Dedukti 3 proof-mode with unification goals

Rehan Malak
j.w.w. Bruno Barras, Frédéric Blanqui, DEDUCTEAM

LSV, ENS Paris-Saclay

9 November 2020

1) Introduction

Why proving ? What do we want to prove ?

- proof obligations from certified software
  - ⇒ one way to ensure there is no bug (unit-testing not sufficient)
  - ⇒ embedded OS in medical devices, power plant, aerospace engineering, ...
- pure mathematics
  - ⇒ Kepler conjecture, 4-color theorem, Feit-Thompson odd-order group theorem
  - ⇒ Kapranov-Voevodsky (1991...2013) error
  - ⇒ Mochizuki's proof (2012) of ABC Conjecture published this year but is considered as flawed by the majority of the mathematical community

Homotopy Type Theory book (2013) :

Imagine a not-too-distant future when it will be possible for mathematicians to verify the correctness of their own papers [...], formalized in a proof assistant.

Why proving ? What do we want to prove ?

- proof obligations from certified software
  - ⇒ one way to ensure there is no bug (unit-testing not sufficient)
  - ⇒ embedded OS in medical devices, power plant, aerospace engineering, ...
- pure mathematics
  - ⇒ Kepler conjecture, 4-color theorem, Feit-Thompson odd-order group theorem
  - ⇒ Kapranov-Voevodsky (1991...2013) error
  - ⇒ Mochizuki's proof (2012) of ABC Conjecture published this year but is considered as flawed by the majority of the mathematical community

Homotopy Type Theory book (2013) :

Imagine a not-too-distant future when it will be possible for mathematicians to verify the correctness of their own papers [...], formalized in a proof assistant.

Why proving ? What do we want to prove ?

- proof obligations from certified software
  - ⇒ one way to ensure there is no bug (unit-testing not sufficient)
  - ⇒ embedded OS in medical devices, power plant, aerospace engineering, . . .
- pure mathematics
  - ⇒ Kepler conjecture, 4-color theorem, Feit-Thompson odd-order group theorem
  - ⇒ Kapranov-Voevodsky (1991. . . 2013) error
  - ⇒ Mochizuki's proof (2012) of ABC Conjecture published this year but is considered as flawed by the majority of the mathematical community

Homotopy Type Theory book (2013) :

Imagine a not-too-distant future when it will be possible for mathematicians to verify the correctness of their own papers [. . .], formalized in a proof assistant.

Type theory in brief :

- notion of type is primitive, no preexistence of objects without a type, no heterogeneous collections
- functions are given explicitly $f : A \to B$ defined by $f(x) := b$ with $b : B$, one can compute $f(a) \hookrightarrow b[a/x]$ if $a : A$
- type theory is its own deductive system (no need of two layers as in set theoretic foundations with propositions + sets in first order logic)

$\Rightarrow$ rigid constructions well suited for computers

Type theory in brief :

- notion of type is primitive, no preexistence of objects without a type, no heterogeneous collections
- functions are given explicitly $f : A \to B$ defined by $f(x) := b$ with $b : B$, one can compute $f(a) \hookrightarrow b[a/x]$ if $a : A$
- type theory is its own deductive system (no need of two layers as in set theoretic foundations with propositions + sets in first order logic)

$\Rightarrow$ rigid constructions well suited for computers

Curry-Howard correspondence in brief :

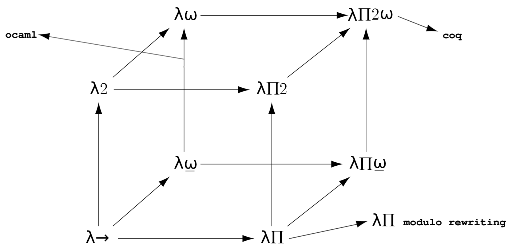
- *propositions as types* and *proofs as terms* (of this type)

$\Rightarrow$ proving a proposition = constructing an element (of this type)

Type formation captures logical operation :

Types	Logic	Sets interpretation
A	propositon	set
$a : A$	proof	element
$B(x)$	predicate	family of sets
$b(x) : B(x)$	conditional proof	family of elements
$A \to B = \prod_{x:A} B$	$A \Rightarrow B$	set of functions
$\prod_{x:A} B(x)$	$\forall_{x:A} B(x)$	product

Barendregt cube :



Three directions :

- values depending on types (polymorphic) $\Rightarrow \lambda 2 =$ System F
- types depending on types (type operators) $\Rightarrow \lambda\omega$
- types depending on values $\Rightarrow \lambda\Pi =$ Logical Framework
  - $\Rightarrow$ can re-encode first-order logic

With the **dependent** functions of $\lambda\Pi$:

- express concatenation of vectors with specified sizes
  `concat` : $\mathit{Vector}\ n \to \mathit{Vector}\ m \to \mathit{Vector}\ (n+m)$

$\lambda\Pi$-terms inductive definition :

$$t, u ::= TYPE | KIND | x | f | tu | \lambda x : t, u | \Pi x : t, u$$

$\lambda\Pi$-calculus modulo rewriting extends $\lambda\Pi$:

⇒ **define function and type symbols with rewriting rules**

In particular :

$$\dfrac{\Gamma \vdash a : A \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash b : B}\ \textsc{ExtendedConversionRule}$$

$Vector(2+2) \equiv_{\beta\Gamma} Vector\ 4$

⇒ **strict equality**

not a proposition to prove !

⇒ $\equiv_{\beta\Gamma}$ is the reflexive symmetric transitive closure of $\rightarrow_{\beta}$ or Γ

⇒ constrain rules so that **type checking remains decidable**

⇒ confluence and termination can be checked by external tools at the meta-theory level

$\lambda\Pi$-calculus modulo rewriting has advantages on other systems :

- simpler
- powerful enough to encode and check proofs developed in other systems : Coq, HOL Light, . . .

Interoperability :

- natural choice to translate one proof from a system to another
- building proofs assembling lemmas developed in different systems
- “universal” encyclopedia of mathematical theorems

⇒ Dedukti 2 is an implementation of the type-checker and comes with the translation tools

Logipedia

2) From a type-checker to a proof-assistant

Why not using directly this framework to formalize mathematics ?

		Dedukti 2 Type-checker	Dedukti 3 Proof-assistant
type inference :	`type a`	YES	YES
type check :	`assert a:A`	YES	YES
evaluate :	`compute a`	YES	YES
equality check :	`assert a=b`	YES	YES
build incrementally :	`?a : A`	NO	YES
equality on holes :	`a =? b`	NO	YES
some degree of automation		NO	YES

⇒ meta-variables for "inhabitation goals", tactics

⇒ "conversion goals" or "unification goals", tactics

This work :

⇒ use Dedukti 3 to formalize (categorical) models of type theory

⇒ add unification goals alongside the usual inhabitation goals

3) An example of formalization

Model theory in general :

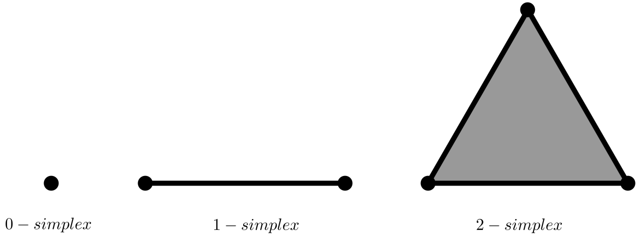
- ≃"theory of relations between theories"
- prove coherence, independence of a particular axiom, . . .
- interpretation of a language (eg. : geometrical interpretation)

Model of intensional dependent type theory :

- identity types (propositional equality) are not trivial

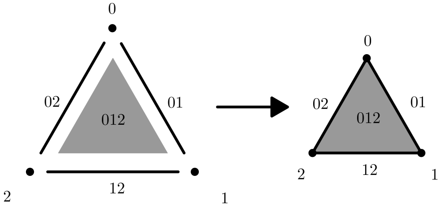
⇒ inhabited by terms behaving as path in homotopy theory

⇒ **simplicial sets** $\hat{\Delta} := \Delta^{\mathrm{op}} \to Set$ where the objects of Δ are $[n] := \{0, \ldots, n\}$ and the morphisms are the order-preserving maps



Extensional set theory vs intensional type theory :

- models usually relying on set-theoretic foundations
- interesting to interpret directly in type theory ("HoTT univalent foundations")
- simplicial sets are difficult to formalize in intensional type theory because of the **coherence conditions**



$$
\begin{aligned}
\partial_0 01 &= 1\\
\partial_1 01 &= 0\\
\partial_1 12 &= 2\\
\partial_2 12 &= 1\\
\partial_0 02 &= 2\\
\partial_2 02 &= 0\\
\partial_0 012 &= 12\\
\partial_1 012 &= 02\\
\partial_2 012 &= 01
\end{aligned}
$$

Dedukti can help :

$\Rightarrow$ $\lambda\Pi$-modulo-rewriting provides a decidable **strict equality**

The formalization of **semi**-simplicial sets has then been turned into a model of a **non-dependent** type theory : System F.

$\Rightarrow$ Types2020 Book of Abstracts

To reach the formalization of a full intensional **dependent** type theory :

- category with families
- semi-simplicial sets $\rightsquigarrow$ simplicial-sets $\rightsquigarrow$ Kan simplicial-sets

This has been tried by B.Barras on Dedukti 2 :

$\Rightarrow$ turned out to be impractical without a real proof-assistant and "holes" development

$\Rightarrow$ one really needs **interactivity** with **unification goals**

4) Unification goals implementation in Dedukti 3

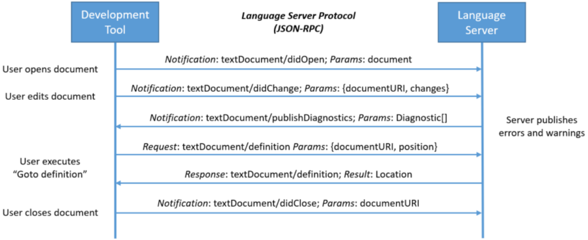
Language Server Protocol (LSP) :

⇒ resolves the "matrix problem" between programming languages and Integrated Development Environment (IDE).

Instead of :

	No LSP	LSP
M IDE's & N languages	M x N plugins	$M + N$ plugins

- user stays in his/her favorite IDE
- language designer focuses on the server side
- IDE designer focuses on the client side
- they can talk to each other via a standardized protocol, (here) via textual JSON documents



```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z       + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z       × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpretation
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp       All (29,0)    <N>    (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDoc Abbrev)   [eglot:lambdapi]
```

```
require open tests.lib

// Is it true that 2 * x = x + x ???
symbol my_theorem : Πx, P (eq (2 × x) (x + x))
```

```
U:---  demo.lp      All (5,0)     <Vl>   (LambdaPi +5 Flymake:Wait[0 0 3] Undo-Tree ElDoc Abbrev)   [eglot:lambdapi]
Use +,-,0 for further adjustment
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp      All (29,0)    <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDo
```

```
require open tests.lib

// Is it true that 2 * x = x + x ???
symbol my_theorem : Πx, P (eq (2 × x) (x + x)) ≔
begin
  assume x
end
```

```
U:---  demo.lp      All (8,0)     <N>   (LambdaPi +5 Flymake:Wait[1 0 3] Undo-Tree ElDoc Abbrev)  [eglot:lambdapi]
```

```
x: N
--------------------------------------------------
Goal 67: P (eq (2 × x) (x + x))
```

```
U:%*-  *Goals*      All (5,0)     <N>   (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp      All (29,0)     <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDo
```

```
require open tests.lib

// Is it true that 2 * x = x + x ???
symbol my_theorem : Πx, P (eq (2 × x) (x + x)) ≔
begin
  assume x
  simpl
end
```

```
U:---  demo.lp       All (7,0)     <N>    (LambdaPi +5 Flymake:Wait[1 0 4] Undo-Tree ElDoc Abbrev)   [eglot:lambdapi]
```

```
x: N
------------------------------------------------------------
Goal 77: P (eq (x + x) (x + x))
```

```
U:%*-  *Goals*       All (4,0)     <N>    (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n      ↪ $n
with (s $m) + $n      ↪ s ($m + $n)
with $m     + z       ↪ $m
with $m     + (s $n)  ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _       ↪ z
with (s $m) × $n      ↪ $n + $m × $n
with _      × z       ↪ z
with $m     × (s $n)  ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp        All (29,0)    <N>    (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDo
```

```
require open tests.lib

// Is it true that 2 * x = x + x ???
symbol my_theorem : Πx, P (eq (2 × x) (x + x)) ≔
begin
  assume x
  simpl
  refine refl (add x x)
end
```

U:--- demo.lp All (8,0) <N> (LambdaPi +5 Flymake:Wait[0 0 6] Undo-Tree ElDoc Abbrev) [eglot:lambdapi]

U:%*- *Goals* All (1,0) <N> (Fundamental +5)

Beginning of line

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

U:--- lib.lp All (29,0) <N> (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDo

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin
  assume x
  simpl
  refine refl (add x x)
end
```

```
U:---  demo.lp      All (8,0)     <N>   (LambdaPi +5 Flymake:Wait[1 0 5] Undo-Tree ElDoc Abbrev)  [eglot:lambdapi]
```

```
x: N
--------------------------------------------------
Goal 107: P (eq (x + (x + x)) (x + x))
```

```
U:**-  *Goals*      All (4,0)     <N>   (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp       All (29,0)    <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree ElDo
```

Unification can fail if :

- the user made a mistake and the type is not well formed
- the default unification algorithm fails

Solution :

- no need for a proof script if unification + typing are OK
- if not, don't fail immediately and let the user interact

⇒ interactive mode with inhabitation **+ unification goals**

⇒ interactive mode for theorems **+ symbol declarations** (unification can fail even if there is no inhabitation goals)

⇒ new tactics

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin

end
```

```
U:---  demo.lp     All (6,0)    <N>   (LambdaPi +5 Flymake:Wait[1 0 2] Undo-Tree ElDoc Abbrev)  [eglot:lambdapi]
```

```
--------------------------------------------------------
Unif     : N ≡ N

--------------------------------------------------------
Typ     101: Πx: N, P (eq (3 × x) (x + x))
```

```
U:%*-  *Goals*     Top (1,0)    <N>   (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp     All (1,0)    <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree El
```

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin
  solve
end
```

```
U:---  demo.lp      All (6,0)     <N>   (LambdaPi +5 Flymake:Wait[1 0 3] Undo-Tree ElDoc Abbrev)  [eglot:lambdapi]
```

```
--------------------------------------------------
Typ    103: Πx: N, P (eq (3 × x) (x + x))
```

```
U:%*-  *Goals*      All (1,0)     <N>   (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp       All (1,0)     <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree El
```

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin
  solve
  assume x
end
```

```
U:**-  demo.lp    All (7,0)    <N>    (LambdaPi +5 Flymake[1 0 4] Undo-Tree ElDoc Abbrev)  [eglot:lambdapi]
```

```
x: N
-----------------------------------------------------
Typ    108: P (eq (3 × x) (x + x))
```

```
U:%*-  *Goals*    All (1,0)    <N>    (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp    All (1,0)    <N>    (LambdaPi +3 Flymake[0 0 24] Undo-Tree El
```

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin
  solve
  assume x
  simpl
end
```

```
U:---  demo.lp        All (8,0)     <N>    (LambdaPi +5 Flymake:Wait[1 0 5] Undo-Tree ElDoc Abbrev)   [eglot:lambdapi]
```

```
x: N
--------------------------------------------------------
Typ    182: P (eq (x + (x + x)) (x + x))
```

```
U:%*-  *Goals*        All (1,0)     <N>    (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z      + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z      × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp         All (1,0)     <N>    (LambdaPi +3 Flymake[0 0 24] Undo-Tree El
```

```
require open tests.lib

// Is it true that 3 * x = x + x ???
symbol my_theorem : Πx, P (eq (3 × x) (x + x)) ≔
begin
  solve
  assume x
  simpl
  refine refl (add x x)
end
```

```
U:---  demo.lp        All (9,0)     <N>   (LambdaPi +5 Flymake:Wait[1 0 6] Undo-Tree ElDoc Abbrev)   [eglot:lambdapi]
```

```
--------------------------------------------------
Unif     : x + x ≡ x + (x + x)
```

```
U:%*-  *Goals*        All (1,0)     <N>   (Fundamental +5)
```

```
// Natural numbers.
constant symbol N : TYPE
constant symbol z : N
constant symbol s : N → N
set builtin "0"  ≔ z
set builtin "+1" ≔ s

// Addition function.
symbol add : N → N → N
set infix left 6 "+" ≔ add
rule z       + $n     ↪ $n
with (s $m) + $n     ↪ s ($m + $n)
with $m     + z      ↪ $m
with $m     + (s $n) ↪ s ($m + $n)

// Multiplication function.
symbol mul : N → N → N
set infix left 7 "×" ≔ mul
rule z       × _      ↪ z
with (s $m) × $n     ↪ $n + $m × $n
with _      × z      ↪ z
with $m     × (s $n) ↪ $m + $m × $n

// Type of propositions and their interpret.
constant symbol Prop : TYPE
injective symbol P : Prop → TYPE
constant symbol eq : N → N → Prop
constant symbol refl : Π x, P (eq x x)
```

```
U:---  lib.lp        All (1,0)     <N>   (LambdaPi +3 Flymake[0 0 24] Undo-Tree El
```

5) Conclusion

To sum up :

- Dedukti is a natural choice for interoperability :
  - $\lambda\Pi$-calculus modulo rewriting as a logical framework is powerful
  - can export a proof from a system to another
- Dedukti 3 :
  - proof-assistant with tactics suitable for proof developments
  - gradually improving the user interface
  - Emacs and VSCode IDE's using state-of-the-art LSP protocol
- This work made contributions to :
  - a library formalizing the category of semi-simplicial sets and a model of a **non-dependent** type theory (System F)

  $\Rightarrow$ exposed in Types2020 book of abstracts
  - make the possibility for the user to manipulate unification goals

Work in progress :

$\Rightarrow$ investigate formalization of a model of **dependent** type theory

$\Rightarrow$ unification goals $\rightsquigarrow$ unification tactics ($\simeq$ pieces of the unification algorithm)