Petsy: Polymorphic Enumerative Type-Guided Synthesis

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:

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

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 problem is:

```
\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). As a component based program, it is idiomatic and concise, and doesn't rely on pattern matching. It is potentially non-intuitive, though, and could be difficult for programmers to find on their own.

We present Petsy, a tool that allows users to provide a type signature and optional examples, and get back a code snippet that consists of a composition of Haskell library functions. This allows users to speed up their programming by using our tool to synthesize small Haskell programs instead of doing it themselves.

2 RELATED WORK

The problem we are exploring is polymorphic type-guided synthesis via enumerative search. Related problems have been explored before.

MYTH. Myth [3] already explores enumerative search for type-directed synthesis, but it does not support polymorphism or typeclasses like we'd like Petsy to.

Tygar. Tygar [2] 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. Because of its use of Petri nets, however, it has to reduce everything to first order, meaning that it can't make use of lambdas, which limits the space of programs that Tygar can pull from.

SYNQUID. Synquid [4] makes use of an enumerative search approach, lambdas, and polymorphism. Its type system, however, restricts type variables from unifying with arrow types. This prevents, for example, `Maybe` $a \to a$ and $\alpha_1 \to \alpha_0 \to b$ from unifying, even though in theory they could if `Maybe` a unified with α_1, and a unfied with $\alpha_0 \to b$. This leads to some programs being unreachable and, in the context of higher-order components, limits the search space.

Authors' addresses: Darya Verzhbinsky, dverzhbi@ucsd.edu, University of California, San Diego; Daniel Wang, University of California, San Diego, daw086@ucsd.edu.

3 APPROACH

3.1 Synthesis Problem

Our synthesis *problem* consists of a component library, a query type, and optional input-output examples. A *solution* to the synthesis problem is a Haskell code snippet that has the desired type and works for the optionally given examples.

3.2 Extending Synquid's enumeration strategy

Petsy builds upon Synquid, which solves its synthesis task by performing a top-down enemerative search and taking advantage of refinement types that are included in the initial type query.

However, Synquid doesn't support cases where 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.

3.3 Memoization (caching subproblems during enumeration)

Because of its enumerative and iterative deepening qualities, Petsy 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, which Myth [3] was able to solve in a first-order setting. In our polymorphic context, however, memoization is non-trivial. With polymorphism, goal types can now contain free type variables. This causes the following issues.

Redundancy. In Petsy, free variables are represented as α's. If you look at 2 goals, $\alpha_0 \to Int$ and $\alpha_1 \to Int$, even though they are syntactically different because of the subscripts, they are semantically equivalent because each is essentially saying "I want a program that takes in anything and returns an `Int`". Because of their semantic difference, however, they would be stored in 2 different locations in the memo map, resulting in the exact same programs being stored in 2 different locations. How can we avoid this and make these two goals equivalent?

To resolve this, we rename any incoming free type variables to be "fresh" β's before using them as the map lookup key. That way, if two different goals with free variables come in, say $\alpha_0 \to Int$ and $\alpha_1 \to Int$, they will each be renamed to $\beta_0 \to Int$ and $\beta_0 \to Int$, meaning that they will hit the same entry in the map, despite having different original names.

Type unificiation clashes. Another issue comes into play when we're trying to synthesize arguments to a component and the component's type contains free variables (that are yet to be unified with a concrete type). Say, for example, we generate and store a program that has α_3 in its type, and at at the time α_3 is free. When we retrieve this program later in the search, α_3 might now be in use for something else, and we would be linking the two free variables unintentially and creating a clash.

To avoid this clash, instead of relying on the stored type, which in this case depends on α_3, we infer the type of the retrieved program by generating a "fresh" type, A_0 for example, independantly

Table 1. Tygar vs. Petsy

N	Name	Query	Time (s): Tygar	Time (s): Petsy	Speedup
1	appBoth	(a -> b) -> (a -> c) -> a -> (b, c)	8.5	7.5	1.137333333
2	test	Bool -> a -> Maybe a	8.0	6.0	1.337815126
3	both	(a -> b) -> (a, a) -> (b, b)	4.1	-	
4	firstJust	a -> [Maybe a] -> a	18.4	6.5	2.824615385
5	mapEither	(a -> Either b c) -> [a] -> ([b], [c])	4.8	3.2	1.49375
6	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b	8.0	5.2	1.544230769
7	mergeEither	Either a (Either a b) -> Either a b	3.8	181.5	0.02077707357
8	mbToEither	Maybe a -> b -> Either a b	12.7	4.2	3.026128266
9	cartProduct	[a] -> [b] -> [[(a,b)]]	-	-	
10	multiAppPair	(a -> b, a -> c) -> a -> (b, c)	14.0	-	
11	map	(a->b)->[a]->[b]	2.1	3.7	0.5564516129
12	repl-funcs	(a->b)->Int->[a->b]	1.1	1.2	0.9411764706
13	mbAppFirst	b -> (a -> b) -> [a] -> b	3.2	9.5	0.3396226415
14	2partApp	(a->b)->(b->c)->[a]->[c]	3.4	10.6	0.3163841808
15	resolveEither	Either a b -> (a->b) -> b	4.0	10.3	0.3912621359
16	dedupe	Eq a => [a] -> [a]	19.0	-	
17	multiApp	(a -> b -> c) -> (a -> b) -> a -> c	6.9	9.6	0.7165271967
18	singleList	a -> [a]	7.7	4.4	1.750566893
19	head-last	[a] -> (a,a)	257.3	12.0	21.4093178
20	head-rest	[a] -> (a, [a])	1.5	3.3	0.4693251534

from the stored type, so that the stored α_3 can be discarded and there is no longer a conflict with the current α_3.

4 RESULTS

We tested our results on 20 queries, that were based on the queries from Tygar's [2] paper and λ^2 [1], and compared Petsy with Tygar, which is the tool that most resembles our own in terms of input and outputs. The results can be found in Table 1.

Analysis. Tygar did better than Petsy in (TODO) out of the 20 tests. For most of the tests, the difference between the 2 tools is quite small. There are additionally some tests that we do much better in (TODO list them) and some tests that we do much worse in (TODO list them). Test #(TODO) shows how we are able to synthesize lambdas whereas Tygar cannot.

These results are quite encouraging. For one, since Petsy takes advantage of lambdas and Tygar doesn't, Petsy is more expressive and therefore searches through more programs, yet is still competitive. We also have only implemented the most basic form of memoization, and plan to improve upon it to make Petsy even faster (see Future Work).

Quality of Results. For the most part, both Tygar and Petsy 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

[1] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

[2] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (2020), 12:1–12:28. https://doi.org/10.1145/3371080

[3] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

[4] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093