CS 199 Final Report

Bretton Chen

December 9, 2022

My research project for this quarter focused on improving the `coq-synth` program synthesis tool for Coq, as a component of the `lfind` lemma synthesis tool. The main additions I made were a new interface, example-based filtering, simplification and deduplication of synthesis terms, lazy output, and support for polymorphism. I also contributed to the integration of `coq-synth` into `lfind`. Overall, these improvements will allow `lfind` to support more complex Coq projects than before.

1 Background

`lfind` is a tool that performs lemma synthesis for the Coq interactive theorem prover [1]. Given a proof state where the user is stuck, it generates possible helper lemmas that might allow the completion of the proof.

The design of `lfind` itself depends on three other tools for Coq: a property-based tester, a data-driven program synthesizer, and an automated prover. The initial implementation of `lfind` used QuickChick as the property-based tester and Proverbot9001 as the automated prover. However, there did not exist a data-driven program synthesizer for Coq. Therefore, a data-driven synthesizer for OCaml called Myth was used, along with Coq's program extraction feature and the `coq-of-ocaml` tool to translate between Coq and OCaml. This approach had some limitations and was brittle since each of these tools only supports a subset of Coq or OCaml, leading to many uses of `lfind` failing because the input could not be understood by Myth.

To solve this problem, we decided to write our own data-driven synthesizer for Coq which can be used directly by `lfind` and circumvents the complicated and error-prone process of using Myth. I wrote the initial prototype of this tool, uncreatively named `coq-synth`, at the end of last quarter, but it was not able to replace Myth yet. Myth is a type-and-example based program synthesizer, which means that it takes as input both the type of the program that you want to synthesize and examples of inputs and outputs of the program that describe its behavior. At the start of this quarter, `coq-synth` was only type-based, but the data-driven nature of `lfind` required the use of input-output examples as well. Therefore, my initial goal was to implement example-based filtering for `coq-synth` so that it could be integrated into `lfind`, and afterwards add further improvements to `coq-synth` which may even go beyond what Myth is capable of in some areas.

2 Core design

I will first describe the core design of `coq-synth` independent of the improvements, which has largely remained the same since the start of the quarter. Any modifications will be described in the next section.

`coq-synth` generates terms of Gallina, the specification language of Coq. The terms that it generates only consist of references or applications of terms. References are the names of constructors, constants, or variables. Constants are things that are defined globally while variables are local to the current section.

$$\begin{aligned} \mathit{term} &::= \mathit{ref} \mid \mathit{term}\ \mathit{term} \\ \mathit{ref} &::= \mathit{ctor} \mid \mathit{const} \mid \mathit{var} \end{aligned}$$

This is obviously a very small subset of all possible Gallina terms. Unlike Myth, `coq-synth` is not capable of synthesizing lambda expressions, pattern matching, or recursion, let alone dependently-typed constructs of Gallina which are not present in OCaml. However, since `coq-synth` was designed for use in `lfind` and not as a general purpose synthesizer for arbitrary Gallina programs, and `lfind` only needs terms in this form, the limited scope of `coq-synth` is sufficient.

`coq-synth` takes four main inputs: a Coq module providing the global environment E, the parameters for synthesis $P_1, \ldots, P_n$, the type of the synthesized expression τ, and extra references $r_1, \ldots, r_m$. Each parameter $P_i = (p_i : t_i)$ is a pair of a name and a type, with no definition. The parameter names will then be possible variables to include in the resulting synthesized expression. An extra reference r_i is an already defined name that is in scope in the given module which may also be included in the synthesized expression. The type of an extra reference does not need to be specified since `coq-synth` can look it up in E.

For instance, if E contains

```
Inductive lst := Nil : lst | Cons : nat -> lst -> lst.
Definition rev : lst -> lst -> lst.
```

and `coq-synth` is given $P_1 =$ `(l1 : lst)`, $P_2 =$ `(n : nat)`, $\tau =$ `lst`, and $r_1 =$ `rev`, one possible output might be `Cons n (rev l1)`.

When called by `lfind`, E will be the environment of the module in which the user is running the `lfind` tactic, $P_1, \ldots, P_n$ will be variables for synthesis as determined by `lfind` (a combination of free variables in the original goal state and new variables created during generalization), τ will be the type of the hole of the lemma sketch, and $r_1, \ldots, r_m$ will be all the names that are referenced recursively from the stuck state. See the `lfind` paper for more details.

`coq-synth` enumerates all possible terms of the given type τ following the above grammar in increasing order of depth, using the provided parameters and extra references as possible values for references in addition to constructors, which are always used. The depth of a term is defined recursively by

$$\begin{aligned}\text{depth}(\mathit{ref}) &= 0\\ \text{depth}(\mathit{term}_1\ \mathit{term}_2) &= \max(\text{depth}(\mathit{term}_1), \text{depth}(\mathit{term}_2)) + 1\end{aligned}$$

The maximum depth to enumerate to must be passed as an input as well.

The synthesizer works roughly as follows. Let A be a set of terms defined by

$$A = \{\text{all constructors}\} \cup \{p_1, \ldots, p_n\} \cup \{r_1, \ldots, r_m\}$$

$R : \mathit{type} \to \mathcal{P}(\mathit{type})$ be defined recursively by

$$\begin{aligned}R(u \to t) &= \{u \to t\} \cup R(t)\\ R(t) &= \varnothing && \text{where } t \text{ is not a function type}\end{aligned}$$

T be a set of types defined by

$$T = \bigcup_{(a:t)\in A} R(t)$$

$D : \mathbb{N} \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$ be defined by

$$D(k) = \{(0, k), (1, k), \ldots, (k-1, k), (k, k), (k, k-1), \ldots, (k, 1), (k, 0)\}$$

and $S : \mathbb{N} \times \textit{type} \to \mathcal{P}(\textit{term})$ be defined recursively by

$$\begin{aligned} S(0,t) &= \{a \in A \mid a : t\} \\ S(k+1,t) &= \{f\ x \mid (u \to t) \in T, (i,j) \in D(k), x \in S(i,u), f \in S(j,(u \to t))\} \end{aligned}$$

$S(n,t)$ is the set of all terms (of the limited grammar) of depth n and type t. Therefore, given a maximum depth N, the output of `coq-synth` is the concatenation of $S(0,\tau), S(1,\tau), \ldots, S(N,\tau)$.

The implementation of `coq-synth` is in OCaml and mainly uses the implementation of Coq itself as an OCaml library to perform operations on internal Coq objects. It also uses part of the implementation of the higher-level `coq-serapi` tool as a library in a few places to perform a few more complex operations on Coq documents. In addition, it uses the alternative `base` standard library from Jane Street, the `res` library for resizable arrays, and the `cmdliner` library for command line parsing.

3 Improvements

I will now describe the major improvements made to `coq-synth` this quarter in chronological order.

3.1 New interface

The original interface of `coq-synth` was not suitable for use with `lfind` as it required a custom Coq input file that specified the synthesis parameters as section variables and the result type as the goal type of an open proof. I had initially used this design instead of taking them as command line parameters due to Coq's internal representation of local variables as De Bruijn indices, which were hard to work with for the things that I wanted to do with them.

I eventually looked more into the Coq implementation and worked around this issue so I switched to an interface where all inputs are taken via command line. This also means that the Coq module the user is calling `lfind` from can be used directly as an input to `coq-synth`, eliminating the need for separate files to be generated like `lfind` does for QuickChick and Myth.

Originally the command line interface was intended only for testing and I had planned to create an alternate server-like interface where a single `coq-synth` process would run continuously and communicate with `lfind` via standard IO. This would remove the overhead of process spawning and loading the user's Coq modules for every call to `coq-synth`. However we decided to abandon that approach for now because it was considered to be more difficult to implement on the `lfind` side.

3.2 Example-based filtering

I integrated input-output examples into the synthesis process by adding filtering as the last step where we check that the term is valid according to the given examples and skip over it if it isn't.

More formally, let $X_1, X_2, \ldots, X_k$ be examples, where $X_i = (\langle i_{i1}, i_{i2}, \ldots, i_{in}\rangle, o_i)$, $i_{ij} : t_j$ for all j, and $o_i : \tau$ for all i. That is, each example is a vector of inputs and an output, where the number and types of the inputs match up with the number and types of the parameters, and the type of the output matches the type of the synthesized expression. Then the output of `coq-synth` is now the concatenation of

$$\{x \in S(d, \tau) \mid \forall i,\ E[\varnothing] \vdash x[i_{ij}/p_j]_j =_{\beta\delta\iota\zeta\eta} o_i\}$$

for each $d \in \mathbb{N}$ up to the maximum depth N, where $=_{\beta\delta\iota\zeta\eta}$ is the convertibility relation defined in Coq [2].

For instance, if $P_1 = (\texttt{l1 : lst})$, $P_2 = (\texttt{n : nat})$, $\tau = \texttt{lst}$, and $r_1 = \texttt{rev}$ as before, and we have

$$X_1 = (\langle \texttt{Nil}, 4\rangle, \texttt{Cons 4 Nil})$$
$$X_2 = (\langle \texttt{Cons 1 (Cons 0 Nil)}, 2\rangle, \texttt{Cons 2 (Cons 0 (Cons 1 Nil))})$$

then `Cons n (rev l1)` would be a valid output while `Nil`, `Cons n l1`, `rev l1`, or `Cons 2 (rev l1)` wouldn't be.

We cannot simply check that $x[i_{ij}/p_j]_j$ and o_i are syntactically equal since $x[i_{ij}/p_j]_j$ might be something like `rev [1; 2; 3]` and o_i might be `[3; 2; 1]`, which are semantically but not syntactically equal. (By semantic equality I mean equality as determined by the = operator defined in Coq.) Therefore both terms are reduced to a normal form and then compared. Assuming that the extra references $r_1, \ldots, r_m$ and the examples $X_1, \ldots, X_k$ only contain regular computable variables and functions like you would get in OCaml, this is guaranteed to be complete, i.e. any $x[i_{ij}/p_j]_j$ and o_i which are semantically equal will be syntactically equal after reduction. This is because after substitution there should be no more opaque variables left in $x[i_{ij}/p_j]_j$, so it is able to be evaluated to a value. Likewise, o_i is evaluated to a value, and values are syntactically equal if and only if they are semantically equal. Hence we will not skip over any synthesized expression that actually matched the examples under these assumptions, which are true for the way that `lfind` invokes `coq-synth`.

3.3 Simplification and deduplication

After adding the convertibility check I realized that I can simplify the outputted terms themselves as well. Furthermore, I can perform simplification during the synthesis process by reducing all the terms in $S(i,t)$ for all intermediate i and t before proceeding to the next step. This is useful because often we generate many terms that are convertible to each other, for instance `Cons n l1`, `append (Cons n Nil) l1`, `append Nil (Cons n l1)`, `append (reverse (Cons n Nil)) l1`. If we reduce all terms in $S(i,t)$ and filter out reduced terms that are already present in some $S(j,t)$ where $j \leq i$, we can cut down on the total number of synthesis terms significantly and do a lot less work, because then the duplicate terms in $S(i,t)$ would not be used as part of generating more terms in the future.

To do this, I call the implementation of the `cbn` tactic with the β-, δ-, ι-, and ζ-reduction flags enabled on each $f\ x$ term constructed in the recursive $S(k+1,t)$ case. (The `cbn` tactic is a faster and more predictable version of the popular `simpl` tactic commonly used in proofs.) Then I remove any reduced terms already present in any $S(i,t)$ where $i \leq k$ or any duplicate terms in $S(k+1,t)$ itself.

In the above example, `Cons n l1` would be first generated in $S(2,\texttt{lst})$, and so when `append (Cons n Nil) l1` is generated at $S(4,\texttt{lst})$ it will be simplified to `Cons n l1` and filtered out. `append Nil (Cons n l1)` will never be generated at all since its component term `append Nil` already simplifies to `(fun x : lst => x)`, which when then applied to `Cons n l1` will yield `Cons n l1`, and `append (reverse (Cons n Nil)) l1` will never be generated either since its component term `reverse (Cons n Nil)` should already be simplified to `Cons n Nil`.

Formally, the recursive case for S is now

$$S(k+1,t) = \{\mathrm{cbn}(f\ x) \mid (u \to t) \in T, (i,j) \in D(k), x \in S(i,u), f \in S(j,(u \to t))\} \setminus \left(\bigcup_{i \leq k} S(i,t)\right)$$

One consequence of this change is that the first argument to S, and the maximum depth parameter N, no longer represent the structural depth of the term, since the term may become smaller (or occasionally larger) after being reduced with `cbn`. Therefore we now reinterpret depth not as a syntactic notion but rather as the recursion depth to which the synthesis algorithm searches to first find this term.

Also, as a consequence of using `cbn`, the results of S and therefore the output of `coq-synth` are not strictly limited to the initial limited term grammar anymore. In particular, δ-reduction will generate terms originating from the body of a definition which can indeed be any arbitrary Gallina term. Conversely, we of course do not generate all possible terms of the limited grammar anymore either, as that is the whole point of performing simplification. There should be no loss in expressiveness though as we only skip terms that are semantically equal to some term that is already generated.

Another point to note is that, unlike in the example filtering step, the semantic equality check here is not necessarily complete. This is because when performing intermediate simplification of terms, the terms may contain parameters which are opaque since they do not have a definition. Under these circumstances, the method of performing reduction and then comparing for syntactic equality is incomplete with respect to semantic equality. For example, `cbn` is able to reduce `append Nil l1` to `l1`, where `l1` is a parameter, by unfolding the definition of `append`, which performs a pattern match on its first argument and if it is `Nil` returns its second argument. However, it is not able to reduce `append l1 Nil` to `l1` as that requires induction; this equality usually needs to be established in Coq proofs using a human-provided lemma. Therefore, `coq-synth` may still generate different terms which are semantically equal, and we cannot guarantee that the generated terms are semantically unique.

3.4 Lazy output

I rewrote the synthesis algorithm to generate terms lazily and output them as they are generated instead of outputting all the terms at the end. This means that `lfind` will be able to start filtering and ranking the first result while `coq-synth` is still synthesizing the rest of the results in parallel, which should speed up the whole lemma synthesis process, although the current implementation of `lfind` does not do this yet.

This change also makes memory usage constant instead of scaling with the number of terms generated (observed empirically by watching the RAM usage in my system monitor not creep up and make my laptop run out of memory anymore), allowing the synthesis of larger terms to be limited only by time and not space.

Finally, making the program lazy means that the `max-depth` parameter is not required anymore; the synthesizer now simply runs forever and generates an infinite stream of terms if no maximum depth is specified (assuming there do exist infinitely many terms of the given type). Another parameter `num-terms` was also added which stops synthesis after producing the given number of terms; this is the one that is currently used by `lfind`.

3.5 Polymorphism

Previously, `coq-synth` only supported monomorphic inductive data types as the types of the parameters $P_1, \ldots, P_n$, the synthesis type τ, the types of the extra references $r_1, \ldots, r_m$, and the types of the examples $X_1, \ldots, X_k$.

There were two major changes in adding support for polymorphism. First, all of the above types are now allowed to be any expression of sort `Type` and not just a single reference to a type. For instance, `list nat`, where `list : Type -> Type` is the list type as defined in the Coq standard library, is now allowed. Second, for each parameter $P_i = (p_i : t_i)$, in addition to t_i being allowed to be an expression of sort `Type`, it is also allowed to be `Type` (or `Set`) itself; that is, p_i itself may also be of sort `Type`. In this case, p_i is now a type parameter, and it may be used in subsequent t_j where $j > i$, in τ, or in $r_1, \ldots, r_m$, as with the normal dependent typing rules. For instance, we can have $P_1 =$ (`t : Type`), $P_2 =$ (`l1 : list t`), $P_3 =$ (`x : t`), and $\tau =$ `list t`. Along with this, $r_1, \ldots, r_m$ can now be any expression instead of only single references as well. This is mainly to support applying type arguments to polymorphic functions, e.g. we could have $r_1 =$ `@app t` in the previous example (the `@` notation allows all arguments to be explicitly applied, since the type argument is normally implicit), but you could also pass arbitrary lambda expressions for instance as extra references. With these changes, not only can `coq-synth` use inputs containing applied type constructors, it can also synthesize polymorphic terms.

Note that when providing examples, the type parameters have to be explicitly supplied as well. For instance,

$$X_1 = (\langle \texttt{bool}, \texttt{@nil bool}, \texttt{true}\rangle, \texttt{@cons bool true (@nil bool)})$$

for the previous example, so that `t` $=$ `bool`. This also means that when `coq-synth` is synthesizing polymorphic terms you can pass examples of different types to it. For instance, the second example could be

$$\begin{aligned}X_2 = (&\langle \texttt{nat}, \texttt{@cons nat 1 (@cons nat 0 (@nil nat))}, 2\rangle,\\ &\quad \texttt{@cons nat 2 (@cons nat 0 (@cons nat 1 (@nil nat)))})\end{aligned}$$

and it would check that the resulting polymorphic terms satisfy both examples.

All the type arguments in the examples and extra references need to be explicitly applied because while they are normally implicit and able to be inferred by Coq from the surrounding context, when they are parsed in a standalone way as command line parameters to `coq-synth` Coq does not have this information and so implicit argument inference fails. I couldn't figure out how to supply the appropriate context to Coq, since implicit argument inference is performed at quite an early stage in interpretation. Nevertheless, this does not really matter for use with `lfind` as `lfind` always passes fully explicitly applied terms to `coq-synth`.

For anything that can now be an expression instead of a single reference, including $t_1, \ldots, t_n$, τ, and $r_1, \ldots, r_m$, we apply `cbn` before continuing with the rest of the synthesis algorithm, so that for the types we don't end up with duplicate types where $t = u$ semantically but $S(i,t) \neq S(i,u)$, and for the values the initial terms in $S(0,t)$ are also reduced.

Mathematically, the synthesis algorithm can be modeled in much the same way as before, if "all constructors" in the definition of A is considered to mean all monomorphic constructors of all fully-applied types; that is, we would have terms like (`@cons nat` : `nat -> list nat -> list nat`) $\in A$ instead of (`cons` : $\forall$(`t` : `Type`), `t -> list t -> list t`) $\in A$. However, in terms of the implementation, since "all constructors" is of course infinite we only lazily added the constructors of a given type to A whenever we first needed to synthesize terms of that type. Now that constructors can be polymorphic, we need a way to correctly add the monomorphized version of the constructors given a fully-applied type, but this is easy since the order of the type parameters of the polymorphic type is guaranteed to be the same as the order of the type parameters of the constructors. Therefore, if we want to obtain the constructors for some fully-applied type $t\ a_1 \cdots a_p$, we simply look up each constructor of t which will be in the form $c : \forall \alpha_1 \cdots \alpha_p,\ \phi(\alpha_1, \ldots, \alpha_p)$ for some ϕ, and add the term $c\ a_1 \cdots a_p : \phi(a_1, \ldots, a_p)$ to A. However, this is not possible for arbitrary polymorphic functions, since they may take type arguments in arbitrary ways. Therefore we do not monomorphize the extra references automatically and instead require the user to pass in monomorphic versions of any polymorphic functions they may wish to use. For instance, if the user wants to include `app : forall (t : Type), list t -> list t -> list t` in the synthesis results, they must know which types it may be used with, say `nat` and `bool`, and then explicitly pass in r_1 = `@app nat` and r_2 = `@app bool`.

Also, the context under which typing judgments were made was left implicit in the earlier descriptions of the synthesis procedure, since they were always $E[\varnothing]$, i.e. the global environment with an empty local context.

Now that we have type parameters in context, we actually have formally $E[\varnothing] \vdash p_1 : t_1$, $E[\{p_1 : t_1, \ldots, p_i : t_i\}] \vdash p_{i+1} : t_{i+1}$ for each i, $E[\{p_i : t_i\}_i] \vdash \tau : \texttt{Type}$, $E[\{p_i : t_i\}_i] \vdash r_j : \rho_j$, and generally $E[\{p_i : t_i\}_i]$ as the context for all judgments in the description of the procedure itself. However, we still have $E[\varnothing] \vdash i_{ij}$ for all i, j and $E[\varnothing] \vdash o_i$ for all i, and also still $E[\varnothing]$ as the context for the $=_{\beta\delta\iota\zeta\eta}$ convertibility check in the example-filtering step, since the parameters should not be present in the examples.

Note that when I say polymorphism I mean parametric polymorphism in the style of ML data types, and not just any types that take type arguments. In ML these are the same thing, but in Coq inductive data types can have type *indices* in addition to type *parameters*. Type parameters do not affect the structure of an inductive data type, but type indices do. For example, the dependently-typed vector type

```
Inductive vec (t:Type) : nat -> Type :=
| vnil : vec t 0
| vcons : forall (_:t) (n:nat), vec t n -> vec t (S n).
```

has one type parameter `t` : `Type` and one type index `n` : `nat`. The choice of `t` does not affect the structure of a vector but the choice of `n` does, namely, it determines its length. Supporting type indices would require major changes to the synthesis algorithm to take into account dependent types, so for now `coq-synth` only supports fully parametrically polymorphic data types.

3.6 Integration into `lfind`

This is not really an improvement to `coq-synth` itself, but I contributed to integrating `coq-synth` into `lfind` as well as running the benchmarks from the `lfind` paper with the new `coq-synth` backend to check for regressions compared to Myth.

4 Comparison to Myth and future work

In terms of features, both Myth and `coq-synth` now support taking input-output examples and only producing terms that behave according to those examples. However, Myth does this in a different way than `coq-synth`. While `coq-synth` simply filters out the invalid terms at the end of the synthesis process, Myth uses the examples to guide the synthesis process and only synthesizes terms that are valid according to the examples in the first place. This means that Myth performs faster, particularly in cases where only a small fraction of all possible terms are valid according to the examples. As a result, when we integrated `coq-synth` into `lfind` we observed a slowdown compared to using the Myth backend. I initially chose to use the simpler filtering approach because the smarter integrated approach would have necessitated a redesign of the synthesis algorithm and taken longer to develop while we wanted to switch to `coq-synth` in `lfind` as soon as possible. However, this is now a good area for improvement and in the future I will probably rewrite `coq-synth` to use Myth's approach.

On the other hand, as far as I know Myth does not perform simplification of terms or support polymorphism, so `coq-synth` is an improvement in these respects.

Nevertheless, the biggest advantage of `coq-synth` over Myth from `lfind`'s perspective is that it works directly with Coq instead of requiring translation to and from OCaml. Switching to `coq-synth` was overall beneficial for `lfind`, as we were hitting a lot of problems in the translation of Coq to the subset of OCaml that Myth understands, particularly in benchmarking real-world Coq projects. Since `coq-synth` directly uses the Coq API, it understands arbitrary Coq code and so we will not run into this issue anymore. Furthermore, we can now extend `lfind` to support things like polymorphism that we were previously unable to do due to Myth's limitations.

The other area for improvement is continuing to expand the subset of Coq that `coq-synth` supports for synthesis. With parametric polymorphism done, the next step is probably to look into how to synthesize dependently-typed terms. This will be interesting as Coq's type system is much more complex than OCaml's.

5 Repository

`coq-synth` is open source, and the implementation of the results described in this report can be viewed at `https://github.com/qsctr/coq-synth`.

References

[1] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-Driven Lemma Synthesis for Interactive Proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 143 (October 2022), 27 pages. `https://doi.org/10.1145/3563306`

[2] Conversion rules. `https://coq.inria.fr/refman/language/core/conversion.html`