The simply-typed λ-calculus

Julio Mariño

Theory of Programming Languages
Master in Formal Methods for Software Engineering
Universidad (Politécnica | Complutense | Autónoma) de Madrid

November 22, 2021

the simply-typed λ-calculus

Russell enriched set theory with a *type system* to escape from the paradox, only allowing well-founded sets. Church thinks something similar could eliminate paradoxes from the λ-calculus, forbidding self-application in terms like $\lambda x.x\,x$, and keeping the intuition that λ-abstractions denote functions.

Types may be atomic ($a, b, c \ldots$) or functional ($\sigma \to \tau$, where σ and τ are types).

Church's approach

Only well typed terms are allowed into the system, i.e. syntax of λ-terms is modified so that only typed expressions exist:

$$x_a, x_{a\to b}, \lambda x_a.x_a, \lambda x_b.x_b, \lambda x_{a\to b}.x_{a\to b}, \lambda x_a.\lambda y_b b.x_a$$

$$\lambda x_a.\lambda y_a.x_a, \lambda f_{a\to b\to c}.\lambda x_a.\lambda y_b.f_{a\to b\to c}\, x_a\, y_b$$

Curry's approach

Syntax is free, as before, but there is a type assignment theory, so that some term may have several typings, or no possible typing.

$$\frac{x : \tau \quad e : \tau'}{(\lambda x.\, e) : \tau \to \tau'}$$

$$\frac{e : \tau' \to \tau \quad e' : \tau'}{(e\, e') : \tau}$$

λ-calculi with simple types

church or curry – which flavour do you prefer?

- In the Church term system variables are typed, so there are infinitely many versions of identity:

$$\lambda x_a.x_a, \lambda x_{a\to b}.x_{a\to b} \ldots$$

- ... and no term for ω, Ω, Y, etc.
- We can save some ink by just annotating variables in abstractions, when terms are closed:

$$\lambda x : a.x, \lambda x : (a \to b).x \ldots$$

  which makes it similar to programming languages with explicit static typing, e.g. Java.
- Curry, on the other hand, thinks that forbidding the terms with self-application terms and the proliferation of identities is an unnecessary complication.
- In Curry's type assignment system, there is a single λ-term for identity (with infinitely many type assignments) and ω, Ω and Y are still valid terms with no type assignment.

Definition (type assignment)

A type assignment is any expression $M : \tau$ where M (the subject) is a λ-term and τ is a simple type (the predicate).

Definition (type context)

A type context Γ is a finite assignment of types to variables $x_1 : \tau_1, \ldots, x_n : \tau_n$. The assignment is *consistent* if no variable is assigned different types.

Definition (type judgements)

A TA$_\lambda$ formula, or judgement is a triple (Γ, M, τ) usually written

$$\Gamma \vdash M : \tau$$

and read "under the assumptions Γ the term M can be assigned the type τ".

TA$_\lambda$

type assignment as a deduction system

- Infinite axioms:

$$x : \tau \vdash x : \tau \quad (\text{VAR})$$

- Two deduction rules:

$$\frac{\Gamma \cup x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}\ \text{ABS}$$

$$\frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma' \vdash M' : \tau}{\Gamma \cup \Gamma' \vdash MM' : \tau'}\ \text{APP}$$

- This presentation requires one structural rule: *context strengthening*.
- $\Gamma \cup \Gamma'$ must be consistent. Inductively, this ensures that all the contexts assumed in TA$_\lambda$ proofs are consistent.

TA$_\lambda$ in natural deduction style

1. $\vdash \lambda x.x : a \to a$

$$\frac{[x : a] \qquad [x : a]}{\lambda x.x : a \to a}$$

2. $\lambda x.\lambda y.x : a \to b \to a$

$$\frac{[x : a] \qquad \dfrac{[y : b] \qquad [x : a]}{\lambda y.x : b \to a}}{\lambda x.\lambda y.x : a \to b \to a}$$

3. $\lambda f.\lambda g.\lambda x.f\ (g\ x) : (b \to c) \to (a \to b) \to (a \to c)$

assumptions must be consistent!

subject reduction

- The type discipline should not only forbid the *problematic* terms (ω, Ω, Y), but also any other term that reduces to them.
- There must be some connection between type assignment and reduction.

Theorem (subject reduction)

If $\Gamma \vdash P : \tau$ and $P \rightsquigarrow^*_{\beta\eta} Q$, then

$$\Gamma \vdash Q : \tau$$

- So, reduction preserves types, and it is not possible that a typable term reduces to the problematic terms above.

normalization properties of typed terms

- We have just seen that no typable term reduces to a term with self-application.
- Is this also true for all the divergent terms?

Weak Normalization Theorem

Every TA_λ-typable term has a β and a $\beta\eta$ normal form

Strong Normalization Theorem

Every $\beta\eta$-reduction starting in a TA_λ-typable term is finite

principal types

- All the types that can be assigned to a term are instances of a principal (most general) type.
- This is more or less obvious taking into account that the typing proofs for a term in TA_λ only differ in the assumptions.
- From a deduction system to constraint-based type inference:

$$\frac{\Gamma \cup x : \tau_1 \vdash M : \tau_2 \qquad \Gamma \vdash \tau_3 = \tau_1 \to \tau_2}{\Gamma \vdash \lambda x.M : \tau_3}\ \mathrm{ABS}$$

$$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash M' : \tau_2 \qquad \Gamma \vdash \tau_1 = \tau_2 \to \tau_3}{\Gamma \vdash MM' : \tau_3}\ \mathrm{APP}$$

- In a proof tree type variables corresponding to *different* variables must be different.

the principal type reconstruction algorithm

- Example: $(\lambda x.x)(\lambda x.x)$

$$\dfrac{\dfrac{x : \tau_1}{\lambda x.x : \tau_3} \qquad \dfrac{x : \tau_2}{\lambda x.x : \tau_4}}{(\lambda x.x)(\lambda x.x) : \tau_5}$$

- It generates the following system of type equations:

$$\begin{aligned} \tau_3 &= \tau_4 \to \tau_5 \\ \tau_3 &= \tau_1 \to \tau_1 \\ \tau_4 &= \tau_2 \to \tau_2 \end{aligned}$$

- The system is solvable if it admits a *most general unifier* (mgu):

$$\begin{aligned} \tau_3 &= \tau_4 \to \tau_5 \\ \tau_1 &= \tau_4 \\ \tau_1 &= \tau_5 \\ \tau_4 &= \tau_2 \to \tau_2 \end{aligned} \qquad \begin{aligned} \tau_3 &= \tau_4 \to \tau_5 \\ \tau_1 &= \tau_4 \\ \tau_4 &= \tau_5 \\ \tau_4 &= \tau_2 \to \tau_2 \end{aligned} \qquad \begin{aligned} \tau_3 &= \tau_4 \to \tau_5 \\ \tau_1 &= \tau_4 \\ \tau_4 &= \tau_5 \\ \tau_5 &= \tau_2 \to \tau_2 \end{aligned} \qquad [\tau_2 \mapsto a] \qquad \begin{aligned} \tau_3 &= (a \to a) \to (a \to a) \\ \tau_1 &= a \to a \\ \tau_4 &= a \to a \\ \tau_5 &= a \to a \end{aligned}$$