EXAMPLE-DIRECTED SYNTHESIS OF A (SMALL) SUBSET OF ELM

Theodore Liu
theodore_liu@college.harvard.edu

Marcy Park
marcellapark@college.harvard.edu

May 10, 2019

ABSTRACT

As our final project for CS 252: Advanced Topics in Programming Languages, we implemented some of the synthesis techniques presented in the 2016 Frankle, Osera, Walker, and Zdancewic paper "Example-Directed Synthesis: A Type-Theoretic Interpretation." Elm, being a pure functional language, is compatible with the type-centric approach indicated in the paper's title. We chose to target Elm (as opposed to the ML-like language targeted in the paper) in the hopes of working out synthesis for the parts of the language that make it useful for building webapps, though this component of the project remains unrealized. We did manage, however, to reproduce the synthesis of several recursive functions based on input-output examples and measure our synthesizer's performance on those test cases.

1 Overview of "Example-Directed Synthesis: A Type-Theoretic Interpretation"

Type-centric program synthesis relies on the Curry-Howard correspondence to conceive of program synthesis as a proof search, in which an inhabitant of a type is to that type as a proof is to a theorem. In the paper, input-output examples are thought of as refinement types that can undergo intersection and union. This intersection and union comes in handy as the synthesizer combines and breaks examples down into a search space for expressions that inhabit the combination of the individual types that the examples indicate. The proof search is done breadth-first, in a proof system (one that in this case can be classified as a sequent calculus, because of the style of logical argumentation that goes on under the hood) expressed as a set of rules that the synthesizer could use to build up a "proof," an expression in the target language that satisfies the input-output examples provided.

The paper also spends a lot of ites time delineating a number of optimizations that make synthesis feasible in a reasonable amount time (i.e. fast enough to be better than just having a human write the program). The most important optimization dealt with how to find the correct variables in context that a function can be applied to. It turns out that naively considering all possible combinations of applications leads to an exponential explosion of the search space. Through clever manipulations and narrowing of that space via type enumeration, they were able to generate small and uniform search spaces which then allowed their synthesis to complete in a reasonable amount of time.

2 Implementation

While the theory of the synthesizer was relatively straightforward and the inference rules were easy to understand, implementing those rules turned out to be a little tricky. It turns out that in many of the rules, there were implicit assumptions being made about the "worlds" that were being refined, which we had to capture in our implementation.

2.1 Understanding prj

In the inference rules, there is frequent use of the judgment

$$\Gamma \vdash x : r \text{ prj}$$

A quick glance at the inference rules for this judgment is opaque and not easy to implement. However, if we follow the rule to its recursive conclusion, we see that this means intuitively that, within context Γ, x is an intersection type like

$$(r_1 \wedge (r_2 \wedge (r_3 \wedge ...)))$$

where for one of those r_i, it has the shape of r. It was possible for there to be multiple r_i that satisfied the "shape" of r (as is the case with functions which are the intersection of many examples), so in our implementation we actually return the list of *all* possible refinement types within x that satisfy r.

2.2 Order and Depth of Inference Rules

In the paper, it is stated that we want to find the *smallest* program that satisfies our examples. We take smallest to mean using the least number of recursive inference rules. To accomplish this, we simply keep a count of the number of inference rules already used and increment this when we use a particular inference rule. When our count exceeds a predetermined limit, we simply return false for any further uses of the inference rules. In the overall synthesizer, we start this global limit at 1 and then search for a program. If nothing is found, we then increase the limit to 2 and repeat.

The order of inference rules is also not specified in the paper. We chose to go the simple route and simply tried all inference rules in order, recursing on them if we found a match. Although we did not try this, the paper mentions that there are actually inference rules that can be "greedily" applied. Essentially, it is always valid to apply those inference rules if they fit the example worlds without potentially "missing" a valid synthesized program.

We also found that changing the order in which the rules were applied could lead to faster synthesis although this could just be attributed to those orderings being better suited to the functions we were trying to generate. Also, it's feasible that there are two programs that match the examples that are of the same depth. Our synthesizer will only find one of those examples, and the ordering of application of the inference rules will determine that.

2.3 Algebraic Data Types

The authors of the paper briefly mention the extension of their system with generic algebraic data types and `match` constructs but never explicitly give the rules for those constructs. We derived and implemented the inference rules for algebraic lists. We give those rules below:

$$\textsc{S-Nil}\ \frac{\forall \langle \Gamma \vdash r\rangle \in \mathcal{W} : \text{Nil} \leq r}{\mathcal{W} \rightsquigarrow \text{Nil sync}}$$

$$\textsc{S-Cons}\ \frac{\langle \Gamma \vdash r_{a1}\rangle \cdots \langle \Gamma \vdash r_{an}\rangle \rightsquigarrow s_1 \text{ sync} \qquad \langle \Gamma \vdash r_{b1}\rangle \cdots \langle \Gamma \vdash r_{bn}\rangle \rightsquigarrow s_2 \text{ sync}}{\langle \Gamma \vdash \text{Cons}(r_{a1}, r_{b1})\rangle \cdots \langle \Gamma \vdash \text{Cons}(r_{an}, r_{bn})\rangle \rightsquigarrow \text{Cons}(s_1, s_2) \text{ sync}}$$

$$\textsc{S-Match}\ \frac{\begin{array}{c}\forall \langle \Gamma \vdash r\rangle \in \mathcal{W}_1 : \Gamma \vdash x : \text{Cons}(r_{ai}, r_{bi}) \text{ prj} \quad \forall \langle \Gamma \vdash r\rangle \in \mathcal{W}_2 : \Gamma \vdash x : \text{Nil prj} \\ h, t \notin \Gamma_1, \ldots, \Gamma_n \quad \langle \Gamma_1 \vdash r_1\rangle \cdots \langle \Gamma_k \vdash r_k\rangle = \mathcal{W}_1 \\ \langle \Gamma_1, h : r_{a1}, t : r_{b1} \vdash r_1\rangle \cdots \langle \Gamma_k, h : r_{ak}, t : r_{bk} \vdash r_k\rangle \rightsquigarrow s_2 \quad \mathcal{W}_2 \rightsquigarrow s_1\end{array}}{\mathcal{W}_1\mathcal{W}_2 \rightsquigarrow \text{match } x \text{ with Nil} \rightarrow s_1 \mid \text{Cons}(h, t) \rightarrow s_2}$$

These are clearly not very pleasant to look at (especially match), but they're simpler to implement than it might initially seem.

2.4 Recursive (Fixed) Functions

Again, the authors say they implemented this feature but give no type theory. We derive the inference rules as follows (based on previous work in the first example-directed synthesis paper):

$$\textsc{S-Fix}\ \frac{\begin{array}{c} f, x \notin \Gamma_1, \ldots, \Gamma_n \quad \langle \Gamma_1, x : r_{a1}, f : (r_{a2} \to r_{b2} \wedge \cdots \wedge (r_{an} \to r_{bn}) \vdash r_{b1}\rangle \\ \cdots \langle \Gamma_n, x : r_{an}, f : (r_{a1} \to r_{b1} \wedge \cdots \wedge (r_{a(n-1)} \to r_{b(n-1)}) \vdash r_{bn}\rangle \rightsquigarrow s \text{ sync}\end{array}}{\langle \Gamma_1 \vdash r_{a1} \to r_{b1}\rangle \cdots \langle \Gamma_n \vdash r_{an} \to r_{bn}\rangle \rightsquigarrow \text{fix } f\, x.s \text{ sync}}$$

Note that in each new "world" i, we leave out the example $f : r_{ai} \to r_{bi}$ since this leads to nonsensical programs like

$$\text{fix } f\, x.f\, x$$

2.5 Optimizations

As we discuss in the results section, our synthesizer is surprisingly fast but starts to struggle when it needs to synthesize recursive functions. Our initial implementation did not use any of optimizations mentioned in the paper. This initial try was able to synthesize almost any non-recursive function but got immediately stuck when given a recursive case.

The only part of the optimizations we ended up implementing was the idea of argument "sudoku" where we essentially eliminated from consideration the functions that did not produce the desired refinement. This turned out to be sufficient to synthesize some recursive functions. Further exploring their type enumeration is something we could forsee doing in the future.

2.6 Lowering to Elm

The end result of our main synthesis portion is actually an abstract syntax tree rather than an actual text program. The final step of our synthesis is a "stringification" step where we take the AST and convert each expression into its equivalent string form in the Elm syntax. We add support for newlines and such so that the program doesn't look completely unreadable.

2.7 Artifact

Find the actual code at `https://github.com/theodoretliu/cs252-project/`.

3 Results

In the table below, we detail the results of our synthesizer. It includes the programs it was able to synthesize and the time it took to synthesize, as well as any comments we might have

Program	Time to Synthesize (s)	Comments
bool-id	0.00003	
bool-neg	0.00005	
bool-or	0.00017	
bool-uncurry-and	0.00013	
bool-impl	0.00026	
bool-xor	0.0006	
list-head	0.00008	
not-list-head	0.000162	
list-tail	0.00005	
list-id	0.00002	
not-of-list	0.00050	
list-fold-and	2.14	
list-fold-or	0.00071	
list-cons	0.00008	
list-append	0.00488	finished but incorrect
list-uncurry-append	0.0272	correct!

In most cases, our synthesizer is very fast! It started to run into trouble with list-append though. In the curried version of append, the code generated was completely incorrect since it made a non-sensical recursive call that would lead to infinite recursion. We suspect this is because the propagation of intersection types for recursive functions is not yet robust enough to handle curried functions. The uncurried version of list-append was able to generate the correct function! That was cool but also expected and seems to solidify the point that there is some shortcoming in our implementation of recursive functions, which means they can only deal with one argument.

4 Limitations (Of Which There Are Many) and Next Steps

As exciting as it was to watch our synthesizer come up with functions to handle booleans and recursive functions like not-of-list, list-and, list-or, and list-uncurry-and, there is much more it will need to do to become useful.

- **list-append**. There are some more complex recursive functions, such as list-append which takes two curried arguments, that at the moment, our synthesizer is unable to correctly construct. In the specific case of list-append, we feel this is a bug with our implementation or derivation of the inference rule for "fix" functions that we can correct in the future. However, as we try to synthesize more and more complicated functions with more complex examples, we will eventually run into a search space problem. At that point, we will have to implement some or all of the type enumeration optimizations that are explained in the latter half of the paper.

- **More types**. As of now, our synthesizer knows of booleans and lists of booleans, but not of numbers, strings, trees, or other data types. Some of these may be natural extensions of what's already constructed. For example, trees follow naturally from the inference rules we derived for algebraic lists. Even natural numbers, which the authors implement, are thought of as algebraic constructs using the zero-successor formulation. Strings could be thought of as character lists. More complicated types are completely beyond the scope of what this paper describes and would require further research. For example, floats or doubles would be difficult to synthesize since they cannot be expressed as singleton types. Even a theory of negative numbers is not really provided by the paper. These are further things to explore.
- **Library sampling**. One way Frankle, Osera, Walker, and Zdancewic get more expressiveness out of their synthesizer is by sampling from libraries. This might be a nice way to add the functionality that usually comes with the datatypes widely available in most popular programming languages. For example, with integers, we get addition, multiplication, comparison, etc. for free. The challenge in allowing our synthesizer to call into libraries would be generating the right refinements to describe the behavior of library functions while we are synthesizing. Again, there is the potential for combinatorial explosion if we broaden our search too much with the presence of too many libraries. The paper mentions a balance of ahead-of-time and just-in-time sampling during argument selection. The downfall of ahead-of-time sampling is the expense of generating massive tables describing various library expressions' behavior on different arguments, and the downfall of just-in-time sampling is that the synthesizer will not always have candidate arguments expressed as specific enough refinements to undergo standard evaluation as the arguments of the library functions being considered.
- **Specify to Elm lang**. We make it no secret that our synthesizer currently only targets the simple functional part of Elm. But Elm is a rich language and the Elm architecture makes manipulations of the DOM and state first-class members, such as the HTML messages that represent user interactions with the DOM. If we incorporate more of the theory behind algebraic datatypes and library sampling, it could be possible in the future to synthesize *actually interesting* components of Elm webapps.

5 Conclusion

In implementing this paper, we have gained a much deeper understanding of example-directed, type-guided synthesis and a great appreciation for the difficulty and complexity of cutting-edge programming languages research. On the one hand, we are pleasantly surprised that we are able to synthesize anything at all given a few small examples. On the other hand, it's slightly disappointing that all this type theory and code can only generate seemingly trivial programs that anyone who has taken CS51 could write. Still, program synthesis has been incredibly interesting to study and seems like a current and open problem within programming languages, so we are excited for what the future holds.

Thank you for a great course!