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 a `Just` and applies it to `x`, or applies the identity function to `x` when given `Nothing` (which is the same as returning `x` unchanged). As a component-based program, (2) is idiomatic and concise and doesn't rely on if-else or pattern matching. But finding it is potentially non-intuitive.

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

Tygar. Tygar [1] 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. However, its encoding into Petri nets reduces the type system to first-order, so programs with lambdas are not in the space of programs that Tygar can pull from.

SYNQUID. Synquid [3] makes use of an enumerative search approach, lambdas, and polymorphism. Its type system, however, restricts type variables (represented here with α) from unifying with arrow types. For example, this prevents `Maybe` $a \to a$ from unifying with $\alpha_1 \to \alpha_0 \to b$, even though in theory they could if `Maybe` a unified with α_1, and a unified with $\alpha_0 \to b$. This means certain programs that unify type variables with arrow types are not in the search space for Synquid.

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

Synquid also solves its synthesis task via top-down enumerative search. The task it solves is similar, the only difference being that the types are not just polymorphic – they are liquid (refinement) types. Among other reasons, Synquid's method of supporting refinements requires it to determine whether a type is a scalar without having to substitute its type variables. This is made possible by restricting type variables to not unify with arrow types. Because Petsy employs an enumerative search method instead, there is no reason to have this restriction.

Petsy is built on top of Synquid, ignoring all the refinement type logic since our synthesis task does not involve refinements. We also change the type system by letting type variables unify with arrows. This means Petsy can synthesize a wider range of programs. Defining :: to mean "has type", one concrete example is the query $(a \to b, a) \to b$, with components [`fst` $:: \forall x.\, \forall y.\, (x, y) \to x$, `snd` $:: \forall x.\, \forall y.\, (x, y) \to y$]. The solution is `fst p (snd p)`, which is only possible if $\forall x.\, \forall y.\, (x, y)$ in `fst` $:: \forall x.\, \forall y.\, (x, y) \to x$, unifies with $p :: (a \to b, a)$. Because the type variable x needs to unify with the arrow type $a \to b$, Petsy is able to consider the program `fst p (snd p)` while Synquid cannot.

3.3 Memoization (caching subproblems during enumeration)

The nature of enumerative search with iterative deepening is that many repeated subproblems are generated over the course of the search. Redoing the same subproblems wastes a lot of time. This makes memoization crucial to practical top-down enumeration, which Myth [2] is able to solve in a first-order setting.

However, memoization is mechanically difficult to deal with when facing polymorphic type variables. We must be careful when dealing with goals that overlap, for example: $\alpha_0 \to$ `Int` and $\alpha_1 \to$ `Int` and (`Int` $\to$ `Int`) $\to$ `Int`. We discovered the following methods to resolve problems that arose due to this complexity.

3.3.1 Overlapping Goals. To resolve a space explosion issue where equivalent goals $\alpha_0 \to$ `Int` and $\alpha_1 \to$ `Int` contain the exact same programs being stored in two different locations, the cache renames all free type variables in goals – the first type variable becomes β_0, the second becomes β_1, and so on. This treats both goals as $\beta_0 \to$ `Int`, making syntactically different goals semantically equivalent.

3.3.2 Preserving Unification. The previous method with β's introduces a small issue in program retrieval. Consider how the program `length` of type $[\alpha_1] \to$ `Int` solves the goal $\alpha_0 \to$ `Int`. This unifies $[\alpha_1]$ with α_0, which is important information for Synquid's algorithm. With memoization, however, the cache stores `length` for the more general goal $\beta_0 \to$ `Int`. When `length` is retrieved from the cache to solve $\alpha_0 \to$ `Int`, we lose the fact that α_0 now unifies with a list, as this information is never stored.

Instead of worrying about how to store this information, we decided to simply unify the type of `length`, $[\alpha_1] \to$ `Int`, with the goal type $\alpha_0 \to$ `Int` at the time of retrieval. This recovers the unification information. Although obtaining the type `length` is easy, as it is a single component, in general we obtain the type of the retrieved program via *type inference*.

Table 1. Tygar vs. Petsy

N	Name	Query	Tygar	Petsy
1	test	Bool -> a -> Maybe a	8.0 s	6.0 s
2	firstJust	a -> [Maybe a] -> a	18.4 s	6.5 s
3	mapEither	(a -> Either b c) -> [a] -> ([b], [c])	4.8 s	3.2 s
4	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b	8.0 s	5.2 s
5	mergeEither	Either a (Either a b) -> Either a b	3.8 s	181.5 s
6	map	(a->b)->[a]->[b]	2.1 s	3.7 s
7	repl-funcs	(a->b)->Int->[a->b]	1.1 s	1.2 s
8	mbAppFirst	b -> (a -> b) -> [a] -> b	3.2 s	9.5 s
9	resolveEither	Either a b -> (a->b) -> b	4.0 s	10.3 s
10	multiApp	(a -> b -> c) -> (a -> b) -> a -> c	6.9 s	9.6 s
11	singleList	a -> [a]	7.7 s	4.4 s
12	head-last	[a] -> (a,a)	257.3 s	12.0 s
13	firstMatch	[a] -> (a -> Bool) -> a	6.0 s	6.9 s
14	rights	[Either a b] -> Either a [b]	1.9 s	2.6 s
15	firstKey	[(a,b)] -> a	1.5 s	3.4 s
16	firstRight	[Either a b] -> Either a b	4.4 s	9.6 s
17	zipWithResult	(a->b)->[a]->[(a,b)]	268.1 s	10.8 s
18	applyNtimes	(a->a) -> a -> Int -> a	30.2 s	18.0 s
19	mbElem	Eq a => a -> [a] -> Maybe a	3.7 s	295.6 s
20	flatten	[[[a]]] -> [a]	4.7 s	1.4 s

4 RESULTS

We tested our results on 20 benchmarks that we took from Tygar's [1] paper. We used the same set of 130 components in all experiments. Because Tygar's search space was most similar to ours, we chose to compare Petsy with Tygar. The results can be found in Table 1.

Analysis. Petsy and Tygar are comparable. While there are some tests that Petsy does much better in – #12 and #17 – and some tests that Petsy does much worse in – #5 and #19 – both tools perform about equally well (within a few seconds) on the other benchmarks.

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 (because Petsy found a solution using lambdas), the programs were equivalent, so there is no significant difference in program quality between the two tools.

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] 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

[2] 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

[3] 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