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

Lambdapi,
a proof assistant
featuring rewriting

Frédéric Blanqui

(URLs and purple texts are clickable)

Lambdapi contributors

Work started in 2017 with various contributors over the years:

Alessio Coltellacci (2023-), **Gabriel Hondet** (2019-2022), Ashish Kumar Barnawal (2020-2021), Emilio Gallego (2018-2021), Aurélien Castre (2021), Yann Leray (2021), Diego Riverio (2020), Amélie Ledein (2020), François Lefoulon (2020), Rehan Malak (2019-2020), Yacine El Haddad (2019), Guillaume Genestier (2019), Houda Mouzoun (2019), Aristomenis-Dionysios Papadopoulos (2019), Franck Slama (2019), Jui-Hsuan Wu (2019), Christophe Raffalli (2017-2018), **Rodolphe Lepigre** (2017-2020)

What is Lambdapi?

- a proof assistant
  software to build and check formal proofs (interactively)

What is Lambdapi?

- a proof assistant
  software to build and check formal proofs (interactively)
- a logical framework
  one can define its own logic

What is Lambdapi?

- a proof assistant

  software to build and check formal proofs (interactively)
- a logical framework

  one can define its own logic
- based on the $\lambda\Pi$-calculus modulo rewriting
  - functions are first-class expressions
  - expressions must be well-typed
  - allows dependent types, e.g. array(n)
  - both functions and types can be defined by rewrite rules

What is Lambdapi?

- a proof assistant
  software to build and check formal proofs (interactively)
- a logical framework
  one can define its own logic
- based on the $\lambda\Pi$-calculus modulo rewriting
  - functions are first-class expressions
  - expressions must be well-typed
  - allows dependent types, e.g. array(n)
  - both functions and types can be defined by rewrite rules
- providing tools to check important properties
  - local confluence
  - subject reduction, aka preservation of typing by rewiting

What is Lambdapi?

- a proof assistant
  software to build and check formal proofs (interactively)
- a logical framework
  one can define its own logic
- based on the $\lambda\Pi$-calculus modulo rewriting
  - functions are first-class expressions
  - expressions must be well-typed
  - allows dependent types, e.g. array(n)
  - both functions and types can be defined by rewrite rules
- providing tools to check important properties
  - local confluence
  - subject reduction, aka preservation of typing by rewiting
- and import/export other formats
  - XTC (termination checkers)
  - HRS (confluence checkers)
  - dk (Dedukti)
  - v (Coq)

Outline

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

How to use Lambdapi to rewrite terms and build proofs?

How to check the properties of a $\lambda\Pi/\mathcal{R}$ theory?

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 $l \hookrightarrow r$

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 $l \hookrightarrow r$

terms $t, u =$

TYPE	sort of types
f	global constant
x	local variable
tu	application
$\lambda x : t, u$	abstraction
$\Pi x : t, u$	dependent product
$t \rightarrow u$	abbreviation for $\Pi x : t, u$ when $x \notin u$

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

theory $=$

Σ sequence of type declarations for global constants

$+ \mathcal{R}$ set of rewrite rules $l \hookrightarrow r$

including rules on types!

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

theory $=$

Σ sequence of type declarations for global constants

$+\ \mathcal{R}$ set of rewrite rules $l \hookrightarrow r$

including rules on types!

typing $= \ldots +$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A, B : \textsc{type}}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}$$

Γ: types of local variables

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

$$\frac{\Gamma \vdash t : A \quad A \equiv_{\beta\mathcal{R}} B}{\Gamma \vdash t : B}$$

$\equiv_{\beta\mathcal{R}}$: equational theory generated by β and $\mathcal{R}$

concat : $\Pi p : \mathbb{N}, \text{array } p \to \Pi q : \mathbb{N}, \text{array } q \to \text{array}(p+q)$

concat 2 a 3 b : $\text{array}(2+3) \equiv_{\beta\mathcal{R}} \text{array}(5)$

Hierarchy of terms in $\lambda\Pi/\mathcal{R}$

there is a priori no distinction between terms and types
yet typing rules induce the following hierarchy on terms:

object t	:	**type-family** A	:	**type-arity** K
0	:	$\mathbb{N}$	:	$\mathtt{TYPE}$
s	:	$\mathbb{N} \to \mathbb{N}$	:	$\mathtt{TYPE}$
	:	array	:	$\mathbb{N} \to \mathtt{TYPE}$
empty	:	array 0	:	$\mathtt{TYPE}$

class	**grammar**
type-arities K	$\mathtt{TYPE} \mid \Pi x : A, K$
type-families A	$X \mid At \mid \Pi x : A, A \mid \lambda x : A, A$
objects t	$x \mid tt \mid \lambda x : A, t$

Properties of the $\lambda\Pi$-calculus modulo rewriting

$\lambda\Pi/\mathcal{R}$ **enjoys all the properties of** $\lambda\Pi$**:**

- unicity of types modulo $\equiv_{\beta\mathcal{R}}$
- decidability of $\equiv_{\beta\mathcal{R}}$ and type-checking

assuming that $\hookrightarrow_{\beta\mathcal{R}}$**:**

- terminates: there is no infinite $\hookrightarrow_{\beta\mathcal{R}}$ sequences
- is confluent: the order of $\hookrightarrow_{\beta\mathcal{R}}$ steps does not matter
- $\mathcal{R}$ preserves typing: if $l\theta : A$ and $l \hookrightarrow r \in \mathcal{R}$ then $r\theta : A$

All these properties are undecidable

Fortunately, we have theorems and tools for checking those properties in some cases (see later)

Outline

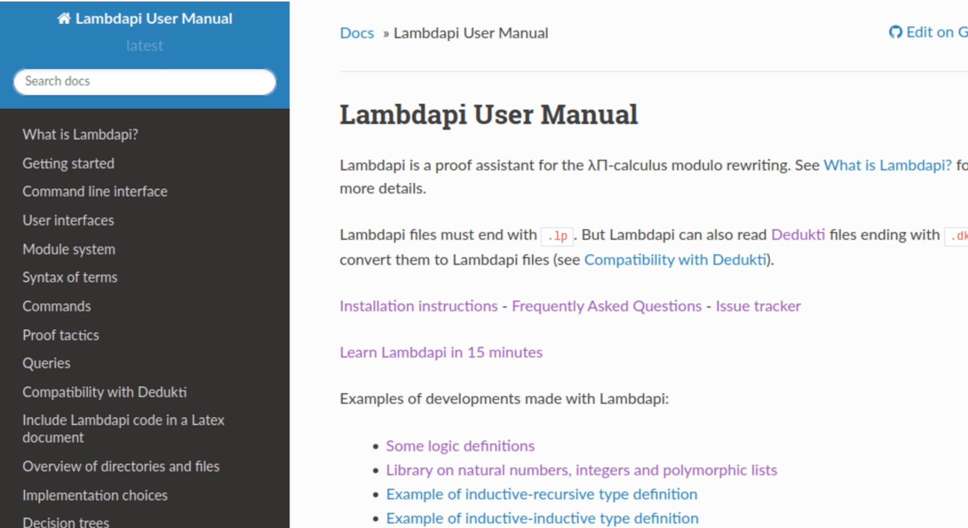
What is the $\lambda\Pi$-calculus modulo rewriting?

How to use Lambdapi to rewrite terms and build proofs?

How to check the properties of a $\lambda\Pi/\mathcal{R}$ theory?

Where to find Lambdapi?

Website: `https://github.com/Deducteam/lambdapi`
Libraries: `https://github.com/Deducteam/opam-lambdapi-repository`
User manual: `https://lambdapi.readthedocs.io/`



Lambdapi User Manual

Lambdapi is a proof assistant for the λΠ-calculus modulo rewriting. See What is Lambdapi? fo more details.

Lambdapi files must end with `.lp`. But Lambdapi can also read Dedukti files ending with `.dk` convert them to Lambdapi files (see Compatibility with Dedukti).

Installation instructions - Frequently Asked Questions - Issue tracker

Learn Lambdapi in 15 minutes

Examples of developments made with Lambdapi:

- Some logic definitions
- Library on natural numbers, integers and polymorphic lists
- Example of inductive-recursive type definition
- Example of inductive-inductive type definition

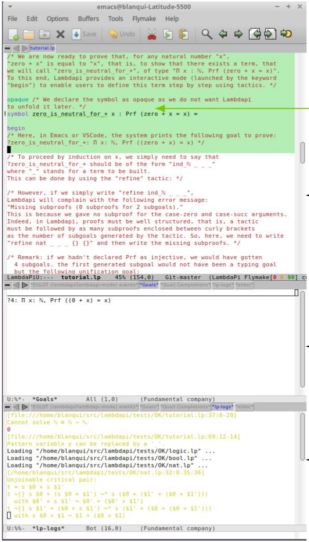
How to use Lambdapi?

- **Batch mode:**

```
lambdapi check file.lp
```

- **Interactive mode** through an editor using a LSP server:
  - **Emacs** (package available on MELPA)
  - **VSCode** (package available on VSCode Marketplace)

Emacs interface



```
/* We are now ready to prove that, for any natural number "x",
"zero + x" is equal to "x", that is, to show that there exists a term, that
we will call "zero_is_neutral_for_+", of type "Π x : ℕ, Prf (zero + x = x)".
To this end, Lambdapi provides an interactive mode (launched by the keyword
"begin") to enable users to define this term step by step using tactics. */

opaque /* We declare the symbol as opaque as we do not want Lambdapi
to unfold it later. */
!symbol zero_is_neutral_for_+ x : Prf (zero + x = x) ≔

begin
/* Here, in Emacs or VSCode, the system prints the following goal to prove:
?zero_is_neutral_for_+: Π x: ℕ, Prf ((zero + x) = x) */

/* To proceed by induction on x, we simply need to say that
?zero_is_neutral_for_+ should be of the form "ind_ℕ _ _ _"
where "_" stands for a term to be built.
This can be done by using the "refine" tactic: */

/* However, if we simply write "refine ind_ℕ _ _ _",
Lambdapi will complain with the following error message:
"Missing subproofs (0 subproofs for 2 subgoals)."
This is because we gave no subproof for the case-zero and case-succ arguments.
Indeed, in Lambdapi, proofs must be well structured, that is, a tactic
must be followed by as many subproofs enclosed between curly brackets
as the number of subgoals generated by the tactic. So, here, we need to write
"refine nat _ _ _ {} {}" and then write the missing subproofs. */

/* Remark: if we hadn't declared Prf as injective, we would have gotten
  4 subgoals. the first generated subgoal would not have been a typing goal
  but the following unification goal:
LambdaPiU:---  tutorial.lp    45% (154,0)    Git-master  (LambdaPi Flymake[0 0 90] c
```

```
?4: Π x: ℕ, Prf ((0 + x) = x)






U:%*-  *Goals*       All (1,0)      (Fundamental company)
```

```
[file:///home/blanqui/src/lambdapi/tests/OK/tutorial.lp:37:0-20]
Cannot solve ℕ ≡ ℕ → ℕ.
0
[file:///home/blanqui/src/lambdapi/tests/OK/tutorial.lp:69:12-14]
Pattern variable y can be replaced by a '_'.
Loading "/home/blanqui/src/lambdapi/tests/OK/logic.lp" ...
Loading "/home/blanqui/src/lambdapi/tests/OK/bool.lp" ...
Loading "/home/blanqui/src/lambdapi/tests/OK/nat.lp" ...
[/home/blanqui/src/lambdapi/tests/OK/nat.lp:32:0-35:36]
Unjoinable critical pair:
t ≔ s $0 × s $1'
t ↪[] s $0 + (s $0 × $1') ↪* s ($0 + ($1' + ($0 × $1')))
  with $0' × s $1' ↪ $0' + ($0' × $1')
t ↪[] s $1' + ($0 × s $1') ↪* s ($1' + ($0 + ($0 × $1')))
  with s $0 × $1 ↪ $1 + ($0 × $1)
U:%%-  *lp-logs*     Bot (16,0)     (Fundamental company)
```

checked part

edition buffer

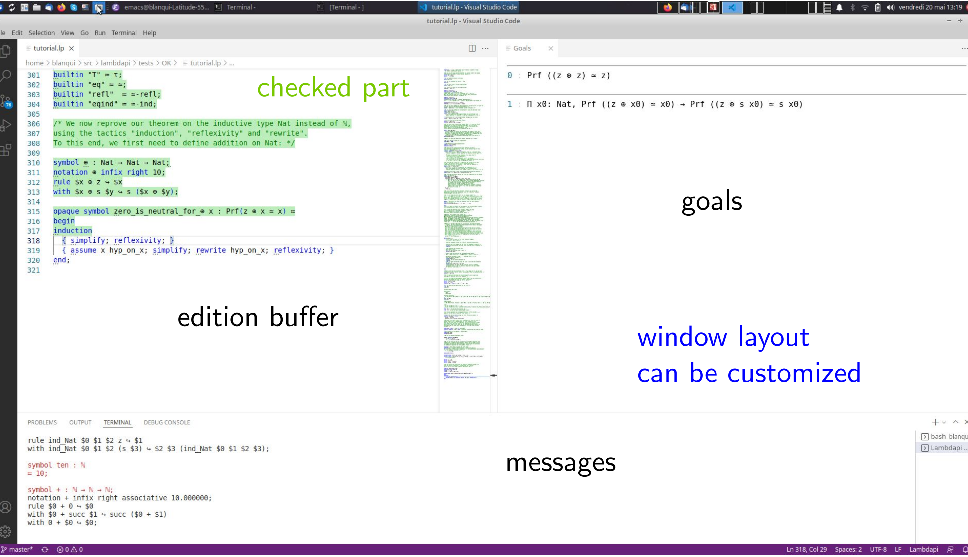
goals

messages

window layout
can be customized

shortcuts: `https://lambdapi.readthedocs.io/en/latest/emacs.html`

VSCode interface



```
builtin "T" ≔ τ;
builtin "eq" ≔ ≃;
builtin "refl"  ≔ ≃-refl;
builtin "eqind" ≔ ≃-ind;

/* We now reprove our theorem on the inductive type Nat instead of ℕ,
using the tactics "induction", "reflexivity" and "rewrite".
To this end, we first need to define addition on Nat: */

symbol ⊕ : Nat → Nat → Nat;
notation ⊕ infix right 10;
rule $x ⊕ z ↪ $x
with $x ⊕ s $y ↪ s ($x ⊕ $y);

opaque symbol zero_is_neutral_for_⊕ x : Prf(z ⊕ x ≃ x) ≔
begin
induction
  { simplify; reflexivity; }
  { assume x hyp_on_x; simplify; rewrite hyp_on_x; reflexivity; }
end;
```

```
0 : Prf ((z ⊕ z) ≃ z)

1 : Π x0: Nat, Prf ((z ⊕ x0) ≃ x0) → Prf ((z ⊕ s x0) ≃ s x0)
```

```
rule ind_Nat $0 $1 $2 z ↪ $1
with ind_Nat $0 $1 $2 (s $3) ↪ $2 $3 (ind_Nat $0 $1 $2 $3);

symbol ten : ℕ
≔ 10;

symbol + : ℕ → ℕ → ℕ;
notation + infix right associative 10.000000;
rule $0 + 0 ↪ $0
with $0 + succ $1 ↪ succ ($0 + $1)
with 0 + $0 ↪ $0;
```

checked part

goals

edition buffer

window layout
can be customized

messages

shortcuts: `https://lambdapi.readthedocs.io/en/latest/vscode.html`

Lambdapi syntax

file extension: `.lp`

BNF grammar:

https://raw.githubusercontent.com/Deducteam/lambdapi/master/doc/lambdapi.bnf

comments: `/* ... /*... */... */` or `// ...`

identifiers: UTF16 characters and `{|` arbitrary string `|}`

commands for defining a $\lambda\Pi/\mathcal{R}$ theory:

- `symbol` for declaring/defining a symbol
- `rule` for adding a (set of) rewrite rules

Syntax of terms

`TYPE`	sort for types
(*id* .)**id*	variable or constant
term *term* . . . *term*	application
λ *id* [: *term*] , *term*	abstraction
Π *id* [: *term*] , *term*	dependent product
term $\to$ *term*	non-dependent product
_	unknown term
`let` *id* [: *term*] $\coloneqq$ *term* `in` *term*	
(*term*)	

Command for declaring/defining a symbol

$$\textit{modifier}^{*}\ \texttt{symbol}\ \textit{id}\ \textit{param}^{*}\ [\texttt{:}\ \textit{term}\ ]\ [\texttt{:=}\ \textit{term}\ ]\ [\texttt{begin}\ \textit{proof}\ \texttt{end}]\ \texttt{;}$$

$$\textit{param} = \textit{id} \mid \texttt{\_} \mid \texttt{(}\ \textit{id}^{+}\ \texttt{:}\ \textit{term}\ \texttt{)} \mid \underset{\textit{implicit parameters}}{\texttt{[}\ \textit{id}^{+}\ \texttt{:}\ \textit{term}\ \texttt{]}}$$

```
symbol N : TYPE;
symbol 0 : N;
symbol s : N → N;
symbol + : N → N → N; notation + infix right 10;
symbol × : N → N → N; notation × infix right 20;
```

Symbol modifiers

- `constant`: not definable
- `opaque`: never unfolded
- `associative`
- `commutative`
- `private`: not exported
- `protected`: exported but usable in rule left-hand sides only
- `sequential`: reduction strategy
- `injective`: unification hint

Handling of C/AC symbols in Lambdapi

When a symbol is declared C/AC, Lambdapi implicitly put terms in some canonical form wrt C/AC

On the implementation of construction functions for non-free concrete data types, ESOP 2007, with Thérèse Hardin, Pierre Weis

This is sufficient to handle simple functions without using matching modulo AC

Command for adding rewrite rules

$$\texttt{rule}\ \textit{term} \hookrightarrow \textit{term}\ \big(\texttt{with}\ \textit{term} \hookrightarrow \textit{term}\ \big)^{*}\ ;$$

---

pattern variables must be prefixed by `$`:

```
rule $x + 0 ↪ $x
with $x + s $y ↪ s ($x + $y);
```

Lambdapi tries to automatically check:

- **local confluence** (AC symbols/HO patterns not handled yet)
- **preservation of typing** (aka subject reduction)

Rules accepted by Lambdapi

overlapping rules

```
rule $x + 0 ↪ $x
with $x + s $y ↪ s ($x + $y)
with 0 + $x ↪ $x
with s $x + $y ↪ s ($x + $y);
```

matching on defined symbols

```
rule ($x + $y) + $z ↪ $x + ($y + $z);
```

non-linear patterns

```
rule $x - $x ↪ 0;
```

higher-order patterns

```
symbol R:TYPE; symbol 0:R; symbol sin:R → R;
symbol cos:R → R; symbol D:(R → R) → (R → R);

rule D (λ x, sin $F.[x]) ↪ λ x, D $F.[x] × cos $F.[x];
rule D (λ x, $V.[]) ↪ λ x, 0;
```

Example: decision procedure for group theory

```
symbol G : TYPE;
symbol 1 : G;
symbol · : G → G → G; notation · infix 10;
symbol inv : G → G;

rule ($x · $y) · $z ↪ $x · ($y · $z)
with 1 · $x ↪ $x
with $x · 1 ↪ $x
with inv $x · $x ↪ 1
with $x · inv $x ↪ 1
with inv $x · ($x · $y) ↪ $y
with $x · (inv $x · $y) ↪ $y
with inv 1 ↪ 1
with inv (inv $x) ↪ $x
with inv ($x · $y) ↪ inv $y · inv $x;
```

Rewrite engine implementation

The new rewriting engine of Dedukti
Gabriel Hondet and Frédéric Blanqui, **FSCD 2020**

extension of Luc Maranget's decision trees for OCaml
to higher-order and non-linear patterns

Queries and assertions

`print` *id* ;
`type` *term* ;
`compute` *term* ;
(`assert` | `assertnot`) *id* * ⊢ *term* (: | ≡) *term* ;

```
print N; // constructors and induction principle
print +; // type and rules

type ×;
compute 2 × 5;

assert 0 : N;
assertnot 0 : N → N;

assert x y z ⊢ x + y × z ≡ x + (y × z);
assertnot x y z ⊢ x + y × z ≡ (x + y) × z;
```

How to use Lambdapi to check proofs?

By reducing proof-checking to type-checking:

```
// type of propositions
symbol Prop : TYPE;
... // constructors of Prop (connectives, quantifiers)

// interpretation of propositions as types
// (Curry-Howard isomorphism)
symbol Prf : Prop → TYPE;
... // rules defining Prf
```

Proving P:Prop now reduces to finding a term of type Prf(P)

Stating an axiom vs Proving a theorem

Stating an axiom: symbol declaration

```
symbol 0_is_neutral_for_+ x : Prf (0 + x = x);
// no definition given now
// one can still be given later with a rule
```

Proving a theorem: symbol definition

```
opaque symbol 0_is_neutral_for_+ x : Prf (0 + x = x) ≔
// generates the typing goal Prf (0 + x = x)
// a proof must be given now
begin
  ... // proof script
end;
```

Goals and proofs

symbol declarations/definitions may generate:

- **typing goals** $\boxed{x_1 : A_1, \ldots, x_n : A_n \vdash \; ? : B}$

  we have to find a term ? of type B assuming $x_1 : A_1, \ldots, x_n : A_n$

- **unification goals** $\boxed{x_1 : A_1, \ldots, x_n : A_n \vdash t \equiv u}$

  we have to prove that $t \equiv_{\beta\mathcal{R}} u$ assuming $x_1 : A_1, \ldots, x_n : A_n$

these goals can be solved by writing *proof*'s:

$$\begin{aligned} \mathit{proof} &::= (\mathit{proof\_step}\ \texttt{;})^* \\ \mathit{proof\_step} &::= \mathit{tactic}\ (\texttt{\{}\ \mathit{proof}\ \texttt{\}})^* \end{aligned}$$

- a *proof* is a `;`-separated sequence of *proof_step*'s
- a *proof_step* is a *tactic* followed by as many *proof*'s enclosed in curly braces as the number of goals generated by the *tactic*

Example of proof

https://raw.githubusercontent.com/Deducteam/lambdapi/master/tests/OK/tutorial.lp

```
opaque symbol 0_is_neutral_for_+ x : Prf(0 + x = x) ≔
begin
  induction
    {simplify; reflexivity}
    {assume x h; simplify; rewrite h; reflexivity}
end;
```

Tactics

- `solve` for unification goals, applied automatically
- `simplify` [*id*]
- `refine` *term*
- `assume` id^+
- `generalize` *id*
- `apply` *term*
- `induction`
- `have` *id* `:` *term*
- `reflexivity`
- `symmetry`
- `rewrite` [`right`] [*pattern*] *term* like Coq SSReflect
- `why3` call external provers

Using Lambdapi as logical framework

Lambdapi does not come with a pre-defined logic
One has to define its own axioms and deduction rules:

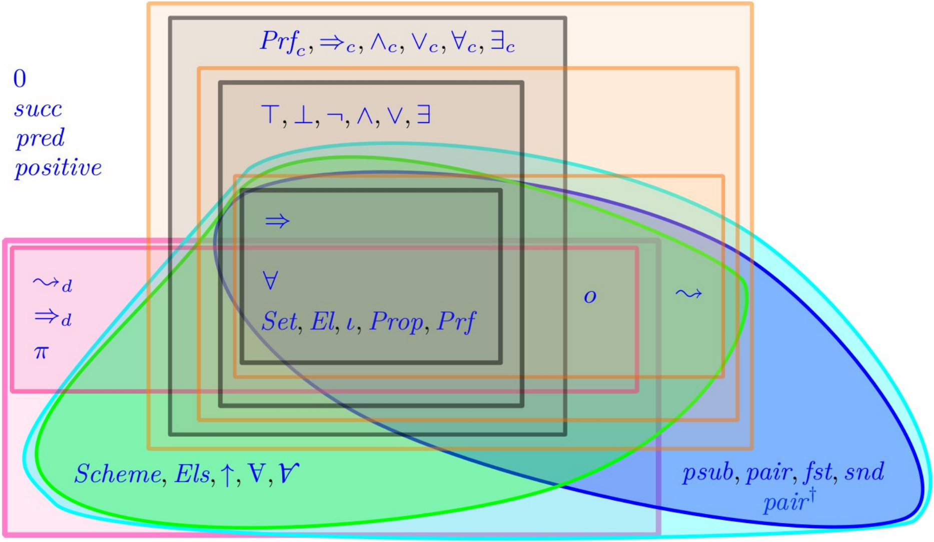
A modular construction of type theories
Frédéric Blanqui, Gilles Dowek, Emilie Grienenberger, Gabriel Hondet, François Thiré, **FSCD 2021** and **LMCS 19(1), 2023**

Definiton of a $\lambda\Pi/\mathcal{R}$ theory U whose sub-theories correspond to many known logic systems from first-order logic, to higher-order logic and the calculus of constructions

Repository of logics defined in Lambdapi: TFF, U, PTS, etc.

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

38 symbols, 28 rules, 13 sub-theories



$Prf_c, \Rightarrow_c, \wedge_c, \vee_c, \forall_c, \exists_c$

0
$succ$
$pred$
$positive$

$\top, \bot, \neg, \wedge, \vee, \exists$

$\Rightarrow$

$\leadsto_d$
$\Rightarrow_d$
π

$\forall$

o

$\leadsto$

$Set, El, \iota, Prop, Prf$

$Scheme, Els, \uparrow, \forall, \not\forall$

$psub, pair, fst, snd$
$pair^{\dagger}$

Beyond U: type systems with universe polymorphism

Some systems like Agda, Coq or Lean use an infinite hierarchy of universes (= inaccessible cardinals in set theory)

Predicative universe levels are expressed in the max-suc algebra with the symbols 0, successor and max interpreted in $\mathbb{N}$

This can be also be handled in Lambdapi:

Encoding type universes without using matching modulo AC
FSCD 2022, using a specific ordering for AC-canonical forms

Outline

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

How to use Lambdapi to rewrite terms and build proofs?

How to check the properties of a $\lambda\Pi/\mathcal{R}$ theory?

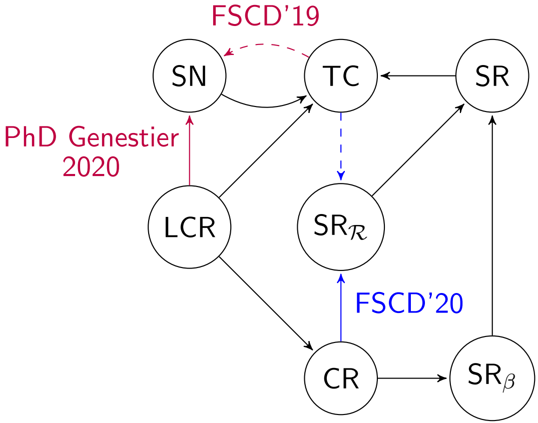
Required properties

TC	decidability of the typing relation
SN	termination of $\rightarrow_{\beta\mathcal{R}}$ from typable terms
SR_β	preservation of typing by $\rightarrow_\beta$
$\mathrm{SR}_\mathcal{R}$	preservation of typing by $\rightarrow_\mathcal{R}$
LCR	local confluence of $\rightarrow_{\beta\mathcal{R}}$ on arbitrary terms
CR	confluence of $\rightarrow_{\beta\mathcal{R}}$ from typable terms

What are the dependencies between those properties ?

For more details, see the slides and video of my talk at IWC 2020!

Dependencies between properties



$\dashrightarrow$ for dependency wrt a strict subset of $\mathcal{R}$

FSCD'19: Dependency Pairs in Dependent Type Theory Modulo

FSCD'20: Type Safety of Rewrite Rules in Dependent Types

Which tools can be used to check confluence automatically?

Lambdapi can export user-defined rewrite rules to the HRS format used in the confluence competition but, in this format:

- terms must be simply-typed
- rewriting is modulo $\beta\eta$
- rewrite rules must be of base type

We therefore need to encode $\lambda\Pi/\mathcal{R}$-terms into the following HRS signature for untyped λ-calculus:

- $A : t \to t \to t$ for application
- $L : t \to (t \to t) \to t$ for λ
- $P : t \to (t \to t) \to t$ for Π
- $A(L(x), y) \hookrightarrow x\,y$ for β-reduction

Available tools: CSIˆho (not developed anymore), SOL

Which tools can be used to check termination automatically?

- Lambdapi can export user-defined rewrite rules to the XTC format used in the termination competition but:
  - XTC does not support dependent types
  - the termination of $\mathcal{R}(\cup\beta)$ on simply-typed terms may not imply the termination of $\mathcal{R}\cup\beta$ on well-typed $\lambda\Pi/\mathcal{R}$ terms

- SizeChangeTool (Genestier, 2020) accepts input problems in the Dedukti format and in an extension of the XTC format allowing dependent types but:
  - requires local confluence (LCR)

How to check local confluence incrementally?

To provide a useful feedback to users,
Lambdapi checks LCR each time a set of rules is added

Problem: assuming that R is LCR,
what do we need to do to check that $R \cup S$ is LCR too?

How to check local confluence incrementally?

A system R is LCR if every critical pair of R is joinable

The set of critical pairs of R is $CP(R) = CP^*(R, R)$ where:

- $CP^*(R, S) = \bigcup\{CP^*(l \to r, g \to d) \mid l \to r \in R, g \to d \in S\}$
- $CP^*(l \to r, g \to d) = \bigcup\{CP(l \to r, p, g \to d) \mid p \in FPos(l)\}$
- $CP(l \to r, p, g \to d) = \{(r\sigma, l[d]_p\sigma) \mid \sigma = mgu(l|_p, g)\}$

So we have:

$$\boxed{CP(R \cup S) = CP(R) \cup CP^*(R, S) \cup CP^*(S, R \cup S)}$$

Remarks:

- S is usually small wrt R
- $CP(R)$ does not need to be computed and checked again
- The set $\{(l, r, p, l|_p) \mid l \to r \in R, p \in FPos(l)\}$ can be computed and recorded once to later check $CP^*(R, S)$ quickly

How to check subject reduction automatically?

$SR(l \hookrightarrow r)$: $\boxed{\forall \Gamma, \sigma, A, \quad \Gamma \vdash l\sigma : A \quad \Rightarrow \quad \Gamma \vdash r\sigma : A}$

- compute the equations $\mathcal{E}$ that must be satisfied for having $l : X$
- simplify $\mathcal{E}$ using confluence and injectivity hints
- turn $\mathcal{E}$ into a convergent system $\mathcal{S}$ using Knuth-Bendix
- check that $r : X$ holds in $\lambda\Pi/(\mathcal{R} + \mathcal{S})$

For more details, see my slides and video at FSCD'20!

Conclusion

Lambdapi is a recent system offering unique features

Remarks and contributions are very welcome!

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