A Lisp Way to Type Theory and Formal Proofs

Frederic Peschanski

A Lisp Way to Type Theory and Formal Proofs

(Submitted January 28 2017)

Frederic Peschanski
UPMC Sorbonne Universités – LIP6
4 place Jussieu
Paris, France
first.last@lip6.fr

ABSTRACT

In this paper we describe the LaTTe proof assistant, a software that promotes the Lisp notation for the formalization of and reasoning about mathematical contents. LaTTe is based on type theory and implemented as a Clojure library with top-level forms for specifying axioms, definitions, theorems and proofs. As a pure library, LaTTe can exploit the advanced interactive coding experience provided by modern development environments. Moreover, LaTTe enables a form of proving in the large by leveraging the Clojar/Maven ecosystem. It also introduces a very simple and concise domain-specific proof language that is deeply rooted in natural deduction proof theory. And when pure logic is not enough, the system allows to take advantage of the host language: a Lisp way to proof automation.

CCS Concepts

•**Theory of computation** → **Logic; Type theory;**

Keywords

Logic; Type Theory; Proof Assistant; Clojure

1. INTRODUCTION

Proof assistants realize an ancient logician's dream of (re)constructing mathematics based on purely mechanical principles. Most proof assistants (perhaps with the exception of [6]) are complex pieces of software.

One important factor of this complexity is that proof assistants generally try to mimic the common mathematical notation, which is a complex parsing problem that very often get in the way of the user. LaTTe, in comparison, totally surrenders to the simplicity (and sheer beauty) of a Lisp notation. One immediate gain is that the complex issue of parsing vanishes. It also makes the definitions more explicit and less ambiguous than if written with more traditional (and highly informal) computerized variants of mathematical notations.

ELS '16 April AA–BB, 2017, Brussels, Beldium

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

Another important characteristic is that LaTTe is implemented as a library. The activity of doing mathematics is here considered as a prolongation (and not as an alternative) to that of programming. We see this as a clear advantage if compared to proof assistants designed as standalone tools such as Isabelle [10] or Coq [13]. First, this gives a better separation of concerns, only a small library is to be maintained, rather than a complex tool-chain. Moreover, modern Lisp implementations often come with very advanced tools for interactive programming that enables *live-coding mathematics*[1].

Another important factor of complexity in proof assistants, both for the developers and the users, is the language in which proof are written. The LaTTe proof language is from this respect extremely small and simple. It only introduces two constructs: **assume** and **have** with obvious semantics. The language is also arguably less ad-hoc than most other proof languages in that it is deeply rooted in natural deduction proof theory [11]. As a summary, LaTTe aims at *minimalism*. It has less features than most other proof assistants and each implemented feature strives for simplicity and concision. We argue, however, that this set of features is already plenty enough for the purpose of formalizing mathematics on a computer.

The outline of the paper is as follows. First, in section 2 LaTTe is introduced from a user perspective. The general principles underlying the LaTTe kernel are then discussed in section 3. In particular, we provide a bidirectional type systems that is used in LaTTe to perform both proof/type-checking and type inference. Perhaps the most significant feature of LaTTe is the introduction of a *domain-specific language* (DSL) for proof scripts. This is presented and illustrated with examples in section 4. In section 5 we discuss the way parts of proofs can be automated, the Lisp-way. In section 6 we discuss some (non-)features of LaTTe in comparison with other proof assistants.

2. A LOGICIAN'S DREAM

Quoting [5], the main functionality of a proof assistant is to:

- formalize mathematical contents on a computer
- assist in proving theorems about such contents

Mathematical contents, as can be found in textbooks and proof assitants, is mostly based on definitions, and statement

[1] cf. https://www.youtube.com/watch?v=5YTCY7wm0Nw

```
(proof compose-injective
  :script
  ;; Our hypothesis is that f and g are injective.
  (assume [Hf (injective U V f)
           Hg (injective T U g)]
    ;; We now have to prove that the composition is injective.
    ;; For this we consider two arbitrary elements x and y
    ;; such that f ∘ g(x) = f ∘ g(y)
    (assume [x T
             y T
             Hinj (equal V (f (g x)) (f (g y)))]
      ;; Since f is injective we have: g(x) = g(y).
      (have ⟨a⟩ (equal U (g x) (g y))
            :by (Hf (g x) (g y) Hinj))
      ;; And since g is also injective we obtain: x = y.
      (have ⟨b⟩ (equal T x y) :by (Hg x y ⟨a⟩))
      ;; Since x and y are arbitrary, f ∘ g is thus injective.
      (have ⟨c⟩ (injective T V (λ [x T] (f (g x))))
            :discharge [x y Hinj ⟨b⟩]))
    ;; Which is enough to conclude the proof.  □
    (qed ⟨c⟩)))
```

Table 1: A declarative proof script in LaTTe.

of axioms and theorems. For example, we can define the notion of an injective function as follows in LaTTe:

```
(definition injective
  "An injective function."
  [[T ⋆] [U ⋆] [F (⟹ T U)]]
  (forall [x y T]
     (⟹ (equal U (F x) (F y))
        (equal T x y))))
```

We see that a Lisp notation (with Clojure extensions and Unicode characters) is used for the definition. With a bit of practice and a good editor, this notation shoud become a Mathematician's best friend.

After defining mathematical concepts, the next step is the statement of theorems. An important property of injective functions is stated below.

```
(defthm compose-injective
  "The composition of two injective functions
  is injective."
  [[T ⋆] [U ⋆] [V ⋆] [f (⟹ U V)] [g [⟹ T U]]]
  (⟹ (injective U V f)
     (injective T U g)
     (injective T V (λ [x T] (f (g x))))))
```

The **defthm** form only declares a theorem. In the next step, we must provide a formal proof so that the theorem can be used in further developments.

There are two main families of proof languages. First there is the LCF-style tactic languages as found in e.g. Coq [13] or HOL light [6]. This is an imperative formalism that works on a goal-state to conduct the proofs. The main drawback of such an approach is that the proofs are then very remote from the mathematical practice. Moreover, they cannot be understood without interacting with the tool. Proofs do not have a life on their own in such environments. The declarative proof languages, such as Isar[2] in Isabelle [10], on the contrary, are designed to be closer to standard "pencil and paper" proofs. In LaTTe a proof can be written in two ways: either by supplying a lambda-term (as explained in the next section), or more interestingly using a declarative

[2]Isar stands for "Intelligible semi-automated reasoning".

```
(definition D "⟨doc⟩"
  [[x1 t1] ... [xn tn]]
  ⟨term⟩)
```

⟨term⟩	t, u	::=	$\square$	(kind)
			$\mid \star$	(type)
			$\mid x$	(variable)
			$\mid [t\ u]$	(application)
			$\mid (\lambda\ [x\ t]\ u)$	(abstraction)
			$\mid (\Pi\ [x\ t]\ u)$	(product)
			$\mid (D\ t_1\ t_2\ \ldots\ t_n)$	(instantiation)

Table 2: The syntax of the LaTTe calculus

proof script. The language for proof scripts is probably the most distinctive feature of LaTTe. It is described more thoroughly in section 4, but we illustrate its use by the proof of the theorem compose-injective, given in Table 1. One important characteristic of this proof is that the formal arguments (in Lisp forms) are in close correspondence with the informal proof steps (in comments). An important difference with a proof language such as Isar is that of simplicity: only two constructs are introduced: **assume** and **have**. And the theory behind is just standard natural deduction proof theory [11].

3. LAMBDA THE ULTIMATE

The kernel of the LaTTe library is a computerized version of a simple, although very expressive lambda-calculus. It is a variant of λD as described in the book [8], which corresponds to the *calculus of constructions* [4] (without the prop/type distinction) enriched with definitional features.

3.1 Syntax

The basic syntax of the LaTTe calculus is given in Table 2. There are various syntactic sugars that will for some of them be introduced later on, and there is also a **defnotation** mechanism to introduce new notations when needed (e.g. for the existential quantifier, which is a derived principle in type theory). Perhaps the most notable feature of the calculus is that it is compatible with the *extended data notation*[3], i.e. it is a subset of the Clojure language[4]. As a dependently-typed lambda-calculus, there is no syntactic distinction between terms and types. A type is simply a term whose type is $\star$, called the sort of types. The type of $\star$ itself is a sort called a *kind* and denoted by $\square$[5]. The kernel of any lambda-calculus is formed by the *triptych*: variable occurrences $x, y, \ldots$, function applications $[t\ u]$ and abstractions $(\lambda\ [x\ t]\ u)$. The latter expresses a function with an argument x of type t (hence a term of type $\star$) and with term u as body. The type of a lambda-abstraction is called a *product* and is denoted by $(\Pi\ [x\ t]\ u)$. The intended meaning is that of universal quantification: "for all x of type t, it is the case that u". As an example, the term $(\lambda\ [A\ \star]\ (\lambda\ [x\ A]\ x))$ corresponds to a type-generic identity function. Its type is $(\Pi\ [A\ \star]\ (\Pi\ [x\ A]\ A))$. An important syntactic sugar that we will largely exploit is that if in $(\Pi\ [x\ t]\ u)$ the variable x

[3]cf. https://github.com/edn-format/edn

[4]This means that the lambda-terms in LaTTe can be quoted in Clojure, and thus used to feed macros.

[5]For the sake of logical consistency, the kind $\square$ has no type, which makes LaTTe an *impredicative* type theory.

has no free occurrence in the body u, then we can rewrite the product as $(\Longrightarrow\ t\ u)$. The intended meaning is that of an arrow type of a function that from an input of type t yields a value of type u, or equivalently by the logical proposition that "t implies u". When the logical point of view is adopted, the universal quantified symbol $\forall$ can be used instead of the more esoteric Π. For the type-generic identity function, this finally gives $(\forall\ [A\ \star]\ (\Longrightarrow\ A\ A))$, i.e. for any type (proposition) A, it is the case that A implies A. This gives a first glimpse of the tight connection between computation and logic in such a typed lambda-calculus, namely the *Curry-Howard correspondence* [12].

Because LaTTe is aimed at *practice* and not only theory, the basic lambda-calculus must be enriched by *definitional principles*. First, parameterized definitions can be introduced using the **definition** form. Then, such definitions can be *instantiated* to produce *unfolded* terms. In LaTTe, parenthesized (and desugared) expressions that do not begin with λ or Π are considered as instantiations of definitions.

For example, we can introduce a definition of the type-generic identity function as follows:

```
(definition identity
  "the identity function"
  [[A ⋆][x A]]
  x)
```

Then, an expression such as (identity nat 42) would be instantiated to 42 (through δ-reduction, as discussed below). In theory, explicit definitions and instantiations are not required since they can be simulated by lambda-abstractions and applications, but in practice it is very important to give names to mathematical concepts (as it is important to give names to computations using function definitions).

3.2 Semantics

The semantics of lambda-calculus generally rely on a rewriting rule named β-reduction and its application under a context:

- (conversion) $[(\lambda[x\ t]u)\ v] \xrightarrow{\beta} u\{v/x\}$
- (context) if $t \xrightarrow{\beta} t'$ then $C[t] \xrightarrow{\beta} C[t']$, for any *single-hole context* C.

The notation $u\{v/x\}$ means that in the term u all the free occurrences of the variable x must be substituted by the term v. For example, we have $[a\ [(\lambda\ [x]\ [bx])\ [c\ d]]] \xrightarrow{\beta} [a\ [b\ [c\ d]]]$. This is because if we let $t = [(\lambda\ [x]\ [bx])\ [c\ d]]$ and $t' = [b\ [c\ d]]$ then $t \xrightarrow{\beta} t'$ by the conversion rule. And if we define the context $C[X] = [a\ X]$ with hole X, then $C[t] \xrightarrow{\beta} C[t']$ by the context rule. While β-reduction seems trivial, it is in fact *not* the case, at least at the implementation level. One difficulty is that lambda-terms must be quotiented according to α-equivalence. For example, $(\lambda\ [x\ t]\ u) \equiv_\alpha (\lambda\ [y\ t]\ u)$ because we do not want to distinguish the lambda-terms based on their bound variables. Reasoning about such issues is in fact not trivial, cf. e.g. [3]. A lambda-calculus aimed at logical reasoning has to fulfill two important requirements:

- *strong normalization*: no lambda-term t yields an infinite sequence of β-reductions
- *confluence*: if $t \xrightarrow{\beta^*} t_1$ and $t \xrightarrow{\beta^*} t_2$ then there exist a term u such that $t_1 \xrightarrow{\beta^*} u$ and $t_2 \xrightarrow{\beta^*} u$ (up-to α-equivalence)[6]

As a consequence, each lambda-term t posseses a unique *normal form* $\widetilde{t}$ (up-to α-equivalence). Thus, two terms t_1 and t_2 are β-equivalent, denoted by $t_1 =_\beta t_2$, iff $\widetilde{t_1} \equiv_\alpha \widetilde{t_2}$. In a proof assistant based on type theory, the α-equivalence and β-reduction relations are not enough, for example to implement the definitional features. The LaTTe kernel uses a δ-reduction relation, similar to that of [8], to allow the instantiation of definitions.

If we consider a definition D of the form given in Table 2 then the rules are as follows:

- (instantiation) $(D\ t_1\ \ldots\ t_n) \xrightarrow{\delta} u\{t_1/x_1, \ldots, t_n/x_n\}$
- (context) if $t \xrightarrow{\delta} t'$ then $C[t] \xrightarrow{\delta} C[t']$, for any *single-hole context* C.

At the theoretical level, the overlap between β and δ-reductions is relatively unsettling but in practice, β-reduction works by copying terms while δ-reduction uses names and references, and is thus much more economical. Moreover, in mathematics giving names to definitions and theorems is of primary importance so the issue must be addressed with rigour. LaTTe here still roughly follows [8].

LaTTe introduces a further σ-reduction relation for **defspecial**'s. This is discussed in section 5.

3.3 Type inference

There are three interesting algorithmic problems related to the typing of lambda-terms. First, there is the *type checking* problem. Given a term t and a term u, check that u is the type of t. In LaTTe this problem occurs given:

- a *definitional environment* Γ containing an unorded map of definition names to definitions. For example, if D is a definition then $\Gamma[D(t_1, \ldots, t_n)]$ gives the lambda-term corresponding to the definition contents.
- a *context* Δ containing an ordered list of associations from (bound) variable names to types. If x is a variable in the context, then $\Delta[x]$ is its type.

A term t that has type u in environment Γ and context Δ is denoted by: $\Gamma; \Delta \vdash t :: u$. It is not very difficult to show that type checking in LaTTe is decidable. This would be a relatively straightforward elaboration for λD in [8]. Suppose that we know only the type part. Thus, we have to find a term to replace the question mark in $\Gamma; \Delta \vdash ? :: u$. This *term synthesis* problem is not decidable in LaTTe and the intuition is that we would then have an algorithmic way to automatically find a proof for an arbitrary theorem. Term synthesis can still be partially solved in some occasions, and it is an interesting approach for proof automation. On the other hand, one may want to replace the question mark in the following problem: $\Gamma; \Delta \vdash t ::?$. Now we are looking for the type of a given term, which is called the *type inference* problem. LaTTe has been designed so that this problem is decidable and can be solved efficiently. If the inferred type of term t is A, then we write: $\Gamma; \Delta \vdash t :> A$.

[6]The notation $t \xrightarrow{\beta^*} t'$ means zero or more β-reductions from t to t', it is the reflexive and transitive closure of the relation of β-reduction under context.

$$\frac{}{\Gamma;\Delta \vdash \star :> \square}\ (type) \quad \frac{\Gamma;\Delta \vdash \widetilde{A} :: s \quad s \in \{\star, \square\}}{\Gamma;\Delta, x :: A \vdash x :> A}\ (var) \quad \frac{\Gamma;\Delta \vdash A :> s_1 \quad \Gamma;\Delta, x :: A \vdash B :> s_2 \quad \widetilde{s_1}, \widetilde{s_2} \in \{\star, \square\}}{\Gamma;\Delta \vdash (\Pi\ [x\ A]\ B) :> s_2}\ (prod)$$

$$\frac{\Gamma;\Delta, x :: A \vdash t :> B \quad \Gamma;\Delta \vdash (\Pi\ [x\ A]\ B) :> s \quad \widetilde{s} \in \{\star, \square\}}{\Gamma;\Delta \vdash (\lambda\ [x\ A]\ t) :> (\Pi\ [x\ A]\ B)}\ (abs) \quad \frac{\Gamma;\Delta \vdash t :> (\Pi\ [x\ A]\ B) \quad \Gamma;\Delta \vdash u :: A}{\Gamma;\Delta \vdash [t\ u] :> B\{u/x\}}\ (app)$$

$$\frac{\begin{array}{l}\Gamma[D] :: [x_1\ t_1]\ [x_2\ t_2]\ \cdots\ [x_n\ t_n] \to t \\ \Gamma;\Delta \vdash e_1 :: t_1 \quad \Gamma;\Delta, x_1 :: t_1 \vdash e_2 :: t_2 \quad \cdots \quad \Gamma;\Delta, x_1 :: t_1, x2 :: t_2, \ldots, x_{m-1} :: t_{m-1} \vdash e_m :: t_m\end{array}}{\Gamma;\Delta \vdash (D\ u_1\ u_2\ \ldots\ u_m) :> (\Pi\ [x_{m+1}\ t_{m+1}]\ \ldots\ (\Pi\ [x_n\ t_n]\ t\{u_1/x_1, u_2/x_2, \ldots, u_m/x_m\})\cdots)}\ (ref)$$

Table 3: Type inference rules

```
(defn type-of-var [def-env ctx x]
  (if-let [ty (ctx-fetch ctx x)]
    (let [[status sort] (let [ty' (norm/normalize def-env ctx ty)]
                          (if (stx/kind? ty')
                            [:ok ty']
                            (type-of-term def-env ctx ty)))]
      (if (= status :ko)
        [:ko {:msg "Cannot calculate type of variable." :term x :from sort}]
        (if (stx/sort? sort)
           [:ok ty]
           [:ko {:msg "Not a correct type (super-type is not a sort)" :term x :type ty :sort sort}])))
    [:ko {:msg "No such variable in type context" :term x}]))


(example
 (type-of-var {} '[[bool ⋆] [x bool]] 'x) ⇒ '[:ok bool])

(example
 (type-of-var {} '[[x bool]] 'x)
 ⇒ '[:ko {:msg "Cannot calculate type of variable.",
          :term x :from {:msg "No such variable in type context", :term bool}}])
```

Table 4: The Clojure source for the (var) inference rule.

Table 3 summarizes the type inference rules used in LaTTe. Each rule corresponds to a Clojure function, we will take the (var) rule as an example. Its implementation is a Clojure function named type-of-var, whose complete definition is given in Table 4. For a variable x present in the context Δ (parameter ctx in the source code) with type A, the (var) rule first normalizes A (using the norm/normalize function) and compares its type with a sort $\star$ or $\square$. This checks that A is effectively a type. In the conclusion of the rule, the notation $x :> A$ is to be interpreted as "the inferred type for x is A". In the source code, this corresponds to the value of the variable ty. Note that only the denormalized version of the type is inferred, which is an important memory saving measure. The other rules are connected similarly to a rather straightforward Clojure function. One subtlety in the (app) rule for application is that the operand term u must be checked against an inferred type A. It is possible to implement a separate type-checker. For one thing, type-checking can be done more efficiently than type inference. Moreover, it is a simpler algorithm and is useful for separate proof checking. However, there is a large overlap between the two algorithms and it is not really worth the duplication. Indeed, a type-checking algorithm can be obtained "for free" using the following fact:

$$\Gamma;\Delta \vdash t :: u \text{ iff } \Gamma;\Delta \vdash t :> v \text{ and } v =_\beta u$$

⟨proof⟩ P	::=	(**proof** thm :term t)	(direct proof)
		| (**proof** thm :script ρ)	(proof script)
⟨script⟩ ρ	::=	$\sigma\ \rho$	(proof step σ)
		| (**assume** $[H\ t]\ \rho$)	(global assumption)
		| (**qed** t)	(proof term t)
⟨step⟩ σ ::=			
		(**assume** $[H\ t]\ \rho$)	(local assumption)
		| (**have** $\langle a \rangle\ A$:by t)	(proof of A with term t)
		| (**have** $\langle a \rangle\ A$:discharge $[x_1 \cdots x_n\ t]$)	(discharge assumptions)

Table 5: The proof language of LaTTe

The complete implementation of the type inference algorithm is less than 400 lines of commented code, and is available on the github repository[7].

4. A DSL FOR PROOF SCRIPTS

The language of mathematical proofs is very literary and remote from the formal requirements of a computer system.

[7] cf.https://github.com/latte-central/LaTTe/blob/master/src/latte/kernel/typing.clj

$$\frac{\Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash t :: P}{\Gamma, thm(x_1 :: t_1, \ldots, x_n :: t_n)\triangleleft? :: P \vdash (\textbf{proof}\ thm\ \text{:term}\ t) \dashv \Gamma, thm \triangleleft t :: P}\ (term) \qquad \frac{\Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash \rho \Rrightarrow t \dashv \Gamma' \quad \Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash t :: P}{\Gamma, thm(x_1 :: t_1, \ldots, x_n :: t_n)\triangleleft? :: P \vdash (\textbf{proof}\ thm\ \text{:script}\ \rho) \dashv \Gamma, thm \triangleleft t :: P}\ (script)$$

$$\frac{\Gamma; \Delta \vdash \sigma \Rrightarrow t \dashv \Gamma' \quad \Gamma'; \Delta \vdash \rho \Rrightarrow u \dashv \Gamma''}{\Gamma; \Delta \vdash \sigma\ \rho \Rrightarrow u \dashv \Gamma''}\ (step) \qquad \frac{\Gamma; \Delta, H :: t \vdash \rho \Rrightarrow u}{\Gamma; \Delta \vdash (\textbf{assume}\ [H\ t]\ \rho) \Rrightarrow u}\ (glob) \qquad \frac{}{\Gamma; \Delta \vdash (\textbf{qed}\ t) \Rrightarrow t}\ (qed)$$

$$\frac{\Gamma; \Delta, H :: t \vdash \rho \Rrightarrow u \dashv \Gamma'}{\Gamma; \Delta \vdash (\textbf{assume}\ [H\ t]\ \rho) \dashv \Gamma'}\ (loc) \qquad \frac{\Gamma; \Delta \vdash t :: A}{\Gamma; \Delta \vdash (\textbf{have}\ \langle a \rangle\ A\ \text{:by}\ t) \dashv \Gamma, \langle a \rangle \triangleleft t :: A}\ (by)$$

$$\frac{\Gamma; \Delta \vdash t' :: A \quad t' \equiv (\lambda\ [x_1 :: t_1]\ \cdots\ (\lambda\ [x_n :: t_n]\ t) \cdots)}{\Gamma; \Delta, x_1 :: t_1, \ldots, x_n :: t_n \vdash (\textbf{have}\ \langle a \rangle\ A\ \text{:discharge}\ [x_1\ \cdots\ x_n\ t]) \dashv \Gamma, \langle a \rangle \triangleleft t' :: A}\ (hyp)$$

Table 6: The semantics of LaTTe proof scripts

H. $(\mathsf{P} \implies \mathsf{Q}) \wedge (\neg\mathsf{R} \implies \neg\mathsf{Q})$	
⟨a⟩ $\mathsf{P} \implies \mathsf{Q}$	$\wedge$ **Elim:**H
x. P	
⟨b⟩ Q	$\implies$ **Elim:**⟨a⟩, x
⟨c⟩ $\neg\mathsf{R} \implies \neg\mathsf{Q}$	$\wedge$ **Elim:**H
Hr. $\neg$R	
⟨d⟩ $\neg$Q	$\implies$ **Elim:**⟨c⟩, Hr
⟨e⟩ Q	**Repeat:**⟨b⟩
⟨f⟩ R	**Absurd:**Hr
⟨g⟩ $\mathsf{P} \implies \mathsf{R}$	$\implies$ **Intro:**x, ⟨f⟩

Table 7: A Fitch-style proof (from [11])

As discussed in [5], a proof should be not just a *certificate of truthness* but also, and perhaps most importantly, an *explanation* about the theorem under study. Proof assistants that use a tactic language (such as Coq or HOL) do not produce readable proofs. To understand the proof, one generally has to replay it step-by-step on a computer. A language such Isabelle/Isar allows for declarative proof scripts, that with some practice can be read and understood like classical proofs on papers. However Isar is arguably a complex an rather *ad hoc* language, with only informal semantics. The domain specific language (DSL) for declarative proof scripts in LaTTe is in comparison very simple. It is an implementation, in the context of type theory and LaTTe, of *fitch-style* natural deduction proofs [11], and is thus deeply rooted in logic and proof theory. The syntax of the proof language is very concise (cf. Table 5) and with simple and formal semantics (cf. Table 6).

As an illustration, we consider the following proposition:

$$\phi \equiv ((P \implies Q) \wedge (\neg R \implies \neg Q)) \implies (P \implies R)$$

A natural deduction proof of ϕ, said in Fitch-style and adapted from [11], is given in Table 7. We will now see how to translate such a proof to the LaTTe proof language. Initially, the environment Γ contains at least the theorem to prove, but without a type, i.e. something of the form: $thm(P :: \star, Q :: \star, R :: \star)\triangleleft? : \phi$. The context Δ contains the three bindings: $P :: \star, Q :: \star, R :: \star$

The beginning of our LaTTe proof is as follows:

```
(proof thm :script
  (assume(H (and (⟹ P Q)
                 (⟹ (not P) (not Q))))
    ...
```

According to rule (*glob*) of Table 6, the hypothesis H and its type (the stated proposition) is introduced in the context Δ. The term u generated by the body of the **assume** block will be propagated. The first step is as follows:

```
... continuing
(have ⟨a⟩ (⟹ P Q) :by (p/and-elim-left% H))
...
```

The justification (and-elim-left% H) is a **defspecial** that will be discussed more precisely in the next section. But it is simply a function that takes the proof of a conjunction and generates the proof of the left operand of the conjunction. The result of a **have** step, handled by the rule named (*by*), is to add a new definition to the environment Γ of the form:

$$\langle a \rangle \triangleleft t :: (\implies P\ Q)$$

with t the left-elimination of assumption H. Of course, this only works if the type-checker agrees: each **have** step is checked for correctness.

Ultimately, each accepted step is recorded as a local theorem recorded in Γ.

The hypothesis x of type P is assumed and in the next steps we have:

```
... continuing
(assume [x P]
  (have ⟨b⟩ Q :by (⟨a⟩ x))
  (have ⟨c⟩ (⟹ (not R) (not Q))
        :by (p/and-elim-right% H))
    ...
```

Now, still through rule (*by*) the environment Γ is extended with definitions $\langle b \rangle$, obtained by applying x on $\langle a \rangle$, and $\langle c \rangle$, obtained by right-elimination of H. For the moment we remain very close to the original Fitch-style proof. In the next step, the objective is to perform a *reductio ab absurdum*. We first state $\neg R$ and derive a contradiction from it. This gives:

```
... continuing
(assume [Hr (not R)]
  (have ⟨d⟩ (not Q) :by (⟨c⟩ Hr))
  (have ⟨e⟩ absurd :by (⟨d⟩ ⟨b⟩))
  (have ⟨f1⟩ (not (not R))
        :discharge [Hr ⟨e⟩]))
...
```

In the type theory of LaTTe, the proposition (not P) corresponds to (⟹ P absurd) with absurd an type without inhabitant, classically: $(\forall\ [A\ \star]\ A)$. Hence after having obtained (not Q) through step $\langle c \rangle$ we obtain step $\langle e \rangle$. The :discharge step $\langle f1 \rangle$ corresponds to the generation of a lambda term of the form: (λ [Hr (not R)] ⟨e⟩) hence a term of type (==> (not R) absurd), thus (not (not R)). Since we discharged the hypothesis Hr we can close the corresponding **assume** scope[8].

In the Fitch-style proof at step $\langle f \rangle$ we deduce R by contradiction. This reasoning step can only be performed in classical logic. In fact the proposition (⟹ (not (not R)) R) is equivalent to the axiom of the excluded middle, and is thus classical in essence. In LaTTe, we must rely on the classic namespace to perform the corresponding step, as follows.

```
... continuing
(have ⟨f2⟩ R
      :by ((classic/not-not-impl R) ⟨f1⟩))
...
```

At this point we are able to assert the conclusion of the rule, and finish the proof.

```
... continuing
(have <g> (==> P R) :discharge [x ⟨f2⟩]))
(qed ⟨g⟩))))
```

The term synthetized at step $\langle g \rangle$ is propagated to the (*script*) rule using the rule (*qed*). Finally, the type-checking problem $\langle g \rangle :: \phi$ is decided, which leads to the acceptation or refutal of the proof. Hence, the natural deduction proof script is only to elaborate, step-by-step, a proof candidate. Ultimately, the type-checker will decide if the proof is correct or not.

5. PROOF AUTOMATION, THE LISP WAY

As illustrated in the previous section, each **have** step of the form (have ⟨a⟩ A :by t) involves the following chain of events:

1. stating a proposition as a type A
2. finding a candidate term t
3. checking that the term t effectively has type A

In the normal usage, the user needs to perform steps 1 and 2, and LaTTe automatically performs step 3. In some situations, the user can benefit from the LaTTe implementation to either state the proposition, or even receive help in finding a candidate term.

Given the term t, the type inference algorithm of Table 3 may be used to obtain proposition A automatically. The syntax of such a proof state is: (have ⟨a⟩ _ :by t). In many situations, it is not recommended because it may make a proof unintelligible, however sometimes this is useful to avoid redundancies in the proofs.

The most interesting situation is the converse: when the proposition A is known but it remains to find the candidate term t. The term synthesis problem is not decidable in general, but it is of course possible to help in the finding process.

The LaTTe proof assistant follows the Lisp tradition of allowing users to write extensions of the system in the host language itself (namely Clojure). This is the purpose of the **defspecial** form that we introduce on a simple example.

The left-elimination rule for conjunction is declared as follows in LaTTe:

```
(defthm and-elim-left "..."
  [[A :type] [B :type]]
  (==> (and A B)
       A))
```

when using this theorem in a **have** step, one needs to provide the types A and B as well as a proof of (and A B), i.e. something of the form:

```
(have ⟨a⟩ A :by ((and-elim-left A B) p))
```

with p a term of type (and A B). But if p has a conjunction type, then it seems redundant having to state propositions A and B explicitely. This is where we introduce a special rule and-elim-left% as follows.

```
(defspecial and-elim-left% "..."
  [def-env ctx and-term]
  (let [[status ty]
             (type-of-term def-env ctx and-term)]
    (if (= status :ko)
      (throw (ex-info "Cannot type term." ...))
      (let [[status A B]
            (decompose-and-type def-env ctx ty)]
        (if (= status :ko)
          (throw (ex-info "Not an 'and'-type." ...))
          [(list #'and-elim-left A B) and-term])))))
```

A **defspecial** is similar to a regular Clojure funtion, except that it may only be called during a **have** proof step. It receives as arguments the current environment and context as well as the argument calls. In the case of the left elimination rule, only one supplementary argument is passed to the special: the term whose type must be a conjunction (parameter and-term in the code above). In the first step, the type of the term is calculated using the inference algorithm. If a type has been successfully derived, an auxiliary function named decompose-and-type analyzes the type to check if it is a conjunction type. If it is the case then the two conjuncts A and B are returned. Ultimately, a **defspecial** form must either throw an exception or return a synthetized term. In our case, the non-special term ((and-elim-left A B) and-term) is returned.

In a proof, the **have** step for left-elimination is now simpler:

```
(have ⟨a⟩ A :by (and-elim-left% p))
```

This is only a small example, there are many other use of specials in the LaTTe library.

The **defspecial** form is quite expressive since the computational content of a special can exploit the full power of the host language. We might wonder if allowing such

[8]In the current version of LaTTe the :discharge steps are performed automatically when closing the assume blocks. Thus, they do not have to (and in fact cannot) be written by the user.

computations within proofs is safe. Thanks to the ultimate type-checking step, there is no risk of introducing any inconsistency using specials. In fact the only "real" danger is to introduce an infinite loop (or a too costly computation) in the proof process. But then the proof cannot be finished so we are still on the safe side of things.

6. DISCUSSIONS

In this section we discuss a few common features of proof assistants, and the way they are supported (or not) in LaTTe.

Implicit arguments

Proof assistants such as Coq [13] and Agda [2] allow to make some arguments of definitions *implicit*. The idea is that such arguments may be filled automatically using a unification algorithm. The advantage is that the notations can thus be simplified, which removes some burden on the user. A first drawback is that because higher-order unification is not decidable, it is sometimes required to fill the arguments manually. Moreover the implicit arguments may hide some important information: it is not because an argument can be synthesized automatically that it is not useful in its explicit form. In LaTTe all arguments must be explicited. However, it is possible to refine definitions by partial applications. For example, the general equality predicate in LaTTe is of the form `(equal T x y)`, which states that `x` and `y` of the same type `T` are equal. In the arithmetic library[9], the equality on integer is defined as follows:

```
(definition =
  "The equality on integers."
  [[n int] [m int]]
  (equal int n m))
```

Hence, we can write `(= n m)` instead of `(equal int n m)` when comparing integers. Thanks to the powerful namespacing facilities of Clojure, partial application becomes a good alternative to implicit arguments. And there is no requirement for a complex unification algorithm.

Holes in proofs

The proof assistant Agda [2] allows to put holes (i.e. unification variables) in proof terms, which gives an alternative way to perform proofs in a stepwise way. Such a partial proof can be type-checked (assuming the holes have the correct type), and suggestions (or even completions) for holes can be provided by a synthesis algorithm. For the moment, LaTTe does not integrate such a feature but it is planned for the next version of the proof engine. The idea is to reject a proof but return possible mappings for the holes.

Inductives and Σ-terms

In Coq [13] and Agda [2] the term languages is much more complex than that of LaTTe. In particular *inductives* and Σ-terms are proposed. It is then much easier to introduce inductive types and recursive functions in the assistants. Moreover, this gives a way of performing *proofs by (recursive) computation*. The main disadvantage is that the uniqueness of typing is lost, and of course the underlying implementation becomes much more complex. Moreover, *universe levels* must be introduced because inductives do not seem to deal well with *impredicativity*. In LaTTe we adopt the approach of Isabelle [9] and HOL-light [6] (among others) of introducing inductive sets and recursion as mathematical libraries. Proof automation is then needed to recover a form of proof by computation. In LaTTe we just started to implement inductive sets and recursion theorems [10]. The next step will be to automate recursive computations using **defspecial** forms. The Σ-types are much easier to implement than inductives. They offer a way to encode *subsets*, i.e. a term $\Sigma x : T.P(x)$ is the subset of the elements of type T that satisfies the predicate P. This is not needed in type theory and LaTTe since such a subset can simply be coded by a predicate $(\lambda\ [x\ T]\ (Px))$. We haven't found any strong argument in favor of Σ-types in the literature.

User interfaces

Most proof assistants are provided with dedicated user interfaces, in general based on an extensible editor such as Emacs. An example of such an environment is *Proof general* [1] that is working with Coq and was also working with Isabelle until version 2014. Proof general was also working with other proof assistants, but support has been dropped. The major weak points are maintainability and evolvability. There is in general much more motivation to work on the kernel of a proof assistant rather than its user interface. The user interfaces for proof assistants can be seen as *live-coding environments*. In most Lisps, and of course Clojure, development environments are designed for a thorough live-coding experience. This observation is one of the two reasons why LaTTe was designed as a library and not as a standalone tool. Our experience is that the Clojure coding environments (Emacs/cider, Cursive, Gorilla Repl, etc.) are perfectly suited for proof assistance. In a way LaTTe has a very powerful interactive environment, maintained by rather large communities, and all this for free!

Proving in the large

The second reason of the design of LaTTe as library is to leverage the *Clojure ecosystem* for proving in the large. Mathematical content can be developed as Clojure libraries, using *namespaces* for modularization. The mathematical libraries can be very easily deployed using Clojars (and Maven) and then used as dependencies in further development. Since all proof forms are macros, the proof checking is performed at compile-time and thus the deployed libraries are already checked for correctness. In this way, although LaTTe is not (yet) a very popular proof assistant, its features for proving in the large are already much more advanced if compared to all the proof assistants we know of. This is of course obtained for free thanks to the way Clojure and its ecosystem is (very thoroughly) designed.

[9] cf. https://github.com/latte-central/latte-integers

[10] cf. https://github.com/latte-central/fixed-points

7. CONCLUSION AND FUTURE WORK

In this paper we described the LaTTe proof asistant in much details. The ways a dependent type theory might be implemented in practice is not very often described in the litterature, a notable exception being [7]. LaTTe can be seen as a rather minimalistic implementation of a proof assistant based on type theory. It is not, however, just a toy implementation for demonstration purpose. It has been used, and is used, to formalize various theories such as a part of typed set-theory, the foundations of integer arithmetic and some developments about fixed points and inductive sets. These are available on the projet page[11].

Beyond the formalization of important parts of mathematics (especially the real numbers), we have a few plans concerning the implementation itself. The terms manipulated in type theory can become quite large in the case of long proofs. This is a rather sparsely studied aspect of type theory, as most of the implementation aspects. We already experimented a more efficient term representation, but the performance gains were limited and the price to pay – give up the internal Lisp representation – much too high. We also introduced a memoization scheme for the type inference algorithm (which is a known bottleneck) but the ratio memory increase vs. CPU gains is not very good. The best way to circumvent this performance issues is to split the proof in separately-compiled subproofs. An automatic proof splitting algorithm is under study. Note, however, that these performance issues only occur at compile-time because this is when the proofs are checked for correctness. This has no impact when using the mathematical libraries. Most of the other planned features revolve around higher-order pattern matching and (partial) unification. One functionality that would then be possible is the notion of proof refinement using holes. This would also enable the development of search algorithms for theorems.

[11] https://github.com/latte-central

8. REFERENCES

[1] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS 2000*, volume 1785 of *LNCS*, pages 38–42. Springer, 2000.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - A functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[3] A. Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.

[4] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[5] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[6] J. Harrison. HOL Light: An overview. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[7] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.

[8] R. Nederpeldt and H. Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[11] F. J. Pelletier and A. P. Hazen. A history of natural deduction. In *Logic: A History of its Central Concepts*, volume 11 of *Handbook of the History of Logic*, pages 341–414. Elsevier, 2012.

[12] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism,*. Elsevier, 2006.

[13] The Coq development team. The coq proof assistant. https://coq.inria.fr/.