equational programming
2018 10 29
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
Femke van Raamsdonk
f.van.raamsdonk at vu.nl, T446

exercise classes:
Tuesdays 09.00-10.45 and Fridays 11.00-12.45
Eui Yeon (Alex) Jang

Haskell lab:
Tuesdays 13.30-17.00 and Fridays 13.30-17.00
Ionut Boicu
material via canvas

course notes lambda calculus

course notes equational specifications

slides and exercise sheets

Haskell assignments

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)

minimum 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
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:

\[
\text{sum} \ [\] = 0 \\
\text{sum} \ (n:ns) = n + \text{sum} \ ns
\]

type of sum:

\[\text{Num} \ a \Rightarrow [a] \rightarrow a\]

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 \([1,2,3]\)

\[
\text{sum} \ [1,2,3]
\]
evaluation by equational reasoning

definition: double x = x + x
evaluation:

double 2
= \{ \text{unfold definition double} \}
2 + 2
= \{ \text{applying +} \}
4

double (double 2)
= \{ \text{unfold definition inner double} \}
double (2 + 2)
= \{ \text{unfold definition double} \}
(2 + 2) + (2 + 2)
= \{ \text{apply first +} \}
4 + (2+2)
= \{ \text{apply last +} \}
4 + 4
= \{ \text{apply +} \}
8
functional programming: properties

high level of abstraction
concise programs
more confidence in correctness
(read, check, prove correct)
higher-order functions
equational reasoning
Haskell: properties

lazy evaluation strategy

powerful type system
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

a group containing ao Philip Wadler and Simon Peyton Jones
### Functional Programming Languages

<table>
<thead>
<tr>
<th></th>
<th>Typed</th>
<th>Untyped</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>ML</td>
<td>Lisp</td>
</tr>
<tr>
<td>lazy</td>
<td>Haskell</td>
<td></td>
</tr>
</tbody>
</table>

See also **F#** (Microsoft), **Erlang** (Ericsson), **Scala** (Java plus ML)
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)
Corrado Böhm (1923–2017)

PhD thesis (1954, ETH Zürich): meta-circular compiler
lambda calculus
combinatory logic
semantics of functional programming languages
course equational programming (EP)

lambda calculus

algebraic specifications

exercises functional programming: Haskell
overview

- practical issues
- introductory remarks
- lambda terms
- material
lambda calculus

inventor: Alonzo Church (1936)

foundations of mathematics

foundations of concept ‘computability’

restriction to functions

basis of functional programming
notation for (anonymous) functions

mathematical notation:
\[ f : \text{nat} \to \text{nat} \]
\[ f(x) = \text{square}(x) \]

or also:
\[ f : \text{nat} \to \text{nat} \]
\[ f : x \mapsto \text{square}(x) \]

lambda notation:
\[ \lambda x. \text{square} \, x \]

we start with the untyped $\lambda$-calculus
lambda terms: intuition

abstraction:

\( \lambda x. M \) is the function mapping \( x \) to \( M \)

\( \lambda x. \text{square} \ x \) is the function mapping \( x \) to \( \text{square} \ x \)

application:

\( F \ M \) is the application

(not the result of applying)

of the function \( F \) to its argument \( M \)
lambda terms: inductive definition

we assume a countably infinite set of variables \((x, y, z \ldots)\)
sometimes we in addition assume a set of constants

the set of \(\lambda\)-terms is defined inductively by the following clauses:

1. a variable \(x\) is a \(\lambda\)-term
2. a constant \(c\) is a \(\lambda\)-term
3. if \(M\) is a \(\lambda\)-term, then \(\lambda x. M\) is a \(\lambda\)-term, called an abstraction
4. if \(F\) and \(M\) are \(\lambda\)-terms, then \(FM\) is a \(\lambda\)-term, called an application
famous terms

\[ I = \lambda x. x \]

\[ K = \lambda x. \lambda y. x \]

\[ S = \lambda x. \lambda y. \lambda z. x z (y z) \]

\[ \Omega = (\lambda x. x x) (\lambda x. x x) \]
terms as trees: example

```
( (λx y x) x )
```
terms as trees: general

\[ \lambda x \]

\[ \begin{array}{c}
  x \\
  M \\
\end{array} \]

\[ \begin{array}{c}
  \emptyset \\
  F \\
  M \\
\end{array} \]
subterm

part of a term that corresponds to a subtree of the syntax tree
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
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 \( \lambda (x, y). \text{plus} \ x \ y \)

\((\lambda x. \lambda y. x + y) \ 3\) is an example of partial application
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
bound variables: definition

\( x \) is bound by the first \( \lambda x \) above it in the term tree

examples: the underlined \( x \) is bound in

\( \lambda x. x \)

\( \lambda x. xx \)

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

\( \lambda x. y x \)
free variables: definition

a variable that is not bound is free

alternatively: define recursively the set \( FV(M) \) of free variables of \( M \):

\[
FV(x) = \{x\} \\
FV(c) = \emptyset \\
FV(\lambda x. M) = FV(M) \setminus \{x\} \\
FV(F P) = FV(F) \cup FV(P)
\]

a term is **closed** if it has no free variables
substitution: intuition

\[ M[x := N] \] means:

the result of replacing in \( M \) all free occurrences of \( x \) by \( N \)
substitution: recursive definition

substitution in a variable or a constant:

\[ x[x := N] = N \]

\[ a[x := N] = a \text{ with } a \neq x \text{ a variable or a constant} \]

substitution in an application:

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

substitution in an abstraction:

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

\[(\lambda y. \ P)[x := N] = \lambda y. (P[x := N]) \text{ if } x \neq y \text{ and } y \not\in \text{FV}(N) \]

\[(\lambda y. \ P)[x := N] = \lambda z. (P[y := z][x := N]) \text{ if } x \neq y \text{ and } z \not\in \text{FV}(N) \cup \text{FV}(P) \text{ and } y \in \text{FV}(N)) \]
substitution: examples

\[(\lambda x. x)[x := c] = \lambda x. x\]

\[(\lambda x. y)[y := c] = \lambda x. c\]

\[(\lambda x. y)[y := x] = \lambda z. x\]
alpha conversion

alpha conversion:
bound variables may be renamed

example:
\[ \lambda x. x =_{\alpha} \lambda y. y \]

compare with:
\[ f : x \mapsto x^2 \text{ is } f : y \mapsto y^2 \]
\[ \forall x. P(x) \text{ is } \forall y. P(y) \]

identification of alpha-equivalent terms
we work with equivalence classes modulo \( \alpha \)
now we know the statics of the lambda-calculus

we consider $\lambda$-terms modulo $\alpha$-conversion

application and abstraction

bound and free variables

currying

substitution

we continue with the dynamics: $\beta$-reduction
course notes chapter *Terms and Reduction*

Haskell pages