erlang (Ericsson), Scala (Java plus ML)

functional programming and lambda calculus

Based on the lambda calculus, Lisp rapidly became ...
(from: wikipedia page John McCarthy)

Haskell is based on the lambda calculus, hence the lambda we use as a logo.
(from: the Haskell website)

Historically, ML stands for metalanguage: it was conceived to develop proof tactics in the LCF theorem prover (whose language, pplambda, a combination of the first-order predicate calculus and the simply typed polymorphic lambda calculus, had ML as its metalanguage).
(from: wikipedia page of ML)

course equational programming (EP)

lambda calculus

equational specifications

exercises functional programming: Haskell

overview

- practical issues
- introductory remarks
- lambda terms
- material

lambda calculus

inventor: Alonzo Church (1936)

a language expressing functions or algorithms

concept of computability and basis of functional programming

a language expressing proofs

untyped and typed

historical note: notation for functions

Frege defined the graph of a function (1893)

Russell and Whitehead and Russell (1910)

Schönfinkel defined function calculus (1920)

Curry defined combinary logic (1920)

notation for (anonymous) functions

mathematical notation:

$$f : \mathsf{nat} \to \mathsf{nat}$$
$$f(x) = \mathsf{square}(x)$$

or also:

$$f : \mathsf{nat} \to \mathsf{nat}$$
$$f : x \mapsto \mathsf{square}(x)$$

lambda notation:

$$\lambda x.\, \mathsf{square}\, x$$

we start with the untyped λ-calculus

lambda terms: intuition

abstraction:

$\lambda x.\, M$ is the function mapping x to M

$\lambda x.\, x$ is the function mapping x to x

$\lambda x.\, \mathsf{square}\, x$ is the function mapping x to $\mathsf{square}\, x$

application:

$F\ M$ is the application of the function F to its argument M

(not the result of applying)

lambda terms: inductive definition

we assume a countably infinite set of variables $(x, y, z \ldots)$

sometimes we in addition assume a set of contstants

the set of λ-terms is defined inductively by the following clauses:

a variable x is a λ-term

a constant c is a λ-term

if M is a λ-term, then $\lambda x.\, M$ is a λ-term, called an abstraction

if F and M are λ-terms, then $F\, M$ is a λ-term, called an application

famous terms

$I = \lambda x.\, x$

$K = \lambda x.\, \lambda y.\, x$

$\mathsf{S} = \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z)$

$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$

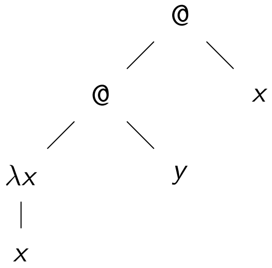
omit outermost parentheses

application is associative to the left

abstraction is associative to the right

lambda extends to the right as far as possible

terms as trees: example



terms as trees: general

a subterm corresponds to a subtree

subterms of $\lambda x.\, y$ are $\lambda x.\, y$ and y

parentheses

application is associative to the left
$(M\,N\,P)$ instead of $((M\,N)\,P)$

outermost parentheses are omitted
$M\,N\,P$ instead of $(M\,N\,P)$

lambda extends to the right as far as possible
$\lambda x.\,M\,N$ instead of $\lambda x.\,(M\,N)$

sometimes we combine lambdas
$\lambda x_1 \ldots x_n.\,M$ instead of $\lambda x_1 . \ldots . \lambda x_n.\,M$

more notation

$(\lambda x.\, \lambda y.\, M)$ instead of $(\lambda x.\, (\lambda y.\, M))$

$(M\, \lambda x.\, N)$ instead of $(M\, (\lambda x.\, N))$

$\lambda xy.\, M$ instead of $\lambda x.\, \lambda y.\, M$

inductive definition of terms

definitions recursively on the definition of terms

example: definition of the free variables of a term

proofs by induction on the definition of terms

example: every term has finitely many free variables

bound variables: definition

x is bound by the first λx above it in the term tree

examples: the underlined x is bound in

$\lambda x.\, \underline{x}$

$\lambda x.\, \underline{x}\, \underline{x}$

$(\lambda x.\, \underline{x})\, x$

$\lambda x.\, y\, \underline{x}$

$\lambda x.\, \lambda x.\, \underline{x}$

free variables: definition

a variable that is not bound is free

alternatively: define recursively the set $\mathsf{FV}(M)$ of free variables of M:

$$
\begin{aligned}
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(c) &= \emptyset \\
\mathsf{FV}(\lambda x.\, M) &= \mathsf{FV}(M) \backslash \{x\} \\
\mathsf{FV}(F\, P) &= \mathsf{FV}(F) \cup \mathsf{FV}(P)
\end{aligned}
$$

a term is closed if it has no free variables

currying

reduce a function with several arguments to functions with single arguments

example:

$f : x \mapsto x + x$ becomes $\lambda x.\, x + x$

$g : (x, y) \mapsto x + y$ becomes $\lambda x.\, \lambda y.\, x + y$, not $\lambda(x, y).\, \mathrm{plus}\, x\, y$

$(\lambda x.\, \lambda y.\, x + y)\, 3$ is an example of partial application

history:

due to Frege, Schönfinkel, and Curry

related to the isomorphism between $A \times B \to C$ and $A \to (B \to C)$

towards computation

we will use terms to compute, as for example in

$$(\lambda x.\, f\, x)\, 5 \rightarrow_{\beta} (f\, x)[x := 5] = f\, 5$$

the definition of substitution requires more preparation

intuitive meaning of $M[x := N]$:

the result of replacing in M all free occurrences of x by N

bound variables: definition

x is bound by the first λx above it in the term tree

examples: the underlined x is bound in

$\lambda x.\, \underline{x}$

$\lambda x.\, \underline{x}\, \underline{x}$

$(\lambda x.\, \underline{x})\, x$

$\lambda x.\, y\, \underline{x}$

free variabeles: definition

a variable that is not bound is free

alternatively: define recursively the set $\mathsf{FV}(M)$ of free variables of M:

$$\begin{array}{rcl}
\mathsf{FV}(x) & = & \{x\} \\
\mathsf{FV}(c) & = & \emptyset \\
\mathsf{FV}(\lambda x.\, M) & = & \mathsf{FV}(M) \backslash \{x\} \\
\mathsf{FV}(F\, P) & = & \mathsf{FV}(F) \cup \mathsf{FV}(P)
\end{array}$$

a term is closed if it has no free variables

substititition: recursive definition

substitution in a variable or a constant:

$x[x := N] = N$

$a[x := N] = a$ with $a \neq x$ a variable or a constant

substitution in an application:

$(P\,Q)[x := N] = (P[x := N])\,(Q[x := N])$

substitution in an abstraction:

$(\lambda x.\, P)[x := N] = \lambda x.\, P$

$(\lambda y.\, P)[x := N] = \lambda y.\, (P[x := N])$ if $x \neq y$ and $y \notin \mathrm{FV}(N)$

$(\lambda y.\, P)[x := N] = \lambda z.\, (P[y := z][x := N])$
if $x \neq y$ and $z \notin \mathrm{FV}(N) \cup \mathrm{FV}(P)$ and $y \in \mathrm{FV}(N))$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] =$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] =$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] =$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] = \lambda z.\, x$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] = \lambda z.\, x$

$(\lambda y.\, x\,(\lambda w.\, v\,w\,x))[x := u\,v] =$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] = \lambda z.\, x$

$(\lambda y.\, x\,(\lambda w.\, v\, w\, x))[x := u\, v] = \lambda y.\, u\, v(\lambda w.\, v\, w\,(u\, v))$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] = \lambda z.\, x$

$(\lambda y.\, x\,(\lambda w.\, v\, w\, x))[x := u\, v] = \lambda y.\, u\, v(\lambda w.\, v\, w\,(u\, v))$

$(\lambda y.\, x\,(\lambda x.\, x))[x := \lambda y.\, x\, y] =$

substitution: examples

$(\lambda x.\, x)[x := \mathsf{c}] = \lambda x.\, x$

$(\lambda x.\, y)[y := \mathsf{c}] = \lambda x.\, \mathsf{c}$

$(\lambda x.\, y)[y := x] = \lambda z.\, x$

$(\lambda y.\, x\,(\lambda w.\, v\, w\, x))[x := u\, v] = \lambda y.\, u\, v(\lambda w.\, v\, w\,(u\, v))$

$(\lambda y.\, x\,(\lambda x.\, x))[x := \lambda y.\, x\, y] = \ \lambda y.(\lambda y.\, x\, y)\,(\lambda x.\, x)$

alpha conversion

intuition:

bound variables may be renamed

example:

$\lambda x.\, x =_\alpha \lambda y.\, y$

definition α-conversion axiom:

$\lambda x.\, M =_\alpha \lambda y.\, M[x := y]$ with $y \notin FV(M)$

definition α-equivalence relation $=_\alpha$: on terms

$P =_\alpha Q$ if Q can be obtained from P

by finitely many 'uses' of the α-conversion axiom

that is: by finitely many renamings of bound variables in context

alpha equivalence classes

we identify α-equivalent λ-terms

just as we identify $f : x \mapsto x^2$ and $f : y \mapsto y^2$

and $\forall x.\, P(x)$ is $\forall y.\, P(y)$

we work with equivalence classes modulo α

alpha-conversion and substitution: intuitive approach

we defined first substitution $[x := P]$ and then α using substitution $[x := y]$

an alternative intuitive approach:

define α as renaming of bound variables

work modulo α

define substitution $M[x := N]$ using renaming of bound variables:

replace all free occurrences of x in M by N,

rename bound variables if necessary

example: $(\lambda x.y)[y := x] =_\alpha (\lambda x'.y)[y := x] = \lambda x'.\,x$

now we know the statics of the lambda-calculus

we consider λ-terms modulo α-conversion

application and abstraction

bound and free variables

currying

substitution

we continue with the dynamics: β-reduction

material

course notes chapter *Terms and Reduction*

Haskell pages