Enumerative Type-Guided Program Synthesis for Higher-Order Polymorphic Language

DARYA VERZHBINSKY, University of California, San Diego
DANIEL WANG, University of California, San Diego

1 INTRODUCTION

Consider the task of implementing a function that has the following description: "If I give you a function, apply it to `x`. Otherwise, return `x` unchanged." We can translate this function description into a Haskell type signature as follows:

$$f : Maybe(a \to a) \to x : a \to a \tag{1}$$

where `f` represents a possible function (hence the `Maybe` type), and `x` is of the same type that `f` takes in and returns. A possible solution to this type signature would be:

```
\f x -> fromMaybe (\y -> y) f x
```
(2)

which either extracts the function from the `Maybe` and applies it to `x`, or applies the identity function to `x` if the `Maybe` is `Nothing` (which is the same as returning `x` unchanged). This solution, though, is potentially non-intuitive and could be difficult for programmers to find on their own. So, as Haskell programmers, we would probably look to Hoogle [reference] to figure it out for us. Here is some of what Hoogle gives you with query `Maybe (a->a) -> a -> a`:

- `(??) :: Functor f => f (a->b) -> a -> f b`
- `(<$~>) :: Functor f0 => f0 (a -> b) -> a -> f0 b`
- `flap :: Functor f => f (a -> b) -> a -> f b`

(**TODO:** we're not quite sure what these functions are so we aren't sure what the problem with them is.) Hoogle also only gives you single library functions back, and not programs consisting of compositions of library functions like our desired solution requires.

We set out to build an enumerative synthesis tool that does just this. We wanted users to be able to provide a type signature and optional examples, and get back a program that consists of a composition of Haskell library functions. This would allow them to speed up their programming by using our tool to synthesize small programs instead of doing it themselves.

2 RELATED WORK

The novel problem we explore is polymorphic type-guided synthesis via enumerative search. This problem has been explored before.

MYTH. Myth already explores enumerative search for type-directed synthesis, but it does not support polymorphism or typeclasses like we'd like our tool to. It has some interesting enumeration techniques that we took inspriation from. See Approach for more details about this.

TYGAR. TYGAR [link] uses petri nets to tackle type-guided Haskell program synthesis, and is remarkable in that it also supports polymorphic types, typeclasses, and can check user-provided input-output examples. We focus on the same problem, but provide a simpler method: enumerative search over the space of all programs. In addition, TYGAR searches a more limited search space of functions, as it does not make use of lambdas.

Authors' addresses: Darya Verzhbinsky, dverzhbi@ucsd.edu, University of California, San Diego; Daniel Wang, University of California, San Diego, @ucsd.edu.

SYNQUID. Synquid makes use of an enumerative search approach, lambdas, and polymorphism. It also includes refinement types, though, which we don't care about for our problem. While it is very close to our desired functionality, it considers a limited search space in comparison to the one we'd like to search, as its type unfier doesn't allow for types of different arrities to unify. For example, $Maybe\ a \to a$ and $\alpha_1 \to \alpha_0 \to b$ wouldn't unify (but they could if $Maybe\ a$ unified with α_1, and a unfied with $\alpha_0 \to b$). This becomes an issue when dealing with higher-order queries, as it leads to certain programs being unreachable.

3 APPROACH

Our approach is based on Synquid, and on top of that we address three problems:

- Limitations with Synquid's enumeration strategy
- Memoization (caching subproblems during enumeration)
- Extending an optimization from Myth

3.1 Limitations with Synquid's enumeration strategy

Synquid synthesizes function applications in the following way. Given a goal $?? :: T$ that is an application, split it into two goals for the function and argument: $(?? :: \alpha \to T)$ and $(?? :: \alpha)$. Repeating this process in the first goal lets us use functions that take in more arguments:

$$(?? :: \alpha_0 \to T)(?? :: \alpha_0)$$
$$(?? :: \alpha_1 \to \alpha_0 \to T)(?? :: \alpha_1)(?? :: \alpha_0)$$
$$(?? :: \alpha_2 \to \alpha_1 \to \alpha_0 \to T)(?? :: \alpha_2)(?? :: \alpha_1)(?? :: \alpha_0)$$

This could repeat forever. To ensure search terminates, the components decide the cutoff point: if the longest arrow type in the components is $a \to b \to c \to d$, then we stop splitting the goal once we see $\alpha_2 \to \alpha_1 \to \alpha_0 \to T$.

However, this approach fails when type variables are instantiated with arrow types. One concrete example is the query $(a \to b, a) \to b$ with components `[fst, snd]`. The solution is `fst p (snd p)`, which uses `fst` as a two-argument function, which means `fst` must have filled the goal $\alpha_1 \to \alpha_0 \to T$. However, `fst` and `snd` are one-argument functions – the cutoff is $\alpha_0 \to T$, so we never end up considering the goal $\alpha_1 \to \alpha_0 \to T$.

Our solution is to use iterative deepening on the size of programs, rather than depth first search. Because we only enumerate programs up to a certain size anyways, we can be rid of the decision of "where to stop splitting the goal".

We also replace the measure of program size with "number of components" rather than "recursion depth", because size tends to generate deeper programs earlier, and deep programs like `f (g (h x))` tend to be more programmer-relevant than flat programs.

3.2 Memoization (caching subproblems during enumeration)

Because of its enumerative and iterative deepening qualities, our tool generates many repeated subproblems over the course of the search, wasting a lot of time redoing searches it has already done. This makes memoization crucial to practical top-down enumeration, because goals can contain free type variables, which depend on the context surrounding the goal. This means solution retrieval from this cache must take into account the surrounding context – but storing the whole context in the cache is a messy affair.

Our memoization strategy resolves this by renaming every free variable, such that two goals different only in name (like `tau1 -> b` and `tau2 -> c`) are both cached to the same set of solutions. At time of retrieval, we use type inference to infer the type of retrieved programs, which makes

Table 1. TYGAR vs. Our tool

N	Name	Query	Time (s): TYGAR	Time (s): Ours
1	firstMatch	[a] -> (a -> Bool) -> a	6.04	6.92
2	appBoth	(a -> b) -> (a -> c) -> a -> (b, c)	8.53	7.5
3	test	Bool -> a -> Maybe a	7.96	5.95
4	hoogle01	(a -> b) -> [a] -> b	7.58	5.12
5	firstJust	a -> [Maybe a] -> a	18.36	6.5
6	mapEither	(a -> Either b c) -> [a] -> ([b], [c])	4.78	3.2
7	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b	8.03	5.2
8	mergeEither	Either a (Either a b) -> Either a b	3.77	181.45
9	mbToEither	Maybe a -> b -> Either a b	12.74	4.21
10	head-rest	[a] -> (a, [a])	1.53	3.26
11	pred-match	[a] -> (a -> Bool) -> Int	5.18	9.83
12	map	(a->b)->[a]->[b]	2.07	3.72
13	repl-funcs	(a->b)->Int->[a->b]	1.12	1.19
14	mbAppFirst	b -> (a -> b) -> [a] -> b	3.24	9.54
15	2partApp	(a->b)->(b->c)->[a]->[c]	3.36	10.62
16	resolveEither	Either a b -> (a->b) -> b	4.03	10.3
17	splitStr	String -> Char -> [String]	1.29	4.3
18	multiApp	(a -> b -> c) -> (a -> b) -> a -> c	6.85	9.56
19	singleList	a -> [a]	7.72	4.41
20	head-last	[a] -> (a,a)	257.34	12.02

use of the context in which retrieval happens. This tackles the problem of "how do we store the context" by instead adding the context at the site of retrieval.

3.3 Extending an optimization from Myth

For purposes of solution quality, we constrain our synthesizer to only produce programs that make use of all the arguments provided. The Myth paper introduces an optimization for synthesizing function applications of type T that must include some component x. We extend this to the case of multiple arguments.

4 RESULTS AND CONTRIBUTIONS

We tested our results on 20 queries and compared our tool with TYGAR, which is the tool that most resembles our own. The results can be found in Table [1].

4.1 TYGAR vs. Memoization

TYGAR did better than our tool in 11 out of the 20 tests. For the tests that TYGAR did better in, though, it did significantly better (see Test 8, where TYGAR was almost 48 times faster). The tests that we did better in mostly only had a slight speedup. We take this to mean that for now, TYGAR is still the better tool. This doesn't discourage us, however, for a couple reasons:

(1) Our tool takes advantage of lambdas and TYGAR doesn't, meaning that our tool enumerates more programs, making it potentially more expressive. This could be part of the reason ours is slower.
(2) So far, we have only implemented the most basic form of memoization. There is plenty of room to improve upon this (see Future Work), so we are hopeful that we can continue to make our tool faster and have it be a competitive alternative to TYGAR.

Quality of Results. For the most part, both TYGAR and our tool returned the same programs. If they didn't return the same program, the programs were equivalent, so the differences between the results of the two tools is insignificant.

5 FUTURE WORK

Our memoization tool is still very basic and we think there are multiple ways in which we can improve upon what we have:

(1) Re-organize our memo map to store programs first based on size, and then find programs based on query. This would make lookup much faster.
(2) Take advange of sub-typing in the memo keys so that the same programs aren't stored multiple times. For example, all programs in goal $Int \rightarrow Int$ should be in goal $\alpha_0 \rightarrow Int$. When we lookup $\alpha_0 \rightarrow Int$, we could first look at $Int \rightarrow Int$ and then move on to other programs that are more general.

REFERENCES

TODO