CS 395T Final Project: Synthesizing Recursive Programs Using Only Enumeration

James Dong

May 19, 2021

1 Problem Statement

In this paper, I will describe a method for synthesizing recursive programs from input-output examples using only enumeration. Specifically, this approach focuses on three main challenges:

- Base cases are allowed to be arbitrary expressions, not just integer constants. This makes symbolic search difficult.
- The recursion does not have to have a fixed form, but is user provided. This also makes symbolic search somewhat more difficult.
- The recursion is allowed to be of arbitrary dimensionality, which can make reasoning about the recursive case more difficult.

To address these challenges, I make the following trade-offs:

- The recursion must be user-provided. Synthesizing recursion through enumeration raises several challenges, the most important being termination, which is not guaranteed.
- The set of examples must be trace-complete. This is used during the synthesis process to derive the conditions for the recursive case.

1.1 Motivating Exxample

For the rest of this paper, I will be referring to the following motivating example: binary search.

The signature of binary search is

$$\text{bsearch} : (arr : \textbf{int}[], el : \textbf{int}, lo : \textbf{int}, hi : \textbf{int}) \to \textbf{int}.$$

As an informal specification, binary search takes as input a sorted array and an element to search for, as well as a low and high bound, and returns either the index of the element to be found or the two's complement of the index it would be located in if inserted. Note: for an integer i, the two's complement of i is equal to $-i - 1$.

2 DSL Description

The synthesis DSL has three basic data types: booleans, integers, and arrays (of arbitrary dimensionality). In addition, I also support positive and negative infinity (to make taking the maximum and minimum more natural) as well as a special "error" value which is generated on out-of-bounds accesses as well as recursion which is not contained in the example set; if any sub-expression of an expression evaluates to the error value, the whole expression will also evaluate to error. This is used to prevent runtime errors as well as guiding the condition generation for recursive cases.

The following describes the DSL for a program $P : (T_1, \ldots, T_n) \to T$:

$$
\begin{aligned}
P &::= g \to e_T \mid P; g \to e_T \\
g &::= e_{\mathbf{bool}} \mid g \wedge e_{\mathbf{bool}} \\
e_{\mathbf{bool}} &::= e_{\mathbf{int}} = \top \mid \bot \mid e_{\mathbf{int}} = e_{\mathbf{int}} \mid e_{\mathbf{int}} \neq e_{\mathbf{int}} \\
&\quad \mid e_{\mathbf{int}} < e_{\mathbf{int}} \mid e_{\mathbf{int}} > e_{\mathbf{int}} \mid e_{\mathbf{int}} \leq e_{\mathbf{int}} \mid e_{\mathbf{int}} \geq e_{\mathbf{int}} \\
&\quad \mid f_{\mathbf{bool}}(e_{t_1}, \ldots, e_{t_n}) \\
e_{\mathbf{int}} &::= n \mid \pm\infty \mid e_{\mathbf{int}} + e_{\mathbf{int}} \mid e_{\mathbf{int}} - e_{\mathbf{int}} \mid e_{\mathbf{int}} \times e_{\mathbf{int}} \mid e_{\mathbf{int}} / e_{\mathbf{int}} \\
&\quad \mid f_{\mathbf{int}}(e_{t_1}, \ldots, e_{t_n}) \\
e_\tau &::= e_{\tau[]}[e_{\mathbf{int}}] \mid \ldots \\
e_T &::= P(e_{T_1}, \ldots, e_{T_n}) \mid \ldots .
\end{aligned}
$$

Each expression e also has a structural cost c_e, which is $|n|$ for a constant n and generally equal to $1 + \sum_{s \in e} c_s$ for an expression consisting of subexpressions s (for some benchmarks, the cost of some operations is increased to 2, but I have not tested if this makes a difference).

As an example, the motivating example binary search can be expressed as the following DSL program (used as the reference solution to generate trace-complete examples):

$$
\begin{aligned}
hi \leq lo &\to -lo - 1 \\
arr\left[\frac{lo+hi}{2}\right] = el &\to \frac{lo+hi}{2} \\
arr\left[\frac{lo+hi}{2}\right] > el &\to \text{bsearch}\left(arr, el, lo, \frac{lo+hi}{2}\right) \\
arr\left[\frac{lo+hi}{2}\right] < el &\to \text{bsearch}\left(arr, el, \frac{lo+hi}{2} + 1, hi\right).
\end{aligned}
$$

3 Overview

The overall synthesis algorithm is very simple. First, enumerative search with pruning using observational equivalence is used to find an expression that min-

imizes a loss function, defined as

$$L(e) = c_e + \alpha \frac{|\{(i,o) \in E \mid [\![e]\!]_i \neq o\}|}{|E|},$$

where E is the set of input-output examples, α is a parameter that controls the trade-off between simplicity and generality, and $[\![e]\!]_i$ is the value of e when applied to the input i. Furthermore, we require that at least one example is satisfied to ensure forward progress.

Next, after having generated a candidate expression, we then generate the boolean guard for that expression. We do this by once again using enumerative search, where this time the set of input-output examples is

$$E_e^{\mathbf{bool}} = \{(i, [\![e]\!]_i = o) \mid (i,o) \in E\}.$$

However, there are a few caveats. First, a guard is never allowed to return the error value. Next, neither $\top$ nor $\bot$ are permissible as guards, except when all examples are satisifed.

At this point, for a guard g, we may have false positives (where an incorrect answer would be produced, which cannot be allowed) and false negatives (which are OK). To eliminate false positives, we continue using enumerative search only on the set of examples for which g returns true. This process is repeated until all examples are satisfied.

Since at every point the partial program satisfies an increasing subset of this examples and terminates when this subset equals the whole set, this procedure is trivially sound with respect to the examples. Furthermore, since progress is always possible (in the worst case, covering one example each time), it is also complete.

3.1 Enumeration Approach

A key part of this approach is *cost-based enumeration*, which allows the search to guarantee optimality of the loss function at each iteration. The idea that allows this is that the number of examples unsatisfied is always at least zero, so if the current optimum has loss L, it is impossible for any expression with $c_e > L$ to be optimal. This means if we search in order of increasing cost, we can stop the search as soon as we encounter such an expression.

The search methodology can be framed as graph search: we connect each expression to all expressions that contain it as a direct sub-expression; then we use Dijkstra's algorithm, which visits expressions in order of increasing cost.

However, this naive approach is obviously intractable as each expression is contained in an infinite number of expressions. As such, I employ a symmetry-reducing optimization which connects each expression to a finite number of containing expressions, one for each grammar rule, and add horizontal links between expressions of the same "level".

First of all, looking at it backwards, the natural choice for the *child* of an expression is its first direct sub-expression. As such, we connect each expression

only to expressions that contain it as its *first* child, which obviously does not impact completeness. Next, among these expressions, the expression with lowest cost is the one in which all other children have lowest cost as well. For example, the expression $a + 0$ is cost-minimal among all expressions of the form $a + e$.

Next, to form the rest of these expressions, we add *horizontal* links which increase cost. First, we define $\text{succ}(e)$ to be the expression explored after e with the same type. This defines a canonical ordering of expressions in order of increasing cost. From this, we can easily define the horizontal links for binary operations: the expression $a + b$ is linked to $a + \text{succ}(b)$. To generalize this for operations with arbitrary arity, we first eliminate symmetry by requiring sub-expressions be increased from left to right; then for an expression $f(e_1; e_2, \ldots, e_n)$, we add the following links:

$$\begin{aligned} f(e_1; e_2, \ldots, e_n) &\to f(e_1; \text{succ}(e_2), e_3, \ldots, e_n) \\ f(e_1; e_2, \ldots, e_n) &\to f(e_1, e_2; \text{succ}(e_3), e_4, \ldots, e_n) \\ &\vdots \\ f(e_1; e_2, \ldots, e_n) &\to f(e_1, \ldots, e_{n-1}; \text{succ}(e_n)) \end{aligned}$$

where the notation $f(e_1, \ldots, e_i; e_{i+1}, \ldots, e_n)$ indicates that the expressions left of the semicolon are *fixed* and should not be involved in any horizontal links.

4 Evaluation

I implemented my tool using Kotlin; the implementation is not very optimized and runs on a single thread, so performance can likely be substantially improved. Furthermore, during the synthesis loop a timeout of 10 minutes is applied (this is only triggered during levenshtein). The code is available at https://github.com/j-dong/basecase.

I evaluated the tool on 6 benchmarks collected from textbooks (I would have collected more, but it took me a long time to get the tool working as evaluation uncovered many bugs). Evaluating against baselines is left as an exercise to the reader; benchmarks are available in the `benchmarks` folder.

- add: add two integers
- fib: Fibonacci sequence
- cut_rod: rod-cutting problem from CLRS
- bsearch: binary search where not found returns -1
- bsearch_fancy: binary search where not found returns $-i - 1$
- levenshtein: edit distance (LCS, not actually Levenshtein distance)

For each benchmark, the following information is provided: the name of the benchmark and its signature, a reference implementation (not provided to the synthesizer but used only to expand the inputs), a set of "seed" expressions including recursive calls (a seed expression can be used to prioritize certain high-cost expressions), a set of initial inputs (which is expanded using the reference implementation to be trace-complete), the set of allowable grammar rules and built-in functions, and the correctness parameter α.

5 Results

All benchmarks but levenshtein were able to be synthesized correctly.

add : $(x : \textbf{int}, y : \textbf{int}) \to \textbf{int}$ (synthesized in 0.02s)

$\top \to x + y$

fib : $(x : \textbf{int}) \to \textbf{int}$ (synthesized in 0.04s)

$1 < x \to \text{fib}(x - 1) + \text{fib}(x - 2)$

$\top \to x$

cut_rod : $(p : \textbf{int}[], n : \textbf{int}, i : \textbf{int}) \to \textbf{int}$ (synthesized in 108s)

$i \neq 1 \to \max(\text{cut\_rod}(p, n - i, n - i) + p[i - 1], \text{cut\_rod}(p, n, i - 1))$

$0 < n \to -\infty$

$\top \to n$

bsearch : $(arr : \textbf{int}[], el : \textbf{int}, lo : \textbf{int}, hi : \textbf{int}) \to \textbf{int}$ (synthesized in 37s)

$$\frac{lo + hi}{2} \neq lo \wedge el < arr\left[\frac{lo + hi}{2}\right] \to \text{bsearch}\left(arr, el, lo, \frac{lo + hi}{2}\right)$$

$$\frac{lo + hi}{2} \neq lo \wedge el \neq arr\left[\frac{lo + hi}{2}\right] \to \text{bsearch}\left(arr, el, \frac{lo + hi}{2} + 1, hi\right)$$

$$lo \geq \frac{lo + hi}{2} \wedge hi \leq \frac{lo + hi}{2} \to -1$$

$$el = arr\left[\frac{lo + hi}{2}\right] \to \frac{lo + hi}{2}$$

$\top \to -1$

$\text{bsearch\_fancy} : (arr : \textbf{int}[], el : \textbf{int}, lo : \textbf{int}, hi : \textbf{int}) \to \textbf{int}$ (synthesized in 63s)

$$\frac{lo+hi}{2} \neq lo \wedge el < arr\left[\frac{lo+hi}{2}\right] \to \text{bsearch\_fancy}\left(arr, el, lo, \frac{lo+hi}{2}\right)$$

$$\frac{lo+hi}{2} \neq lo \wedge el \neq arr\left[\frac{lo+hi}{2}\right] \to \text{bsearch\_fancy}\left(arr, el, \frac{lo+hi}{2}+1, hi\right)$$

$$\frac{lo+hi}{2} \neq lo \to \frac{lo+hi}{2}$$

$$hi \leq \frac{lo+hi}{2} \to -1 - hi$$

$$el = arr\left[\frac{lo+hi}{2}\right] \to \frac{lo+hi}{2}$$

$$el > arr\left[\frac{lo+hi}{2}\right] \to \text{bsearch\_fancy}\left(arr, el, \frac{lo+hi}{2}+1, hi\right)$$

$$\top \to \text{bsearch\_fancy}\left(arr, el, lo, \frac{lo+hi}{2}\right)$$

The following incorrect program is synthesized for levenshtein (note: this was synthesized before I adjusted how α worked; as a result, the loss function used is $L(e) = c_e + |\text{incorrect}|$. The same result could likely be produced by choosing an appropriate α.)

$\text{levenshtein} : (a : \textbf{int}[], b : \textbf{int}[], x : \textbf{int}, y : \textbf{int}) \to \textbf{int}$ (synthesized in 2874s)

$$\min(y, x) \neq 0 \wedge a[x-1] \neq b[y-1] \to$$
$$\min(\text{levenshtein}(a, b, x, y-1), \text{levenshtein}(a, b, x-1, y)) + 1$$

$$3 \geq \min(y, x) \to -\min(y - x, x - y)$$

$$\top \to \text{levenshtein}(a, b, x-1, y-1)$$

The example input given is ("algorithm", "altruistic", 9, 10), where the each character is mapped to $1000 + c$, where c is 1 for 'a', 2 for 'b', etc. We see that this program overfits the example input, as it exploits the fact that both strings start with the substring "al", as seen in the second branch. This could be fixed by adding more examples.

6 Related Work

The "inspiration" for this project is "Synthesis of First-Order Dynamic Programming Algorithms" by Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. The three main challenges I address are the main limitations of this paper, which is due to their symbolic approach.

"Synthesizing Data Structure Transformations from Input-Output Examples" (a.k.a. λ^2) by John Feser, Swarat Chaudhuri, and Işıl Dillig approaches a similar problem using a top-down approach, but they require a fixed library of

combinators as opposed to a user-provided general recursion scheme. Furthermore, for each combinator, some domain knowledge is necessary to exploit the examples.

Two other papers, "Type-and-Example-Directed Program Synthesis" (a.k.a. MYTH) by Peter-Michael Osera and Steve Zdancewic and "Recursive Program Synthesis" by Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid apply a very similar approach using bottom-up enumeration and unification, but my approach is distinguished from them by the optimality of the enumeration, which is a stronger bias than e.g. the greedy approach used by "Recursive Program Synthesis".

7 Future Work

Eliminating the two major trade-offs of my approach is a possible avenue for future work, but this is likely very challenging for the type of enumeration-based synthesis used.

One issue with my approach is that it is highly dependent on the distribution of examples, which influences which expressions are optimal. This can be seen in the binary search benchmark, where I had to specify several examples to cover ways in which the synthesis algorithm was overfitting.

To address this and also generalize the problem, it is a very promising idea to use CEGIS to generate a solution to a high-level specification not in terms of input-output examples. The CEGIS approach would then automatically take care of distributing examples such that the resulting program generalizes.