Foundations of Programming Languages 2023

The typed λ-calculus: $\lambda_{\rightarrow}$

Sandra Alves

October 2023

The typed λ-calculus: $\lambda_{\rightarrow}$

In the 1900's Russel introduced the theory of types to deal with paradoxes.

When these paradoxes were found in the λ-calculus, both Church and Curry introduced simply typed variants of the λ-calculus.

The relation between simple types and intuitionistic logic was then estabilish by Curry and Howard.

In the context of programming languages types are used identify different objects, enforce behavior, extract and guarentee properties, etc...

Simple types

Let $\mathbb{V}$ be a set of type variables. The *set of simple types*, $\mathbb{T}_C$ is inductively defined as:

$$\begin{aligned} \alpha \in \mathbb{V} &\Rightarrow \alpha \in \mathbb{T}_C \\ \sigma, \tau \in \mathbb{T}_C &\Rightarrow (\sigma \to \tau) \in \mathbb{T}_C \end{aligned}$$

Notation: If $\tau_1, \ldots, \tau_n \in \mathbb{T}_C$, then

$$\tau_1 \to \tau_2 \to \cdots \to \tau_n$$

represents

$$(\tau_1 \to (\tau_2 \to \cdots \to (\tau_{n-1} \to \tau_n) \ldots))$$

that is, the type constructor $\to$ is right associative.

Examples

Types:

$$(\tau_1 \to \tau_1)$$
$$(\tau_1 \to (\tau_2 \to \tau_1))$$
$$((\tau_1 \to (\tau_2 \to \tau_3)) \to ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3))$$

can just be written as:

$$\tau_1 \to \tau_1$$
$$\tau_1 \to \tau_2 \to \tau_1$$
$$(\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$$

Types à la Curry

An environment Γ is a set of pairs:

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

A type declaration:

$$\underbrace{x_i}_{\textit{subject}} : \underbrace{\tau_i}_{\textit{predicate}}$$

Γ is a partial function, $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$, and $\Gamma(x_i) = \tau_i$.

We write $\Gamma, x : \tau$ for $\Gamma \cup \{x : \tau\}$ where $x \notin \mathrm{dom}(\Gamma)$.

Typing judgements

A derivable judgement

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

is obtained from:

$$\Gamma, x : \tau \vdash x : \tau \qquad \text{(Axiom)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad (\to \text{Elim})$$

If $\Gamma \vdash M : \tau$ then we say that M has type τ under Γ.

Examples

Recall the λ-terms:

(i) $I = \lambda x.x$

(ii) $K = \lambda xy.x$

(iii) $S = \lambda xyz.xz(yz)$

For which we have:

(i) $\vdash I : \tau_1 \to \tau_1$

(ii) $\vdash K : \tau_1 \to \tau_2 \to \tau_1$

(iii) $\vdash S : (\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$

Verify...

Decision problems involving types

We consider the following problems, for every system deriving judgements of the form:

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

Type checking: decide if Γ, M and τ verify $\Gamma \vdash M : \tau$.

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

Typability (inference/reconstruction): decide if M is typable.

$$? \vdash M :?$$

Type inhabitation (*emptiness*): decide if τ has a closed term that inhabits it.

$$\vdash ? : \tau$$

All these problems are decidable for $\lambda_{\to}$.

Type checking versus Typability

In the simple type system, type-checking and typability problems can be reduced to each other.

To check if a term M is typeable, where $\mathrm{fv}(M) = \{x_1, \ldots, x_n\}$, we can ask the following question:

$$x_0 : p \vdash \mathbf{K} x_0 (\lambda x_1 \ldots x_n . M) : p$$

This reduces the typability problem to the type checking problem.

On the other hand, type-checking can be reduced to typability (using unification).

Simply typed λ-calculus à la Church

Assume an infinite set of typed variables $\Upsilon_\sigma = \{x_1^\sigma, x_2^\sigma, \dots\}$, for each $\sigma \in \mathbb{T}_C$.

The set of simply typed terms of type σ, denoted Λ_σ is thus defined:

- $\Upsilon_\sigma \subseteq \Lambda_\sigma$;
- If $M \in \Lambda_{\sigma\to\tau}$ and $N \in \Lambda_\sigma$ then $(MN) \in \Lambda_\tau$;
- If $M \in \Lambda_\tau$ and $x^\sigma \in \Upsilon_\sigma$ then $(\lambda x^\sigma M) \in \Lambda_{\sigma\to\tau}$;

The set of simply typed terms is $\cup\{\Lambda_\sigma \mid \sigma \in \mathbb{T}_C\}$.

Another formulation of Church typing

We define raw terms of the Church style $\lambda_{\to}$:

- If $x \in \mathcal{V}$ then x is a raw term;
- If M and N are raw terms then (MN) is a raw term;
- If M is a raw term, $x \in \mathcal{V}$ and $\sigma \in \mathbb{T}_C$ then $(\lambda x : \sigma M)$ is a raw term.

Notions and notation on untyped terms extend to raw terms as expected.

The term $(\lambda x : \tau \lambda y : \sigma . x)$ is a raw term, that we can also write as $(\lambda x^{\tau} \lambda y^{\sigma} . x)$

Typing judgements à la Church

A derivable judgement

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

is obtained from:

$$\Gamma, x : \tau \vdash x : \tau \qquad \text{(Axiom)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad (\to \text{Elim})$$

Examples

As before we have:

(i) $\vdash I : \tau_1 \to \tau_1$

(ii) $\vdash K : \tau_1 \to \tau_2 \to \tau_1$

(iii) $\vdash S : (\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$

But now we consider raw terms:

(i) $I = \lambda x_1^{\tau}.x$

(ii) $K = \lambda x^{\tau_1} \lambda y^{\tau_2}.x$

(iii) $S = \lambda x^{\tau_1 \to \tau_2 \to \tau_3} \lambda y^{\tau_1 \to \tau_2} \lambda z^{\tau_1}.xz(yz)$

Verify...

Church versus Curry

The original Church formulation assumed that types of variables and terms are fixed.

Terms are *"build"* in such a way that well-formed terms are correctly typed by definition.

The difference between typing à la Curry and typing à la Church is the same as the difference between explicitly typed languages (like C or Pascal) and implicitly typed languages (like ML or Haskell).

Note that our second formulation of à la Church type system is somewhere between both (free variables are not explicitly typed).

Raw terms become a typed term à la Church, when associated to an environment, in which case its type is unique (unlike in the Curry system).

Church versus Curry (cont.)

We can define an erasing function $|\cdot|$ from terms in Church to terms in Curry:

$$\begin{aligned} |x| &= x \\ |MN| &= |M||N| \\ |\lambda x : \sigma.M| &= \lambda x.|M| \end{aligned}$$

Which is preserved by reduction and typing:

(i) If $M \to_\beta N$ then $|M| \to_\beta |N|$.

(ii) If $\Gamma \vdash M : \sigma$ in the Church system then $\Gamma \vdash |M| : \sigma$ in the Curry system.

Exercise: Verify...

Church versus Curry (cont.)

Additionally, we can *"lift"* each derivation à la Curry:

$$\Gamma \vdash M : \sigma$$

to a derivation à la Church.

(i) If $M \to_\beta N$ and $M = |M'|$ then $M' \to_\beta N'$, for some N' such that $|N'| = N$.

(ii) If $\Gamma \vdash M : \sigma$ in Curry system then there exists M' such that $|M'| = M$ and $\Gamma \vdash M' : \sigma$ in Church system.

It is possible to translate properties between both systems.

Some properties of $\lambda_{\to}$

Substitution preserves typings:

$$\Gamma, x : \sigma \vdash M : \tau \text{ and } \Gamma \vdash N : \sigma \Rightarrow \Gamma \vdash M[N/x] : \tau$$

Reduction preserves typings:

$$\Gamma \vdash M : \tau \text{ and } M \twoheadrightarrow N \Rightarrow \Gamma \vdash N : \tau$$

$\lambda^{\to}$ is terminating:

$$\Gamma \vdash M : \tau \Rightarrow M \text{ is strongly normalisable}$$

Simple types and termination

We prove that simple types are strongly normalising using the computability/reducibility proof method due to Tait and Girard.

We define the notion of computability inductively on types:

(a) M^{α} of base type α is computable iff M^{α} is terminating.

(b) $M^{\sigma \to \tau}$ is computable iff for every computable N^{σ} the term $(MN)^{\tau}$ is computable.

We now prove:

(i) If M^{τ} is computable then M is terminating

(ii) If $(xM_1 \ldots M_n)^{\tau}$ is terminating then $(xM_1 \ldots M_n)^{\tau}$ is computable.

Simple types and termination (cont.)

Proof: By induction of τ.

- If τ is an atomic type α then:
  - (i) By Def. (a).
  - (ii) If $(xM_1 \dots M_n)^\alpha$ is terminating then $(xM_1 \dots M_n)^\alpha$ is computable by Def. (a).
- If τ is of the form $\tau_1 \to \tau_2$ then:
  - (i) Assume $M^{\tau_1 \to \tau_2}$ computable. Also x^{τ_1} is computable by i.h. (ii). Then, by definition Mx^{τ_2} is computable, therefore terminating by i.h. (i).
  - (ii) Assume $xM_1 \dots M_n^{\tau_1 \to \tau_2}$ is terminating and assume N^{τ_1} is computable, therefore terminating by i.h. (i). Then $xM_1 \dots M_n N^{\tau_2}$ is terminating, therefore $xM_1 \dots M_n N^{\tau_2}$ is computable, which implies $xM_1 \dots M_n^{\tau_1 \to \tau_2}$ is computable.

Simple types and termination (cont.)

We have established that every typable computable term is terminating.

We now prove that every typable term M^σ is computable.

Let us prove a more general result...

Lemma: For every $x_1^{\tau_1}, \ldots, x_n^{\tau_n}$ and computable terms $N_1^{\tau_1}, \ldots, N_n^{\tau_n}$ then $(M^*)^\sigma = M[N_1/x_1] \ldots [N_n/x_n]$ is computable.

For which we will need the following substitution lemma:

Lemma: If $M[N/x]P_1 \ldots P_n$ $(n \geq 0)$ is computable and N is computable, then $(\lambda x.M)NP_1 \ldots P_n$ is also computable.

Proof: Exercise

Simple types and termination (cont.)

Proof: By induction on M.

- $M = x_i$, in which case $\sigma = \tau_i$. Then $M^* = N_i$, and N_i is computable by hypothesis.
- $M = y$ (and y is none of x_i), in which case $M^* = y$ and y is computable.
- $M = M_1 M_2$, then $M^* = M_1^* M_2^*$, which are computable by i.h., therefore M^* is computable.
- $M^\sigma = \lambda x : \sigma_1 . M_1^{\sigma_2}$ and $\sigma = \sigma_1 \to \sigma_2$, and $M^* = \lambda x . M_1^*$. Then M^* is computable, if $(M^* N^{\tau_1})$ is computable, for every N^{τ_1} computable. By i.h. $M_1^*[N/x]$ is computable, therefore $M^* N$ is computable, by the substitution lemma.

Church-Rosser

Recall that:

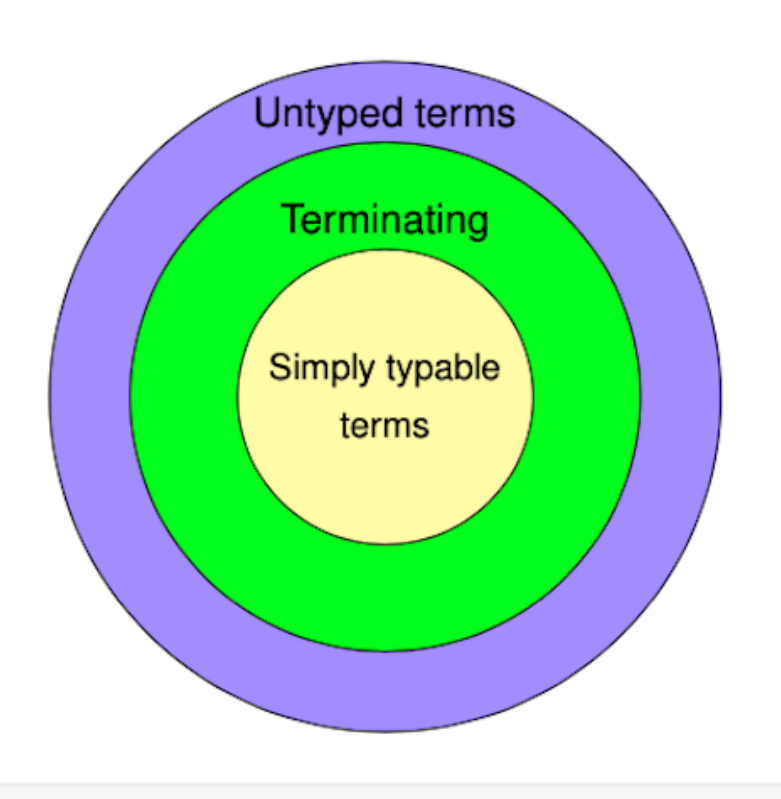
$$\text{Strong Normalization} \wedge \text{Weak-Church Rosser} \Rightarrow \text{Church Rosser}$$

Since we have proved that the simply typed λ-calculus is SN, then confluence follows from WCR.

Exercise: Prove WCR for the simply typed λ-calculus.

Expressivity of simple types

Simply typed λ-terms are terminating, but what exactly do they characterise?



Expressivity of simple types: the extended polynomials

A numeric function $f : \mathbb{N}^k \to \mathbb{N}$ is $\lambda_{\to}$-definable if and only if there exists $\vdash F : \mathsf{int} \to \cdots \to \mathsf{int} \to \mathsf{int}$ such that:

$$F\,\underline{n_1}\ldots\underline{n_k} =_\beta \underline{f(n_1,\ldots,n_k)}$$

where $\mathsf{int} = (\alpha \to \alpha) \to \alpha \to \alpha$.

Extended polynomials is the smallest class of numeric functions containing projections, constant functions the signum function, and closed under addition and multiplication.

Exercise: Prove that the functions above are $\lambda_{\to}$-definable.

Theorem (Schwichtenberg) The $\lambda_{\to}$-definable functions are exactly the extended polynomials.

Beyond simple types: Gödel's System T

System **T** extends $\lambda_{\to}$ with types for numbers and booleans:

$$\sigma ::= \cdots \mid \mathbf{nat} \mid \mathbf{bool}$$

The set of terms adds to λ-terms:

$$M, N, P ::= \cdots \mid \mathbf{Z} \mid \mathbf{S}(M) \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{R}(M, N, P) \mid \mathbf{C}(M, N, P)$$

Gödel's System **T** is far more powerful that $\lambda_{\to}$...

... while still being strongly normalising!

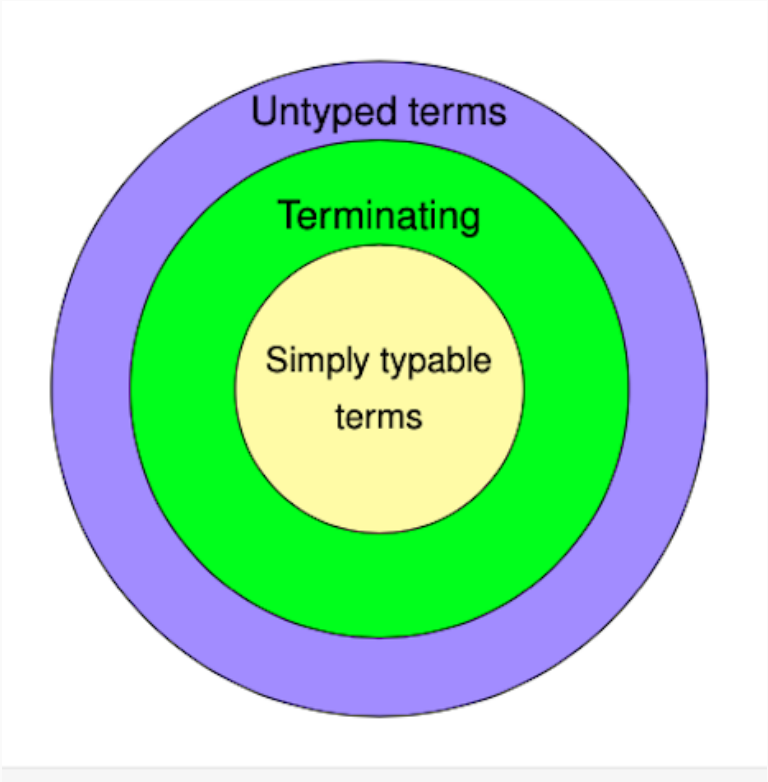
"In other words, the expressive power of the system T is enormous, and much more than what is feasible on a computer!"

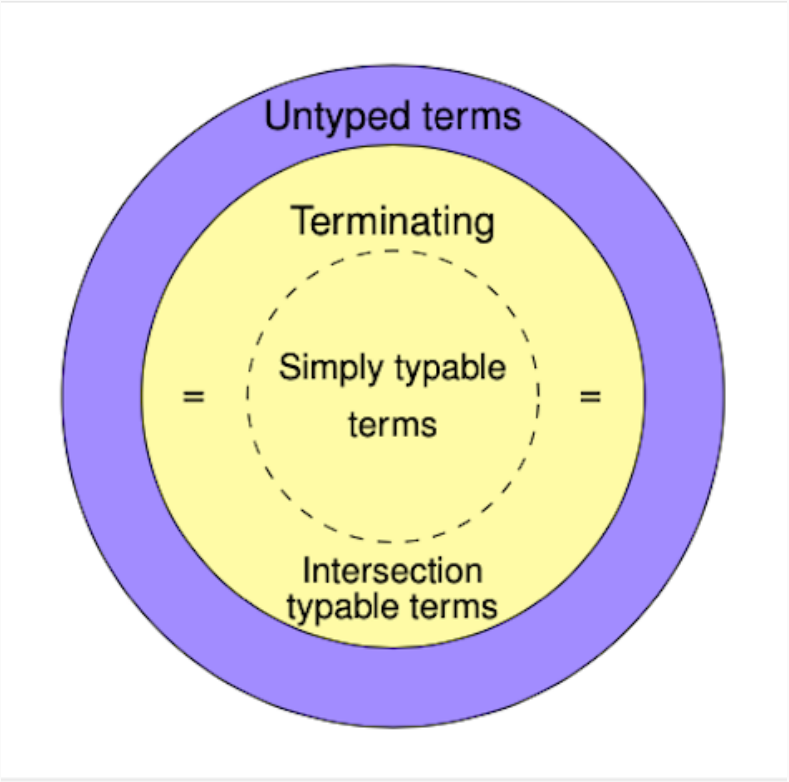
- J.Y. Girard and P. Taylor

Can types characterise termination?[1]

Yes! Intersection types characterise terminating terms.

$$\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \sigma \cap \tau$$





[1] Thanks to Delia Kesner for the pictures!

Some quotes on the λ-calculus

"There may, indeed, be other applications of the system than its use as a logic."

- Alonzo Church

"λ-calculus and combinatory logic are regarded as 'test-beds' in the study of higher-order programming languages: techniques are tried out on these two simple languages, developed, and then applied to other more 'practical' languages."

- Hindley and Seldin

Some quotes on the λ-calculus

"Some people prefer not to commingle the functional, λ-calculus part of a language with the parts that do side effects. It seems they believe in the separation of Church and state."

— Guy Steele

"Okay, I'll just say one more thing: λ-calculus. And if that wets your whistle, you know where to find me. Boop."

- Sheldon Cooper

Some references...

Books:

- *The Lambda Calculus: Its Syntax and Semantics*: Henk Barendregt, 1984.
- *Lambda Calculus with Types*: Henk Barendregt, Wil Dekkers, Richard Statman, 2013.
- *Lectures on the Curry-Howard Isomorphism*: Morten Heine Sørensen, Pawel Urzyczyn, 2006.
- *Proofs and Types*: Jean-Yves Girard, Yves Lafont, Paul Taylor, 1989.
- *Lambda Calculi: A Guide for Computer Scientists*: Chris Hankin, 1994.

Some other references...

Articles:

- *Linear numeral systems*: Ian Mackie, 2018.
- *Parallel Reductions in λ-calculus*: Masako Takahashi, 1995.
- *Perpetual Reductions in λ-Calculus*, Femke van Raamsdonk, Paula Severi, Morten Heine B. Sørensen, Hongwei Xi, 1999.
- *Lambda Calculus Notation with Nameless Dummies*: N. G. De Bruijn, 1972.
- *The Impact of the Lambda Calculus in Logic and Computer Science*: Henk Barendregt, 1997.

Notes:

- *Foundations of Functional Programming*: Larry Paulson.
- *Introduction to Lambda Calculus*: Henk Barendregt and Erik Barendsen, online notes, 2000.

And this leads to...

...THE END!!!

Many thanks to the organisers:
Besik Dundua and Temur Kutsia.