Test-based Solution Filtering for Program Synthesis

ZITENG WANG*, UC San Diego, USA

1 INTRODUCTION

Consider the task of implementing a function `mapEither f xs` with the following type signature:

```
mapEither :: (a → Either b c) → [a] → ([b], [c]).        (1)
```

Many programs may match that signature, but we intend to write one that applies `f` to each element of `xs` and partitions the results by their type constructor into a pair of lists: `Left` for one list and `Right` for the other. To prevent code duplication, we want to see if there is already a library function implementing (1). To do so, we search for it on Hoogle [Mitchell 2004], a search engine for Haskell APIs, but it fails to give us any solutions because no single function in the standard library implements this signature.

Solutions from Hoogle+. Under these circumstances, a type-directed synthesis engine could help us discover a satisfying code snippet. Guo et al. have recently developed Hoogle+ [Guo et al. 2020], a search engine extending Hoogle with synthesis capabilities. In this case, Hoogle+ produces the desired solution

```
\f xs → partitionEithers (map f xs).        (2)
```

Unhelpful Solutions. However, the type signature alone cannot precisely capture programmer intent; some programs of the same type are undesirable. For example, the following programs are also synthesized for `mapEither`:

```
\f xs → curry (last []) f xs,        (3)
\f xs → partitionEithers (repeat (f (last xs))).        (4)
```

Solution (3) is never useful since executing `(last [])` always throws an exception; similarly, solution (4) diverges as `partitionEithers` has to iterate through the infinite list returned by `repeat`. Therefore, these solutions should not be in the candidate set, even though they satisfy the type signature.

Another class of unhelpful solutions are duplicates, where several syntactically distinct solutions have the same behavior. For example, solutions that apply either `head` or `last` to a singleton list would yield the same result.

Check+. In this paper, we present Check+, an extension to Hoogle+ to filter out the uninteresting solutions identified above. Moreover, Check+ can demonstrate how candidate programs behave so that users may confirm if a synthesized solution matches their intent. Check+ tests every synthesized solution with randomly generated inputs, generating feedback to improve the set of solutions Hoogle+ provides. We use QuickCheck [Claessen and Hughes 2000], a property-based testing framework for Haskell, to evaluate solutions inside the Haskell interpreter Hint [Gorin 2007]. Check+ generates properties that are testable for QuickCheck and then feed into Hint for evaluation results.

*Undergraduate Student; Advisor: Nadia Polikarpova; ACM Student Number: 4756401

Author's address: Ziteng Wang, UC San Diego, USA, ziw329@ucsd.edu.

2 ELIMINATING INVALID PROGRAMS

Invalid Solutions. A synthesized program is invalid if it throws an exception or diverges on all tested inputs. As evidenced by (3), invalid terms such as `(last [])` and `(fromJust Nothing)` are frequent because they match any polymorphic type. Unfortunately, there are too many of these terms to manually exclude and therefore we opt for an automated way to evaluate solutions. Check+ generates a set of sample inputs with fair coverage to validate synthesized programs.

Challenges in Evaluating Solutions. We want to filter out functions that throw exceptions or diverge. In removing diverging programs, one challenge will be the treatment of infinite data structures. After all, testing an infinite data structure (e.g. by comparing it with another value) may cause the testing itself to diverge. So how can we distinguish valid programs returning infinite structures (e.g. `repeat x`) from diverging programs (e.g. `length (repeat x)`)? On the other hand, filtering out functions that throw exceptions is not as easy as checking if the program fails on some input. This is because desirable components such as `head` are partial, meaning that they may error out on some arguments. Therefore, our filter for exception-throwing programs will require a nuanced testing approach for components like `head` to appear in our candidate programs.

Definition 2.1 (First Filtering Strategy). Using a random set of inputs, we evaluate a program as follows:

(1) If the program terminates, without error, for all possible inputs, then it is valid and passes the filter.
(2) If the program terminates, without error, for some input, but fails for others, then it is a partial function and also passes the filter.
(3) If the program returns an infinite data structure, then it passes the filter. We can do this by lazily producing output within a timeout.
(4) In all other cases, the program is rejected.

In this implementation, Check+ generates a QuickCheck-compatible property and validates that property within the Hint interpreter. Our property simply tests if the synthesized solution fails on all inputs. If the property failed with a counterexample, we know that the solution is at least *partial* and therefore passes the filter. Otherwise, it is eliminated. Additionally, we can test for a second property: whether a solution succeeds on all inputs. Those solutions for which this property holds not only pass the filter, but can also be tagged as more preferrable.

3 ELIMINATING DUPLICATES

After getting rid of those solutions that never succeed, Check+ tests if a new solution is a duplicate, meaning that it behaves identically to some previously generated solution. Consider the following type query to Hoogle+,

```
lookup :: Eq a => [(a,b)] → a → b,                    (5)
```

describing a function performing a linear search on a dictionary represented by a list of pairs. It is similar to `Prelude.lookup`, but is a partial function. Hoogle+ gives the desired solution,

```
\xs k → fromJust (lookup k xs),                       (6)
```

but will also generate

```
\xs k → head (maybeToList (lookup k xs)),             (7)
\xs k → last (maybeToList (lookup k xs)).             (8)
```

Id	Query	Check+
0	*appBoth* `(a → b) → (a → c) → a → (b, c)`	`(,) (arg2 arg0) (arg1 arg0)` ~~`(,) (arg2 (fromJust Nothing)) (arg1 arg0)`~~ ~~`(,) (arg2 (head [])) (arg1 arg0)`~~ ~~`(,) (arg2 (last [])) (arg1 arg0)`~~

2	*head-tail*
	`[a] → (a,a)`
		`(,) (last arg0) (last arg0)`
		`(,) (head arg0) (head arg0)`
		~~`last (zip [] arg0)`~~
		~~`head (zip [] arg0)`~~

3	*mapEither*
	`(a → Either b c) → [a] → ([b], [c])`
		`partitionEithers (map arg1 arg0)`
		`partitionEithers (repeat (arg1 (head arg0)))`
		~~`partitionEithers (repeat (arg1 (last arg0)))`~~
		~~`curry (last []) arg1 arg0`~~

4	*app3* `(a → b → c → d) → a → c → b → d`	`arg3 arg2 arg0 arg1` ~~`fromMaybe (arg3 arg2 arg0 arg1) Nothing`~~ ~~`arg3 (fromMaybe arg2 Nothing) arg0 arg1`~~ ~~`arg3 arg2 (fromMaybe arg0 Nothing) arg1`~~ ~~`arg3 arg2 arg0 (fromMaybe arg1 Nothing)`~~

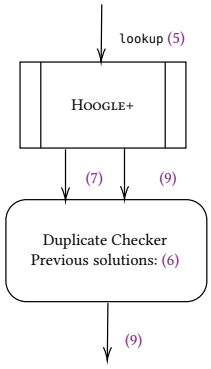

Fig. 1. (left) Synthesized solutions with Check+ crossing out invalid ones. (right) Diagram illustrating how Check+ eliminating duplicates.

While (7) and (8) are syntactically distinct, they are semantically equivalent to (6) because `maybeToList` returns either a singleton or an empty list. We want to remove solutions with identical behavior because they are less interesting to users.

Definition 3.1 (Second Filtering Strategy). As described in the diagram located at the right side of Fig. 1, we evaluate a set of programs over the same, randomly generated inputs and compare their results. Each time the synthesizer suggests a new solution, Check+ uses QuickCheck to test if it behaves identically to at least one of the previous solutions. If so, we know that the solution is a duplicate and it is eliminated. Otherwise, it passes the filter.

For the remaining set of programs, Check+ generates minimal examples of how they differ from one another. We extract these from the counter-examples generated by QuickCheck at the second filtering stage. Before showing them to the user, we shrink them for clarity.

Example 3.2 (Minimal Differentiating Example). Let us continue with the example of `lookup` (5). Check+ filtered out solutions (7) and (8), and then Hoogle+ generated another candidate,

```
\xs k → fromJust (lookup k (cycle xs)).        (9)
```

For this program, Check+ chose `k = 0, xs = [(1,0)]` as the minimal example. With the knowledge that `cycle xs` returns an infinite list repeating `xs`, we see that (9) diverges if the key is not found while (6) does not. This is because (9) terminates only if `k` exists in `xs`. Otherwise, `lookup` diverges while searching for `k` in the rest of the infinite list. Therefore, `k = 0, xs = [(1,0)]` correctly differentiates it from the other solutions, which raise an exception on the input instead.

4 EVALUATION

To directly compare the degree to which Check+ improves the quality of solutions, we present, in Fig. 1, solution sets by Hoogle+ from benchmark queries and then run Check+ to cross out invalid ones based on the strategies discussed above.

5 FUTURE WORK

In developing this project, we will explore two core directions. First, we want to use testing results for ranking solutions, as mentioned in Sec. 2. Second, we aim to communicate testing results using innovative visualizations of program behavior.

REFERENCES

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266

Daniel Gorin. 2007. Hint. http://hackage.haskell.org/package/hint.

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program Synthesis by Type-Guided Abstraction Refinement. *PACMPL* 4, POPL (Jan. 2020), 28. https://doi.org/10.1145/3371080

Neil Mitchell. 2004. Hoogle. https://www.haskell.org/hoogle/.