

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:

$$f : \text{Maybe } (a \rightarrow a) \rightarrow x : a \rightarrow a \quad (1)$$

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:

$$\backslash f \ x \rightarrow \text{fromMaybe } (\backslash y \rightarrow y) \ f \ x \quad (2)$$

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, `2` 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 → a` and $\alpha_1 \rightarrow \alpha_0 \rightarrow b$ from unifying, even though in theory they could if `Maybe a` unified with α_1 , and a unified with $\alpha_0 \rightarrow b$. This leads to some programs being unreachable and, in the context of higher-order components, limits the search space.

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 this 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.

This means PETSy can synthesize a wider range of programs. One concrete example is for the query $(a \rightarrow b, a) \rightarrow b$, with components $[fst :: \forall x. \forall y. (x, y) \rightarrow x, snd :: \forall x. \forall y. (x, y) \rightarrow y]$. The solution is $fst \ p \ (snd \ p)$, which is only possible if $\forall x. \forall y. (x, y)$ in the component $fst :: \forall x. \forall y. (x, y) \rightarrow x$ unifies with $p :: (a \rightarrow b, a)$. Because the type variable x needs to unify with the arrow type $a \rightarrow b$, SYNQUID cannot do this while PETSy can.

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, as 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, $\alpha_0 \rightarrow Int$ and $\alpha_1 \rightarrow Int$, 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 $\alpha_0 \rightarrow Int$ and $\alpha_1 \rightarrow Int$, they will each be renamed to $\beta_0 \rightarrow Int$ and $\beta_0 \rightarrow Int$, meaning that they will hit the same entry in the map, despite having different original names.

Type unification clashes. Because we treat $\alpha_0 \rightarrow Int$ and $\alpha_1 \rightarrow Int$ as the same goal under renaming, a question naturally arises over how to retrieve the programs for this goal such that type variables α_0 and/or α_1 do not clash. We do so by storing untyped programs, using type inference to generate fresh type variables for programs retrieved from the map. This ensures no name clashes.

As an example, say there are two programs featuring different type variables that solved the goal $\alpha_0 \rightarrow Int$. They are `length` and `length`. TODO

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.

Table 1. TYGAR vs. PETSy

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

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 advantage of sub-typing in the memo keys so that the same programs aren't stored multiple times. For example, all programs in goal $\text{Int} \rightarrow \text{Int}$ should be in goal $\alpha_0 \rightarrow \text{Int}$. When we lookup $\alpha_0 \rightarrow \text{Int}$, we could first look at $\text{Int} \rightarrow \text{Int}$ 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>