EuroProofNet

$\lambda\Pi$-calculus modulo rewriting

Frédéric Blanqui

Deduc⊢eam

Inria

école normale supérieure paris—saclay

What is the $\lambda\Pi$-calculus modulo rewriting ?

$\lambda\Pi/\mathcal{R} =$

λ	simply-typed λ-calculus
$+\ \Pi$	dependent types, e.g. `Array` *n*
$+\ \mathcal{R}$	identification of types modulo rewrites rules $l \hookrightarrow r$

Outline

Lambda-calculus

Simple types

Dependent types

Rewriting

What is λ-calculus ?

introduced by Alonzo Church in 1932

the (untyped or pure) λ-calculus is a general framework for defining functions (objects or propositions)

initially thought as a possible foundation for logic
but turned out to be inconsistent

it however provided a foundation for computability theory
and functional programming !

What is λ-calculus ?

only 3 constructions:

- **variables** $x, y, \ldots$
- **application** of a term t to another term u, written tu
- **abstraction** over a variable x in a term t, written $\lambda x, t$

example: the function mapping x to $2x + 1$ is written

$$\lambda x, +(*2x)1$$

but, for the sake of readability, we may still use infix notations

α-equivalence

the names of abstracted variables are theoretically not significant:

$\lambda x, +(*2x)1$ denotes the same function as $\lambda y, +(*2y)1$

terms equivalent modulo valid renamings are said α-equivalent

in theory, one usually works modulo α-equivalence, that is, on α-equivalence classes of terms (hence, one can always rename some abstracted variables if it is more convenient)

$\Rightarrow$ but, then, one has to be careful that functions and relations are actually invariant by α-equivalence!...

in practice, dealing with α-equivalence is not trivial

$\Rightarrow$ this gave raise to a lot of research and tools (still nowadays)!

Example: the set of free variables

a variable is free if it is not abstracted

the set $\mathrm{FV}(t)$ of free variables of a term t is defined as follows:

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(tu) = \mathrm{FV}(t) \cup \mathrm{FV}(u)$
- $\mathrm{FV}(\lambda x, t) = \mathrm{FV}(t) - \{x\}$

one can check that FV is invariant by α-equivalence:

$$\text{if } t \equiv_\alpha u \text{ then } \mathrm{FV}(t) = \mathrm{FV}(u)$$

Substitution

a substitution is a finite map from variables to terms

$$\sigma = \{(x_1, t_1), \dots, (x_n, t_n)\}$$

the domain of a substitution σ is

$$\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$$

how to define the result of applying a substitution σ on a term t ?

- $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$
- $x\sigma = x$ if $x \notin \mathrm{dom}(\sigma)$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x, t)\sigma = \lambda x, (t\sigma)$? example: $(\lambda x, y)\{(y, x)\} = \lambda x, x$?

Substitution

a substitution is a finite map from variables to terms

$$\sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\}$$

the domain of a substitution σ is

$$\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$$

how to define the result of applying a substitution σ on a term t ?

- $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$
- $x\sigma = x$ if $x \notin \mathrm{dom}(\sigma)$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x, t)\sigma = \lambda x, (t\sigma)$? example: $(\lambda x, y)\{(y, x)\} = \lambda x, x$?

definition not invariant by α-equivalence ! $\lambda x, y \equiv_\alpha \lambda z, y$

Substitution

in λ-calculus, substitution is not trivial!

we must rename abstracted variables to avoid name clashes:

$$(\lambda x, t)\sigma = \lambda y, (t\sigma')$$

where $\sigma' = \sigma|_{\mathrm{FV}(\lambda x, t)} \cup \{(x, y)\}$

and $y \notin \mathrm{FV}(\lambda x, t) \cup \bigcup \{\mathrm{FV}(z\sigma) \mid z \in \mathrm{dom}(\sigma|_{\mathrm{FV}(\lambda x, t)})\}$

Operational semantics: β-reduction

applying the term $\lambda x, +(*2x)1$ to 3 should return $+(*23)1$

this is the top β-rewrite relation:

$$(\lambda x, t)u \hookrightarrow_\beta^\varepsilon t_x^u$$

the β-rewrite relation $\hookrightarrow_\beta$ is the closure by context of $\hookrightarrow_\beta^\varepsilon$:

$$\frac{t \hookrightarrow_\beta^\varepsilon u}{t \hookrightarrow_\beta u} \quad \frac{t \hookrightarrow_\beta u}{tv \hookrightarrow_\beta uv} \quad \frac{t \hookrightarrow_\beta u}{vt \hookrightarrow_\beta vu} \quad \frac{t \hookrightarrow_\beta u}{\lambda x, t \hookrightarrow_\beta \lambda x, u}$$

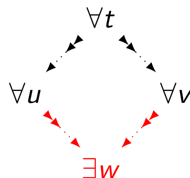
a term is in normal form if it cannot be reduced further

Properties of β-reduction in pure λ-calculus

$\hookrightarrow_\beta$ is **confluent** or has the Church-Rosser property (CR):

if $t \hookrightarrow_\beta^* u$ and $t \hookrightarrow_\beta^* v$
then $u \downarrow_\beta v$

i.e. there is w s.t.
$u \hookrightarrow_\beta^* w$ and $v \hookrightarrow_\beta^* w$



this means that the order of reduction steps does not matter

and every term has <u>at most</u> one normal form

Properties of β-reduction in pure λ-calculus

$\hookrightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \hookrightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

Properties of β-reduction in pure λ-calculus

$\hookrightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \hookrightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t \coloneqq (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \hookrightarrow_\beta tY_t$$

Properties of β-reduction in pure λ-calculus

$\hookrightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \hookrightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t := (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \hookrightarrow_\beta tY_t$$

λ-calculus is Turing-complete/can encode any recursive function

Properties of β-reduction in pure λ-calculus

$\hookrightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \hookrightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t \coloneqq (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \hookrightarrow_\beta tY_t$$

λ-calculus is Turing-complete/can encode any recursive function

a natural number n can be encoded as

$$\lambda f, \lambda x, f^n x$$

where $f^0 x = x$ and $f^{n+1} x = f(f^n x)$

Outline

Lambda-calculus

Simple types

Dependent types

Rewriting

On the origin of type theory

like in unrestricted set theory where every term is a set

in pure λ-calculus, every term is a function

$\Rightarrow$ every term can be applied to another term, including itself!

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx) = Y_\neg$ we have $RR \hookrightarrow_\beta \neg(RR)$

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx) = Y_\neg$ we have $RR \hookrightarrow_\beta \neg(RR)$

proposals to overcome this problem:

- restrict comprehension axiom to already defined sets
  use $\{x \in A \mid P\}$ instead of $\{x \mid P\}$

$\rightsquigarrow$ **modern set theory**

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx) = Y_\neg$ we have $RR \hookrightarrow_\beta \neg(RR)$

proposals to overcome this problem:

- restrict comprehension axiom to already defined sets
  use $\{x \in A \mid P\}$ instead of $\{x \mid P\}$

$\rightsquigarrow$ **modern set theory**

- organize terms into a hierarchy
  - natural numbers are of type ι and propositions of type o
  - unary predicates/sets of natural numbers are of type $\iota \to o$
  - sets of sets of natural numbers are of type $(\iota \to o) \to o$
  - ...

$\rightsquigarrow$ **modern type theory**

Church simply-typed λ-calculus

simple types:

$$A, B \in \mathcal{S} \coloneqq X \in \mathcal{V}_{typ} \mid A \to B$$

- X is a user-defined type variable/constant
- $A \to B$ is the type of functions from A to B

terms:

$$t, u \in \mathcal{T} \coloneqq x \in \mathcal{V}_{obj} \mid \lambda x : A, t \mid tu$$

Assigning types to terms

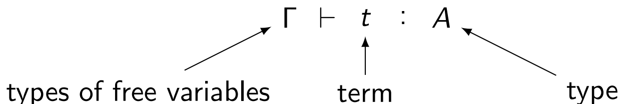
to assign a type to a term, we define a relation

$$\vdash \quad \subseteq \quad (\mathcal{V}_{obj} \stackrel{fin}{\rightarrow} \mathcal{S}) \times \mathcal{T} \times \mathcal{S}$$

where $\mathcal{V}_{obj} \stackrel{fin}{\rightarrow} \mathcal{S}$ is the set of finite maps from variables to types (typing environments) giving the types of free variables

a term t is well-formed in Γ if there is A such that:

$$\Gamma \ \vdash \ t \ : \ A$$



Typing rules for objects

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x : A, t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

Some properties of simply-typed λ-terms

- xx is not typable
- a term has at most one type in a given typing environment
- $\hookrightarrow_\beta$ preserves typing/has the subject-reduction property (SR_β):
  if $\Gamma \vdash t : A$ and $t \hookrightarrow_\beta u$, then $\Gamma \vdash u : A$
- $\hookrightarrow_\beta$ terminates on well-typed terms (SN)
- type-inference $\exists A, \Gamma \vdash t : A$? is decidable
- type-checking $\Gamma \vdash t : A$? is decidable

Outline

Lambda-calculus

Simple types

Dependent types

Rewriting

Dependent types / $\lambda\Pi$-calculus

a dependent type is a type that depends on terms

example: the type ($\texttt{Array}\, n$) of arrays of size n

first introduced by de Bruijn in the Automath system in the 60's

dependent types:

$$A, B \coloneqq X\, t_1 \dots t_n \mid \Pi x\!:\!A, B$$

$A \to B$ is an abbreviation for $\Pi x\!:\!A, B$ when $x \notin \mathrm{FV}(B)$

Example of objects with dependent types

concatenation function on arrays:

$$\texttt{concat} : \Pi p : \mathbb{N}, \texttt{Array}\, p \rightarrow \Pi q : \mathbb{N}, \texttt{Array}\, q \rightarrow \texttt{Array}(p+q)$$

$$\texttt{concat}\, 2\, a\, 3\, b : \texttt{Array}(2+3)$$

Typing rules for objects ?

with simple types

$$\frac{(x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x{:}A\} \vdash t : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}A, t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

with dependent types

$$\frac{(x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x{:}A)\} \vdash t : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}A, t : \Pi x{:}A, B}$$

$$\frac{\Gamma \vdash t : \Pi x{:}A, B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B_x^u}$$

$$\frac{\Gamma \vdash t : A \quad A \downarrow_\beta A' \quad \Gamma \vdash A' : \mathtt{TYPE}}{\Gamma \vdash t : A'}$$

the last rule allows one to identify the types

$$A = \mathtt{Array}((\lambda n : \mathbb{N}, n)3) \quad \text{and} \quad A' = \mathtt{Array}(3)$$

How to make sure that a dependent type is well-formed ?

definition of simply-typed terms:

1. we define types: $A, B \coloneqq X \in \mathcal{V}_{typ} \mid A \to B$
   all types are well-formed by definition
2. we define terms: $t, u \coloneqq x \in \mathcal{V}_{obj} \mid \lambda x\!:\!A, t \mid tu$
3. we define well-formed terms with typing

How to make sure that a dependent type is well-formed ?

definition of simply-typed terms:

1. we define types: $A, B := X \in \mathcal{V}_{typ} \mid A \to B$
   all types are well-formed by definition
2. we define terms: $t, u := x \in \mathcal{V}_{obj} \mid \lambda x : A, t \mid tu$
3. we define well-formed terms with typing

problem with dependent types: types depend on terms

$\Rightarrow$ not all types are well-formed

$\Rightarrow$ we need typing rules for dependent types

What is a simple type ?

a simple type refines the notion of arity:

- it indicates the number of arguments
- but also the type of each argument

instead of saying $+$ takes 2 arguments, we say $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

instead of saying that $2 + 2$ is well-formed, we say $2 + 2 : \mathbb{N}$

What is a simple type ?

a simple type refines the notion of arity:

- it indicates the number of arguments
- but also the type of each argument

instead of saying $+$ takes 2 arguments, we say $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

instead of saying that $2 + 2$ is well-formed, we say $2 + 2 : \mathbb{N}$

$2 + 2 : \mathbb{N}$ also means that $2 + 2$ is an expression of arity 0

types extend arities from function symbols to any expression

we can have partial applications: $2 + _ : \mathbb{N} \to \mathbb{N}$

Arity and typing rules of simple types

we introduce a new constant $\mathtt{TYPE}$ for the arity of simple types

the fact that every simple type is well-formed can be represented by the following typing rules:

$$\frac{X \in \mathcal{V}_{typ}}{\Gamma \vdash X : \mathtt{TYPE}}$$

$$\frac{\Gamma \vdash A : \mathtt{TYPE} \quad \Gamma \vdash B : \mathtt{TYPE}}{\Gamma \vdash A \to B : \mathtt{TYPE}}$$

example: $\mathbb{N} : \mathtt{TYPE} \;\vdash\; \mathbb{N} \to \mathbb{N} : \mathtt{TYPE}$

Arity and typing rules of dependent types

the typing rules for simple types can be easily extended to dependent types as follows:

$$\frac{\Gamma \vdash A : \texttt{TYPE} \quad \Gamma, x : A \vdash B : \texttt{TYPE}}{\Gamma \vdash \Pi x : A, B : \texttt{TYPE}}$$

but what is the arity of `Array` ?

Arity and typing rules of dependent types

the typing rules for simple types can be easily extended to dependent types as follows:

$$\frac{\Gamma \vdash A : \texttt{TYPE} \quad \Gamma, x : A \vdash B : \texttt{TYPE}}{\Gamma \vdash \Pi x : A, B : \texttt{TYPE}}$$

but what is the arity of `Array` ?

`Array` is a function taking a natural number as argument and returning a type: its arity is $\mathbb{N} \to \texttt{TYPE}$

Intermediate summary

we now have 3 sorts of expressions:

- **objects**: 0, $+$, $2+2$, etc.
- **families**, the arities of objects: $\mathbb{N}$, $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, $\texttt{Array}\,3$, $\Pi p : \mathbb{N}, \texttt{Array}\,p$, etc.
- **kinds**, the arities of families: $\texttt{TYPE}$, $\mathbb{N} \to \texttt{TYPE}$, $\texttt{Array}\,3 \to \texttt{TYPE}$, etc.

we have typing rules to make sure that an object is well-formed
we have typing rules to make sure that a family is well-formed

we have no typing rules to make sure that a kind is well-formed
yet a kind may contain families and objects

How to make sure that a kind is well-formed ?

to type families, we introduced the constant TYPE and typing rules on families

to type kinds, we introduce a new constant KIND and the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{TYPE} : \texttt{KIND}}$$

$$\frac{\Gamma \vdash A : \texttt{TYPE} \quad \Gamma, x : A \vdash K : \texttt{KIND}}{\Gamma \vdash \Pi x : A, K : \texttt{KIND}}$$

example: $\mathbb{N} : \texttt{TYPE} \ \vdash \ \mathbb{N} \to \texttt{TYPE} : \texttt{KIND}$

Adding abstractions in families

finally, we can easily add abstractions in families like

$$\lambda n : \mathbb{N}, \mathtt{Array}\, n$$

by adding the following rules:

$$\frac{\Gamma, x\!:\!A \vdash B : K}{\Gamma \vdash \lambda x\!:\!A, B : \Pi x\!:\!A, K}$$

$$\frac{\Gamma \vdash A : K \quad K \downarrow_\beta K' \quad \Gamma \vdash K' : \mathtt{KIND}}{\Gamma \vdash A : K'}$$

the last rule allows one to identify the types

$$K = (\lambda n : \mathbb{N}, \mathtt{Array}\, n) 3 \quad \text{and} \quad K' = \mathtt{Array}\, 3$$

Rules to make sure that a typing environment is well-formed

$\Gamma \vdash$ means that Γ is a well-formed:

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A : \texttt{TYPE} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash K : \texttt{KIND} \quad X \notin \mathrm{dom}(\Gamma)}{\Gamma, X : K \vdash}$$

All rules on one slide

typing environments:

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A : \mathtt{TYPE} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash K : \mathtt{KIND} \quad X \notin \mathrm{dom}(\Gamma)}{\Gamma, X : K \vdash}$$

objects:

$$\frac{\Gamma \vdash \quad (x, A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \quad A \downarrow_\beta A' \quad \Gamma \vdash A' : \mathtt{TYPE}}{\Gamma \vdash t : A'}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B} \qquad \frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B_x^u}$$

families:

$$\frac{\Gamma \vdash \quad (X, K) \in \Gamma}{\Gamma \vdash X : K} \qquad \frac{\Gamma, x : A \vdash B : \mathtt{TYPE}}{\Gamma \vdash \Pi x : A, B : \mathtt{TYPE}}$$

$$\frac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \lambda x : A, B : \Pi x : A, K} \qquad \frac{\Gamma \vdash T : \Pi x : A, K \quad \Gamma \vdash u : B}{\Gamma \vdash Tu : K_x^u}$$

$$\frac{\Gamma \vdash A : K \quad K \downarrow_\beta K' \quad \Gamma \vdash K' : \mathtt{KIND}}{\Gamma \vdash A : K'}$$

kinds:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathtt{TYPE} : \mathtt{KIND}} \qquad \frac{\Gamma, x : A \vdash K : \mathtt{KIND}}{\Gamma \vdash \Pi x : A, K : \mathtt{KIND}}$$

All rules on one slide (PTS presentation)

$$s \in \mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$$

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A : s \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \qquad (x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B} \qquad \frac{\Gamma \vdash t : \Pi x : A, B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B_x^u}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{TYPE} \quad : \texttt{KIND}} \qquad \frac{\Gamma \vdash A : \texttt{TYPE} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s}$$

$$\frac{\Gamma \vdash t : A \qquad A \downarrow_\beta A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash t : A'}$$