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:

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 rewriting
- and import/export other formats
  - XTC (termination checkers)
  - HRS (confluence checkers)
  - dk (Dedukti)
  - v (Coq)
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 rewriting
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 rewriting
- 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 $\lambda\Pi/R$ theory?
What is the $\lambda\Pi$-calculus modulo rewriting?

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

- $\lambda$ simply-typed $\lambda$-calculus
- $\Pi$ dependent types, e.g. $\text{array}(n)$
- $\mathcal{R}$ identification of types modulo rewrites rules $l \leftrightarrow r$
What is the $\lambda\Pi$-calculus modulo rewriting?

$\lambda\Pi/R =$

$\lambda$ simply-typed $\lambda$-calculus

$+ \Pi$ dependent types, e.g. array($n$)

$+ R$ identification of types modulo rewrites rules $l \leftrightarrow r$

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

$\Rightarrow u$ abbreviation for $\Pi x : t, u$ when $x \notin u$
What is the $\lambda\Pi$-calculus modulo rewriting?

theory =

$\Sigma$ sequence of type declarations for global constants

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

including rules on types!
What is the $\lambda\Pi$-calculus modulo rewriting?

**theory** =

- $\Sigma$ sequence of type declarations for global constants
- $R$ set of rewrite rules $l \leftrightarrow r$
  - including rules on types!

**typing** = . . . +

\[
\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A, B : \text{TYPE} \\
\Gamma \vdash \lambda x : A, t : \Pi x : A, B
\]

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

\[
\Gamma \vdash t : A \quad A \equiv_{\beta R} B \\
\Gamma \vdash t : B
\]

≡$\beta R$: equational theory generated by $\beta$ and $R$

concat : $\Pi p : \mathbb{N}, \text{array} p \rightarrow \Pi q : \mathbb{N}, \text{array} q \rightarrow \text{array}(p + q)$
concat $2$ $a$ $3$ $b : \text{array}(2 + 3) \equiv_{\beta R} \text{array}(5)$
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:

<table>
<thead>
<tr>
<th><strong>object</strong> $t$</th>
<th><strong>type-family</strong> $A$</th>
<th><strong>type-arity</strong> $K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$\mathbb{N}$</td>
<td>TYPE</td>
</tr>
<tr>
<td>$s$</td>
<td>$\mathbb{N} \to \mathbb{N}$</td>
<td>TYPE</td>
</tr>
<tr>
<td></td>
<td>array</td>
<td>$\mathbb{N} \to$ TYPE</td>
</tr>
<tr>
<td>empty</td>
<td>array $0$</td>
<td>TYPE</td>
</tr>
</tbody>
</table>

**class** | **grammar**

| type-arities $K$ | TYPE, $\Pi x : A, K$ |
| type-families $A$ | $X$, $At$, $\Pi x : A, A$, $\lambda x : A, A$ |
| objects $t$ | $x$, $tt$, $\lambda x : A, t$ |
Properties of the $\lambda\Pi$-calculus modulo rewriting

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

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

assuming that $\ ightarrow_{\beta\cal{R}}$:

- terminates: there is no infinite $\ ightarrow_{\beta\cal{R}}$ sequences
- is confluent: the order of $\ ightarrow_{\beta\cal{R}}$ steps does not matter
- $\cal{R}$ preserves typing: if $l\theta : A$ and $l \rightarrow r \in \cal{R}$ 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 $\lambda\Pi/R$ 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/
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

- checked part
- edition buffer
- goals
- messages
- window layout can be customized

VSCode interface

checked part

goals

edition buffer

can be customized

window layout

messages

file extension: .lp

BNF grammar:

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
(id .)* id

term term ... term

λ id [ : term ] , term

Π id [ : term ] , term

term → term

_ _

let id [ : term ] := term in term

( term )

sort for types
variable or constant
application
abstraction
dependent product
non-dependent product
unknown term
Command for declaring/defining a symbol

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

\[
\text{param} = id \mid \_ \mid (id \ + : \ \text{term}) \mid [id \ + : \ \text{term}]
\]

\text{symbol} \ N : \ \text{TYPE};
\text{symbol} \ 0 : N;
\text{symbol} \ s : N \rightarrow N;
\text{symbol} \ + : N \rightarrow N \rightarrow N; \ \text{notation} \ + \ \text{infix right 10};
\text{symbol} \ \times : N \rightarrow N \rightarrow N; \ \text{notation} \ \times \ \text{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
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

```
rule term → term (with term → term )* ;
```

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)
overlapping rules

\begin{verbatim}
rule $x + 0 \rightarrow x$
with $x + s \ y \rightarrow s \ (x + y)$
with $0 + \ x \rightarrow \ x$
with $s \ x + \ y \rightarrow s \ (x + y)$;
\end{verbatim}

matching on defined symbols

\begin{verbatim}
rule (x + y) + z \rightarrow x + (y + z);
\end{verbatim}

non-linear patterns

\begin{verbatim}
rule $x - x \rightarrow 0$;
\end{verbatim}

higher-order patterns

\begin{verbatim}
symbol R:TYPE; symbol 0:R; symbol sin:R \rightarrow R;
symbol cos:R \rightarrow R; symbol D:(R \rightarrow R) \rightarrow (R \rightarrow R);

rule D (\lambda x, \sin \ F .[x]) \rightarrow \lambda x, D \ F .[x] \times \cos \ F .[x];
rule D (\lambda x, \ V .[]) \rightarrow \lambda x, 0;
\end{verbatim}
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;
```
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

\texttt{\textbf{print} \textit{id} ;}
\texttt{\textbf{type} \textit{term} ;}
\texttt{\textbf{compute} \textit{term} ;}
\texttt{(\textbf{assert} \mid \textbf{assertnot}) \textit{id} \ast \vdash \textit{term} (\vdash \equiv) \textit{term} ;}

\texttt{\textbf{print} \textit{N}; \quad \textit{\textit{// constructors and induction principle}}}
\texttt{\textbf{print} \textbf{+}; \quad \textit{\textit{// type and rules}}}

\texttt{\textbf{type} \times ;}
\texttt{\textbf{compute} \textbf{2} \times \textbf{5} ;}

\texttt{\textbf{assert} \textbf{0} : \textit{N} ;}
\texttt{\textbf{assertnot} \textbf{0} : \textit{N} \rightarrow \textit{N} ;}

\texttt{\textbf{assert} \textbf{x} \textbf{y} \textbf{z} \vdash \textbf{x} + \textbf{y} \times \textbf{z} \equiv \textbf{x} + (\textbf{y} \times \textbf{z}) ;}
\texttt{\textbf{assertnot} \textbf{x} \textbf{y} \textbf{z} \vdash \textbf{x} + \textbf{y} \times \textbf{z} \equiv (\textbf{x} + \textbf{y}) \times \textbf{z} ;}
By reducing proof-checking to type-checking:

```plaintext
// 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

```haskell
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

```haskell
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**

\[ x_1 : A_1, \ldots, x_n : A_n \vdash ? : B \]

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

▶ **unification goals**

\[ x_1 : A_1, \ldots, x_n : A_n \vdash t \equiv u \]

we have to prove that \( t \equiv_{\beta \eta} u \) assuming \( x_1 : A_1, \ldots, x_n : A_n \)

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

\[
\text{proof ::= (proof_step ;)*} \\
\text{proof_step ::= tactic \{ proof \})*}
\]

▶ 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

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 \textit{term}
- assume \textit{id}^+
- generalize \textit{id}
- apply \textit{term}
- induction
- have \textit{id} : \textit{term}
- reflexivity
- symmetry
- rewrite \textit{right} \textit{pattern} \textit{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

Definition of a $\lambda\Pi/R$ 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 $\lambda \Pi/R$ theory $U$ and its sub-theories

38 symbols, 28 rules, 13 sub-theories
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

<table>
<thead>
<tr>
<th>TC</th>
<th>decidability of the typing relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN</td>
<td>termination of $\rightarrow_{\beta R}$ from typable terms</td>
</tr>
<tr>
<td>$\text{SR}_{\beta}$</td>
<td>preservation of typing by $\rightarrow_{\beta}$</td>
</tr>
<tr>
<td>$\text{SR}_{R}$</td>
<td>preservation of typing by $\rightarrow_{R}$</td>
</tr>
<tr>
<td>LCR</td>
<td>local confluence of $\rightarrow_{\beta R}$ on arbitrary terms</td>
</tr>
<tr>
<td>CR</td>
<td>confluence of $\rightarrow_{\beta R}$ from typable terms</td>
</tr>
</tbody>
</table>

What are the dependencies between those properties?

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

- - → 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 $\lambda\Pi/R$-terms into the following HRS signature for untyped $\lambda$-calculus:

- $A : t \to t \to t$ for application
- $L : t \to (t \to t) \to t$ for $\lambda$
- $P : t \to (t \to t) \to t$ for $\Pi$
- $A(L(x), y) \leftrightarrow x y$ for $\beta$-reduction

Available tools: CSI^ho (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:

$\begin{align*}
\&CP^*(R, S) = \bigcup \{ CP^*(l \rightarrow r, g \rightarrow d) \mid l \rightarrow r \in R, g \rightarrow d \in S \} \\
\&CP^*(l \rightarrow r, g \rightarrow d) = \bigcup \{ CP(l \rightarrow r, p, g \rightarrow d) \mid p \in FPpos(l) \} \\
\&CP(l \rightarrow r, p, g \rightarrow d) = \{ (r\sigma, l[d]_p\sigma) \mid \sigma = mgu(l|_p, g) \}
\end{align*}$

So we have:

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

Remarks:

$\begin{align*}
\&S \text{ is usually small wrt } R \\
\&CP(R) \text{ does not need to be computed and checked again} \\
\&\text{The set } \{ (l, r, p, l|_p) \mid l \rightarrow r \in R, p \in FPpos(l) \} \text{ can be computed and recorded once to later check } CP^*(R, S) \text{ quickly}
\end{align*}$
How to check subject reduction automatically?

\[ SR(l \rightarrow r) : \forall \Gamma, \sigma, A, \quad \Gamma \vdash l\sigma : A \Rightarrow \Gamma \vdash r\sigma : A \]

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

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/