A Case for Tactics with (Limited) Side Effects

Benjamin Delaware
MIT CSAIL
bendy@csail.mit.edu

With the notable exception of `abstract`, Coq's standard collection of tactics are free from side effects. In this talk, I propose to make the case that additional tactics with limited side effects that are "local" to a given proof can have a considerable impact on performance, presenting as evidence a set of ML tactics that have been developed as part of the Fiat deductive synthesis framework. In particular, I'll show how a version of `abstract` that adds transparent definitions to the global context and tactics that manipulate hint databases can shrink both the memory usage and processing time of a proof script by an order of magnitude.

Deductive Synthesis in Fiat We begin with a brief overview of the Fiat [1] framework for deriving implementations of abstract data types (ADTs) in Coq, highlighting two facets of this domain where side effecting tactics are useful. Fiat ADTs are specified using nondeterministic reference implementations whose operations are defined in terms of a model of internal state. Figure 1 gives an example specification of an cache ADT modelling its internal state with mathematical sets and featuring with empty, insert, update, and lookup methods. The body of each of these methods lives in the nondeterminism monad, leaving behaviors such as the cache eviction policy underspecified.

```
ADT CacheSpec implementing CacheSig :=
  rep := Set of (Key × Value)
  constructor "empty" := return ∅
  method "insert" (r : rep, k : Key, v : Value) : rep :=
    return ((k, v) ∪ r)
  method "update" (r : rep, k : Key, f : Value → Value) : rep :=
    {r′ | ∀v. (k, v) ∈ r  →  ∀kv. kv ∈ r′  →  kv = (k, f(v))
                                             ∨ kv ∈ RemoveKey(k, r)}
  method "lookup" (r : rep, k : Key) : option Value :=
    {v_opt | ∀v. v_opt = Some v  →  (k, v) ∈ r}
```

Figure 1. An abstract data type for caches

An implementation of this ADT is the first component of the refinement type $\{\mathsf{impl} \mid \mathsf{CacheSpec} \succ_{\approx} \mathsf{impl}\}$, where $\succ_{\approx}$ is simply an *abstraction relation* [2, 3] lifted to the nondeterminism monad. The proof component of this refinement type guarantees that impl is correct-by-construction. An inhabitant of this refinement type is developed using Coq's standard interactive proof development mechanism, using a combination of setoid rewriting and custom Ltac tactics for building ADT refinement proofs. By constraining the specifications to specific subdomains, derivations can be further automated by codifying design decisions as refinement lemmas that are intelligently selected by domain-specific tactics. As an example, the distribution of Fiat includes a library for specifying and implementing ADTs with SQL-like operations. Similar domain-specific synthesis libraries for parsers and cryptographic primitives are currently under development. This quick overview is enough to highlight the two challenges to performant synthesis that side-effecting tactics can help address: *concise domain-specific specifications* and the *synthesis of intermediate terms*.

Concise domain-specific specifications Fiat ADTs are shallowly embedded, allowing users to draw on all of Gallina when writing specifications and allowing the synthesis process to produce easily extractable implementations. ADTs are encoded as records parameterized over a signature containing method names and their domain and codomain. Fiat's domain-specific synthesis libraries similarly use parameterized definitions for modelling internal state. Figure 2 shows the definition of a "schema" used by our SQL-like synthesis library to model state using ensembles for the "tables" of an ADT.

```
Definition MessagesSchema :=
Query Structure Schema
  [ relation messages has
      schema <"number"::nat, "date":: nat, "msg"::list string>;
    relation contacts has
      schema <"number"::nat, "name"::string> ].
```

Figure 2. Fiat specification for an address-book database

In both cases, these parameterized definitions allow for concise, natural-looking specifications that can be generically manipulated by tactics when synthesizing a term. The downside is that the types used to index these definitions are repeated throughout the proof term. Since derivations consist of sequences of small refinement steps, seemingly innocuous inefficiencies in the definition of an index can quickly cause a memory blowup. The natural choice of strings for identifiers in specifications, for example, leads to slightly larger specifications, but this penalty is compounded with each refinement step. Initial versions of the library created auxiliary definitions for each string literal to avoid this problem, improving performance at the cost of clunkier specifications. Maximizing performance required aliases for each component of the schema, i.e. the headings for each table, quickly leading to unwieldy specification.

Synthesis of intermediate terms Predefining aliases is not a complete solution, however, as automated tactics actually synthesize terms during a derivation. As an example, the synthesis tactic for the SQL domain selects appropriate "indexes" on tables using combinations of data structures drawn from a library of implementation strategies. Defining aliases for these intermediate terms forces users to make implementation choices instead of synthesis tactics.

Side-Effecting Tactics to the Rescue We solve both of these problems using `cache_term`, a transparent version of `abstract` implemented as an ML tactic. Combining this tactic with Ltac's contextual goal matching allows refinement tactics to identify terms, transparently abstract them, and fold the newly introduced definition to reduce the size of the goal. In order to avoid introducing duplicate definitions, we have also built ML tactics for adding hints to a database and for iterating a tactic over the hints in a database. The latter is used to keep track of abstracted definitions, which are then folded in subsequent goals using the latter tactic. Figure 3 summarizes each of these tactics.

Tactic	**Summary**
cache_term δ as ι	Add a definition with identifier ι for the term δ to the global context
cache_term δ run κ	Add a definition with a fresh identifier for the term δ to the global context and run the tactic κ on the new identifier
add_hint δ to η	Add a resolution hint for definition δ to the database η
foreach η run κ	Applies the tactic κ to each resolution hint in η

Figure 3. Summary of the side-effecting tactics included in Fiat.

Evaluation To demonstrate the value of these side-effecting tactics, summarizes the evolving performance on the three benchmarks included in the original Fiat paper. Our initial attempts at allowing tactics to construct implementations resulted in a large jump in memory usage for each benchmark: after the switch, the largest benchmark took 23GB of memory and ran for close to 100 minutes. Our next iteration introduced local definitions for constants by let binding them in the proof context, reducing the memory footprint and runtime by 5x for our examples. Using the side-effecting tactics presented here led to an additional 3x improvement in both memory usage and runtime for all the examples.

Discussion A pleasant characteristic of these tactics is that their side effects are localized, in the sense that the performance gains observed above aren't impacted if the effects of these tactics aren't observable outside of the proof they in which they are utilized and can be discarded after the final proof term is checked. In this regard, they should be compatible with the support for parallel processing of definitions in the upcoming 8.5 release. We posit that additional variants of `abstract` may also be useful when synthesizing terms via tactics, particularly lifting the `abstract`'s current restriction to goals without existential variables.

References

[1] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, 2015.

[2] J. He, C. Hoare, and J. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer Berlin Heidelberg, 1986.

[3] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

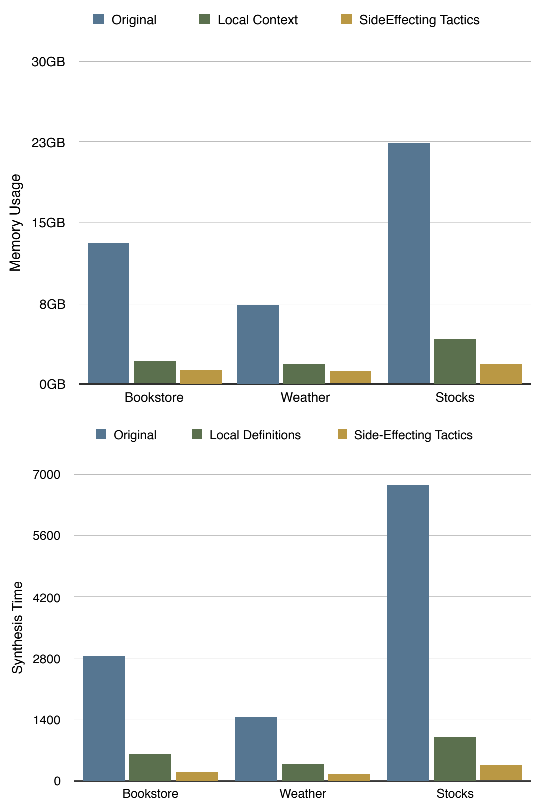

Figure 4. Memory usage and derivation times for ADTs with SQL-like operations.