EuroProofNet

λΠ-calculus modulo rewriting

Frédéric Blanqui

Deducteam

$\lambda\Pi$-calculus (Harper, Honsell and Plotkin, 1993)

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 (arities of objects):

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

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

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

kinds (arities of families):

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

$\lambda\Pi$-calculus (PTS presentation, Berardi, Terlouw, 1989)

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

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

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

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

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

Pure Type Systems (PTS, Berardi, Terlouw, 1989)

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

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

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

$$\frac{\Gamma \vdash}{\Gamma \vdash \cancel{\mathtt{TYPE}}\, s_1 : \cancel{\mathtt{KIND}}\, s_2} \qquad \frac{\Gamma \vdash A : \cancel{\mathtt{TYPE}}\, s_1 \qquad \Gamma, x : A \vdash B : \cancel{s}\, s_2}{\Gamma \vdash \Pi x : A, B : \cancel{s}\, s_3}$$

$$(s_1, s_2) \in \mathcal{A} \subseteq \mathcal{S}^2 \qquad ((s_1, s_2), s_3) \in \mathcal{P} \subseteq \mathcal{S}^2 \times \mathcal{S}$$

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

Pure Type Systems (PTS)

PTS's are a family of type systems parametrized by:

- a set $\mathcal{S}$ of sorts
- a relation $\mathcal{A} \subseteq \mathcal{S}^2$ giving the sort type of some sorts

$$\frac{\Gamma \vdash}{\Gamma \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A}$$

- a relation $\mathcal{P} \subseteq \mathcal{S}^2 \times \mathcal{S}$ describing in what sorts live products depending on the sorts of their arguments

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A, B : s_3} \quad ((s_1, s_2), s_3) \in \mathcal{P}$$

a PTS is functional if $\mathcal{A}$ and $\mathcal{P}$ are functional relations

in this case types are unique up to $\downarrow_\beta$-equivalence:

if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $A \downarrow_\beta B$

Examples of (functional) PTS's

- simply-typed λ-calculus ($\lambda^{\rightarrow}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE})\}$

Examples of (functional) PTS's

- simply-typed λ-calculus ($\lambda^{\rightarrow}$):
  - $\mathcal{S} = \{\mathtt{TYPE}, \mathtt{KIND}\}$
  - $\mathcal{A} = \{(\mathtt{TYPE}, \mathtt{KIND})\}$
  - $\mathcal{P} = \{((\{\mathtt{TYPE}, \mathtt{TYPE}), \mathtt{TYPE})\}$
- $\lambda^{\rightarrow}$ + type constructors (e.g. List : $\mathtt{TYPE} \rightarrow \mathtt{TYPE}$):
  - $\mathcal{S} = \{\mathtt{TYPE}, \mathtt{KIND}\}$
  - $\mathcal{A} = \{(\mathtt{TYPE}, \mathtt{KIND})\}$
  - $\mathcal{P} = \{((\{\mathtt{TYPE}, \mathtt{TYPE}), \mathtt{TYPE}), ((\{\mathtt{KIND}, \mathtt{KIND}), \mathtt{KIND})\}$

Examples of (functional) PTS's

- simply-typed λ-calculus ($\lambda^{\rightarrow}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE})\}$

- $\lambda^{\rightarrow}$ + type constructors (e.g. List : $\texttt{TYPE} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\{\texttt{KIND}, \texttt{KIND}), \texttt{KIND})\}$

- $\lambda\Pi$-calculus = $\lambda^{\rightarrow}$ + dependent types (e.g. Array : $\mathbb{N} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\texttt{TYPE}, \texttt{KIND}), \texttt{KIND})\}$

Examples of (functional) PTS's

- simply-typed λ-calculus ($\lambda^{\rightarrow}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE})\}$

- $\lambda^{\rightarrow}$ + type constructors (e.g. List : $\texttt{TYPE} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\{\texttt{KIND}, \texttt{KIND}), \texttt{KIND})\}$

- $\lambda\Pi$-calculus = $\lambda^{\rightarrow}$ + dependent types (e.g. Array : $\mathbb{N} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\texttt{TYPE}, \texttt{KIND}), \texttt{KIND})\}$

- $\lambda^{\rightarrow}$ + polymorphic types (e.g. id : $\Pi A : \texttt{TYPE}, A \rightarrow A$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\{\texttt{KIND}, \texttt{TYPE}), \texttt{TYPE})\}$

Examples of (functional) PTS's

- simply-typed λ-calculus ($\lambda^{\rightarrow}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE})\}$

- $\lambda^{\rightarrow}$ + type constructors (e.g. List : $\texttt{TYPE} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\{\texttt{KIND}, \texttt{KIND}), \texttt{KIND})\}$

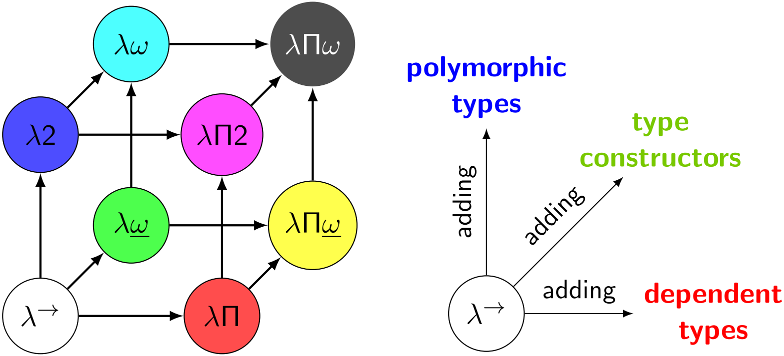
- $\lambda\Pi$-calculus = $\lambda^{\rightarrow}$ + dependent types (e.g. Array : $\mathbb{N} \rightarrow \texttt{TYPE}$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\texttt{TYPE}, \texttt{KIND}), \texttt{KIND})\}$

- $\lambda^{\rightarrow}$ + polymorphic types (e.g. id : $\Pi A : \texttt{TYPE}, A \rightarrow A$):
  - $\mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$
  - $\mathcal{A} = \{(\texttt{TYPE}, \texttt{KIND})\}$
  - $\mathcal{P} = \{((\{\texttt{TYPE}, \texttt{TYPE}), \texttt{TYPE}), ((\{\texttt{KIND}, \texttt{TYPE}), \texttt{TYPE})\}$

Remark: in all these examples, $\mathcal{P}(s_1, s_2) = s_2$

Barendregt's λ-cube

feature	PTS rule
simple types	TYPE, TYPE
polymorphic types	KIND, TYPE
dependent types	TYPE, KIND
type constructors	KIND, KIND



Examples of PTS's with infinitely many sorts

- **Agda** base type system:
  - $\mathcal{S} = \{\texttt{Set}_i | i \in \mathbb{N}\}$
  - $\mathcal{A} = \{(\texttt{Set}_i, \texttt{Set}_{i+1}) | i \in \mathbb{N}\}$
  - $\mathcal{P} = \{((\{\texttt{Set}_i, \texttt{Set}_j), \texttt{Set}_{\max(i,j)}) | i, j \in \mathbb{N}\}$
- **Lean** base type system:
  - $\mathcal{S} = \{\texttt{Sort}_i | i \in \mathbb{N}\}$
  - $\mathcal{A} = \{(\texttt{Sort}_i, \texttt{Sort}_{i+1}) | i \in \mathbb{N}\}$ $(\texttt{Sort}_0 = \texttt{Prop})$
  - $\mathcal{P} = \{((\{\texttt{Sort}_i, \texttt{Sort}_j), \texttt{Sort}_{\max(i,j)}) | i, j \in \mathbb{N}\} \cup \{((\{\texttt{Sort}_i, \texttt{Sort}_0), \texttt{Sort}_0) | i \in \mathbb{N}\}$

Rocq base type system is the same as Lean + subtyping:

$$\frac{}{\texttt{Type}_i \leq \texttt{Type}_{i+1}} \qquad \frac{B \leq B'}{\Pi x : A, B \leq \Pi x : A, B'}$$

Properties of the $\lambda\Pi$-calculus

- equivalence of types: if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $A \downarrow_\beta B$
- $\hookrightarrow_\beta$ terminates on well-typed terms (SN)
- $\hookrightarrow_\beta$ preserves typing (SR_β)
- type-inference $\exists A, \Gamma \vdash t : A$? is decidable
- type-checking $\Gamma \vdash t : A$? is decidable

Signature in the $\lambda\Pi$-calculus

a typing environment can be split in two parts:

1. a fixed part Σ representing global constants
2. a variable part Γ for local variables

in the following, we write

$$\Gamma \vdash_\Sigma t : A \text{ or simply } \Gamma \vdash t : A$$

instead of

$$\Sigma, \Gamma \vdash t : A$$

The need for more identifications

we have seen that the types

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

can be identified thanks to the typing rule

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

but not the types

$$A = \texttt{Array}(2 + 3) \quad \text{and} \quad A' = \texttt{Array}(5)$$

Outline

Rewriting

What is rewriting ?

introduced at the end of the 60's (Knuth)

a rewrite rule $l \hookrightarrow r$ is an equation $l = r$ used from left-to-right

rewriting simply consists in repeatedly replacing a subterm $l\sigma$ by $r\sigma$
(rewriting is Turing-complete)

it can be used to decide equational theories:

given a set $\mathcal{E}$ of equations, $\equiv_{\mathcal{E}}$ is decidable
if there is a rewrite system $\mathcal{R}$ such that:

- $\hookrightarrow_{\mathcal{R}}$ terminates (SN)
- $\hookrightarrow_{\mathcal{R}}$ is confluent (CR)
- $\equiv_{\mathcal{R}} = \equiv_{\mathcal{E}}$

where $\hookrightarrow_{\mathcal{R}}$ is the closure by context and
substitution of $\mathcal{R}$

$\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)

The $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$) simply extends the $\lambda\Pi$-calculus by identifying types modulo a set $\mathcal{R}$ of rewriting rules on a signature Σ:

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

remark: if $\Gamma \vdash t : A$ then $A = \texttt{KIND}$ or $\Gamma \vdash A : s$

$\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)

The $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$) simply extends the $\lambda\Pi$-calculus by identifying types modulo a set $\mathcal{R}$ of rewriting rules on a signature Σ:

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

remark: if $\Gamma \vdash t : A$ then $A = \mathtt{KIND}$ or $\Gamma \vdash A : s$

therefore it is equivalent to use the more symmetric rule

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : s \quad A \downarrow_{\beta\mathcal{R}} A' \quad \Gamma \vdash A' : s'}{\Gamma \vdash t : A'}$$

What is a $\lambda\Pi/\mathcal{R}$ theory ?

a theory in the $\lambda\Pi$-calculus modulo rewriting is given by:

- a signature Σ
- a set $\mathcal{R}$ of rewrite rules on Σ

such that:

- $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ terminates (SN)
- $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is confluent (CR)
- every rule $l \hookrightarrow r$ preserves typing ($SR_\mathcal{R}$):
  if $\Gamma \vdash l\sigma : A$ then $\Gamma \vdash r\sigma : A$

EuroProofNet

The Lambdapi proof assistant

Frédéric Blanqui

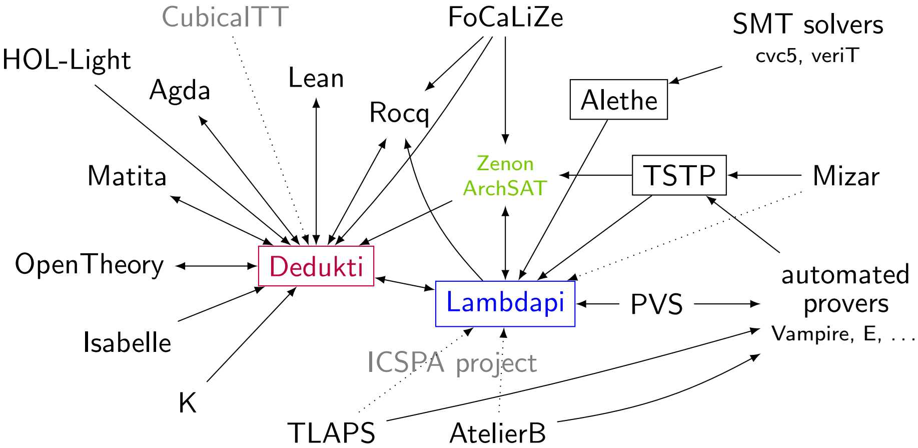
Deduc⊢eam

Inria

école normale supérieure paris—saclay

LMF

Dedukti, an assembly language for proof systems



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

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

Lambdapi

Lambdapi is an *interactive* **proof assistant for** $\lambda\Pi/\mathcal{R}$

- has its own syntax and file extension `.lp`
- can read and output `.dk` files
- symbols can have implicit arguments
- symbol declaration/definition generates typing/unification goals
- goals can be solved by structured proof scripts (tactic trees)
- ...

Where to find Lambdapi?

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

User manual: `https://lambdapi.readthedocs.io/`

Libraries:
`https://github.com/Deducteam/opam-lambdapi-repository`

How to install Lambdapi?

Using Opam:

```
opam install lambdapi
```

Compilation from the sources:

```
git clone https://github.com/Deducteam/lambdapi.git
cd lambdapi
make
make install
```

How to use Lambdapi?

Command line (batch mode):

```
lambdapi check file.lp
```

Through an editor (interactive mode):

- Emacs
- VSCode

Lambdapi automatically (re)compiles dependencies if necessary

How to install the Emacs interface?

3 possibilities:

1. Nothing to do when installing Lambdapi with opam

2. From Emacs using MELPA:

```
M-x package-install RET lambdapi-mode
```

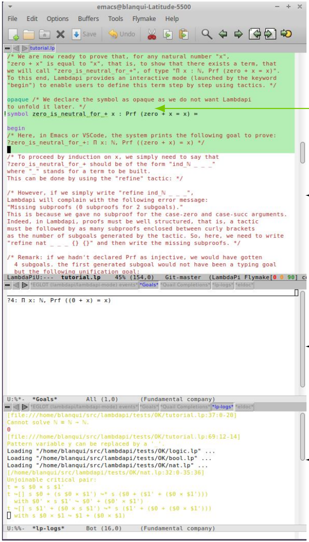
3. From sources:

```
make install_emacs
```

+ add in `~/.emacs`:

```
(load "lambdapi-site-file")
```

Emacs interface



checked part

edition buffer

goals

messages

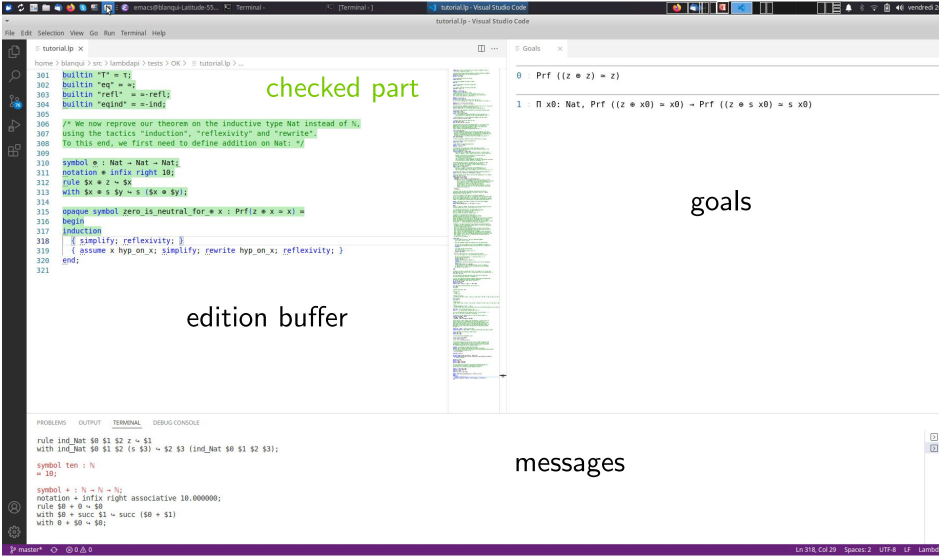
window layout
can be customized

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

How to install the VSCode interface?

From the VSCode Marketplace

VSCode interface



File `lambdapi.pkg`

developments must have a file `lambdapi.pkg` describing where to install the files relatively to the root of all installed libraries

```
package_name = my_lib
root_path = logical.path.from.root.to.my_lib
```

Importing the declarations of other files

lambdapi.pkg:

```
package_name = unary
root_path = nat.unary
```

file1.lp:

```
symbol A : TYPE;
```

file2.lp:

```
require nat.unary.file1;
symbol a : nat.unary.file1.A;
open nat.unary.file1;
symbol a' : A;
```

file3.lp:

```
require open nat.unary.file1 nat.unary.file2;
symbol b ≔ a;
```

Lambdapi syntax

BNF grammar:

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

file extension: `.lp`

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

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

Lambdapi syntax for 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 $\rightarrow$ *term*	non-dependent product
(*term*)	
_	unknown term
`let` *id* [: *term*] ≔ *term* `in` *term*	

Command for declaring/defining a symbol

*modifier** `symbol` *id* *param** [: *term*] [≔ *term*] [`begin` *proof* `end`] ;

param = *id* | `_` | `(` *id* $^+$ `:` *term* `)` | `[` *id* $^+$ `:` *term* `]`

implicit parameters

modifier's:

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

Examples of symbol declarations

```
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;
```

Command for declaring rewrite rules

rule *term* $\hookrightarrow$ *term* (**with** *term* $\hookrightarrow$ *term*)* ;

pattern variables must be prefixed by `$`:

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

Lambdapi tries to automatically check:

preservation of typing by rewrite rules (aka subject reduction)

Command for adding rewrite rules

Lambdapi supports:

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;
```

Lambdapi tries to automatically check:

local confluence (AC symbols/HO patterns not handled yet)

Higher-order pattern-matching

```
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;
```

Non-linear matching

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;
```

Queries and assertions

print *id* ;

type *term* ;

compute *term* ;

$(\texttt{assert} \mid \texttt{assertnot})\ \ id* \ \vdash\ term\ (: \mid \equiv)\ \ term\ ;$

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

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;
```

Reducing proof checking to type checking

(aka the Curry-de Bruijn-Howard isomorphism)

```
// type of propositions
symbol Prop : TYPE;
symbol = : N → N → Prop; notation = infix 1;

// interpretation of propositions as types
// (Curry-de Bruijn-Howard isomorphism)
symbol Prf : Prop → TYPE;

// examples of axioms
symbol refl x : Prf(x = x);
symbol s-mon x y : Prf(x = y) → Prf(s x = s y);
symbol ind_N (p : N → Prop)
  (case_0: Prf(p 0))
  (case_s: Π x : N, Prf(p x) → Prf(p(s x)))
  (n : N) : Prf(p n);
```

Stating an axiom vs Proving a theorem

Stating an axiom:

```
opaque 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:

```
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 can generate:

- typing goals $x_1 : A_1, \ldots, x_n : A_n \vdash ? : B$
- unification goals $x_1 : A_1, \ldots, x_n : A_n \vdash t \equiv^? u$

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

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

- 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*

tactic 's for unification goals:

- `solve` (applied automatically)

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
    {reflexivity}
    {assume x h; simplify; rewrite h; reflexivity}
end;
```

Tactics for typing goals

- `simplify` [*id*]
- `refine` *term*
  - `assume` id^+
  - `generalize` *id*
  - `apply` *term*
  - `induction`
  - `have` *id* : *term*
  - `reflexivity`
  - `symmetry`
  - `rewrite` [`right`] [*pattern*] *term* — like Rocq SSReflect
- `why3` — calls external prover

Defining inductive-recursive types

because symbol and rule declarations are separated, one can easily define inductive-recursive types in Dedukti or Lambdapi:

```
// lists without duplicated elements

constant symbol L : TYPE;

symbol ∉ : N → L → Prop; notation ∉ infix 20;

constant symbol nil : L;
constant symbol cons x l : Prf(x ∉ l) → L;

rule _ ∉ nil ↪ ⊤
with $x ∉ cons $y $l _ ↪ $x ≠ $y ∧ $x ∉ $l;
```

Command for generating induction principles

(currently for strictly positive parametric inductive types only)

```
inductive N : TYPE ≔ 0 : N | s : N → N;
```

is equivalent to:

```
symbol N : TYPE;
symbol 0 : N;
symbol s : N → N;
symbol ind_N (p : N → Prop)
  (case_0: Prf(p 0))
  (case_s: Π x : N, Prf(p x) → Prf(p(s x)))
  (n : N) : Prf(p n);
rule ind_N $p $c0 $cs 0 ↪ $c0
with ind_N $p $c0 $cs (s $x)
  ↪ $cs $x (ind_N $p $c0 $cs $x)
```

Example of inductive-inductive type

```
/* contexts and types in dependent type theory
Forsberg's 2013 PhD thesis */

// contexts
inductive Ctx : TYPE ≔
| □ : Ctx
| · Γ : Ty Γ → Ctx

// types
with Ty : Ctx → TYPE ≔
| U Γ : Ty Γ
| P Γ a : Ty (· Γ a) → Ty Γ;
```

Lambdapi's additional features wrt Dkcheck/Kocheck

Lambdapi is an *interactive* proof assistant for $\lambda\Pi/\mathcal{R}$

- has its own syntax and file extension `.lp`
- can read and output `dk` files
- supports Unicode characters and infix operators
- symbols can have implicit arguments
- can implicitly apply a type coercion in case of type mismatch
- symbol declaration/definition generates typing/unification goals
- goals can be solved by structured proof scripts (tactic trees)
- provides a `rewrite` tactic similar to Rocq/SSReflect
- can call external (first-order) automated theorem provers
- provides a command for generating induction principles
- provides a local confluence checker
- handles associative-commutative symbols differently
- supports user-defined unification rules and tactics

Practical session

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

have a look at tests/OK/tutorial.lp

modify or add some declarations/definitions/proofs