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

Ignacio Tiraboschi  
FAMAF, UNC  
Córdoba, Argentina  
ignatirabob@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 $\forall \alpha . \text{mtac2}$ will ensure that the value returned will indeed have type $\alpha$.

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 inspectes values on which the types of other values depend. For example, imagine a function to compute the maximum:

$\text{max} (S : \text{Set}) : \text{M} (S \rightarrow \text{M} \text{nat}) ;$

$\text{mtmmatch} l \text{as} l \Rightarrow \text{match} l \text{with end} .$$

We cannot instantiate $P$ to introduce an hypothesis such as $H$: The best we can do is $\forall l, M (l \not= \text{nil} \rightarrow \text{x})$, 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 un-currying 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 $\text{mfix}$ and $\text{mtmmatch}$, respectively. Using dependent pairs under the hood, i.e. transparently, they allow us to write a monadic version of $\text{list\_max\_nat\_pure}$ in a carefree way.¹

Definition $\text{list\_max\_nat} : \forall l, l \not= \text{nil} \rightarrow \text{M} \text{nat} :=$

$f (l : \text{list} \text{nat}) : l \not= \text{nil} \rightarrow \text{M} \text{nat} :=$

$\text{mtmmatch} l \text{as} l \Rightarrow \text{fun} H \Rightarrow \text{ret} e$

Definition $\text{max} (S : \text{Set}) : \text{M} (S \rightarrow \text{M} \text{nat}) ;$

$\text{mtmmatch} l \text{as} l \Rightarrow \text{let} x := \text{Nat} \text{.max} e1 e2 \text{in}$

We would then like to generalize $\text{nat}$ in $\text{list\_max\_nat}$ to an arbitrary set $S$, bind the result of $\text{max} S$ and us it to compute the maximum:

Definition $\text{max} (S : \text{Set}) : \forall l, l \not= \text{nil} \rightarrow \text{M} S :=$

We cannot instantiate $P$ to introduce an hypothesis such as $H$: The best we can do is $\forall l, M (l \not= \text{nil} \rightarrow \text{x})$, 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 un-currying 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 $\text{mfix}$ and $\text{mtmmatch}$, respectively. Using dependent pairs under the hood, i.e. transparently, they allow us to write a monadic version of $\text{list\_max\_nat\_pure}$ in a carefree way.¹

Definition $\text{list\_max\_nat} : \forall l, l \not= \text{nil} \rightarrow \text{M} \text{nat} :=$

$f (l : \text{list} \text{nat}) : l \not= \text{nil} \rightarrow \text{M} \text{nat} :=$

$\text{mtmmatch} l \text{as} l \Rightarrow \text{let} x := \text{Nat} \text{.max} e1 e2 \text{in}$

¹$\text{mtmmatch}$ patterns can bind arbitrary (sub)terms and its patterns use the notation $[? a \ldots z]$ to name these terms.
However, types do not match: \texttt{Mtac2}'s \texttt{bind} operator (written using the traditional notation \(a \leftarrow f; g\)) has type:
\[
\texttt{bind} : \forall (A : \text{Type}), M A \to (A \to M B) \to M B
\]
In our example, the \texttt{mfix} term returns \(\forall l, 1 \leftrightarrow \text{nil} \to M S\): the \(M\) does not occur at the top-most position in the type as \texttt{bind} expects it. Thus, without a suitably generalized version of \texttt{bind}, we once again cannot just apply the convoy pattern.\(^2\)

In this work we present a new meta-meta-program \texttt{lift} that provides a semi-automatic solution: given any meta-program or operator (like \texttt{bind}) and a list of dependencies (what lies behind the last \(\to\) in the type of \texttt{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 \texttt{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 \texttt{bind} more general. Specifically we wish to get the following function \texttt{bind}^*.
\[
\texttt{bind}^* : \forall (A B : \text{Type}), \forall l (H : l \leftrightarrow \text{nil}) (\text{Type}),
\begin{align*}
\forall &\ 1 H, M (A 1 H) \to \\
\forall &\ 1 H, (A 1 H \to M (B 1 H)) \to \\
\forall &\ 1 H, M (B 1 H)
\end{align*}
\]

This signature is derived from that of the fixpoint, that gets two arguments \(l\) and \(H\) before returning the (monadic) value. Note the following: in our case, the first (non-implicit) argument of \texttt{bind}^* will be \texttt{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 \texttt{max} \(S\) must also be \texttt{lifted} to include such dependencies.

In concrete, we can lift the \texttt{bind} operator to have a type that matches the expected type for our example by writing:
\[
\texttt{@bind} \ [	exttt{t} : (\texttt{l : list S}) (\_ : l \leftrightarrow \text{nil})]
\]
The \(@\) is Coq's syntax to not insert implicit variables (\(A\) and \(B\) for \texttt{bind}); the \(\lambda\) is notation for the lift function; and \([\texttt{t} : a \ ... z]\) is notation for a \texttt{telescope} listing the dependencies a to z (which are binders).

We can then write the new \texttt{list\_max} program as follow:
\[
\textbf{Definition} \ \texttt{list\_max} \ (S : \text{Set}) : \forall l, 1 \leftrightarrow \text{nil} \to M S := \\
(\texttt{@bind} \ [\texttt{t} : (\texttt{l : list S}) (\_ : l \leftrightarrow \text{nil})]) \_ \_ \\
\quad \text{(max} \ S \ [\texttt{t} : (\texttt{l : list S}) (\_ : l \leftrightarrow \text{nil})]))
\]
\[
(\texttt{fun} \ l (H : l \leftrightarrow \text{nil}) (\texttt{max} : S \to S) \Rightarrow \\
\texttt{mfix2} f (1 : \texttt{l : list S}) (H : l \leftrightarrow \text{nil}) : M S := \\
\texttt{mtmatch} 1 \lambda \texttt{l} \texttt{return} 1' \leftrightarrow \text{nil} \to M S \texttt{with} \\
\quad [\_ \ e] [\_] \Rightarrow \texttt{ret} e \_ \_ [\texttt{t} : (\_ : [\_ e] \leftrightarrow \text{nil})]
\]
\(^2\)It is possible to introduce all arguments and beta-expand the fixpoint but we consider this inadvisable for reasons of readability and maintainability.

3 Technicalities
Under the hood, \texttt{lift} is quite involved. The code can be downloaded from
\href{https://github.com/ignatirabo/Mtac2_lift}{https://github.com/ignatirabo/Mtac2_lift}

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

The actual lifting occurs in \texttt{lift}'. \texttt{lift}' takes a function \(f\), our lifting candidate, with its encoded type, and the corresponding telescope, then returning a \(\Sigma\)-type, wrapped in the monad \(M\), with the new signature and function.

4 Conclusion and Future work
At the moment, \texttt{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 \texttt{bind} and \texttt{ret}.

Performance wise, \texttt{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 \texttt{lift} into \texttt{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 \texttt{list\_max} example from Section 2, we should be able to obtain the returned type \texttt{forall l : list S, 1 \leftrightarrow \text{nil} \to M S} and construct from it the telescope. Ideally, we should be able to also find that the argument of \texttt{bind} (\texttt{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 uplifted ones when no convoy pattern is required.
References


