# Extended Abstract: Generalization of Meta-Programs with Dependent Types in Mtac2 with Mtac2

Ignacio Tiraboschi
FAMAF, UNC
Córdoba, Argentina
ignatirabo@gmail.com

Jan-Oliver Kaiser
MPI-SWS
Saarbrücken, Germany
janno@mpi-sws.org

Beta Ziliani
FAMAF, UNC and CONICET
Córdoba, Argentina
beta@mpi-sws.org

## 1 Motivation

Meta-languages are becoming an essential part of proof assistants, as they enable the proof developer to automate her proofs. Hence, an increasing number of provers are adopting different meta-languages. In the particular case of the Coq proof assistant, to day there exists a myriad of meta-languages: [1, 3, 5–7]. From these, Ltac [3] is the standard *de facto*, although it is expected that others will quickly catch-up. These languages share something in common: they do not provide static guarantees over the Coq terms they manipulate. In contrast, the Mtac and Mtac2 meta-languages [4, 8] take a different path, coding meta-programs within a monad in Coq to obtain *typed* meta-programs. That is, an Mtac2 meta-program with type `M A` will ensure that the value returned will indeed have type `A`.

However, the combination of monads and dependent types presents an interesting challenge: The *convoy pattern*[2]—an extremely useful and often necessary tool for dependent programming in Coq—is not automatically supported. The convoy pattern is necessary when dependent pattern matching inspects values on which *the types* of other values depend. For example, imagine a function to compute the maximum of a list `l: list nat` whose non-emptiness is witnessed by hypothesis `H: l <> nil`. Seasoned Coq programmers will know that to implement this function, the pattern matching within needs to be generalized over `l` *and* `H`:

```
(fix list_max_nat_pure l : l <> nil → X :=
   match l as l' return l' <> nil → X with
   | [] ⇒ fun H ⇒ match H eq_refl with end
   | [e] ⇒ fun _ ⇒ e
   | (e1 :: e2 :: l') ⇒ fun H ⇒
     let x := Nat.max e1 e2 in
     list_max_nat_pure (x :: l') cons_not_nil
   end) l H
```

The convoy pattern describes the necessity of generalizing over `H`, as `H`'s type is different in every branch. And it is exactly this generalization that causes friction in the monadic setting as can be seen in the type of Mtac2's fixpoint combinator `mfix1`:

```
∀ X (P: X → Type),
((∀  x, M (P x)) → ∀ x, M (P x)) → ∀ x, M (P x)
```

We cannot instantiate `P` to introduce an hypothesis such as `H`: The best we can do is `∀ l, M (l <> nil → X)`, giving us access to the proof only once we return a pure value from within the monad. This restriction applies to all monadic operators in Mtac2 and, thus, prohibits us from applying the convoy pattern as one would in non-monadic code.

Experienced functional programmers might suggest uncurrying the arguments by packing them into a dependent pair. While possible, this approach requires extra care in meta-programs, where pattern matching may distinguish convertible but syntactically different terms.

To avoid this particular pitfall, Mtac2 provides generalized fixpoint and pattern matching operators, written `mfix` and `mtmmatch`, respectively. Using dependent pairs *under the hood*, i. e. transparently, they allow us to write a monadic version of `list_max_nat_pure` in a carefree way.[1]

```
Definition list_max_nat: ∀ l, l <> nil → M nat :=
  mfix f (l: list nat) : l <> nil → M nat :=
    mtmmatch l as l' return l' <> nil → M nat with
    | [? e] [e] ⇒ fun H ⇒ ret e
    | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun H ⇒
      let x := Nat.max e1 e2 in
      f (x :: l') cons_not_nil
    end.
```

However, Mtac2 offers many more monadic operations and users are free to write new ones. Do we need to manually generalize all of them? To answer that question, let us introduce `bind` into our program. To motivate the use of `bind`, we change our program take in lists of *arbitrary type*, using another meta-program, `max`, to compute a suitable comparison function.

```
Definition max (S: Set) : M (S → S → S) :=
```

We would then like to generalize `nat` in `list_max_nat` to an arbitrary set `S`, *bind* the result of `max S` and us it to compute the maximum:

```
Definition list_max (S: Set): ∀ l, l <> nil → M S :=
  max_f ← max S;
  mfix f (l: list S) : l <> nil → M S :=
    ... <use max_f instead of Nat.max> ...
```

[1] `mtmmatch` patterns can bind arbitrary (sub)terms and its patterns use the notation `[? a ... z]` to name these terms.

However, types do not match: Mtac2's `bind` operator (written using the traditional notation `a ← f; g`) has type:

```
bind : forall {A B : Type}, M A → (A → M B) → M B
```

In our example, the `mfix` term returns `∀ l, l <> nil → M S`: the `M` does not occur at the top-most position in the type as `bind` expects it. Thus, without a suitably generalized version of `bind`, we once again cannot *just* apply the convoy pattern.[2]

In this work we present a new meta-meta-program `lift` that provides a semi-automatic solution: given any meta-program or operator (like `bind`) and a list of dependencies (what lies behind the last → in the type of `mfix` above), it generates a new operator that can be used in a context where such dependencies are expected. It is important to mention that **we use Mtac2 as is as its own meta-language!**

## 2 Result

At the moment, we have a working solution that requires the developer to explicitly provide the list of dependencies, even when they can be inferred from the context. We discuss this in section 4.

In this case, we would wish to make `bind` more general. Specifically we wish to get the following function `bind^`.

```
bind^ : ∀{A B : ∀l (H : l <> nil), Type},
  ∀l H, M (A l H) →
  ∀l H, (A l H → M (B l H)) →
  ∀l H, M (B l H)
```

This signature is derived from that of the fixpoint, which gets two arguments `l` and `H` before returning the (monadic) value. Note the following: in our case, the first (non-implicit) argument of `bind^` will be `max S`, and therefore will not make use of the dependencies available. However, if we think of the general case, we have to include them. What this means, in essence, is that `max S` must also be *lifted* to include such dependencies.

In concrete, we can lift the `bind` operator to have a type that matches the expected type for our example by writing:

```
@bind † [t: (l: list S) (_ : l <> nil)]
```

The `@` is Coq's syntax to not insert implicit variables (`A` and `B` for `bind`); the † is notation for the `lift` function; and `[t: a ... z]` is notation for a *telescope* listing the dependencies `a` to `z` (which are binders).

We can then write the new `list_max` program as follow:

```
Definition list_max (S: Set) : ∀l, l <> nil → M S :=
  (@bind † [t: (l: list S) (_ : l <> nil)]) _ _
    (max S † [t: (l:list S) (_ : l <> nil)])
  (fun l (H: l <> nil) (max : S → S → S) ⇒
  (mfix2 f (l: list S) (H : l <> nil) : M S :=
    (mtmmatch l as l' return l' <> nil → M S with
    | [? e] [e] ⇒ ret e † [t: (_ : [e] <> nil)]
    | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun H ⇒
      let m := max e1 e2 in
      f (m :: l') cons_not_nil
    end) H) l H).
```

[2]It is possible to introduce all arguments and beta-expand the fixpoint but we consider this inadvisable for reasons of readability and maintainability.

While it certainly looks verbose, we are certain we can trim down the boilerplate and make it look as a regular 'bind' with two extensions, that we leave for future work: 1) as said before, we know from the context the types the telescope should have, we must be able to construct them automatically; 2) similarly, if we know a parameter is not using the dependencies, like our `max` function above, `lift` should not introduce unnecessary dependencies.

## 3 Technicalities

Under the hood, `lift` is quite involved. The code can be downloaded from

https://github.com/ignatirabo/Mtac2_lift

It is implemented in Mtac2 itself as a recursive function over the signature of the target meta-program, after *reflecting* it in a datatype `TyTree` to allow a proper manipulation. In order to map from and to a Coq type, we define functions `to_ty : TyTree → Type`, which trivially translates a `TyTree` to a Coq type, and the inverse `to_tree : Type → M TyTree`, which pattern matches on the type to obtain the corresponding `TyTree`. Therefore, this last function must be monadic.

The actual lifting occurs in `lift'`. `lift'` takes a function `f`, our lifting candidate, with its encoded type, and the corresponding telescope, then returning a Σ-type, wrapped in the monad `M`, with the new signature and function.

## 4 Conclusion and Future work

At the moment, `lift` has been implemented and is working as expected. We are currently studying different scenarios to find its limitations (we currently have no proof of its completeness). Yet, so far we found it can lift many useful operators, besides of `bind` and `ret`.

Performance wise, `lift` has linear complexity as is a simple recursion over the signature of the target function, and it is in practice rather fast.

In the future, we plan on integrating `lift` into Mtac2 as a standard feature, but for that, we need to improve the notation for lifting to automatically infer the telescope by analyzing the current context. For example, in the `list_max` example from Section 2, we should be able to obtain the returned type `forall l:list S, l <> nil → M S` and construct from it the telescope. Ideally, we should be able to also find that the argument of bind (`max S`) does not make use of the dependencies and avoid generating them for the argument.

What we expect in the end is to be able to use lifted functions and operators as one would do without the unlifted ones when no convoy pattern is required.

## 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 (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

[2] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[3] 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 (Lecture Notes in Computer Science)*, Michel Parigot and Andrei Voronkov (Eds.), Vol. 1955. Springer, 85–95. https://doi.org/10.1007/3-540-44404-1_7

[4] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. *PACMPL* 2, ICFP (2018), 78:1–78:31. https://doi.org/10.1145/3236773

[5] 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 (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 532–559. https://doi.org/10.1007/978-3-662-49498-1_21

[6] Pierre-Marie Pédrot. [n.d.]. *Ltac2*. https://coq.inria.fr/distrib/current/refman/proof-engine/ltac2.html

[7] Enrico Tassi. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi $\lambda$Prolog dialect). (Jan. 2018). https://hal.inria.fr/hal-01637063 working paper or preprint.

[8] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000118