EECS598-006 Final Report: Shmyth

Andrew, David, Eric

December 10, 2020

Abstract

Shmyth is an extension to the Hazel structure editor for interactive program synthesis with the Smyth synthesis engine. It circumvents the fully automatic synthesis pipeline of Smyth to give the user direct and fine grained control over synthesis as it occurs.

1 Introduction

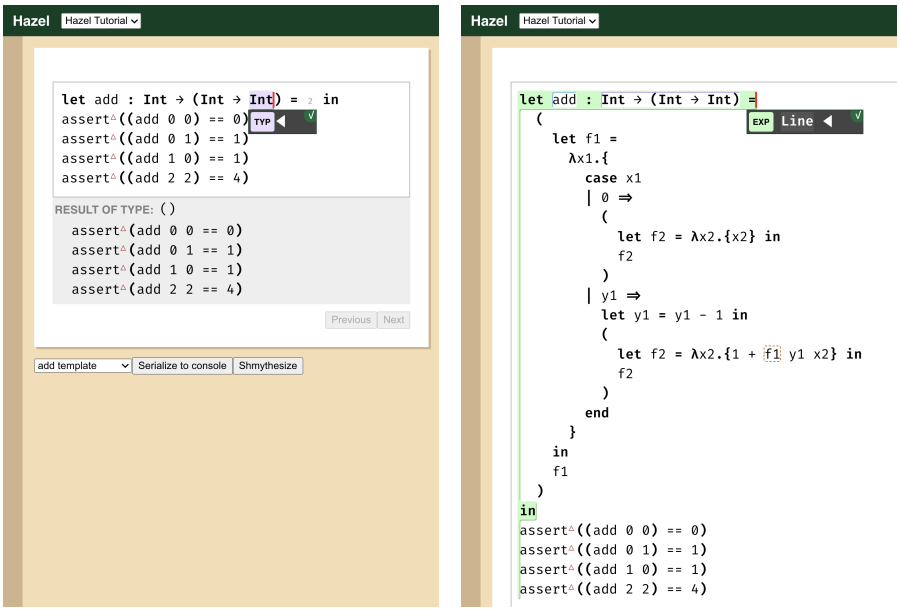
For our term project, we explored the feasibility of incorporating program synthesis into a structured editing context. Our exploration centered on the development of a prototype interactive program synthesizer. We built a prototype by embedding Smyth [1], a type-and-example-directed program synthesizer, into the Hazel [4] structure editing environment, along with a custom user interface for navigating and steering the synthesis processes.

This work was motivated by our observation that programming is increasingly becoming a social activity. Programmers often spend much of their time working in isolation, only soliciting feedback from others when they are paralyzed by too little insight or too many choices. We conjecture that bringing a synthesizer into this conversation is helpful on these fronts and wanted to gauge the level of technical difficulty in pursuing this line of inquiry.

More concretely, our exploration of hybrid PL systems with semi-autonomous assistants was motivated and guided by a simple task. Figure 1 shows a sketch and full implementation of an `add` function for addition over Peano-encoded natural numbers, written in the Hazel programming language. The sketch in Figure 1a consists of a function signature followed by some Hazel assertions that can be reduced to input-output examples. The full program in Figure 1a shows the solution we expected to get from the Smyth program synthesizer after a brief series of user interactions.

In this report, we detail our effort to integrate Hazel with Smyth:

- We propose an interaction model based on our exploratory UI design for Hazel. Our design exposes the intermediate results of the Smyth synthesizer as a series of constraints and sketch refinements, enabling the user to have an interactive dialog with the synthesizer as it is running.



(a) Initial sketch (b) Final implementation

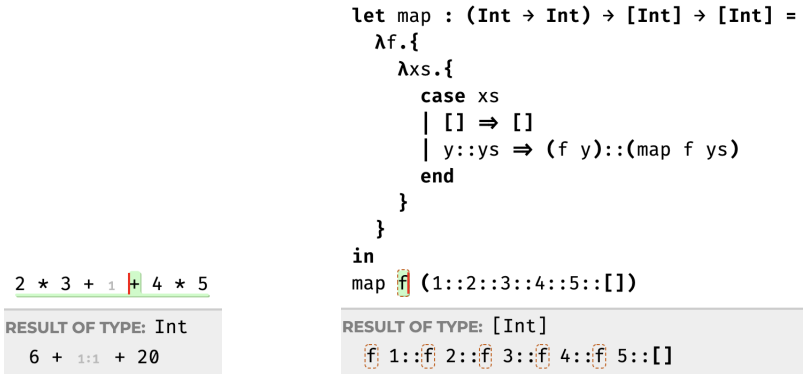
Figure 1: Synthesizing a program for adding Peano-encoded natural numbers.

- We discuss how we altered Smyth to operate in an interactive, incremental manner, including the integration of back end components with a custom UI in Hazel's existing web-based front end.
- We pose several user-centered research questions that outline our plans for future work.

2 Background

2.1 Hazel

Hazel [4] is a statically typed programming language as well as a type-aware structure editor for the language. Hazel has primitive support for well-typed holes [3], making it an interesting target for sketch-based programming synthesis. In Hazel, every expression has a well-defined type and evaluation result, including holes. Figure 2a show a simple arithmetic expression with a hole for the middle term. Using a technique called *live evaluation* [2], Hazel can partially evaluate ill-typed programs by placing expressions with inconsistent types into holes and evaluating around them. For example, in Figure 2b where the subject function `f` of a standard list `map` is undefined, Hazel partially evaluates `map f` into a list of applications of `f`.



(a) (b)

Figure 2: Live evaluation in Hazel

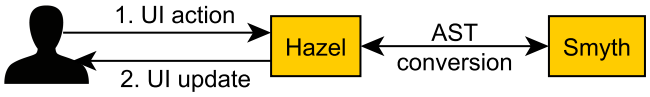


Figure 3: User-directed interaction model

2.2 Smyth

Smyth [1] is a type-and-example-directed synthesis algorithm. It uses a novel technique called *live unevaluation* to refine partial programs with assertions into extra synthesis constraints. By combining live evaluation with live unevaluation to form yet another novel technique called *live bidirectional evaluation*, Smyth overcomes a key limitation in prior work that required assertions be trace complete, i.e., they must cover all base cases as well as a complete set of inductive cases.

3 Synthesis-Guided Structure Editing

Figure 3 depicts our interaction model. When the Hazel edit cursor is on an empty hole, the user can initiate a "shmythesize" action. Hazel feeds our embedded Smyth engine the entire program, along with an identifier for the current hole and a hard-coded prelude discussed in section 3.2 that defines several primitive Hazel data types in terms that Smyth can understand. Smyth returns a list of candidate hole fillings which may themselves contain new holes. The user is then free to choose a candidate result. If the chosen candidate has holes, synthesis proceeds on whichever hole the cursor inhabits. Whenever Smyth is

$$\frac{h \in dom(U) \quad \Delta(h) = (\Gamma \vdash \bullet : T) \quad U(h) = X \quad F\,;\, (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\, \Delta' \quad \Sigma\,;\, \Delta +\!\!+ \Delta'\,;\, Merge((U \setminus h\,;\, F) \oplus K) \triangleright K' \quad \Sigma\,;\, \Delta +\!\!+ \Delta'\,;\, Solve(K') \rightsquigarrow F'\,;\, \Delta''}{\Sigma\,;\, \Delta\,;\, Solve(U\,;\, F) \rightsquigarrow F'\,;\, \Delta''} \text{[Solve-One]}$$

Figure 4: Our interface to the Smyth operational semantics

not active, the user is free to commit the current state, backtrack one hole filling at a time, or abort the entire synthesis process.

Working our interaction model around the Smyth formalism was straight forward, as we neither altered the existing semantics nor introduced any new substantial rules. Instead, we produced a harness around the Solve-One rule, as replicated in Figure 4, that fixes the $h \in dom(U)$ to always refer to the hole that contains the user's cursor.

3.1 Implementation

Both Hazel and Smyth are implemented in OCaml, so we were able to get Smyth running within Hazel with a few tweaks to the build config files. Our main challenges consisted of:

(1) serializing and deserializing between Hazel and Smyth terms;

(2) restricting Smyth's synthesis procedure to a single iteration for a user-specified hole; and

(3) implementing the keyboard-navigable user interface for navigating intermediate synthesis results.

All three of us touched all parts of this project, but in rough strokes, Andrew and Eric worked together on (1) and (2) while David worked on (3).

3.2 Conversion and De/Serialization

While Hazel and Smyth operate on similar languages, the devil was in the details when it came to getting them to work properly together, a process which required both functional compromises and special-casing. Hazel does not currently support algebraic data types, though this is currently in the works. Types in Hazel are currently limited to base types for booleans, integers, and strings, as well as function types, lists, and anonymous (n-ary) product and (binary) sum types. In addition, while Smyth supports a fixed-point operator—essentially a lambda with an internally-specified name to enable recursive self-reference—Hazel only permits recursive lambdas when they are directly let-bound to a name. Thus, accurately translating fixes from Smyth to Hazel results in the rather ugly construction seen in Figure 1b. Moreover, because our process is incremental, there is no opportunity to perform a whole-program analysis and selectively rewrite non-recursive fixes into simpler forms; we have to leave the

```
type Bool
  = True ()
  | False ()

type Nat
  = Z ()
  | S Nat

type NatList
  = Nil ()
  | Cons (Nat, NatList)
```

Figure 5: The Shmyth prelude

option open. In our current prototype, we decided to simply dispense with generating fixes, effectively limiting recursion to top-level definitions only.

To resolve the data type issue, we decided to limit ourselves to Booleans, Naturals, and Lists of Naturals, all of which have natural bijections between the languages. We created a prelude defining these ADTs, as shown in Figure 5, which is attached to every Smyth program we generate prior to beginning synthesis. We additionally had to implement special-case translations of the successor function into addition, and the successor pattern into the special-case internal let-binding also seen in Figure 1b. This creates ergonomic issues as a user might attempt to interactively modify these generated subterms in a way which precludes further synthesis. In fact, during testing we would often find ourselves doing something like deleting the "`1`" in an addition and attempting to synthesize the resulting hole. This of course is not possible as the "`+`" operator is not a primitive notion in our translation. Similar hacks were involved in translating "`Cons`" patterns for lists. We are very keen to eliminate this restriction once algebraic data types exist in Hazel.

3.3 Incrementalizing Synthesis

Due to clean code and help from Justin Lubin, this process was generally straightforward, though not without a few snags. The Smyth top-level synthesis loop essentially maintains a stack of hole-numbers and their associated constraints. It recurses through this stack, applying guess-and-check and refinement sub-procedures, which themselves may result in new holes/constraints being added to the stack, until the stack is empty.

We were originally concerned with how we would incrementalize this procedure, thinking we'd need to implement a legitimately interactive stop-and-start process to avoid the costly loss of intermediate results when turning over control to the user. However, we discovered that the primary time costs of synthesis come from the highly non-deterministic branching cases which are precisely the

points of user intervention. Thus, we decided to avoid retaining any synthesis state, and to simply restart the algorithm with the new state of the sketch after the user fills a hole. The result is an approach which is incremental in the sense of this top-level loop, which essentially determines the selections of case splittings, lambdas, and nested applications including literals and variable references, as the latter is handled by a separate guess-and-check procedure. In the future, we might consider extending our incrementalization into the latter, so that only one level of application is generated at a time; at least to see how that effects the feel of the workflow.

But for the moment, our incrementalization modifies the original code so that instead of recurring until the hole stack is empty, we gather the hole numbers in the user's current sketch, and only make the recursive call once over each hole. Originally, we only iterated a single time, over the hole we were trying to fill, until we realized that this was causing problems in the multi-holed case. Generally this resulted in no synthesis results being returned, as each each hole was not able to see the constraints generated by possible synthesis options of the others.

3.4 Synthesis Navigation

The main challenge in implementing the user interface lay in creating an appropriate data structure for representing the user's navigation state within the synthesis tree. Figure 6 shows our encoding of this data structure as a recursive datatype in ReasonML. Type `filled_holes` on Line 34 represents a synthesis tree that maps sketch holes (the key of the map data structure `HoleMap.t`) to hole fillings (`UHExp.t`) and their recursive synthesis trees. Type `t` on Line 42 represents the user navigation state within a synthesis tree; this datatype consists of a pair of a hole identifier (`CursorPath.steps`) and a user-fill state (`z`), where the user may either be choosing among a list of synthesized options presented alongside the constraints for the hole (constructor `Filling` on Line 44) or the user may be deeper in the synthesis tree (constructor `Filled` on Line 45). (Disclaimer: the current names are not optimal.) More details may be found here:

`https://github.com/hazelgrove/hazel/pull/455/files#diff-8aedccbccb2ef26dcf3f2d58d5cf06c16d5b337b55e3447acd855881210b12b4R42`

4 Future Directions for UI

One angle we'd like to examine is making the constraint table itself interactive. Currently Smyth only support top-level assertions, but it was always the intention to extend this to inline assertions. We were thinking that one interesting method to expose this would be in tabular form; the user would simply interact with the constraint table in a spreadsheet-like fashion to compose constraints, which would automatically result in pruning of the synthesis options. We would also explore options for persisting the constraints so-entered, either via inline

```
34  type filled_holes =
35    // need constructor to prevent type synonym cycle
36    | F(HoleMap.t((UHExp.t, filled_holes)));
37
38  /**
39   * Top-down zipper representing synthesis navigation
40   */
41  [@deriving sexp]
42  type t = (CursorPath.steps, z)
43  and z =
44    | Filling(ZList.t(UHExp.t, UHExp.t), Shmyth.constraint_data)
45    | Filled(UHExp.t, filled_holes, t);
```

Figure 6: User-navigable synthesis data structure

assertions, or via a process involving unevaluation and heuristics to synthesize top-level assertions satisfying the desired sub-expression constraints.

We're also keen on exploring options to streamline the table presentation, to better support data structures like lists which have less compact literal presentations. One basic idea is to transpose the table, allowing data elements to occupy rows rather than columns. We might also investigate options for only showing one column/'world' at a time, with the ability to use the arrow keys to navigate between them.

We also intend to examine the Smyth implementation with an eye to enabling some sorts of basic, degenerate synthesis interactions which don't seem to currently work. For example, conceptually speaking it might seem that if we specify no assertions, we should be able to use our UI to navigate the space of well-typed programs in a type-directed way, as every program satisfies an empty set of assertions. Likely this doesn't work as the guess-and-check module might simply be non-terminating in this case, but we'd like to explore generalizing it so that we can use our UI as a kind of type-directed auto-completer, enabling interleaving between a purely type-directed workflow and a synthesis-directed one. Combined with the previously mentioned interactive constraints table, we would be free to switch from purely type-driven to type-and-data-driven synthesis without exiting the UI. Regardless of which workflows we surface, we feel that a flexible approach to disengaging the different components of our workflow could help in comparative testing designed to disentangle the relative benefit of different aspects of the tool, as touched on in the next section.

5 Evaluation

We have used Shmyth's interface to interactively synthesize a variety of small programs, including the `add` function described in Figure 1 as well as additional examples drawn from the original Smyth paper. Here is a link to a Google

Drive folder containing screencasts of Shmyth in action for functions `add`, `odd`, and `max`. (An unrelated bug in Hazel causes there to be spurious and persistent errors in the `odd` example.)

`https://drive.google.com/drive/folders/1jUnAUbAPiDTdgIHyvM52WoJClM7W8cbr?usp=sharing`

In future work, we plan to continue to develop Shmyth and evaluate its usability and effectiveness as a programming assistant with human subjects. In the linked directory, we included a screencast of Shmyth failing to synthesize correctly when the initial sketch for `max` is more incomplete. We plan to investigate such failures and consider how to handle such situations from both technical and human factors perspectives. In addition to studying failure modes, we are interested in consider some of the following research questions:

- How well do users understand the synthesized programs?
- In what situations do users find the synthesized programs unhelpful or confusing?
- How many levels do users go in a typical synthesis transaction?
- How does user disambiguation of nondeterminism affect the typical specification burden compared to push-button synthesis?
- How does this interface affect users' test-writing behavior compared to without it?

6 Conclusion

We believe that future programming tools will follow a trend of modeling the compilation process as a collaboration between between humans and machines. Our goal was to explore how granular the synthesis process could become while still offering meaningful feedback that enhances the developer experience. To these ends, we sought to make such collaboration possible by exposing the intermediary stages of synthesis, and to make relevant data structures as fully transparent as possible. Creating Shmyth and using it to build toy programs has prepared us to design and implement more substantial interactive program synthesizers.

References

[1] Lubin, J., Collins, N., Omar, C., and Chugh, R. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages 4*, ICFP (Aug. 2020), 109:1–109:29.

[2] Omar, C., Voysey, I., Chugh, R., and Hammer, M. A. Live Functional Programming with Typed Holes. *PACMPL 3*, POPL (2019).

[3] Omar, C., Voysey, I., Hilton, M., Aldrich, J., and Hammer, M. A. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)* (2017).

[4] Omar, C., Voysey, I., Hilton, M., Sunshine, J., Goues, C. L., Aldrich, J., and Hammer, M. A. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languagess (SNAPL)* (2017), vol. 71 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 11:1–11:12.