A Lisp way to Type Theory and Formal Proofs

a D omain S pecific L anguage Study

Frédéric Peschanski
UPMC - LIP6 - Complex - APR team

Me, myself and I

Associate professor at the University Pierre et Marie Curie

- Research : formal methods, concurrency, automata, combinatorics
- Teaching : programming languages (including Clojure, Ocaml, Scheme)

Amateur programmer and free software enthusiast

- Polyglot : Lisps (for sure !), Ocaml, Haskell, Scala, Java, C++, Python, etc.
- Ex.: LaTTe, cl-jupyter, arbogen, Tikz-editor, pave, piccolo ... (cf. github)

Lisp background

- PhD thesis (co-)advised by Christian Queinnec ⇒ Scheme by “name”
- Programming language programmer ⇒ (Common) Lisp by “value”
- Community member (and converted to FP) ⇒ Clojure by “need”

A LaboratoryofTypeTheoryexperiments https://github.com/latte-central

a Proof Assistant

- Formalize mathematical content (definitions, axioms, theorems, ...) on a computer
- Assist in proving theorems

implemented as a Clojure library

- Small purely functional kernel based on type theory
- "Live coding mathematics" experience (using e.g. cider)
- Proving in the large (compile-time type checking, clojars ecosystems)

and some basic mathematical content

- Integer arithmetics, typed set theory, fixed points theorems (more to come)

In this presentation

LaTTe from the developer point of view

- Proof assistants = a kind of a “deep” D$_{\text{omain}}$ S$_{\text{pecific}}$ L$_{\text{anguage}}$
- (enriched) Lisp as a universal (e.g. mathematical) notation
- The Clojure way : small purely functional kernel, data-oriented, a sip of macros, prog. in the large, ...

LaTTe for the user ?

⇒ cf. LaTTe@Eucoclojure2016 and latte-central on github

Disclaimer

Other proof assistants are much more advanced

- LaTTe is (mostly) a personal project (with a few contributors) focusing on minimalism
- it is aimed at enthusiasts of the Lisp notation (with Clojure enhancements)
  ⇒ most mathematicians favor the (highly informal) “standard” mathematical notations (including all the quirks, ambiguities, historical incidents, ...)

Many of the underlying ideas come from the (excellent) book :

- Type Theory and Formal Proof : an Introduction
  Rob Nederpelt and Herman Geuvers
  Cambridge University Press - 2014

Example: a natural deduction proof

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

(from Francis Jeffry Pelletier, Allen P. Hazen: A History of Natural Deduction, 2012)

... in Coq

Coq is a famous, successful proof assistant

Some notable features :

- External DSL implemented in Ocaml
- based on a very rich type theory (universes, inductives, sigma-types, etc.)
- (thus) has a rather complex kernel implementation
- supports a complex notation system
- use tactic-based imperative proof scripts
- user-defined tactics are written in a dedicated DSL (LTac)
- plugins can be implemented in Ocaml (but it is not for “normal” users)

… in LaTTe

In comparison, some LaTTe features :

- Internal DSL with Clojure as host
- based on a very simple, less expressive, type theory
- (thus) has a very small kernel implementation
- uses the Lisp notation for mathematical contents (with Clojure extensions)
- use declarative proof scripts based on fitch-style natural deduction.
- can be extended (in various ways) directly in the host (Clojure) language.

Claim:

We do *not* claim that LaTTe is better, only smaller and lispier …

Existential question

What makes a DSL

?

Beautiful Domain Specific Languages : a personal Agenda

- An interesting and rich domain
  ⇒ e.g.: html (imho) cannot be a beautiful DSL !
- Internal DSL
  ⇒ proving is programming, programming is proving
- Declarative-first
  ⇒ faithfully convey the domain principles
- (but) Programmable/extendable in the host language
  ⇒ example of (what I think is) a counter-example : syntax-rules ...
- Small kernel
  ⇒ core abstractions vs. “sugars”
- Macros (only) when required
  ⇒ with great power comes great … but hey : GREAT POWER !
- The Clojure way: data-oriented ⇐ new !
  ⇒ for me an important piece of the “Lisp programming puzzle” ...

Proof steps

A proof step in LaTTe is of the form:

```
(have <step> (some proposition P) :by (some proof of P))
```

- (some proposition P) is a type T
- (some proof of P) is a term u

Principle (Curry-Howard) : u has type T ⇔ u is a proof of T ⇔ proposition T is true

⇒ LaTTe first infers a canonical type U of u
⇒ the proof <step> is accepted iff U and T are (β-)equivalent
⇒ in this case <step> becomes a local variable of value u and type T.

Proof automation : synthesize propositions ?

A proof step in LaTTe may also by of the form:

```
(have <step> _ :by (some proof of P))
```

- (some proof of P) is a term u

⇒ LaTTe first infers a canonical type U of u
⇒ the proof <step> is accepted if U is a valid type (term of type ★)
⇒ in this case <step> becomes a local variable of value u and type U.

<u>Remark</u> : such a proof step is not declarative (but can help reduce redundancy)

Proof automation : synthesize terms ?

A proof step in LaTTe is of the form:

```
(have <step> (some proposition P) :by ???)
```

- (some proposition P) is a type T

Question: is there a term `t` of type T? (⇔ is type T inhabited?)

This problem is (thankfully!) not decidable in the general case.
But such a term `t` can be generated programmatically using `defspecial`.

Remark: any term `t` will do, as long as it has type T ⇐ proof irrelevance

Proof automation: the `defspecial` form

```
(defspecial auto%
  [def-env ctx arg1 … argN]
  <arbitrary Clojure code to generate
   a term t of the expected type>
  … )
```

Using a special in a proof step:

```
(have <step> (some proposition P) :by (auto% arg1 … argN))
```

Ultimately type-checking ensures correctness of the automated step
(if `auto%` terminates)

Proof generation ? (ongoing development)

What if you want to generate proofs (or parts of proofs) programmatically ?

⇒ LaTTe proofs are macro-calls:

```
(proof <myproof> :script
   (assume [H1 <blabla>
            H2 <blablabla>]
     (have <a> (good prop P) :by (nice proof p))
     (have <b> (good prop Q) :by (nice proof q))
     …
    …)
 …)
```

⇒ can quote/quasiquote proofs, but it's rather cumbersome to manipulate proofs as lists …
(at least in Clojure) and we lose the benefits of macros (transparency)...

Proof generation: the Clojure (data-oriented) way

Alternative proof representation:

```
[:proof <myproof> :script
   [:assume ['H1 <blabla>
             'H2 <blablabla>]
     [:have '<a> (good prop P) :by (nice proof p)]
     [:have '<b> (good prop Q) :by (nice proof q)]
     …
    …]
 …]
```

⇒ This is a Clojure literal, very easy to generate/manipulate programmatically

Example: the hence form

```
(defmacro hence [prop by proof]
   [:have (gensym "hence") ~prop ~by ~proof])
```

An unexpected limitation (for compile-time type checking):

The literal representation must be generated in a bottom-up way
⇒ the macro-expander works the other way around

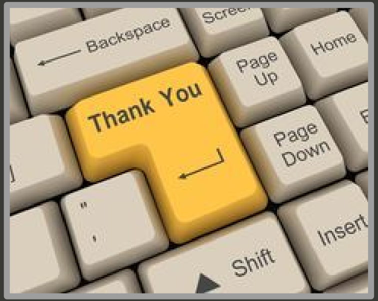
Hence we need a user-level macro-expander
⇒ clojure.core/macroexpand (usable but somewhat limited)
⇒ ridley : a powerful code walker/macro-expander as a library

Conclusion

Lisp and Clojure rock ! (you don’t say …)

- A beautiful universal notation
  (mathematical concepts are just an example)
- Programming as a generalized computer interaction principle
  (doing mathematics is just an example)
- Macros “rock-” ... data-oriented macros “-abilly”

Type theory is a beautiful and rich domain !



Want to try ?

⇒ https://github.com/latte-central/LaTTe