Forging Smyth

The Implementation of *Program Sketching with Live Bidirectional Evaluation*

by

Justin Lubin

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science with Honors

in

Computer Science

at the

University of Chicago

Chicago, Illinois
June 2020

Acknowledgements

First and foremost, I must express my immense gratitude to Ravi Chugh, my advisor and mentor for the past four years, who has invested countless hours in me. You have always looked out for me and encouraged me to take on tasks I never would have dreamed achievable four years ago. I will remain steadfast in my focus on paying forward what you have given me.

Just as important have been Mom and Dad: I cannot overstate my appreciation for your infinite love, patience, and support. Thank you.

Lastly, I would also like to thank Nick Collins and Cyrus Omar for their valiant efforts on the paper that forms the core of this thesis, *Program Sketching with Live Bidirectional Evaluation*, especially in the days leading up to the many deadlines we faced.

Abstract

This thesis is a companion document to the publication *Program Sketching with Live Bidirectional Evaluation* appearing in The International Conference on Functional Programming (ICFP) 2020 by the author of this thesis (Justin Lubin), Nick Collins, Cyrus Omar, and Ravi Chugh. In this thesis, I present the details of the implementation of Smyth (the program synthesizer introduced in *Program Sketching with Live Bidirectional Evaluation*), including critical performance optimizations, an overview of its architecture, and important future work still to be done.

Contents

List of Figures

List of Tables

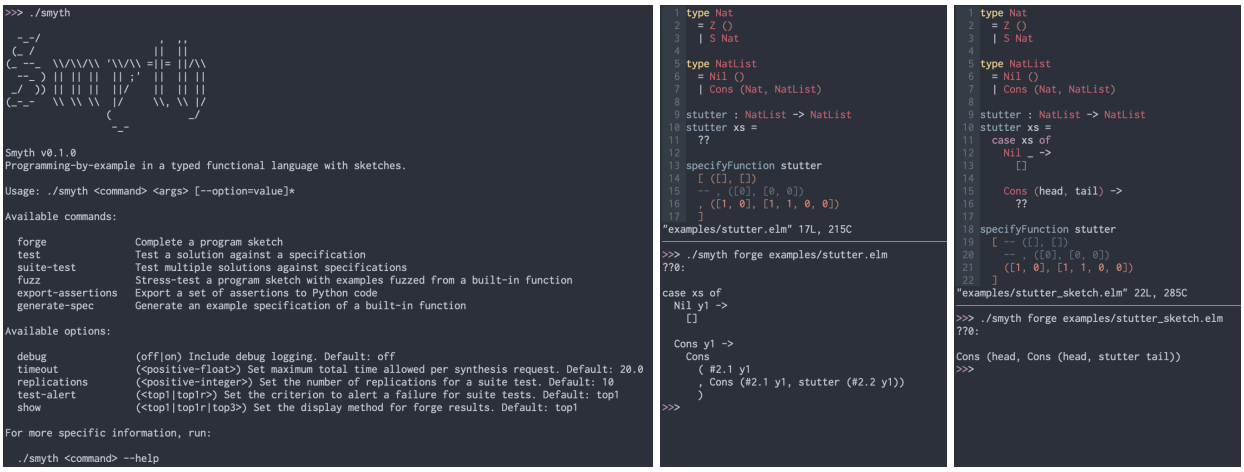

Figure 1: Smyth in action. *Left:* the Smyth help screen. *Center:* "forging" (synthesizing a solution to) an empty sketch. *Right:* forging a sketch that includes the base case.

1 Introduction

This thesis presents the details of my implementation of Smyth, the system introduced in the publication *Program Sketching with Live Bidirectional Evaluation* appearing in The International Conference on Functional Programming (ICFP) 2020 by me (Justin Lubin), Nick Collins, Cyrus Omar, and Ravi Chugh [1].

Smyth is a program synthesizer that fills in "holes" in a program in a typed, functional language (approximately Elm[1]) when given arbitrary assertions about the desired behavior of the code. Smyth is written in OCaml, licensed under the MIT License, and freely available on GitHub at

`https://github.com/UChicago-PL/smyth`.

The Smyth codebase at the time of the submission of this thesis is archived under the `lubin-thesis` tag of the above GitHub repository, and all file paths mentioned in this thesis are relative to the root of that archive.

This project took approximately 16,000 lines of code, with the core algorithm for Smyth taking approximately 6,500 of those lines. The remainder of the code is dedicated to an enhanced standard library of helper functions, the experiments and benchmarks presented in *Program Sketching with Live Bidirectional Evaluation*, and the command-line interface for Smyth (which is depicted in Figure 1).[2]

Although the implementation of Smyth follows the *Program Sketching with Live Bidirectional Evaluation* formalization directly for the most part, there are some critical optimizations and choices that allow Smyth to be realized as code and usably performant. These details are described in Section 2. Following these details is an overview of the architecture of the implementation of Smyth in Section 3, bridging the gap between theory

[1] `https://elm-lang.org/`, accessed June 2020.

[2] Statistics gathered using Tokei, `https://github.com/XAMPPRocky/tokei`, accessed June 2020.

and implementation. Finally, Section 4 concludes with a discussion of interesting avenues of future research at the intersection of human-computer interaction and the theory and implementation of program synthesis.

N.B. The theory of Smyth is in part based on the research detailed in Peter-Michael Osera's PhD thesis [2], which presents its own implementation of a program synthesizer by the name of Myth. In this thesis, I refer to Osera's thesis as "the Myth thesis" and often situate Smyth relative to Myth.

2 Implementation Details and Optimizations

This section begins the overview of the implementation of Smyth by exploring the implementation choices that went into making Smyth realizable as code and usably performant.

2.1 Implementation Techniques Inherited from Myth

First, I will detail the implementation techniques that Smyth inherits from Myth. Notably absent from this section is the notion of preserving information from failed hole fillings across synthesis attempts—an optimization possible in Myth but not directly generalizable to Smyth. Section 4.1.1 discusses this matter further.

2.1.1 Collection Semantics

Myth and Smyth are inherently nondeterministic algorithms. The nondeterminism inherent in the choice of rules (both for unevaluation and for synthesis) in the formalization of Smyth is represented explicitly in the code using the list monad, which corresponds to what Osera describes as "collection semantics" in Section 7.2 of the Myth thesis.

2.1.2 Termination Checking

Myth employs a check during program synthesis to ensure that all recursive calls are *structurally decreasing*, that is, that the argument to a recursive function recursive call is a strict subterm of the parameter to the recursive call.

Smyth employs this same restriction. One limitation of this approach (as implemented) arises from the fact that all functions in Smyth have exactly one parameter (multi-parameter functions are curried). As a consequence, only recursive calls that are structurally decreasing on the "first" argument of multi-parameter (curried) functions are accepted as valid. To sidestep this issue, some of the sketches in the benchmark suites have unconventional orderings of parameters or internal helper functions with one argument.

2.1.3 Timeouts

Like Myth, Smyth cuts off term enumeration after a small amount of time has elapsed (0.25s in both systems). Additionally, Smyth cuts off evaluation after 0.1s to eliminate any unlikely pathological program candidates. Lastly, for usability considerations, Myth and

Smyth both also cut off the entire synthesis procedure if too much time has elapsed since the initial synthesis request.

2.1.4 Staged Synthesis Plan

Myth employs a staged approach to program synthesis whereby specific search parameters begin at highly restricted values and are gradually relaxed as the search with these parameters fails until the search finally succeeds (or times out). The parameters that undergo this progressive staging (known as the "synthesis plan") are:

- *Match depth*, or, how many nested `case` expressions the synthesizer is allowed to synthesize;
- *Scrutinee size*, or, how large the AST of the scrutinee of the synthesized `case` expressions can be; and
- *Guessed term size*, or, how large the AST of the terms that are produced by type-directed term guessing (i.e. "term generation" or "term enumeration") without the guidance from examples can be.

Smyth follows the same approach (with the same parameters) and the same overall synthesis plan as Myth, albeit with slightly more granular stages. The Myth synthesis plan was determined experimentally, and so too was the Smyth synthesis plan. Appendix A of the Myth thesis includes more details.

2.1.5 Informativeness Restriction

Section 7.3.2 of the Myth thesis details how to cut down on redundant pattern matching (an important concern because `case` expressions introduce a high degree of nondeterminism in both Myth and Smyth) by introducing an "informativeness restriction." Smyth follows suit with the same restriction, which disallows pattern matching when fewer than two of the branches get new examples distributed to them (that is, when pattern matching actually does not produce any "new information").

Notably, this restriction prevents underspecification of functions that might actually be desirable. For example, synthesis of the function `head : NatList -> Nat` with the assertion `assert head [1, 2] == 1` might reasonably be expected to produce the following result:

```
head : NatList -> Nat
head xs =
  case xs of
    []     -> ??
    y :: _ -> y
```

However, only the non-empty case of this pattern match has an example distributed to it, so the informativeness restriction criterion fails and this `case` expression cannot be synthesized. The immense performance benefit of the informativeness restriction allows a much broader class of programs to be synthesized while only sacrificing the ability to

synthesize these kinds of underspecified functions more cleverly, making this tradeoff worth implementing overall.

2.1.6 Cached Term Enumeration

Section 7.3.4 of the Myth thesis details how to achieve efficient raw term enumeration based on a careful caching strategy. Smyth follows the same strategy.

2.1.7 Rule Order

Similarly to Myth, Smyth performs "fill" rules in the following order: (1) branch, (2) refine, (3) guess. Branching is prioritized over refinement because constructor refinements idiomatically occur at the leaves of case expressions. Refinement is prioritized over guessing for increased efficiency: there is often no need to "guess" if the example worlds clearly indicate that all examples have a common refinement. However, to be clear, even when refinement is possible, term guessing sometimes actually *is* the correct path to take. This behavior occurs, for example, when Smyth should be synthesizing a call to a helper function whose outputs always have a common refinement, as may be the case, for example, with a helper function that always returns a positive natural number.

2.2 Novel Implementation Techniques

Sketches, live unevaluation, and support for polymorphic type operators pose additional implementation challenges not addressed by Myth. Accordingly, I will now describe the implementation details and optimizations that are novel to Smyth.

2.2.1 Restricting unevaluation rules

The rule U-Case in *Program Sketching with Live Bidirectional Evaluation* presents the most general form of unevaluation onto a case expression, but it is highly nondeterministic and impractical for large synthesis tasks. As an alternative, *Program Sketching with Live Bidirectional Evaluation* presents U-Case-Guess, which "eagerly" fills indeterminate case expressions (that is, case expressions with a hole in the scrutinee) with a "guess." This "guess" truly is just that: only type (and no example) information can be used to fill a scrutinee in an indeterminate case. The implementation of Smyth simply calls raw term enumeration to fill scrutinees of indeterminate case expressions with a term of size 1; slightly larger terms should be, in theory, feasible, but are often unneccessary because the scrutinee of case expressions is typically just a variable name. (This setting worked for all benchmarks and sketches presented in *Program Sketching with Live Bidirectional Evaluation*.)

Smyth first tries to solve synthesis tasks using a restricted form of unevaluation that includes all the rules except for U-Case. If this approach fails, Smyth tries again from scratch using the more general form of unevaluation that includes all the rules except for U-Case-Guess.

2.2.2 Restricting unevaluation depth

To further reduce the nondeterminism inherent in live unevaluation, the implementation of Smyth introduces two new parameters:

- The *unevaluation case budget* (set to 10 in the implementation), which sets a limit to the number of times U-Case is allowed to be called (only applicable when Smyth is in the "unrestricted" unevaluation mode, as described in the previous paragraph).
- The *unevaluation limiter* (set to 10 in the implementation), which immediately cuts off any unevaluation path that exceeds a certain number of nondeterministic possibilities (thereby eliminating pathological search paths that are unlikely to succeed).

2.2.3 Polymorphic Type Operators

Smyth supports both System F universal polymorphism and user-defined polymorphic type operators, an example of the latter being

```
type List a
  = Nil
  | Cons a (List a)
```

(The unapplied type `List` on its own, however, is meaningless in Smyth; full higher-kinded types are not supported.) Chapter 9 of the Myth thesis details how to extend Myth to include System F universal polymorphism (and the details generalize as expected in Smyth), but does not include a description of how to support polymorphic type operators.

The changes to the Smyth codebase to support polymorphic type operators were largely straightforward except for in one place: synthesis of `case` scrutinees. When synthesizing a `case` scrutinee, Myth and Smyth attempt synthesis at every data type in scope. But with the inclusion of a single polymorphic type operator (and a base type), there are an infinite number of data types in scope (for example, `List Nat`, `List (List Nat)`, `List (List (List Nat))`, etc.). All of these are valid types for the scrutinee of a `case` expression. Raw term enumeration occurs at a single type, but in this instance there is an infinite family of types that serve as the goal to term enumeration.

To capture this notion of an infinite class of types, I introduced the notion of a *type wildcard* ($*$) to Smyth and extended the standard syntactic notion of type equality ($=$) to *type matching* ($\equiv_*$):

$$\frac{\tau \text{ Type} \qquad \sigma \text{ Type} \qquad \tau = \sigma}{\tau \equiv_* \sigma} \text{ [Match-Equality]}$$

$$\frac{\tau \text{ Type}}{* \equiv_* \tau} \text{ [Match-Left]} \qquad \frac{\tau \text{ Type}}{\tau \equiv_* *} \text{ [Match-Right]}$$

Scrutinee synthesis then occurs as before (once per data type), but with polymorphic data types instantiated with the wildcard type. Raw term enumeration then substitutes equality for type matching wherever necessary to compensate.

Concept/Figure	File (in `lib/smyth/`)
Syntax of Core Smyth	`lang.ml`
Type checking	`type.mli`
Expression evaluation	`eval.mli`
Resumption	`eval.mli`
Example satisfaction	`example.mli`
Constraint satisfaction	`constraints.mli`
Constraint merging	`constraints.mli`
Live bidirectional example checking	`uneval.mli`
Example unevaluation	`uneval.mli`
Program evaluation	`eval.mli`
Result consistency	`res.mli`
Assertion satisfaction and simplification	`uneval.mli`
Constraint simplification	`solve.mli`
Constraint solving	`solve.mli`
Type-and-example-directed hole synthesis	`fill.mli`
Type-directed guessing (term generation)	`term_gen.mli`
Type-and-example-directed refinement	`refine.mli`
Type-and-example-directed branching	`branch.mli`

Table 1: An injection from concepts in *Program Sketching with Live Bidirectional Evaluation* to the Smyth codebase.

3 Bridging Theory and Implementation

Although *Program Sketching with Live Bidirectional Evaluation* describes the working parts of Smyth at a high level and Section 2 describes them in low-level detail, it may still be hard to visualize the entire structure of the algorithm realized as actual code. This section aims to walk the reader through how to observe the moving parts of Smyth come together in the `UChicago-PL/smyth` repository, serving as a bridge between the formalization and the implementation.

The `lib/smyth` directory of contains all the code for the core implementation of the Smyth synthesis algorithm. The `lib/stdlib2` directory contains helper functions that are used throughout the codebase. The `src` directory contains the code relevant to the command-line interface to Smyth, as well as some code that is used for its experimental evaluation. All parameters used to tweak and tune Smyth are located in the `Params` module found in `lib/smyth/params.ml`.

Table 1 provides a roadmap of where each concept/figure in *Program Sketching with Live Bidirectional Evaluation* can be found in the codebase, in order of presentation in the

paper. The file `lib/smyth/endpoint.ml` contains the `solve` function which takes in a string containing the textual representation of a program sketch and outputs a list of valid hole fillings (synthesis solutions). The `solve` function is called from the command-line interface in `src` and ties all the code together into the "synthesis pipeline" as follows:

1. Parse the given sketch into a `Lang.program` using `Parse.program`.
2. Desugar the `Lang.program` into a `Lang.exp` using `Desugar.program`.
3. Type check the `Lang.exp` against the unit type using `Type.check`.
4. Evaluate the `Lang.exp` using `Eval.eval`.
5. Using the restricted notion of unevaluation from Section 2.2.1, unevaluate any assertions that result from evaluation using `Uneval.simplify_assertions`.
6. Solve the constraints generated by unevaluation using `Solve.solve_any` (the "staged synthesis plan" described in Section 2.1.4 occurs here in the `Solve` module).
7. If Step 6 fails, retry Step 5 with unrestricted unevaluation (as described in Section 2.2.1).
8. Return the result of `Solve.solve_any`; that is, a list of valid hole fillings.

Because this procedure returns a list of hole fillings—all of which are valid—the program must make a choice on how to rank these hole fillings in order of display to the user. In Section 7.2.1 of the Myth thesis, Osera defines the *minimum program principle*, which Myth follows: "in program synthesis, smaller satisfying programs (in terms of program size) are more likely to generalize to the desired behavior intended by the user." Following this same principle, Smyth ranks solutions by AST size, with some special affordances to correct for some necessary boilerplate in our choice of syntax (unit tuples and tuple projections cost nothing to compensate for the fact that all constructors take exactly one argument in Smyth).

4 Future Directions

This section concludes with some avenues of future work, drawing from the theories and practices of both program synthesis and human-computer interaction. Although the challenges to be addressed are here delineated into these two broad categories, true resolutions will need to draw from both disciplines to be maximally successful.

4.1 Avenues in Program Synthesis

4.1.1 Fill Caching

In the implementation of Smyth, when a path in the synthesis search space fails, the corresponding hole filling is discarded. In the implementation of Myth, however, this

information is retained and, in fact, shared between all paths in the synthesis search space. In other words, Myth can store a *single* hole filling that gets updated with information from all paths in the synthesis search space. Partial solutions that have succeeded along the ultimately failed path can then be reused later in the synthesis procedure (such as when the staged parameters are increased).

N.B. Myth keeps track of all synthesis outputs in a data structure called a *refinement tree*, essentially equivalent to Smyth's notion of a *hole filling*. In this thesis, I will refer exclusively (albeit anachronistically) to hole fillings rather than refinement trees.

The reason why Myth can perform this optimization—which I here call *fill caching*—is because hole fillings that are valid in one path in the search space will be valid in *all* paths. This is because new constraints over goals (or, in Smyth, holes) are never generated; all constraints are known up-front. Smyth, on the other hand, can introduce new (possibly conflicting) constraints in different paths in the search space—this is one reason why Smyth does not require trace-complete examples but Myth does.

Hole fillings that are valid in Smyth may only be valid given very difficult or pathological constraints, and might ultimately not lead to the desired solution. Hole fillings in Myth, though, will always be valid, and can thus be cached and shared in a single hole filling accross multiple search branches.

Although it is a fundamental aspect of Smyth that the solutions cannot be directly shared from one branch of the search space to another, a reasonable question is to ask: what knowledge can be gained from failed search paths? Or, put another way, when a synthesis seatch path fails, what can be reused? In Myth, the answer is that all valid work done is reusable. In Smyth, the answer will have to be more nuanced. If there is to be a single hole filling cache, it will likely need to keep track of the exact constraints over other holes used to derive solutions, and cache hits should occur only if these constraints are met in the code relying on the cache. For now, though, this optimization remains an open problem.

4.1.2 Functional Axioms

Systems such as λ^2 [3] have special built-in axioms for particular functions that greatly aid their synthesis search (for example, the axiom that `map g . map f == map (g . f)`). It may be possible to encode these rules as special "introduction forms" for these functions so that examples can be refined according to these axioms, speeding up the synthesis procedure for functions that rely on these general-purpose combinators by taking work done by raw term enumeration (which is just "guessing") and giving it to type-and-example-directed refinement.

4.1.3 Richer Type Systems

Smyth is an evaluator-based program synthesizer and does not rely on logical techniques such as an SMT solver. However, Smyth does take full advantage of a rich (but simple) type system, which indicates that Smyth may generalize to even richer type systems, including

refinement types and more powerful dependent type theories, which may offer paths to merging logical- and evaluator-based synthesis in Smyth.

A simpler direction to extend the type system would be to include base types (such as "native integers" or "native strings"). Naïvely, base types could essentially each be treated as unique constructors over a datatype: example refinement would only occur if each example was exactly the same base type. But this approach does not add much in the ways of expressivity to the system; much more interesting would be to figure out a meaningful way of "refining" base types, perhaps through the addition of functional axioms as described in Section 4.1.2.

4.2 Avenues in Human-Computer Interaction

4.2.1 Solution Ranking

Empirically, the minimum program principle described in Section 3 works well, but it is not the end of the story for program synthesis. Given multiple solutions of similar sizes, which one should a synthesizer display to a user? In what ways can synthesizers explain the *difference* between the synthesized solutions so that the user can be more confident (and accurate) in their selection? Smyth goes no further than the minimum program principle, but if Smyth or other program synthesiers are to ever be usable by a wide audience, these questions must be addressed.

4.2.2 Interactivity

Smyth, via live unevaluation, provides a rich opportunity to take advantage of interactivity by lifting the requirement that all examples be provided up-front. One interesting way to explore this possibility is through the use of a structured editor, such as the lightweight overlay provided by the Deuce editor [4]. Users could hover over expressions in the code and provide examples on-the-fly for what those expressions should evaluate to in a given environment, and Smyth could unevaluate those assertions and perform synthesis with the resulting constraints as usual. This approach and others to enhancing human interactivity in program synthesis will hopefully make program synthesis more accessible and usable by non-experts.

4.2.3 Usability Evaluation

This thesis presents a theoretical formulation and empirical evaluation of Smyth, but stops short of measuring how well actual humans can use the tool. Even simple usability studies to see what *kinds* of examples users provide to a system like Smyth would be a great start toward designing accessible program synthesizers. Other questions that user studies may be able to address include:

- How do users modify their examples when synthesis fails? And how can the synthesizer instruct them in the most effective way to do so?
- When do users reach for program synthesizers over just writing functions themselves? At what point is the burden of writing code greater than that of writing

synthesis specification (and, more specifically to Smyth and other program sketching tools, program assertions)?

Investigation of such questions would be of great value in broadening the class of users who can reap the benefits of research in program synthesis.

References

[1] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. "Program Sketching with Live Bidirectional Evaluation." In: *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming*. ICFP '20. New York, NY, USA: Association for Computing Machinery, 2020. URL: https://arxiv.org/abs/1911.00583.

[2] Peter-Michael Osera. "Program synthesis with types." PhD thesis. University of Pennsylvania, 2015.

[3] John K. Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing Data Structure Transformations from Input-Output Examples." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 229–239. DOI: 10.1145/2737924.2737977.

[4] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. "Deuce: A Lightweight User Interface for Structured Editing." In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 654–664. DOI: 10.1145/3180155.3180165.