The pure (type free) λ-calculus

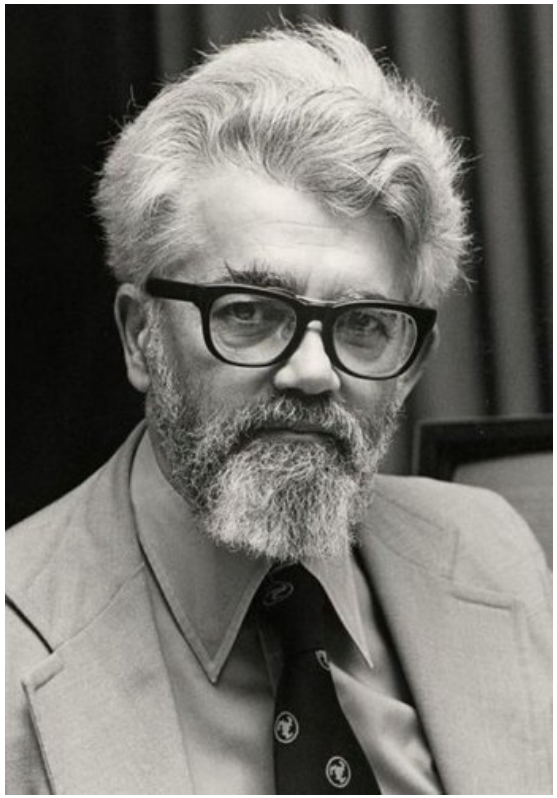
Julio Mariño

Theory of Programming Languages
MASTER IN FORMAL METHODS FOR SOFTWARE ENGINEERING
Universidad (Politécnica | Complutense | Autónoma) de Madrid

November 17, 2020

a brief history of functional programming

- 1930: Alonzo Church develops the theory of the λ-calculus.
- 1960: McCarthy delivers the first version of LISP.
- 1980: United Kingdom: modern functional languages: ML, Miranda, Hope, Haskell ...



the pure λ-calculus

- It is a *calculus of anonymous functions* developed by Alonzo Church in the 1930s as a possible foundation for logic and mathematics.
- While other approaches did exist at the time – e.g. set theory – there was a general concern about devising simple and paradox-free formalisms.
- Alan Turing (who eventually read his Ph.D dissertation under Church's direction) proved in 1937 that Turing machines and the λ-calculus were equivalent in terms of computational power.
- The pure λ-calculus can be seen as the first (formal) programming language and possibly the simplest one.
- It tries to capture (like combinators) the possible ways in which expressions can be manipulated by a mathematician.
- While functional programming languages are largely influenced by the λ-calculus and λ-abstractions can be routinely interpreted as functions, giving a functional interpretation for every λ-term is not an easy task.

pure λ-calculus

syntax and semantics

- language:

$$x ::= x_1 \mid x_2 \mid x_3 \mid \ldots$$

$$e ::= x \mid (\lambda x.\, e) \mid (e e')$$

We will save silly parentheses by associating applications to the left i.e.:

$$e\, f\, g = ((e\, f)\, g)$$

- conversion and reduction rules:

$$\begin{array}{ll} (\alpha) & \lambda x.\, e =_\alpha \lambda y.(e[x \mapsto y]) \\ (\beta) & (\lambda x.\, e)e' \rightsquigarrow_\beta e[x \mapsto e'] \end{array}$$

  - α-conversion expresses the fact that the actual name of the parameters in a function definition should be irrelevant.
  - β-reduction reduces function application by copying actual parameters in place of formal ones.

variable capture

- We have to be very careful when renaming variables:

$$\lambda x.\, f\ x\ y \not\rightarrow_{\alpha} \lambda y.\, f\ y\ y$$

Clearly, the second term is not equivalent to the first one — there is no way to convert it back to the original one.

capture-avoiding substitution

$$
\begin{array}{lcll}
x[x \mapsto t] & = & t & \\
y[x \mapsto t] & = & y & \\
(p\ q)[x \mapsto t] & = & ((p[x \mapsto t])(q[x \mapsto t])) & \\
(\lambda x.p)[x \mapsto t] & = & \lambda x.p & \\
(\lambda y.p)[x \mapsto t] & = & \lambda y.p & \text{if } x \notin FV(p) \\
(\lambda y.p)[x \mapsto t] & = & \lambda y.(p[x \mapsto t]) & \text{if } x \in FV(p) \text{ and } y \notin FV(t) \\
(\lambda y.p)[x \mapsto t] & = & \lambda z.(p[y \mapsto z][x \mapsto t]) & \text{if } x \in FV(p) \text{ and } y \in FV(t)
\end{array}
$$

conversion, revisited

- α-conversion:

$$\lambda x.\ e \rightsquigarrow_\alpha \lambda y.\ e[x \mapsto y] \qquad \text{if } y \notin FV(e)$$

- β-conversion:

$$(\lambda x.\ e)e' \rightsquigarrow_\beta e[x \mapsto e']$$

- Notice how the previous definition of substitution performs *implicit* α*-conversions*:

$$\begin{array}{ll} \lambda x.((\lambda y.\lambda x.\ x\ y)x) & \not\rightsquigarrow_\beta \\ \lambda x.(\lambda x.\ x\ x) & (WRONG!!) \end{array}$$

$$\begin{array}{ll} \lambda x.((\lambda y.\lambda x.\ x\ y)x) & \rightsquigarrow_\alpha \\ \lambda x.((\lambda y.\lambda z.\ z\ y)x) & \rightsquigarrow_\beta \\ \lambda x.(\lambda z.\ z\ x) & (RIGHT!!) \end{array}$$

- Exercise: Program conversions in Haskell, Prolog or your language of choice.

Defining:

$$\text{TRUE} = \lambda x.\lambda y.\ x$$

$$\text{FALSE} = \lambda x.\lambda y.\ y$$

then we can introduce the conditional (*if_then_else*):

$$\text{COND} = \lambda b.\lambda x.\lambda y.\ b\ x\ y$$

Exercise. Prove the following by conversion:

$$\text{COND TRUE}\ x\ y \rightsquigarrow x$$

$$\text{COND FALSE}\ x\ y \rightsquigarrow y$$

boolean logic in the pure λ-calculus (2)

$$
\begin{array}{ll}
\text{COND TRUE } x\, y & = \\
(\lambda b.\lambda x.\lambda y.\ b\ x\ y)\ (\lambda x.\lambda y.\ x)\ x\ y & \rightsquigarrow_{\alpha}^{*} \\
(\lambda b.\lambda z.\lambda w.\ b\ z\ w)\ (\lambda a.\lambda c.\ a)\ x\ y & \rightsquigarrow_{\beta} \\
(\lambda z.\lambda w.\ (\lambda a.\lambda c.\ a)\ z\ w)\ x\ y & \rightsquigarrow_{\beta} \\
(\lambda w.\ (\lambda a.\lambda c.\ a)\ x\ w)\ y & \rightsquigarrow_{\beta} \\
(\lambda a.\lambda c.\ a)\ x\ y & \rightsquigarrow_{\beta} \\
(\lambda c.\ x)\ y & \rightsquigarrow_{\beta} \\
x &
\end{array}
$$

exercises:

the λ-calculus as a programming language

- Define pairs
- Define natural numbers
  - Use the so-called *Church numerals*:

$$\begin{array}{lcll} \overline{0} & = & \lambda f.\ \lambda x.\ x & \\ \overline{1} & = & \lambda f.\ \lambda x.\ f\ x & \\ \overline{2} & = & \lambda f.\ \lambda x.\ f(f\ x) & \\ & \vdots & & \\ \overline{n} & = & \lambda f.\ \lambda x.\ f(f(\ldots f\ x)) & \text{apply } f \text{ } n \text{ times} \end{array}$$

  - Program *addition*, *multiplication* and *exponentiation*.
- Define lists

self-reference, recursion and paradoxes

- self-reference was considered problematic by classic rhetoricians and philosophers — *the definiendum must not appear inside the definiens*
- of course, we all know the computational benefits of a good use of safe recursion schemes...
- ...although bad schemes may lead to nontermination, etc
- self-reference is also in the heart of some well-known paradoxes:

Cantor's paradox

"The set of all sets is not a set"

Russell's paradox

"The set of all sets that do not belong to themselves"

Russell's paradox

- Russell's paradox can be rephrased in a number of ways, showing the inadequacies of natural language, e.g.:

the barber's paradox

In a certain town the barber shaves all those men who cannot shave themselves

the heterology paradox

A property is a *homology* if it is true of itself. A property is a *heterology* if it is true of itself.

- The test for heterologies can be programmed in the untyped λ-calculus:

$$\text{HETEROLOGY} = \lambda x.\ \text{NOT}\ (x\ x)$$

In order to *solve* the paradox, it would suffice to evaluate

$$\text{HETEROLOGY HETEROLOGY}$$

a fixed-point combinator

$$\begin{aligned}\text{HETEROLOGY HETEROLOGY} =& (\lambda x.\text{NOT } (x\ x)\ \text{HETEROLOGY}) && \rightsquigarrow_\beta \\ & \text{NOT } (\text{HETEROLOGY HETEROLOGY}) && \rightsquigarrow_\beta \\ & \text{NOT } (\text{NOT } (\text{HETEROLOGY HETEROLOGY}))\end{aligned}$$

- Indeed, for every f,

$$(\lambda x.f\ (x\ x)\ \lambda x.f\ (x\ x)) \rightsquigarrow_\beta f\ ((\lambda x.f\ (x\ x)\ \lambda x.f\ (x\ x))$$

  so, defining

$$\mathrm{Y} = \lambda f.(\lambda x.f\ (x\ x)\ \lambda x.f\ (x\ x))$$

  we obtain

$$\mathrm{Y}\, f \rightsquigarrow f\ (\mathrm{Y}\, f)$$

- Y is called the *paradoxical combinator* and is one way of expressing general recursion in the λ-calculus

is β reduction function evaluation?

Note that there can be more than one N such that $M \rightsquigarrow_\beta N$. For example

$$(\lambda x.\lambda y.xy)((\lambda z.t)a) \quad \rightsquigarrow_\beta \quad (\lambda x.\lambda y.xy)(t[a/z])$$

and also

$$\rightsquigarrow_\beta \ \lambda y.((\lambda z.\ t)a)\ y$$

or there can be none at all: a very important case.

Definition

If M does not have a redex among its subterms, (thus being β-irreducible) it is said to be a **normal** term, or a term in normal form.

many-step β reduction

Finally we define $\leadsto_\beta^*$: β-reduction (not just a single step) as the transitive closure of $\leadsto_\beta$. That is to say

Definition

M is β-**reducible** to N ($M \leadsto_\beta^* N$) if and only if there are terms $M_1, \ldots, M_k$ such that

$$M \leadsto_\beta M_1, M_1 \leadsto_\beta M_2, \ldots, M_k \leadsto_\beta N.$$

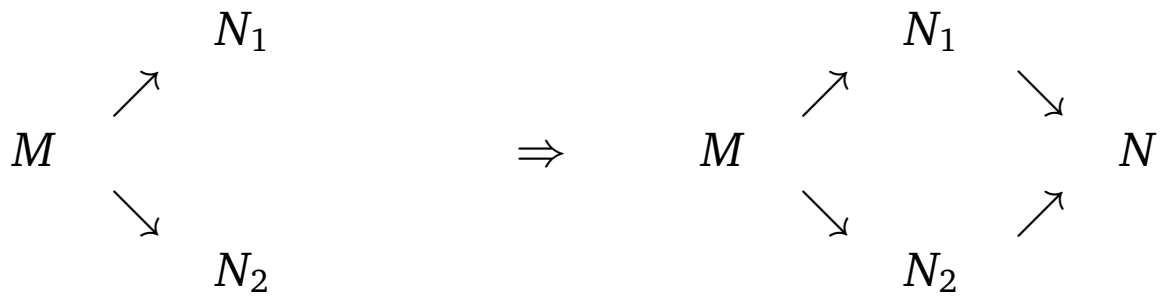
The symmetric and reflexive closure of this relation is called β-**equivalence** $=_\beta$, or also beta-conversion.

is beta reduction function evaluation?

the Church-Rosser property

Theorem (Diamond property)

If $M \rightsquigarrow N_1$ and $M \rightsquigarrow N_2$ then there is a term N such that $N_1 \rightsquigarrow N$ and $N_2 \rightsquigarrow N$. (where $\rightarrow$ can be β or $\beta\eta$).



Corollary: *Normal forms are* **unique**

This would seem to suggest that if M and A are terms, then we can think of the term (MA) as denoting a computation of the operation M with input A and the reduction of such a term to normal form as yielding the (unique) output.
This leaves us wondering when we can be certain that normal forms exist or how to make sure we choose reduction paths that reach them.

Some terms can be *beta*-reduced forever:

$$(\lambda x.(xx))(\lambda x.(xx)) \overset{\beta}{\rightsquigarrow} (\lambda x.(xx))(\lambda x.(xx))$$

This term is called Ω. Also $\omega\omega$ where $\omega = (\lambda x.(xx))$. So: $\Omega \overset{\beta}{\rightsquigarrow} \Omega$.
The behaviour of other terms depends on the reduction *strategy:*

$$\begin{array}{ccccc} (\lambda xy.y)\underline{\Omega} & \overset{\beta}{\rightsquigarrow} & \cdots & & (\lambda xy.y)\Omega \\ \downarrow & & & & \downarrow \\ (\lambda y.y) & & & & (\lambda y.y) \end{array}$$

normalizable terms

Definition

A term M is *normalizable* if there is a finite sequence of β-reductions starting at M end ending with a normal form. M es *strongly normalizable* (SN) if **all** secuences starting at M are finite.

Thus $(\lambda xy.y)\Omega$ is normalizable but not SN.

Theorem (Turing)

It is not decidable if a term in the lambda-calculus is normalizable.

Lemma (uniqueness of normal forms)

Modulo α-conversion, a term M has at most one β-nf.

Definition (leftmost reductions)

The leftmost β-redex-occurrence in a term P is the β-redex-occurrence whose leftmost parenthesis is to the left of the other redexes.
The leftmost β-reduction of a term P is a β-reduction with maximal length in which the leftmost redex is fired at each step.

Theorem (leftmost reductions suffice)

A term M has a β-nf M^*_β iff the leftmost β-reduction of M is finite and ends at M^*_β.

- With the functional interpretation of lambda abstractions, the terms M and $\lambda x.\, M\, x$ should denote the same function (see why?). This is called the *extensionality principle*, and *cannot* be proven from β convertibility.
- So, we define η-reduction as

$$\lambda x.\, M\, x \rightsquigarrow_\eta M$$

  and η-redexes, η-reduction, η-conversion ($=_\eta$), η-normal forms, etc., analogously to the β case.
- The interleaving of η- and β- reduction steps leads to the notions of $\beta\eta$-reduction ($\rightsquigarrow_{\beta\eta}$) and conversion ($=_{\beta\eta}$).
- Fortunately, η-conversion does not add much complexity to the λ-calculus operational semantics:

Theorem

A λ-term M has a $\beta\eta$-normal form iff it has a β-normal form.

completeness

Theorem (Turing)

All computable functions $f : \mathbb{N} \to \mathbb{N}$ *from natural numbers to natural numbers are* representable *in the* λ*-calculus.*