



EuroProofNet

$\lambda\Pi$ -calculus modulo rewriting: theory and application to proof systems interoperability

Frédéric Blanqui

DeducTeam

Inria

école —
normale —
supérieure —
paris — saclay —



Course outline

Lecture 1. Introduction to proof systems interoperability; introduction to the $\lambda\Pi$ -calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)
(part 1): λ -calculus, dependent types

Lecture 2. Introduction to $\lambda\Pi/\mathcal{R}$ (part 2): pure type systems, rewriting; introduction to the **Lambdapi proof assistant, practical session on **Lambdapi****
install on your machine Opam (<https://opam.ocaml.org>)
and Lambdapi (<https://github.com/Deducteam/lambdapi>)

Lecture 3. Encoding logics in $\lambda\Pi/\mathcal{R}$: first-order logic, polymorphism, higher-order logic, pure type systems, ...

Lecture 4. Properties of $\lambda\Pi/\mathcal{R}$: decidability of type-checking, subject-reduction, confluence, termination, dependencies between these properties



EuroProofNet

Introduction to Proof Systems Interoperability

Frédéric Blanqui

DeducTeam

Inria

école —
normale —
supérieure —
paris — saclay —



Gilles Dowek (1966-2025)

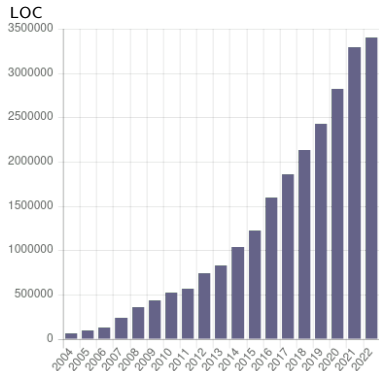
This symposium, and especially this school, is dedicated to the memory of Gilles Dowek who died last July at the age of 58 only, and has been a very important contributor to the development of logical frameworks, and especially the $\lambda\Pi$ -calculus modulo rewriting and its implementation Dedukti.



Libraries of formal proofs today

Library	Nb files	Nb objects*
Rocq Opam	35,000	1,200,000
Isabelle AFP	7,500	280,000
Lean Mathlib	4,200	210,000
Mizar Mathlib	1,400	77,000
HOL-Light	600	35,000
...

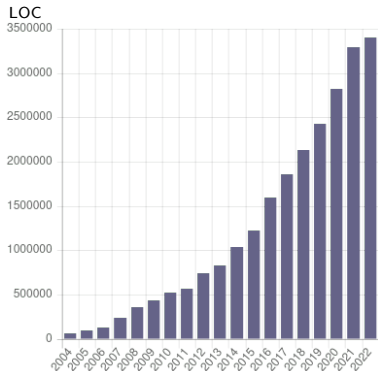
* type, definition, theorem, ... in 2023



Libraries of formal proofs today

Library	Nb files	Nb objects*
Rocq Opam	35,000	1,200,000
Isabelle AFP	7,500	280,000
Lean Mathlib	4,200	210,000
Mizar Mathlib	1,400	77,000
HOL-Light	600	35,000
...

* type, definition, theorem, ... in 2023

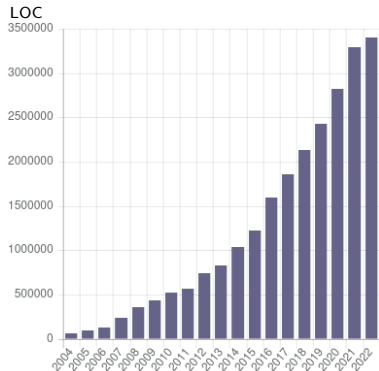


- Every system has basic libraries on integers, lists, ...
- Some definitions/theorems are available in one system only

Libraries of formal proofs today

Library	Nb files	Nb objects*
Rocq Opam	35,000	1,200,000
Isabelle AFP	7,500	280,000
Lean Mathlib	4,200	210,000
Mizar Mathlib	1,400	77,000
HOL-Light	600	35,000
...

* type, definition, theorem, ... in 2023



- Every system has basic libraries on integers, lists, ...
 - Some definitions/theorems are available in one system only
- ⇒ Can't we translate a proof between two systems automatically?

Interest of proof systems interoperability

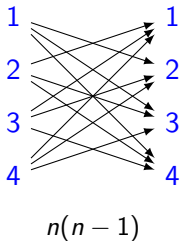
- Avoid duplicating developments and losing time
- Facilitate development of new proof systems
- Increase reliability of formal proofs (cross-checking)
- Facilitate validation by certification authorities
- Relativize the choice of a system (school, industry)
- Provide multi-system data to machine learning

Difficulties of interoperability

- Each system is based on different axioms and deduction rules
- It is usually non trivial and sometimes impossible to translate a proof from one system to the other (e.g. a proof using impredicativity or proof irrelevance in a system not allowing these features)

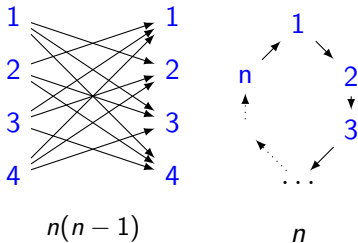
Difficulties of interoperability

- Each system is based on different axioms and deduction rules
- It is usually non trivial and sometimes impossible to translate a proof from one system to the other (e.g. a proof using impredicativity or proof irrelevance in a system not allowing these features)
- Is it reasonable to have $n(n - 1)$ translators for n systems?



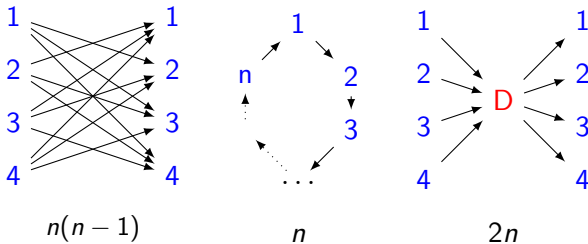
Difficulties of interoperability

- Each system is based on different axioms and deduction rules
- It is usually non trivial and sometimes impossible to translate a proof from one system to the other (e.g. a proof using impredicativity or proof irrelevance in a system not allowing these features)
- Is it reasonable to have $n(n - 1)$ translators for n systems?



Difficulties of interoperability

- Each system is based on different axioms and deduction rules
- It is usually non trivial and sometimes impossible to translate a proof from one system to the other (e.g. a proof using impredicativity or proof irrelevance in a system not allowing these features)
- Is it reasonable to have $n(n - 1)$ translators for n systems?



A common language for proof systems?

Logical framework D

language for describing axioms, deduction rules and proofs of a system S as a theory $D(S)$ in D

Example: $D =$ predicate calculus

allows one to represent S =geometry, S =arithmetic, S =set theory, ...
not well suited for computations and dependent types

A common language for proof systems?

Logical framework D

language for describing axioms, deduction rules and proofs of a system S as a theory $D(S)$ in D

Example: $D =$ predicate calculus

allows one to represent S =geometry, S =arithmetic, S =set theory, ...
not well suited for computations and dependent types

Better: $D = \lambda\Pi$ -calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)

allows one to represent also:

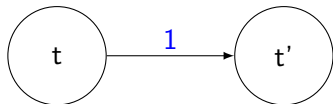
S =HOL, S =Rocq, S =Agda, S =PVS, ...

How to translate a proof $t \in A$ in a proof $u \in B$?

In a logical framework D :

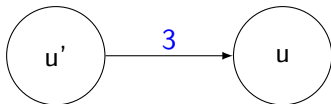
system A

$D(A)$



$D(B)$

system B

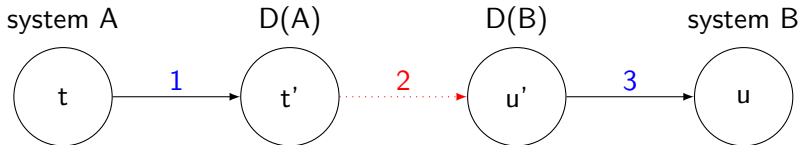


1. translate $t \in A$ in $t' \in D(A)$

3. translate $u' \in D(B)$ in $u \in B$

How to translate a proof $t \in A$ in a proof $u \in B$?

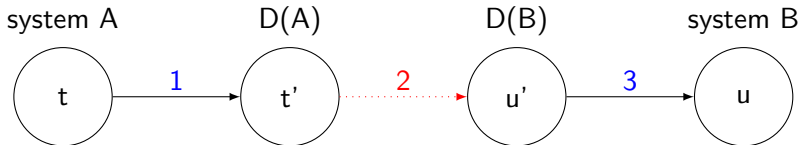
In a logical framework D :



1. translate $t \in A$ in $t' \in D(A)$
2. identify the axioms and deduction rules of A used in t'
translate $t' \in D(A)$ in $u' \in D(B)$ if possible
3. translate $u' \in D(B)$ in $u \in B$

How to translate a proof $t \in A$ in a proof $u \in B$?

In a logical framework D :

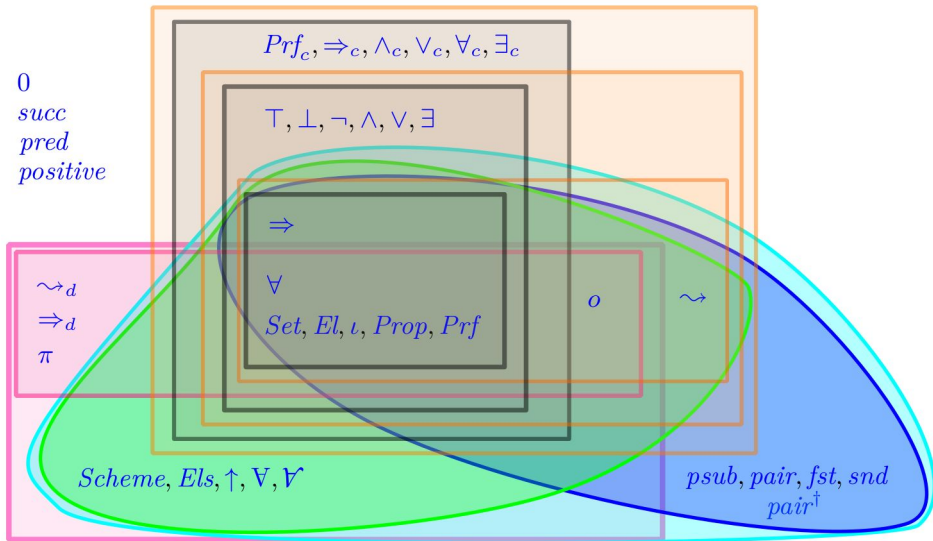


1. translate $t \in A$ in $t' \in D(A)$
2. identify the axioms and deduction rules of A used in t'
translate $t' \in D(A)$ in $u' \in D(B)$ if possible
3. translate $u' \in D(B)$ in $u \in B$

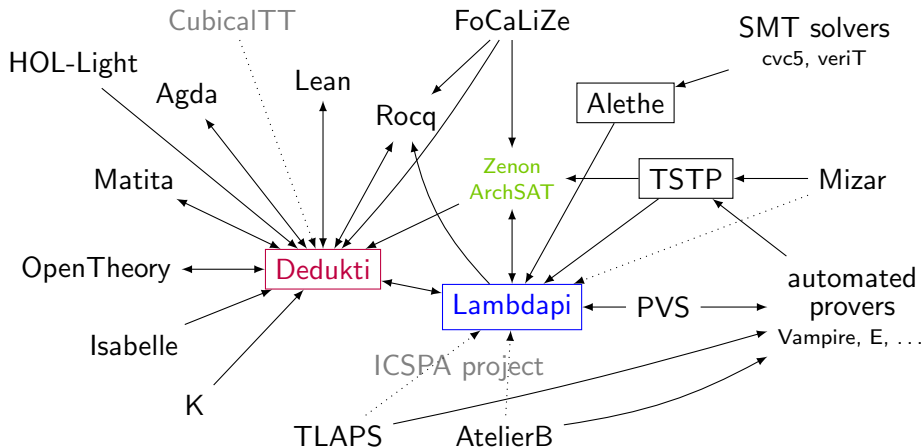
\Rightarrow represent in the same way functionalities common to A and B

The modular $\lambda\Pi/\mathcal{R}$ theory U and its sub-theories

38 symbols, 28 rules, 13 sub-theories



Dedukti, an assembly language for proof systems



Lambdapi = **Dedukti** + implicit arguments/coercions, tactics, ...

<https://github.com/Deducteam/Dedukti>

<https://github.com/Deducteam/lambdapi>

Libraries currently available in Dedukti

System	Libraries
HOL-Light	all libraries
Matita	Arith
Rocq	Stdlib parts, GeoCoq
Isabelle	HOL.Complex_Main
Agda	Stdlib parts ($\pm 25\%$)
PVS	Stdlib parts (statements only)
Lean	Stdlib
TPTP	E 69%, Vampire 83%

Examples of library translations

- [Matita/Arith](#) \longrightarrow [OpenTheory](#), [Rocq](#), [PVS](#), [Lean](#)
<https://logipedia.inria.fr>
- [Matita/Arith](#) \longrightarrow [Agda](#)
https://github.com/Deducteam/matita_lib_in_agda
- [HOL-Light](#) \longrightarrow [Rocq](#)
<https://github.com/Deducteam/coq-hol-light>



EuroProofNet

$\lambda\Pi$ -calculus modulo rewriting

Frédéric Blanqui

Deducteam

Inria

école
normale
supérieure
paris-saclay



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

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

λ

simply-typed λ -calculus

+ Π

dependent types, e.g. `Array n`

+ \mathcal{R}

identification of types modulo rewrites $l \hookrightarrow r$

Outline

Lambda-calculus

Simple types

Dependent types

What is λ -calculus ?

introduced by Alonzo Church in 1932

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

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, \dots
- **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 rise 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 $FV(t)$ of free variables of a term t is defined as follows:

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

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

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

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

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

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

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

Substitution

a substitution is a finite map from variables to terms

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

the domain of a substitution σ is

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

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

- $x\sigma = \sigma(x)$ if $x \in \text{dom}(\sigma)$
- $x\sigma = x$ if $x \notin \text{dom}(\sigma)$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x, t)\sigma = \lambda x, (t\sigma)$? 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 $\sigma' = \sigma|_{\text{FV}(\lambda x, t)} \cup \{(x, y)\}$

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

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_{β} 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

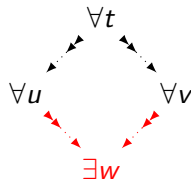
Properties of β -reduction in pure λ -calculus

\hookrightarrow_β 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 at most one normal form

Properties of β -reduction in pure λ -calculus

\hookrightarrow_{β} 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_β does not terminate:

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

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

$$Y_t \hookrightarrow_\beta tY_t$$

Properties of β -reduction in pure λ -calculus

\hookrightarrow_β 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_{β} 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

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

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 $\{x \mid P\}$

\leadsto **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 $\{x \mid P\}$

\leadsto **modern set theory**

- organize terms into a hierarchy

- natural numbers are of type ι and propositions of type o
- sets of natural numbers are of type $\iota \rightarrow o$
- sets of sets of natural numbers are of type $(\iota \rightarrow o) \rightarrow o$
- ...

\leadsto **modern type theory**

Church simply-typed λ -calculus (1940)

simple types:

$$A, B \in \mathcal{S} := X \in \mathcal{V}_{typ} \mid A \rightarrow B$$

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

terms:

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

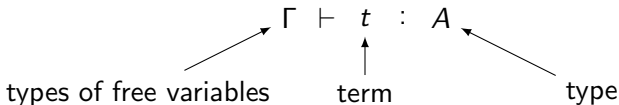
Assigning types to terms

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

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

where $\mathcal{V}_{obj} \xrightarrow{fin} \mathcal{S}$ 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:



Typing rules for objects

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

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

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

Some properties of simply-typed λ -terms

- xx is not typable
- a term has at most one type in a given typing environment
- \hookrightarrow_β 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_β 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

Dependent types / $\lambda\Pi$ -calculus

a dependent type is a type that depends on terms

example: the type `(Array n)` of arrays of size n

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

dependent types:

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

$A \rightarrow B$ is an abbreviation for $\Pi x:A, B$ when $x \notin \text{FV}(B)$

Example of objects with dependent types

concatenation function on arrays:

$$\text{concat} : \prod p : \mathbb{N}, \text{Array } p \rightarrow \prod q : \mathbb{N}, \text{Array } q \rightarrow \text{Array}(p + q)$$
$$\text{concat } 2 \ a \ 3 \ b : \text{Array}(2 + 3)$$

Typing rules for objects ?

with simple types

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

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

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

with dependent types

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

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

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

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

the last rule allows one to identify the types

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

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

definition of simply-typed terms:

1. we define types: $A, B := X \in \mathcal{V}_{typ} \mid A \rightarrow B$
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

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

definition of simply-typed terms:

1. we define types: $A, B := X \in \mathcal{V}_{typ} \mid A \rightarrow B$
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 too

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 $+: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

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 $+: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

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: $2 + _ : \mathbb{N} \rightarrow \mathbb{N}$

Arity and typing rules of simple types

we introduce a new constant `TYPE` for the arity of simple types

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

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

$$\frac{\Gamma \vdash A : \text{TYPE} \quad \Gamma \vdash B : \text{TYPE}}{\Gamma \vdash A \rightarrow B : \text{TYPE}}$$

example: $\mathbb{N} : \text{TYPE} \vdash \mathbb{N} \rightarrow \mathbb{N} : \text{TYPE}$

Arity and typing rules of dependent types

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

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

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:

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

but what is the arity of Array ?

Array is a function mapping natural numbers to types

its arity is $\mathbb{N} \rightarrow \text{TYPE}$

Intermediate summary

we now have 3 sorts of expressions:

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

$$\frac{}{\Gamma \vdash \text{TYPE} : \text{KIND}}$$
$$\frac{\Gamma \vdash A : \text{TYPE} \quad \Gamma, x:A \vdash K : \text{KIND}}{\Gamma \vdash \Pi_{x:A} K : \text{KIND}}$$

example: $\mathbb{N} : \text{TYPE} \vdash \mathbb{N} \rightarrow \text{TYPE} : \text{KIND}$

Adding abstractions in families

finally, we can easily add abstractions in families like

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

by adding the following rules:

$$\frac{\Gamma, x:A \vdash B : K}{\Gamma \vdash \lambda x:A, B : \Pi x:A, K}$$
$$\frac{\Gamma \vdash A : K \quad K \downarrow_{\beta} K' \quad \Gamma \vdash K' : \text{KIND}}{\Gamma \vdash A : K'}$$

the last rule allows one to identify the types

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

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

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

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

All rules on one slide

(Harper, Honsell and Plotkin, 1993)

typing environments:

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

objects:

$$\frac{\Gamma \vdash (x, A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma \vdash t : A \quad A \downarrow_{\beta} A' \quad \Gamma \vdash A' : \text{TYPE}}{\Gamma \vdash t : A'} \\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B} \quad \frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B_x^u}$$

families (arities of objects):

$$\frac{\Gamma \vdash (X, K) \in \Gamma}{\Gamma \vdash X : K} \quad \frac{\Gamma, x : A \vdash B : \text{TYPE}}{\Gamma \vdash \Pi x : A, B : \text{TYPE}} \\ \frac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \lambda x : A, B : \Pi x : A, K} \quad \frac{\Gamma \vdash T : \Pi x : A, K \quad \Gamma \vdash u : A}{\Gamma \vdash Tu : K_x^u} \\ \frac{\Gamma \vdash A : K \quad K \downarrow_{\beta} K' \quad \Gamma \vdash K' : \text{KIND}}{\Gamma \vdash A : K'}$$

kinds (arities of families):

$$\frac{}{\vdash} \quad \frac{}{\vdash} \quad \frac{\Gamma, x : A \vdash K : \text{KIND}}{\vdash}$$