Lecture 2: Functional Programming

Theoretical foundations. The λ-calculus

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

General structure of programming languages

According to Peter Landin (1966):

Programming language = core language + syntactic sugar.

Core language: small programming language that implements basic functionality

Syntactic sugar: other programming constructs (a.k.a. *derived forms*) which are abbreviations of combinations of core language constructs.

- they simplify the writing of programs.

Programs are processed in two steps:

1. First, a **preprocessor** (or macro translator) translates all syntactic sugar into core language $\Rightarrow$ program written in core language
2. Next, a **compiler** or **interpreter** runs the program produced by the preprocessor.

What is the λ-calculus?

The core language of most programming languages, and almost al FP languages, is the λ-calculus

- Invented by Alonzo Church in 1928, long before the advent of electronic digital programmable computers
- The smallest programming language of the world. It consists of
  - A language that can be used to write meaningful expressions
  - A set of transformation rules that indicate how to perform evaluation by manipulating them.

There are only 3 kinds of expressions also known as **terms**. A variable x by itself is a term; the **abstraction** of a variable x from a term t, written $\lambda x.t$, is a term; and the **application** of a term t_1 to another term t_1, written $t_1\ t_2$ is a term.

$$t ::= x \mid \lambda x.t \mid t_1\ t_2$$

where x is a variable. Variables are assumed to be elements of a countably infinite set V.

- Intended reading of $\lambda x.t$: "the function which, for input x returns the value of t."

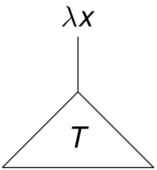
This grammar defines the **syntax** of the language: how to write expressions as strings of characters.

- A **parser** translates terms from syntax into trees of a special kind, called *abstract syntax trees* or *AST*

The λ-calculus

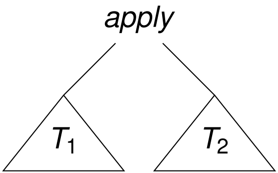
Abstract syntax trees

1. The AST of a variable x is a single node with label x.
2. The AST of an abstraction $\lambda x.t$ is



where T is the AST of t.
λx is called the **binder** of this abstraction.

3. The AST of an application $t_1\ t_2$ is



where T_1 is the AST of t_1 and
T_2 is the AST of t_2.

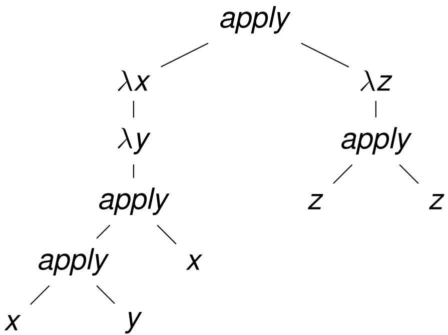
- $t_1\ t_2\ t_3$ is parsed as $(t_1\ t_2)\ t_3$. This means that application is left associative.
- The bodies of abstractions are taken to extend as far to the right as possible, so that, for example,

  $\lambda x.\lambda y.x\ y\ x$ is parsed as $\lambda x.(\lambda y.((x\ y)\ x))$, and
  $x\ \lambda x.y\ x\ x$ is parsed as $x\ (\lambda x.(y\ x)\ x)$.
- Inner abstractions bind more tightly than outer abstractions, so that, for example, $\lambda x.\lambda y.y\ y\ x$ is parsed as $\lambda x.(\lambda y.(y\ y)\ x)$.

Every binder in an AST has a **depth**, which is the number of binders above that binder in the AST.

Parsing
Example

$(\lambda x.\lambda y.x\ y\ x)\ \lambda z.z\ z$ is parsed as $(\lambda x.\lambda y.((x\ y)\ x))\ (\lambda z.(z\ z))$.
Its AST is



The depths of binders λx, λy, λz in this AST are 0,1 and 0, respectively.

Scope. Free variables

- An occurrence o of a variable x is free in an expression t if either
  1. $t = x$, or
  2. $t = \lambda y.t$ with $y \neq x$, and o is a free occurrence or x in t, or
  3. $t = t_1\ t_2$ and o is a free occurrence or x in either t_1 or t_2.

  The set of variables with free occurrences in a term t is denoted by $FVar(t)$, and can be computed as follows:

$$FVar(t) := \begin{cases} \{x\} & \text{if } t = x \in V, \\ FVar(t_1) \setminus \{x\} & \text{if } t = \lambda x.t_1, \\ FVar(t_1) \cup FVar(t_2) & \text{if } t = t_1\ t_2. \end{cases}$$

- The scope of a binder λx of $\lambda x.t$ is t. The occurrences **bound to** λx are the free occurrences of x in t.

Example

If $t = (\lambda z.\lambda s.s\ s\ z)\ (s\ x)$ then $FVar(t) = \{s, x\}$ and the free occurrences of these variables in t are those colored with red.

1. Reference-based representation: makes explicit the relationship between bound variable occurrences and their binders by drawing an arrow from every binder to the variable occurrences bound to it. For example:

$$(\lambda z.\lambda s.s\ s\ z)\ (\underline{s}\ \underline{x}) \quad \text{and} \quad (\lambda n.\lambda t.t\ t\ n)\ (\underline{s}\ \underline{x}).$$

   The underlined occurrences are free variable occurrences.

2. Nameless representation: all binders λx are replaced by λ, and all variable occurrences bound to a binder of depth i in the AST are replaced by i.
   For example, the previous two terms have the same nameless representation: $(\lambda.\lambda.1\ 1\ 0)\ (s\ x)$.

Two terms $t_1, t_2$are α-congruent, and we write $t_1 =_\alpha t_2$, if they have the same nameless representation.

Intuition: the names of variables bound in abstractions are irrelevant.

Capture-avoiding substitution

$[t_1/x]t$ is the operation of replacing all free occurrences of x in t with t_1. This operation is allowed only if there is no free occurrence of x in t which is inside a subterm $\lambda y.t'$ where y occurs in t_1.

- If this happens, then the free variable y of t_1 gets captured.

Example

- $[x/y](\lambda x.y)$ is not capture-free (and, therefore, disallowed); it would produce $\lambda x.x$
- $[z/y](\lambda x.y)$ is capture-free and produces $\lambda x.z$

Safe operations on terms

α-conversion: $\lambda x.t \to_\alpha \lambda y.t'$

if $t' = [y/x]t$ is a capture-avoiding substitution.
Intuition: we are allowed to rename bound variables, is renaming is a capture-avoiding substitution.

β-reduction: $(\lambda x.t_1)\ t_2 \to_\beta [t_2/x]t_1'$

where $\lambda x.t_1' =_\alpha \lambda x.t_1$ such that $[t_2/x]t_1'$ a capture-avoiding substitution.
Intuition: This operation describes how to perform a **function call:** we replace all occurrences of parameter x with the input argument t_2 in the body t_1 of the abstraction.

The intended reading of terms

- $\lambda x.t$: the function which, for input argument x returns the value of t. In particular, $\lambda x.x$ would represent the identity function, and $\lambda x.y$ would represent a constant function which, for any input returns the value if y.
- $t_1\ t_2$: the application of the function represented by t_1 to t_2.

- Function calls are represented by expressions $(\lambda x.t_1)\ t_2$, called β-redexes.

The lambda calculus has no built-in constants or primitive operators. It has no numbers, arithmetic operations, conditionals, records, loops, sequencing, I/O, etc. The only way to compute is by applying functions represented by abstractions to terms – which can also be functions.

Evaluation of expressions is performed by rewriting them with the rule of β-reduction.

- We write $t \Rightarrow_\beta t'$ if we can rewrite t to t' by reducing one β-redex of t.
- Rewriting is defined by four rule of inference:

$$\frac{t \rightarrow_\beta t'}{t \Rightarrow_\beta t'} \qquad \frac{t \Rightarrow_\beta t'}{\lambda x.t \Rightarrow_\beta \lambda x.t'} \qquad \frac{t_1 \Rightarrow_\beta t_1'}{t_1\ t_2 \Rightarrow_\beta t_1'\ t_2} \qquad \frac{t_2 \Rightarrow_\beta t_2'}{t_1\ t_2 \Rightarrow_\beta t_1\ t_2'}$$

Remark

In logic, rules of inference are described by writing $\dfrac{H_1\ \ldots\ H_n}{C}$ with the intended reading: "If H_1 and ... and H_n hold, then C holds."

A **reduction derivation** is a sequence $t_1 \Rightarrow_\beta t_2 \Rightarrow_\beta \ldots \Rightarrow_\beta t_n$ of such rewrite steps, abbreviated $t_1 \Rightarrow^* t_n$.

1. t is a **normal form** if it contains no β-redexes. Also, we say that t is **normalizable** if there exists a normal form t' such that $t \Rightarrow_\beta^* t'$. In this case, we say that t' is a normal form of t.
2. t' is a **functional normal form** if all its β-redexes occur in the body of some abstraction. We say that t' is a functional normal form of t if $t \Rightarrow_\beta^* t'$ and t' is a functional normal form.

Remarkable properties of the λ-calculus

1. Not all terms are normalizable. For example, $\Omega = \omega\ \omega$ where $\omega = \lambda x.x\ x$ is not normalizable because there is only one possible reduction step of Ω, which can be repeated forever:

$$\Omega = \omega\ \omega = (\lambda x.x\ x)\ \omega \Rightarrow_\beta \omega\ \omega \Rightarrow_\beta \ldots$$

2. There may be several derivations starting from an expression: Some of them may terminate whereas other may not. For example, the following are distinct derivations of $(\lambda x.y)\ \Omega$:

$$\underline{(\lambda x.y)\ \Omega} \Rightarrow_\beta [\Omega/x]y = y \quad \text{but} \quad (\lambda x.y)\ \underline{\Omega} \Rightarrow_\beta (\lambda x.y)\ \underline{\Omega} \Rightarrow_\beta \ldots$$

3. If t is normalizable, then all its normal forms are α-congruent. This means that, if $t \Rightarrow^* t_1$, $t \Rightarrow^* t_2$, and t_1, t_2 are normal forms, then $t_1 =_\alpha t_2$.

Evaluation strategies

Reduction derivations are intended to describe the evaluation of terms.

- The evaluation of an expression t is a derivation $t \Rightarrow_\beta^* t'$ which yields an expression t', called the **value** of t.
- Several evaluation strategies have been studied over the years by programming language designers and theorists. We mention only the most important evaluation ones, and illustrate the differences between them by indicating how they evaluate the expression $\mathtt{id}\ (\mathtt{id}\ (\lambda z.\mathtt{id}\ z))$ where $\mathtt{id} = \lambda x.x$.
  1. Normal order
  2. Call-by-name
  3. Call-by-value

Normal order prescribes the selection of the leftmost outermost β-redex at any time. It can be defined as the reduction relation "$\Rightarrow_{\mathrm{n}}$" induced by the following rules of inference:

$$\frac{t \to_\beta t'}{t \Rightarrow_{\mathrm{n}} t'} \qquad \frac{t \Rightarrow_{\mathrm{n}} t'}{\lambda x.t \Rightarrow_{\mathrm{n}} \lambda x.t'} \qquad \frac{t_1 \Rightarrow_{\mathrm{n}} t_1'}{t_1\ t_2 \Rightarrow_{\mathrm{n}} t_1'\ t_2} \qquad \frac{t_1 \not\Rightarrow_{\mathrm{n}} \quad t_2 \Rightarrow_{\mathrm{n}} t_2'}{t_1\ t_2 \Rightarrow_{\mathrm{n}} t_1\ t_2'}$$

Under this strategy, the expression above is evaluated as follows:

$$\begin{aligned}
\mathtt{id}\ (\mathtt{id}\ (\lambda z.\mathtt{id}\ z)) &= \underline{(\lambda x.x)\ (\mathtt{id}\ (\lambda z.\mathtt{id}\ z))} \Rightarrow_{\mathrm{n}} \mathtt{id}\ (\lambda z.\mathtt{id}\ z) \\
&= (\lambda x.x)\ (\lambda z.\mathtt{id}\ z) \Rightarrow_{\mathrm{n}} \lambda z.(\mathtt{id}\ z) \\
&= \lambda z.(\underline{(\lambda x.x)\ z}) \Rightarrow_{\mathrm{n}} \lambda z.z =_\alpha \mathtt{id}
\end{aligned}$$

Call by name prescribes at any time the selection of the leftmost outermost β-redex, except if it is inside the body of an abstraction. Call by name can be defined as the reduction relation "$\Rightarrow_{\mathrm{cbn}}$" induced by the following rules of inference:

$$\frac{t \rightarrow_{\beta} t'}{t \Rightarrow_{\mathrm{cbn}} t'} \qquad \frac{t_1 \Rightarrow_{\mathrm{cbn}} t_1'}{t_1\ t_2 \Rightarrow_{\mathrm{cbn}} t_1'\ t_2} \qquad \frac{t_1 \not\Rightarrow_{\mathrm{cbn}} \qquad t_2 \Rightarrow_{\mathrm{cbn}} t_2'}{t_1\ t_2 \Rightarrow_{\mathrm{cbn}} t_1\ t_2'}$$

Under this strategy, the expression above is evaluated as follows:

$$\underline{\mathtt{id}\ (\mathtt{id}\ (\lambda z.\mathtt{id}\ z))} \Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id}\ (\lambda z.\mathtt{id}\ z)} \Rightarrow_{\mathrm{cbn}} \lambda z.\mathtt{id}\ z$$

Evaluation strategies

Disadvantages of call-by-name. Call-by-need

Call by name reductions have the disadvantage that they may replicate the computation of a needed argument of a function call, as illustrated by the following derivation:

$$
\begin{aligned}
\underline{(\lambda x.x\ (x\ x))\ \ (\mathtt{id\ id})} &\Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id\ id}}\ \mathtt{(id\ id\ (id\ id))} \Rightarrow_{\mathrm{cbn}} \\
&\Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id\ (id\ id\ (id\ id))}} \\
&\Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id\ id}}\ \mathtt{(id\ id)} \\
&\Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id(id\ id)}} \Rightarrow_{\mathrm{cbn}} \underline{\mathtt{id\ id}} \Rightarrow_{\mathrm{cbn}} \mathtt{id}.
\end{aligned}
$$

The first reduction step did replicate three times the need to evaluate the redex argument `id id`. This disadvantage can be eliminated by using an optimized version of call by name, known as **call by need**. This strategy avoids reevaluating an argument each time it is used by rewriting all occurrences of the argument with its value the first time it is evaluated.

Call by value prescribes at any time the selection of the leftmost outermost β-redex which fulfils the following conditions: (1) is not in the body of an abstraction, and (2) the actual argument of the β-redex is a functional normal form. Call by value can be defined as the reduction relation "$\Rightarrow_{\mathrm{cbv}}$" induced by the following rules of inference:

$$\frac{t \to_\beta t'}{t \Rightarrow_{\mathrm{cbv}} t'} \qquad \frac{t_2 \Rightarrow_{\mathrm{cbv}} t_2'}{t_1\ t_2 \Rightarrow_{\mathrm{cbv}} t_1\ t_2'} \qquad \frac{t_1 \Rightarrow_{\mathrm{cbv}} t_1' \quad t_2 \not\Rightarrow_{\mathrm{cbv}}}{t_1\ t_2 \Rightarrow_{\mathrm{cbv}} t_1'\ t_2}$$

Under this strategy, the expression above is evaluated as follows:

$$\mathtt{id}\ (\underline{\mathtt{id}\ (\lambda z.\mathtt{id}\ z)})\ \Rightarrow_{\mathrm{cbv}} \underline{\mathtt{id}\ (\lambda z.\mathtt{id}\ z)} \Rightarrow_{\mathrm{cbv}} \lambda z.\mathtt{id}\ z$$

Strategies implemented by FP languages

Most programming language use the call by value strategy.

- The call by value strategy is **strict**, in the sense that the arguments of function calls are always evaluated, even if they are not used in the body of the function.
  - Racket is a strict FP language
- Call by name and call by need are called **lazy** strategies because they evaluate only the arguments that are used in the body of the function.
  - Haskell is a lazy FP language, based on call-by-need evaluation strategy

Programming in the λ-calculus

The λ-calculus is deceptively simple, but can be used as a full-blown programming language in its own right.

- We will illustrate a number of standard examples of programming in the λ-calculus.
- Note: high-level programming languages provide clearer and more efficient ways to accomplish the tasks described in these examples.

The λ-calculus has no predefined datatypes and values, like integers, booleans, or lists. Instead, the programmer uses **combinators** to represent all the values and operations of a datatype.

- **Combinator** = term without free variables, There is no built support for functions with multiple arguments, but we can simulate them with **higher-order functions** that produce functions as results:

- A function f which computes the value of t for inputs x and y can be defined as $f = \lambda x.\lambda y.t$
  - $f\ v_1$ is $[v_1/x](\lambda y.t) = \lambda y.[v_1/x]t$. This abstraction is for the function which already knows the value of x, and is waiting for the value of y.
  - $f\ v_1\ v_2$ is parsed as $((f\ v_1)\ v_2)$, and reduced in two steps as follows:

$$((f\ v_1)\ v_2) = \underline{((\lambda x.\lambda y.t)\ v_1)}\ v_2 \Rightarrow_\beta \underline{\lambda y.[v_1/x]t\ t_2} \Rightarrow_\beta [v_2/y][v_1/x]t.$$

- This transformation of multi-argument functions into higher-order functions is called currying.

Abstract data type (ADT) = mathematical model for a certain class of data structures with similar behaviour. Is is characterised by

1. a set of constructors, that is, functions that build objects of that type from zero or more input arguments;
2. a set of functions that operate on objects of that type; and
3. a set of equational axioms that describe the properties of the operations.

ADTs can be implemented by specific data types or data structures. In the λ-calculus

there are no built-in constants or primitive operators

$\Rightarrow$ we represent ADT by terms to which we give a meaning, based on some convention.

- Two constructors

$$\texttt{true} : \texttt{Bool} \qquad \texttt{false} : \texttt{Bool}$$

- The operations

$$\begin{array}{rl} \texttt{and} & : \texttt{Bool} \times \texttt{Bool} \to \texttt{Bool} \\ \texttt{or} & : \texttt{Bool} \times \texttt{Bool} \to \texttt{Bool} \end{array} \qquad \begin{array}{rl} \texttt{not} & : \texttt{Bool} \to \texttt{Bool} \\ \texttt{if} & : \texttt{Bool} \times T \times T \to T \end{array}$$

where T can be any other type.

- The equational axioms

$$\begin{array}{lll} \texttt{not}\ \texttt{true} = \texttt{false} & \texttt{and}\ \texttt{true}\ b = b & \texttt{or}\ \texttt{true}\ b = \texttt{true} \\ \texttt{not}\ \texttt{false} = \texttt{true} & \texttt{and}\ \texttt{false}\ b = \texttt{false} & \texttt{or}\ \texttt{false}\ b = b \\ \texttt{if}\ \texttt{true}\ t_1\ t_2 = t_1 & \texttt{if}\ \texttt{false}\ t_1\ t_2 = t_2 & \end{array}$$

For the boolean values "true" and "false" we choose the combinators

$$\texttt{true} = \lambda t.\lambda f.t \quad \texttt{false} = \lambda t.\lambda f.f$$

For the conditional operation `if` and the boolean operators `not`, `and` and `or`, we can choose the following combinators:

$$\begin{array}{rclcrcl} \texttt{if} & = & \lambda x.\lambda y.\lambda z.x\ y\ z & & \texttt{not} & = & \lambda b.b\ \texttt{false}\ \texttt{true} \\ \texttt{and} & = & \lambda b.\lambda c.b\ c\ \texttt{false} & & \texttt{or} & = & \lambda b.\lambda c.b\ \texttt{true}\ c \end{array}$$

These operations are coherent with the interpretation given to them, as seen from the call by name evaluation of the following combinators (where t_1, t_2, and t are combinators):

$$\texttt{if}\ \texttt{true}\ t_1\ t_2 \Rightarrow^*_{\text{cbn}} \texttt{true}\ t_1\ t_2 = \lambda t.\lambda f.t\ t_1\ t_2 \Rightarrow^*_{\text{cbn}} t_1$$

$$\texttt{if}\ \texttt{false}\ t_1\ t_2 \Rightarrow^*_{\text{cbn}} \texttt{false}\ t_1\ t_2 = \lambda t.\lambda f.f\ t_1\ t_2 \Rightarrow^*_{\text{cbn}} t_2$$

$$\texttt{not}\ \texttt{true} \Rightarrow_{\text{cbn}} \texttt{true}\ \texttt{false}\ \texttt{true} \Rightarrow^*_{\text{cbn}} \texttt{false}$$

$$\texttt{not}\ \texttt{false} \Rightarrow_{\text{cbn}} \texttt{false}\ \texttt{false}\ \texttt{true} \Rightarrow^*_{\text{cbn}} \texttt{true}$$

Pair = composite data type that groups two arbitrary values in a compound value. The ADT for pairs has

- A constructor `pair` which takes as arguments two arbitrary values v_1, v_2, such that `pair` $v_1\ v_2$ represents the pair of values (v_1, v_2)
- The selector functions `first` and `second` that expect as input a pair, and return its first and second component, respectively.
- The equational axioms that must be satisfied by these operations are

$$\texttt{first (pair } v_1\ v_2\texttt{)} = v_1 \quad \texttt{second (pair } v_1\ v_2\texttt{)} = v_2$$

A coherent representation is given by

$$\texttt{pair} = \lambda f.\lambda s.\lambda b.b\ f\ s \qquad \begin{aligned} \texttt{first} &= \lambda p.p\ \texttt{true} \\ \texttt{second} &= \lambda p.p\ \texttt{false} \end{aligned}$$

because, for any values v_1 and v_2, we have the call by name evaluation

$$\begin{aligned} \texttt{first (pair } v_1\ v_2\texttt{)} &= \underline{(\lambda p.p\ \texttt{true})\ (\texttt{pair } v_1\ v_2)} \\ &\Rightarrow_{\text{cbn}} \underline{\texttt{pair } v_1\ v_2\ \texttt{true}} \\ &\Rightarrow^*_{\text{cbn}} \texttt{true } v_1\ v_2 \Rightarrow^*_{\text{cbn}} v_1, \end{aligned}$$

and, similarly, we can verify that $\texttt{second (pair } v_1\ v_2\texttt{)} \Rightarrow^*_{\text{cbn}} v_2$.

List = composite data type characterized by

- two constructors: (1) `null` of the empty list; and (2) `cons` v l which takes as arguments a value v and a list l, and creates the list with head v and tail l.
- The recognizer `null?` that recognizes the empty list, and the selector functions `car` and `cdr` which, when applied to a representation of a nonempty list, evaluate to its head and tail, respectively.
- The equational axioms of this data type are

$$\begin{array}{rclcrl} \texttt{null?}\,\texttt{null} &=& \texttt{true} & \quad & \texttt{car (cons } v_1\ v_2\texttt{)} &= v_1 \\ \texttt{null? (cons } v_1\ v_2\texttt{)} &=& \texttt{false} & \quad & \texttt{cdr (cons } v_1\ v_2\texttt{)} &= v_2 \end{array}$$

We can define

$$\texttt{null} = \lambda x.\texttt{true} \qquad \texttt{cons} = \lambda f.\lambda s.\lambda b.b\ f\ s$$
$$\texttt{car} = \lambda p.p\ \texttt{true} \qquad \texttt{cdr} = \lambda p.p\ \texttt{false}$$
$$\texttt{null?} = \lambda p.p\ \lambda f.\lambda s.\texttt{false}$$

This representation is coherent with the ADT for lists because each left side of an equational axiom is reducible, with the call by name strategy, to the corresponding right side. For example:

$$\begin{aligned}
\underline{\texttt{null?}\ (\texttt{cons}\ v_1\ v_2)} &\Rightarrow_{\text{cbn}} (\underline{\texttt{cons}\ v_1\ v_2)\ \lambda f.\lambda s.\texttt{false}} \\
&\Rightarrow^*_{\text{cbn}} \underline{(\lambda b.b\ v_1\ v_2)\ \lambda f.\lambda s.\texttt{false}} \\
&\Rightarrow_{\text{cbn}} (\lambda f.\lambda s.\texttt{false})\ v_1\ v_2 \Rightarrow^*_{\text{cbn}} \texttt{false} \\
\underline{\texttt{null?}\ \texttt{null}} &\Rightarrow_{\text{cbn}} \underline{(\lambda x.\texttt{true})\ \lambda f.\lambda s.\texttt{false}} \Rightarrow_{\text{cbn}} \texttt{true}
\end{aligned}$$

Non-negative integers can be represented in the λ-calculus by *Church numerals*, which are the combinators:

$$c_0 = \lambda s.\lambda z.z \quad c_1 = \lambda s.\lambda z.s\ z \quad c_2 = \lambda s.\lambda z.s\ (s\ z) \quad \ldots$$

In this encoding, each number n is represented by an abstraction c_n that takes two arguments s and z (for "successor" and "zero"), and applies s, n times, to z.

- Like booleans, numbers are encoded as active entities: the number n is represented by a function that does something n times.

All arithmetic operations can be defined to work properly with numbers as Church numerals. For example, successor, predecessor[1], addition, multiplication, test for zero, and test for equality on Church numerals can be defined as follows:

$$\begin{aligned}
\texttt{succ} &= \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)\\
\texttt{pred} &= \lambda m.\texttt{first}\,(m\ \texttt{ss}\ \texttt{zz})\\
\texttt{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z)\\
\texttt{zero?} &= \lambda m.m\ (\lambda x.\texttt{false})\ \texttt{true}\\
\texttt{times} &= \lambda m.\lambda n.m\ (\texttt{plus}\ n)\ c_0\\
\texttt{eq?} &= \lambda m.\lambda n.\texttt{and}\ (\texttt{zero?}\ (m\,\texttt{pred}\,n))\ (\texttt{zero?}\ (n\,\texttt{pred}\,m))
\end{aligned}$$

where ss and zz are used only in the definition of pred:

$$\begin{aligned}
\texttt{zz} &= \texttt{pair}\ c_0\ c_0\\
\texttt{ss} &= \lambda p.\texttt{pair}\ (\texttt{second}\,p)\ (\texttt{plus}\ c_1\ (\texttt{second}\,p))
\end{aligned}$$

[1] The predecessor of a non-negative integer n is assumed to be $n-1$ if $n > 0$, and 0 if $n = 0$.

Other ADTs, including

- trees
- arrays, and
- records

can be encoded using similar techniques.

In FP, recursion is the only way to define repetitive computations.

- How can we encode a recursive function? For example

$$\texttt{fact} : \mathbb{N} \to \mathbb{N}, \quad \texttt{fact}(n) := \begin{cases} 1 & \text{if } n = 0, \\ n \cdot \texttt{fact}(n-1) & \text{if } n > 0. \end{cases}$$

This is called definition with textual recursion, because it relies on the textual use of the function name in its own body.

We want to be able to write

$$\texttt{fact} = \lambda n.\underbrace{\texttt{if (zero? } n\texttt{)}\ c_1\ \texttt{(times } n\ \texttt{(fact (pred } n\texttt{)))}}_{\text{body containing }\texttt{fact}}$$

but the λ-calculus has no explicit support for such definitions.

All recursive functions can be encoded with the combinator

$$\mathtt{Z} = \lambda f.\mathtt{(}\lambda x.f\ \mathtt{(}\lambda y.x\ x\ y\mathtt{))}\ \mathtt{(}\lambda x.f\ \mathtt{(}\lambda y.x\ x\ y\mathtt{))}$$

- The encoding of a recursive function definition of the form $f = \langle\text{body containing } f\rangle$ is

$$\boxed{f = Z\ \lambda f.\langle\text{body containing } f\rangle}$$

For example, $\mathtt{fact} = \mathtt{Z}\ \mathtt{Fact}$ where

$\mathtt{Fact} = \lambda f.\lambda n.\mathtt{if}\ \mathtt{(zero?}\ n\mathtt{)}\ c_1\ \mathtt{(times}\ n\ \mathtt{(}f\ \mathtt{(pred}\ n\mathtt{)))}$

$\mathtt{fact}$ simulates the recursive definition of the factorial function: for all $n \in \mathbb{N}$: $\mathtt{fact}\ c_n \Rightarrow_\beta^* v =_\alpha c_{n!}$

Concluding remarks

1. The λ-calculus is the core language of most FP languages, including Racket and Haskell.
2. **Computation** = reduction of an expression to a value with rewrite derivations that respect a fixed and predictable evaluation strategy:
   - strict languages implement **call-by-value**.
     Example: Racket
   - lazy languages implement **call-by-name** or **call-by-need**.
     Example: Haskell
3. The λ-calculus is a a full-fledged programming language:
   - we can encode the missing things: ADTs, operations on them, recursion, etc.
   - computations in pure λ-calculus is inefficient

   $\Rightarrow$ all modern FP languages implement extensions of the λ-calculus with
     - built-in datatypes
     - built-in support for textual recursion (not via fixed-point combinators)
     - etc.

Bibliographic references

1. B. C. Pierce. Types and Programming Languages. MIT Press, Cambridge, MA, USA, 2002.
2. J. C. Mitchell. Foundations of Programming Languages. MIT Press, Cambridge, MA. London, England, 2000.

   Chapter 1. Introduction. Subsections 1.1-1.3.