Introduction

Johan Montelius

KTH

VT19

Functional programming - what?

Functional expressions

(3 + 8) ∗ (6 □ 3)

(\sqrt[3]{3 + 5^4})

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

- \((3 + 5) \times (6 - 3)\)
- \(8 \times (6 - 3)\)
- \(8 \times 3\)
- 24

- \((3 + 5) \times (6 - 3)\)
- \((3 + 5) \times 3\)
- \(8 \times 3\)
- 24

- \((3 + 5) \times (6 - 3)\)
- \((3 + 5) \times 3\)
- \((9 + 15)\)
- 24

- \(5 \times (4 + 2)\)

- \(17 \mod 5\)

- \(7 \mod 0\)

how about this

- \(5 \times (4 + 2)\)
- \(17 \mod 5\)
- \(7 \mod 0\)

bottoms

- \(5 \mod 0 \equiv \bot\)

\(\bot\) is called bottoms, undefined or ... exception

We extend the domain: \(\mathbb{Z} \cup \{\bot\}\)

How should we interpret: \(5 \times \bot\)

strict functions

A function that is defined to be \(\bot\) if any of its arguments is \(\bot\), is called a strict function, All of our regular arithmetic functions are strict.
What is the value of: \((x - x) \times 5\)

- \((\sqrt{3 + 5^4}) \times (6 - 6)\)
- \((\sqrt{3 + 5^4}) \times 0\)
- 0
- 0
- hmmm, not so good

If all functions are strict:
- then all arguments of the function must be evaluated
- the order does not matter,... or does it?

Assume we have a function \(\text{if}(test, then, else)\) 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: \(\lambda x \rightarrow x + 5\)

Most often written \(\lambda x. x + 5\) but we will use \(\rightarrow\).

So far, functions do not have names.

functions

\[\lambda x \rightarrow x + 5\]
\[(\lambda x \rightarrow x + 5) \ 7\]
\[(7 + 5)\]
\[12\]

application

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

\[(\lambda x \rightarrow \langle E \rangle) \ 7\]

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

\[\ [x/7]\langle x + 5 \rangle \]
\[\ [x/7]\langle \lambda y \rightarrow y + x \rangle \]
\[\ [x/7]\langle \lambda y \rightarrow (xy) * 2 \rangle \]

But, things could go wrong.
scope of declaration

In an expression \( \lambda x \to (E) \), the scope of \( x \) is \( (E) \).

We say that \( x \) is free in \( (E) \) but bound in \( \lambda x \to (E) \).

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

substitution

A substitution \([x/(F)](E)\) is possible if \( (F) \) does not have any free variables ...

... that become bound in \([x/(F)](E)\).

\[
\begin{align*}
(\lambda x \to (\lambda y \to (y + x))(y + 5)) & \quad (\lambda x \to (\lambda z \to (z + x))(y + 5)) \\
[x/(y + 5)](\lambda y \to (y + x)) & \quad [x/(y + 5)](\lambda z \to (z + x)) \\
\lambda y \to (y + (y + 5)) & \quad \lambda z \to (z + (y + 5))
\end{align*}
\]

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: \( A \to B \) ... a description of the expression that should be evaluated: \( \lambda x \to x + 2 \)

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

\( \lambda \) calculus

- The \( \lambda \) 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 \( \lambda \) calculus, it is as powerful as a Turing machine.
- We will use some extensions to the language when we describe functional programming.
A function of two arguments, can be described as function of one argument that evaluates to another function of a second argument.

- \((\lambda x \to (\lambda y \to x + y))\) 7 8
- \((\lambda y \to 7 + y)\) 8
- 7 + 8

We can write:
- \(\lambda xy \to x + y\)

\[\text{let expressions}\]

\[
\lambda x \to \lambda y \to \lambda z \to z + z
\]

\[
\lambda x \to let y = x + 2, z = y + 5 \text{ in } y + y
\]

What does this mean?

So is this,

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

\[
\lambda x \to ((\lambda y \to (\lambda y \to y + y)(y + 5))(x + 2))
\]
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

\( \lambda x \to 2 + x \) \quad \text{fn } x \to 2 + x \text{ end}

\((\lambda y \to 2 + y)(4)\) \quad ((\text{fn } y \to 2 + y \text{ end}).(4))

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

\( \text{fn } x \to y = x + 2; y = y + 5; y + y \text{ end} \)

let expression

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

\( x = 2; y = x + 3; y + y \)
difference Erlang/Elixir

\[
x = 2; \ x = 3; \ x + x
\]

let \(x = 2, x = 3\) 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 \(2 = 3, \ldots\)

function definition

\[
inc \equiv \lambda x \to x + 1
\]

\[
\text{def inc(x) do x + 1 end}
\]

multiple arguments

\[
add \equiv \lambda xy \to x + y
\]

\[
\text{def add(x, y) do x + y end}
\]

literals

- atoms: \(\text{:a, :b, :foo, :bar}\)
- integer: \(1,23, 3456789012345678901233456\)
- bool: \(\text{true, false}\)
- more: there are more but this is enough for now
- tuples: {}, {:foo, :bar}, {:a, 42, true}
- lists: [], [1|[]], [1,:a,3,4] (more on this later)

Next time.