Ltac2: Tactical Warfare

Pierre-Marie Pédrot
INRIA
Nantes, France
pierre-marie.pedrot@inria.fr

Abstract

We present Ltac2, a proposal for the replacement of the Ltac tactic language that is shipped with Coq as the default interface to build up proofs interactively. Ltac2 is primarily motivated by two antagonistic desires, namely extending the expressivity and regularity of the historical tactic language of Coq while maintaining a strong backward compatibility. We thereafter give a bird's eye view of the features and semantics of the current state of Ltac2.

Keywords Coq, Tactic Language, Ltac

1 Introduction

Introduced by Delahaye [2] in Coq 7.0, the Ltac tactic language is probably one of the major ingredients of the success of this proof assistant, as it allows to write proofs in an incremental, more efficient and more robust way than the state of the art of that time. Before Ltac, Coq users either had to write out the proof terms explicitly or had to rely on a very primitive set of tactics that were performing the basic rules of reasoning, without a way to compose them. Ltac provided a way for users to build their own, new proof abstractions by glueing together atomic tactics thanks to an expressive set of combinators. This in turn allowed the development of large pieces of formalized proofs in Coq.

Nonetheless, Ltac has somewhat outgrown the setting it had been designed for. It suffers for a lack of planning ahead for the language, and its features have been added piecewise without a clear concern for uniformity and coherence, leading to a very ad-hoc feel and an overly intricate implementation. Similarly, the Ltac development tried to reconcile two contradictory properties, namely that tactics ought to be at the same time *automagical* and *predictible.* This puts the scalability of the language at stake, a fact that was highlighted as early as the French version of the Coq 7.0 `CHANGES` file.

As the limitations of Ltac are becoming more and more pressing, this has been the root of several recent research lines on the conception of new tactic languages. Let us cite Mtac [9] and Mtac2 [3], Rtac [4] and Template-Coq [1], and Coq-Elpi [8]. Each of these projects take a different standpoint about what a tactic language should look like.

In this blossoming context, we propose Ltac2, an evolution of the original Ltac (henceforth called Ltac1) that should fix most of its technical defects while being as conservative as possible. Ltac2 is an ML: it is typed, has extensible datatypes and battle-tested semantics. Ltac2 can be installed as a plugin for Coq from https://github.com/ppedrot/ltac2.

CoqPL'19, January 13–19, 2019, Lisbon, Portugal
2019.

2 General Design Choices

Ltac2 is essentially a wrapper around the *proof engine* that was introduced in Coq 8.5 [7]. At its heart, it is the OCaml implementation of a monadic type α `tactic` equipped with the minimal API allowing to interact with the proof state [6].

The monadicity of the `tactic` type allows for a systematic language construction atop of it known as Moggi's computational λ-calculus [5]. Ltac2 is the direct result of this process. As such, it is a member of the ML family of languages, i.e. Ltac2 is call-by-value and effectful, supports algebraic datatypes and allows prenex polymorphism.

On top of this sane base language, we add built-in features for Coq metaprogramming together with a rich notation system that allows to emulate most of Ltac1 facilities. We describe more in depth every point in the remainder of this extended abstract.

3 ML Component: Types

There is nothing much to say here. As mentioned above, Ltac2 provides the standard type constructions from ML, namely prenex polymorphism and the ability to define algebraic datatypes. This is already a great step forward compared to Ltac1, that merely has a second-class support for a handful of data structures like lists, and, to add insult to injury, is dynamically typed.

In addition to common base types like integers and strings, Ltac2 provides Coq-specific abstract types like terms, identifiers, constant references and so forth.

Contrarily to languages like Mtac2, the type system of Ltac2 does not enforce much, and many sanity checks are performed at runtime. This is imposed by the backward compatibility constraint.

4 ML Component: Dynamics

Ltac2 itself is written in OCaml, and is currently interpreted rather than compiled to OCaml. In particular, through Moggi's construction, an Ltac2 value $f : \alpha \to \beta$ is intuitively translated into an OCaml function of type $[\![\alpha]\!] \to [\![\beta]\!]$ `tactic` where $[\![\cdot]\!]$ stands for the runtime representation of the corresponding type. Argumentless Ltac1 tactics can be given the type `unit` $\to$ `unit` in this setting. All ML equations are validated, thus it is sufficient to understand the additional expressivity given by the underlying monad. We give a non-exhaustive list of effects below.

General IO Programs can print messages, modify mutable memory and raise dynamic exceptions, called *panics*.

Proof state Programs can modify the global unification state, i.e. a DAG-like structure of terms with holes by filling them with partial proof-terms.

Goals Programs can modify the set of goals under focus. Contrarily to Ltac1, there can be several goals at once, or none. Many primitives expect that there is exactly one goal under focus and will panic dynamically if it is not the case.

Backtrack In order to support logic programming out of the box, Ltac2 has a built-in backtrack effect. It means that in addition to the above effects, a thunk `unit` $\to \alpha$ can be seen as a stream of results of type α. Both the proof state and the goal state are synchronized with this backtrack, but the IO effects, including panics, are not.

These effects allow to give concise types to complex primitives that are interacting with the proof engine. For instance, we can type `refine : (unit → constr) → unit`. Intuitively, it is a function so it modifies the proof state, and furthermore it takes a thunk as argument, because it needs to know the holes generated by it so as to add them to the list of goals to be proven. Another example is the primitive function `hyp : ident → constr` that gives the hypothesis with the provided name if there is exactly one goal under focus or panics otherwise.

5 Metaprogramming

Ltac2 features a `constr` type that stands for Coq terms. In addition to low-level manipulation functions that give full access to the kernel representation, there is a lot of syntactic sugar to seamlessly manipulate terms. As in Ltac1, one can quote a term using the `constr:(...)` syntax.

Contrarily to Ltac1, there is a strict separation between antiquotations and term variables in the context of a term quotation. One has to explicitly write `$x` to refer to the Ltac2 variable `x` bound outside of the quotation, while `x` refers to a Coq term variable introduced inside the quotation by a term binder. Both are checked statically.

Sometimes, the fact that a variable is in the goal scope cannot be detected statically, because it relies e.g. on the invocation of the tactic in some specific context. This recurring pattern was given a shorthand syntax of the form `&x` which is just sugar for `hyp ident:(x)`.

Once again, the ML type system is completely useless when it comes to checking that a Coq term is well-typed. Just as in Ltac1 this is worked around by a heavy dosis of runtime checks. Unsafe primitives are also provided for efficiency, at the cost of potential runtime anomalies.

6 Macro System

This is the crux of the backward compatibility w.r.t. Ltac1. In a nutshell, Ltac2 comes bundled with a macro system that allows to extend the syntax straightforwardly. Contrarily to the infamous `Tactic Notation` feature from Ltac1, this system is composable by construction. The idea is that notations get elaborated into data structures at *parsing time*, when Ltac1 does so at runtime because it lacks explicit quotations. The basic ingredient of a macro is called a *scope*, which is just a meta function expanding syntax on the fly.

For instance, the `ident` scope has two parsing rules:

- identifiers `id` are turned into `ident:(id)`
- quotations `$x` are turned into the Ltac2 variable `x`

That is, by defining some notation

```
Ltac2 Notation "foo" id(ident) := e
```

this makes Ltac2 perform the following expansions:

```
foo bar  ⇝  let id := ident:(bar) in e
foo $x   ⇝  let id := x in e
```

By leveraging the existence of data structures, Ltac2 also provide scope combinators that expand parsing entries into compound terms. For instance, the `list0` scope transformer takes one scope argument *s* and parses a list of *s* and turns it into the corresponding Ltac2 list. That is, assuming

```
Ltac2 Notation "quz" ids(list0(ident)) := e
```

we obtain the expansion below:

```
quz bar $x qux $y  ⇝
  let ids :=
    [ident:(bar); x; ident:(qux); y] in e
```

More complex scopes are available to the user, e.g. intro patterns are desugared into an algebraic datatype that can be readily used to implement homemade variants of introduction tactic. More generally, this mechanism is heavily used to provide many built-in constructs from Ltac1 as syntactic sugar for mundane ML code, e.g. `match goal`.

Thanks to the similarity of these macros with Ltac1 surface syntax, semi-automated translation was shown to be possible on simple enough proof files. Most of the changes are either minute syntactic variations or quotation errors caught by the type-checker.

7 Conclusion and Future Work

Ltac2 is still in an active development phase, but the foundations of the language have been settled. More than anything, it is in need of users in order to polish the rough edges. While not groundbreaking like other tactic language proposals, Ltac2 should eventually alleviate the pain of the maintenance of Ltac1 and provide a complete way to write low-level code without resorting to OCaml plugins.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

[2] David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*. 85–95. https://doi.org/10.1007/3-540-44404-1_7

[3] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: Typed Tactics for Backward Reasoning in Coq. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming, ICFP 2018, St Louis, USA, September 23-29, 2018*.

[4] Gregory Malecha and Jesper Bengtson. 2016. Extensible and Efficient Automation Through Reflective Tactics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 532–559. https://doi.org/10.1007/978-3-662-49498-1_21

[5] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[6] Pierre-Marie Pédrot. [n. d.]. Ticking like a Clockwork: the New Coq Tactics. http://coqhott.gforge.inria.fr/blog/coq-tactic-engine/

[7] Arnaud Spiwack. 2010. An abstract type for constructing tactics in Coq. In *Proof Search in Type Theory*. Edinburgh, United Kingdom. https://hal.inria.fr/inria-00502500

[8] Enrico Tassi. 2018. Elpi: an extension language for Coq. In *Fourth International Workshop on Coq for Programming Languages*.

[9] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: a monad for typed tactic programming in Coq. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 87–100. https://doi.org/10.1145/2500365.2500579