Lambda calculus

Johan Montelius

KTH

VT21

a small language

- a domain: $\mathbb{Z}$ i.e. $... - 2, -1, 0, 1, 2...$
- a set of primitive functions: $+$, $-$, $*$, mod, div
- syntax: symbols, precedence, parentheses i.e. a way to write expressions

evaluation of expressions

- $(3 + 5) * (6 - 3)$
- $8 * (6 - 3)$
- $8 * 3$
- 24

- $(3 + 5) * (6 - 3)$
- $(3 + 5) * 3$
- $8 * 3$
- 24

- $(3 + 5) * (6 - 3)$
- $(3 + 5) * 3$
- $(9 + 15)$
- 24

how about this

$$5 * (4 + 2)$$

$$17 \bmod 5$$

$$7 \bmod 0$$

bottoms

$5 \bmod 0 \equiv \perp$

$\perp$ is called *bottoms*, *undefined* or ... *exception*

We extend the domain: $\mathbb{Z} \cup \{\perp\}$

How should we interpret: $5 * \perp$

strict functions

A function that is defined to be $\perp$ if any of its arguments is $\perp$, is called a *strict function*,

All of our regular arithmetic functions are strict.

ok, I get it

What is the value of: $(x - x) * 5$

evaluation of expressions

- $(\sqrt[3]{3 + 5^4}) * (6 - 6)$
- $(\sqrt[3]{3 + 5^4}) * 0$
- 0

- $(512 \operatorname{div} 0) * (6 - 6)$
- $(512 \operatorname{div} 0) * 0$
- 0
- hmmm, not so good

order of evaluation

If all functions are strict:

- then all arguments of the function must be evaluated
- the order does not matter,... or does it?

if-then-else

Assume we have a function $if(test, then, else)$ with the obvious definition.

Do we want this function to be a *strict function*?

variables and functions

Too make life more interesting, we introduce

variables: x, y,

and functions: $\lambda x \to x + 5$

Most often written $\lambda x.x + 5$ *but we will use* $\to$.

So far, functions do not have names.

functions

- $\lambda x \to x + 5$
- $(\lambda x \to x + 5)\ 7$
- $(7 + 5)$
- 12

application

We *apply a function* to an argument (or *actual arguments*),

- $(\lambda x \to \langle E \rangle)7$

by *substituting* the parameter (or *formal argument*) of the function with the argument.

- $[x/7]\langle E \rangle$

examples

- $[x/7]\langle x + 5 \rangle \qquad 7 + 5$
- $[x/7]\langle \lambda y \to y + x \rangle \qquad \lambda y \to y + 7$
- $[x/(\lambda z \to z + 2)]\langle \lambda y \to (xy) * 2 \rangle \qquad \lambda y \to ((\lambda z \to z + 2)y) * 2$

But, things could go wrong.

scope of declaration

In an expression $\lambda x \to \langle E \rangle$, the *scope* of x is $\langle E \rangle$.

We say that x is *free* in $\langle E \rangle$ but *bound* in $\lambda x \to \langle E \rangle$.

We can write $\lambda x \to (\lambda x \to (x * x))$*, which does complicate things.*

substitution

A substitution $[x/\langle F \rangle]\langle E \rangle$ is possible if $\langle F \rangle$ does not have any free variables ...

... that become bound in $[x/\langle F \rangle]\langle E \rangle$.

$(\lambda x \to (\lambda y \to (y + x)))(y + 5)$	$(\lambda x \to (\lambda z \to (z + x)))(y + 5)$
$[x/(y + 5)](\lambda y \to (y + x))$	$[x/(y + 5)](\lambda z \to (z + x))$
$\lambda y \to (y + (y + 5))$	$\lambda z \to (z + (y + 5))$

We have to be careful but renaming variables solves the problem.

functions

A function is:

... a many to one mapping from one domain to another: $A \mapsto B$

... a description of the expression that should be evaluated: $\lambda x \rightarrow x + 2$

In mathematics we can work with functions even if we do not know how to compute them.

λ calculus

- The λ calculus was introduced in the 1930s by Alonzo Church.
- Easy to define:
  - only three types of expressions: variable, lambda abstraction, application
  - only one rule: evaluation of application
  - you don't even need data structures nor named functions
- Anything that is *computable* can be expressed in λ calculus, it is as powerful as a *Turing machine*.
- We will use some extensions to the language when we describe functional programming.

currying

A function of two arguments, can be described as function of one argument that evaluates to another function of a second argument.

- $(\lambda x \rightarrow (\lambda y \rightarrow x + y))\ 7\ 8$
- $(\lambda y \rightarrow 7 + y)\ 8$
- $7 + 8$

We can write:

- $\lambda xy \rightarrow x + y$

let expressions

- $\lambda x \rightarrow (x + 2) + (x + 2)$ do we have to evaluate $(x + 2)$ twice?
- $\lambda x \rightarrow ((\lambda y \rightarrow y + y)(x + 2))$ $(x + 2)$ only evaluated once
- $\lambda x \rightarrow \text{let } y = x + 2 \text{ in } y + y$ more readable

no recursive definitions

- $\lambda x \to \text{let } y = x + y \text{ in } y + y$ What does this mean?
- $\lambda x \to ((\lambda y \to y + y)(x + y))$

this is ok

- $\lambda x \to \text{let } y = x + 2, z = y + 5 \text{ in } z + z$
- $\lambda x \to ((\lambda y \to (\lambda z \to z + z)(y + 5))(x + 2))$

So is this,

- $\lambda x \to \text{let } y = x + 2, y = y + 5 \text{ in } y + y$
- $\lambda x \to ((\lambda y \to (\lambda y \to y + y)(y + 5))(x + 2))$

functional programming languages

- λ-calculus
  - not the best syntax - not important
  - no "data structures" - functions are all you need
  - no need for named named functions
  - no defined evaluation order
- functional programming languages:
  - different syntax, some good some strange
  - almost always provide built-in or user defined data structures
  - named function i.e. the program
  - defines the evaluation order

All functional programming languages have a core that can be expressed in λ-calculus.

Elixir

- uses the Erlang virtual machine
- a Ruby like syntax
- a small set of built-in data structures, no user defined
- an "eager evaluation" order i.e. arguments are evaluated before the function is applied

Elixir/Erlang is extended to be able to model concurrency. In the first part of this course we will only use the functional subset.

lambda expression

$\lambda x \to 2 + x$ `fn x -> 2 + x end`

$(\lambda y \to 2 + y)4$ `(fn y -> 2 + y end).(4)`

$\lambda x \to \text{let } y = x + 2, y = y + 5 \text{ in } y + y$

```
fn x -> y = x + 2; y = y + 5; y + y end
```


let expression

$\text{let } x = 2, y = x + 3 \text{ in } y + y$

```
x = 2; y = x + 3; y + y
```


difference Erlang/Elixir

```
x = 2; x = 3; x + x
```

$\text{let } x = 2, x = 3 \text{ in } x + x$

$(\lambda x \to (\lambda x \to x + x)3)2$

$(\lambda z \to z + z)3$

$3 + 3$

Erlang: not allowed, interpreted as `2 = 3, ...`

function definition

$inc \equiv \lambda x \to x + 1$

```
def inc(x) do x + 1 end
```

multiple arguments

$add \equiv \lambda xy \to x + y$

```
def add(x, y) do x + y end
```