Monadic typed tactic programming by reflection*

Liang-Ting Chen
Department of Computer Science
Swansea University
Swansea, Wales, United Kingdom
liang.ting.chen.tw@gmail.com

Abstract

We present a work in progress—a shallow embedding of a typed tactic language *Mtac* using *elaborator reflection* in a dependently typed language to allow users to write high-level tactics within the same language. In contrast to the original implementation of Mtac in Coq, this implementation is completely written in Agda using its reflection mechanism. To focus on the difference from its Coq counterpart, we give an example of tactics and briefly sketch the implementation of the core design and the pattern matching construct.

Keywords reflection, elaboration, tactics, meta-programming, interactive theorem proving, agda, monads

1 Introduction

Mtac is a typed tactic programming language in Coq proposed in [4, 10] to provide a high-level tactic programming with static guarantees and to develop tactics interactively within the Coq development environment. The idea is to encapsulate tactics in a *monad* $\bigcirc$ with a primitive `run` which takes a term of type $\bigcirc\tau$, read as "maybe τ", to execute during type inference. The execution will either produce a value of type τ or an uncaught exception if it terminates.

In order to run tactics during type inference, their implementation is an extension partly written in OCaml changing the infrastructure. It also relies specifically on impredicativity and typical ambiguity.

It is posed by Christiansen and Brady [1] that elaborator reflection in Idris could be used to implement Mtac. To the best of our knowledge, however, such implementation does not exist yet.[1]

Idris's reflection has inspired Ulf Norell to implement a new reflection framework for Agda [8]. For a long time, Agda as a proof assistant has lacked *custom* proof automation, but it has a rich set of language features to support a succinct programming style. Also, it seemed challenging to implement Mtac due to the difference on the universe design. Hence, Agda is chosen to implement Mtac as an interesting case study of elaborator reflection.

*The source code for the work presented here and few more examples are available to download from https://github.com/L-TChen/MtacAR.

[1] There are certainly other typed tactic languages. See [4, 10] for comparison.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

2 A trivial example

Due to the page limit, we demonstrate a somewhat trivial tautology solver taken from [10] with our syntax:

```
{-# TERMINATING #-}
prop-tauto : {P : Set} → ◯ P
prop-tauto {P} = mcase P of
  | ⊤                  ⇒ ⦇ tt ⦈
  | P : _ , Q : _ , P × Q ⇒ ⦇ prop-tauto , prop-tauto ⦈
  | P : _ , Q : _ , P ⊎ Q ⇒ ⦇ inj₁ prop-tauto | inj₂ prop-tauto ⦈
  | P                  ⇒ ⦇⦈
  end
```

The type `Set` is not inductively defined, so the termination checker is not able to decide whether the recursive tactic terminates or not. Hence we switch off the termination checker for it, but it is okay—tactics are executed during type inference and the generated values will be checked.

The tactic starts with an `mcase` construct—pattern matching for *arbitrary* expression which works as expected except that variables are introduced explicitly with type annotation before being used in the pattern. Each case is easy to follow:

1. If P is the unit $\top$, it returns the constructor `tt`.
2. If P is a product $P \times Q$ for some P and Q, it returns a pair of values found by the tactic.
3. If P is a sum $P \uplus Q$, it searches for a value of the left disjunct. If it fails, then it proceeds with the right disjunct.
4. If none of above is matched, then it returns nothing.

Note that there are no explicit arguments or type annotations, as they will be inferred by unification in this particular case.

The idiom bracket notation $⦇ \ldots \mid \ldots ⦈$ with possibly empty choices is a syntax sugar proposed in [5, 6] for applicative functors equipped with a monoidal structure. Its use with mixfix notation [2] is essential for readability. Otherwise, using the `do`-notation as in [10], the second case needs to thread arguments explicitly:

```
| P : _ , Q : _ , P × Q ⇒ (do
  x ← prop-tauto
  y ← prop-tauto
  return (x , y))
```

Instance arguments [3] are used to infer which `Applicative` functor and which `Monad` are being used when invoking idiom brackets, the `do`-notation, etc. Instead the original

Mtac throws an exception using `mtry` when no value is found. As `mtry` is prefixed by `m`, the users need to remember the naming difference if a similar construct has existed already.

Executing a tactic is simple. A tactic is placed in a term position and executed by a *macro* (see below). The following

```
solve : ⊥ ⊎ (⊤ ⊎ List N) × ⊤
solve = Proof prop-tauto ■
```

will have a term $\mathsf{inj}_2\ (\mathsf{inj}_1\ \mathsf{tt}\ ,\ \mathsf{tt})$ filled by the type checker.

The reader who is familiar with Mtac may notice that there is one construct missing in our example—`mfix`. In Coq it is used to define a recursive tactic, but a macro can just call any other function during execution.

3 From the TC monad to the ◯ monad

The elaborator reflection in Agda (and in Idris) is manipulated through the type checking monad

```
TC : Set ℓ → Set ℓ
```

which encapsulates states such as contexts for metavariables. Our ◯ monad is defined as ◯ A = `TC Tac` for every A where

```
data Tac : Set where
  term : Term      → Tac
  error : Exception → Tac
```

and `Term` is the type for the reflected syntax using the de Bruijn index. In [4, 10], ◯ is defined as a *predicate* whose target is the impredicate universe `Prop` to address the circularity when defining `mfix`. Agda does not have an impredicate universe but there is no need to define `mfix` either.

The `return` and `bind` are defined using the primitives

```
quoteTC   : A → TC Term
unquoteTC : Term → TC A
```

where the first function reifies an abstract value to `Term` and the second function translates a `Term` back to a value.

Elaborator reflection allows the language users to invoke type checking, whnf reduction, and unification explicitly via

```
checkType : Term → Type → TC Term
reduce    : Term        → TC Term
unify     : Term → Term → TC ⊤
```

respectively. They are essential to our implementation.

A macro is a function of type $t_1 \to \cdots \to$ `Term` $\to$ `TC` $\top$ defined in a `macro` block. To run a macro, the last argument `Term` is supplied by the type checker and will be the metavariable for instantiation by unification to fill in the term position. If a tactic ◯A is `return (term` t`)`, then t will be unified with the metavariable created by the type checker.

Monad laws are proved by postulating that subject only to α-conversion (i) `TC` satisfies monad laws; (ii) `quoteTC` is injective; (iii) `quoteTC` followed by `unquoteTC` is the `return` of `TC`; (iv) for ◯A, inhabitants of `TC Term` are identified if they cannot be differentiated by `unquoteTC` to A.

4 Implementation of `mcase`

An acute reader may have wondered how the `mcase` construct is defined to accommodate arbitrary expression as there is no typical ambiguity in Agda. For this, we use the kind $\mathtt{Set}_\omega$ of universe polymorphic terms and an optional typing rule $\mathtt{Set}_\omega : \mathtt{Set}_\omega$ which breaks consistency.[2] The rule is only used internally and is not used when checking the generated terms, so no harm will be done.

In detail, the following datatype is used to encode *patt*erns:

```
data Patt (P : A → Set ℓ) : Setω where
  Pbase : (x : A)       → ◯ (P x) → Patt P
  Ptele : (C : Set ℓ′) → (C → Patt P) → Patt P
```

The constructor Ptele is used to introduce a variable into the context and Pbase to store a pattern as the first argument and a body as the second argument. For example, the last clause of `prop-tauto` expends to `Ptele` $(\lambda\, p \to$ `Pbase` $p\ (\!|\,|\!))$.

Contrary to the common conception, de Bruijn indices are rarely manipulated directly. For example, splitting a `Patt P` into a patten and a body is defined by

```
split : Patt P → TC (Term × Term)
split (Pbase x px) = (| quoteTC x , quoteTC px |)
split (Ptele C f)  =
  quoteTC C >>= newMeta >>= unquoteTC >>= λ x → split (f x)
```

replacing every variable occurred by a metavariable.

Checking a non-empty list of patterns is a function

```
mcase_of : (x : A) → Patts P (suc n) → ◯ (P x)
```

trying to unify a list of patterns in order against the first argument where `Patts` is a length-indexed list for `Patt`. If no pattern is matched, then an exception will be thrown.

5 Further work

We have exploited elaborator reflection and Agda's recent features to embed a typed tactic language without impredicativity or typical ambiguity.

However, its poor performance is the main defect to overcome, as tactics are interpreted during type inference. It is also an issue shared with Mtac in Coq, though.

A type-theoretic understanding of elaborator reflection and perhaps its connection with (contextual) modal type theory [7, 9] are also interesting to investigate.

Acknowledgments

The author would like to thank Hsiang-Shang 'Josh' Ko for his helpful comments. This research was supported by the EPSRC project Data Release–Trust, Identity, Privacy and Security (EP/N028139/1 and EP/N027825/1).

[2] $\mathtt{Set}_\omega$ and the option `--omega-in-omega` are introduced in the version 2.6.0.

References

[1] David Christiansen and Edwin Brady. 2016. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*, Vol. 51. ACM Press, New York, New York, USA, 284–297. https://doi.org/10.1145/2951913.2951932

[2] Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages. IFL 2008*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Lecture Notes in Computer Science, Vol. 5836. Springer, Berlin, Heidelberg, 80–99. https://doi.org/10.1007/978-3-642-24452-0_5

[3] Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP '11*, Vol. 46. ACM Press, New York, New York, USA, 143. https://doi.org/10.1145/2034773.2034796

[4] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 1–31. https://doi.org/10.1145/3236773

[5] Conor McBride. 2009. Idiom brackets. https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/idiom.html

[6] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 01 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

[7] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (June 2008), 1–49. https://doi.org/10.1145/1352582.1352591

[8] Ulf Norell. 2016. Agda reflection overhaul. https://lists.chalmers.se/pipermail/agda/2016/008414.html

[9] F. Pfenning. 2002. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc, 221–230. https://doi.org/10.1109/LICS.2001.932499

[10] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming* 25 (Aug. 2015), e12. https://doi.org/10.1017/S0956796815000118