Yiming Fang and Gregory Schare
COMS 3997 BC
Seminar in Program Synthesis
Mark Santolucito

Review: *Digging for Fold: Synthesis-Aided API Discovery for Haskell*

Evaluation

1) A score for your overall evaluation - either accept or reject. Usually, review platforms will also ask for your confidence on a scale of Expert or Knowledgable (there are other options, but you should always be one of these).

We confidently accept the paper. This paper advances the field of program synthesis by combining previous work, namely type-guided synthesis and anti-unification, and presenting several novel algorithms for candidate filtering, ranking, and comprehension. The user study excellently frames the positive results of the paper while pointing to future work. Overall, the paper demonstrates the practical effectiveness of program synthesis in addressing the limitations of API discovery tools. While we are still novices in the field of program synthesis, we are both intermediate Haskell users and we recognize the importance of this contribution.

Summary

2) A summary of the work, accessible to those in the class. This should highlight the major contributions of the work and how it fits into the related work cited in the paper. If you are able, try to also connect the work to topics we have covered in class

Background and related work
Hoogle is a type-directed API discovery tool for Haskell where users can search for library functions by querying the type signature they need to satisfy.

Overview
The motivation for this paper is to address a limitation of Hoogle, namely its inability to provide guidance on how to build programs with compositions of functions. The paper proposes Hoogle+, an API discovery tool implemented with programming synthesis techniques that helps Haskell programmers find matching programs composed from functions in popular Haskell libraries. Hoogle+ uses a three-stage pipeline to (1) perform efficient type-and-example-directed synthesis from user specification, (2) eliminate meaningless and repetitive synthesis results using

effective heuristics, and (3) automatically generate informative feedback to demonstrate program behavior to the user. We will present each of these components in more detail in the following sections.

Type-guided synthesis with user specifications

The paper uses TyGuS as a black box for its synthesis procedure. However, the challenge in using type-guided synthesis with user specification is that users, especially beginner Haskell programmers, may not know the appropriate type signature, particularly with Haskell's complex typeclass constraints. Hoogle+ solves this problem in the following way: it takes user specification either as a type, a set of input-output pairs, or a combination of both. If the user provides no type information, Hoogle+ automatically infers likely type specifications from input/output pairs, which users are often more comfortable providing. Hoogle+ performs type inference to generate a list of type signatures that fit the specification, and then filters and ranks the types to prioritize those that are simple, general, and inhabited.

The type inference algorithm builds upon prior work in anti-unification in order to reconcile examples with different concrete types. For example, the user may provide `"abaa" -> "aba"` and `[1,1,1] -> [1]` as examples, but they have different types: `[Char] -> [Char]` and `[Int] -> [Int]`, respectively. Anti-unification finds a polymorphic type that can be instantiated into either of these concrete types. The paper extends prior work on anti-unification by adding support for Haskell type classes.

The filtering algorithm eliminates uninteresting types by checking for reachability and relevancy. A return type is unreachable if it contains type variables that do not occur in the argument types. For example, there is no non-degenerate way to write a program with type `[Int] -> [a]`. An argument type is irrelevant if it is not used by the return type (even indirectly, as is the case for `a` in `map :: (a -> b) -> [a] -> [b]`).

The ranking algorithm is heuristics-based. It returns the *k* highest ranked candidate types based on three heuristics. For each of the heuristics, we show how it would affect the candidate generalizations of the example concrete type `[Int] -> [Int]`. The heuristics are: (1) penalize abstracting away a complex type, e.g. penalize the generalization `a -> a`; (2) prioritize like generalizations, e.g., prefer `[a] -> [a]` over `[a] -> [Int]`; (3) prioritize general types over specific types, e.g. prefer `[a] -> [a]` over `[Int] -> [Int]`. Together these heuristics help prioritize types that might be more relevant to the user.

Filtering candidate programs

Since the search space of candidate programs is infinite, Hoogle+ filters out irrelevant and duplicate programs and suggests only a finite number of "interesting" candidates. Hoogle+ makes use of the SmallCheck property-based testing library to test each program for

meaningfulness and uniqueness. Meaningfulness is determined by exhaustively evaluating the program on a finite number of test inputs, called witnesses, that are smaller than a given constructor depth. This is like the bottom-up enumerative search algorithm from the first week of class, but for verification instead of synthesis. If the program terminates in a given timeout, Hoogle+ marks it meaningful and presents it to the user. Uniqueness is determined by observational equivalence with previous candidates.

One weakness of this approach is loss due to misclassification, where a good candidate program is filtered out because Hoogle+ was unable to quickly generate a small witness of the program's meaningfulness or uniqueness. However, the authors' stance is that it is better to miss a few relevant results sometimes than to overwhelm the user with many irrelevant results every time.

User comprehension aids

In order to aid the user in choosing the candidate program that best suits their needs, Hoogle+ displays each program's behavior on a small set of inputs that maximize comprehension. Inputs are generated such that they demonstrate the differences between candidate programs in three categories: (1) meaningfulness, by including success and failure examples; (2) uniqueness, by showing examples that differentiate the program from its predecessors; and (3) functionality, by picking inputs that are different from all previously-generated inputs in order to demonstrate a wider range of behavior.

In addition to these aids, the user can insert their own inputs in order to view the behavior of each program on a particular example of interest. These user-provided examples persist across multiple searches so that the user can iteratively refine their query in response to the information provided by the comprehension aids.

Benchmarking

Because type inference plays a critical role in the type-guided synthesis process, the authors include benchmarks to evaluate the quality of the inferred types. Both user provided test cases and randomly generated test cases are used, and in both settings the heuristics that this paper uses prove reasonably successful. Specifically, in most tests, the heuristics are able to rank the correct type as the top or second choice. However, the authors also notice that the quality of type inference tends to deteriorate as the number of input/output pairs decrease, resulting in a few failed test cases.

The elimination process, i.e., filtering out unhelpful programs returned by TyGAR is also evaluated using benchmarking. In particular, the authors wish to ensure that the number of false negatives are low, so that meaningful programs do not get mistakenly eliminated often. During this evaluation, the authors identify a tradeoff between misclassification and testing overhead: to

test for the uniqueness of each program decreases the rate of misclassification, but increases the runtime overhead, and vice versa.

User study

A user study with 30 participants of varying Haskell proficiency from both industry and academia shows that programmers equipped with Hoogle+ generally solve tasks faster and were able to solve more tasks compared to traditional API search techniques using Hoogle. Users with less experience in Haskell tended to rely more on input-output specifications rather than type specifications when compared with experienced Haskellers, but all users demonstrated a strong preference for providing tests in their queries. The usability-focused features of Hoogle+ were mostly helpful for interpreting results, with the aids appealing more to beginners, who used them more. Overall, the users provided positive feedback, with the notable exception of complaints about long synthesis times. The usefulness of Hoogle+ for solving the problem was spiky across the four tasks, which the authors conjecture is due to the complexity of the Haskell features involved.

Related work

Besides Hoogle, tools such as code completion in IDEs can provide similar functionality, but most of these are still limited to single-component synthesis. Hoogle+ retains an edge over most other tools due to its ability to synthesize compositions.

Other type-based synthesis solutions include proof search, but this fails to scale. Graph-based inhabitation algorithms scale better, but may not handle n-ary functions or polymorphism. TyGAR builds on this previous work on inhabitation and allows n-ary functions and polymorphism.

Other work in program synthesis usability is unfortunately limited. Hoogle+ stands out in this area because of its target audience: programmers of all experience levels working in a general-purpose language.

Filtering and ranking are popular features in program synthesis. Related work explores ranking via statistical methods and syntax-based ranking. Hoogle+, since it uses hand-written heuristics, is aligned with previous work in hand-crafted ranking. Filtering is already a necessary part of any synthesizer, but Hoogle+ builds on TyGAR's built-in filtering by also filtering the synthesized results. Hoogle+ repurposes property-based testing, which was developed for automating testing and bug-finding, for use in elimination and comprehension.

Hoogle+ differs from other work in inferring types from examples by focusing on static inference of polymorphic function signatures, where the existing literature has a different context.

We were unable to fully evaluate the differences between Hoogle+ and MagicHaskeller since the web services are both down and we could not install the program locally. However, we note one key way in which Hoogle+ improves on MagicHaskeller: MagicHaskeller checks uniqueness through random testing, while Hoogle+ uses a much more advanced property-based testing approach. There are also some aspects where they adopt similar approaches: both tools include some form of elimination of irrelevant results, e.g., MagicHaskeller hides functions that do not use their arguments; and both allow the user to easily access API documentation through the user interface.

Critical Review

3) A critical review of the work. What evidence are you using to support your overall evaluation score? Are there issues with the work/writing/evaluation? What stands out as truly impactful about this work? What excites you about this work?

Strengths

We appreciate that Hoogle+ solves a real problem motivated by the limitations of existing tools for Haskell API discovery. Moreover, the results of the paper are overwhelmingly positive, especially since this is one of the few examples we have seen where a programmer-oriented correct-by-construction synthesis tool for API discovery in a general-purpose language has demonstrated effectiveness and usability for non-experts. The space of equally-successful related work is mostly taken up by methods which are not correct-by-construction, such as machine learning based code completion.

The paper really "does it all": it takes up a reasonable problem, builds on previous work, presents several novel algorithms that can be adapted to other synthesis tools, implements the proposed tool with a friendly interface, benchmarks the implementation, and evaluates the usability of the implementation via a sensible user study. The novel algorithms that form a major part of this paper's contributions are written in abstract pseudocode such that they are not tied to a particular implementation or bogged down by particularities. Due to this broad range of contributions, the paper stands out as a model of program synthesis research.

Furthermore, the paper itself is well-written. It is structured in a sensible way that introduces information in a natural flow with increasing complexity. It is thorough in its explanations of both the existing work it builds on and the novel contributions of the paper.

This paper encapsulates many of the ideas we have explored in class, including bottom-up enumeration, type-directed synthesis, specification via input-output examples, usability, and more. The work appeals to us, the reviewers, because of its relevance to our academic interests and experience; as intermediate Haskell programmers studying type theory, the fact that the bulk

of the paper deals with type inference, filtering, and ranking is a great plus. We are impressed by the positive feedback by the user study and compelled to try out the tool ourselves, as it may help in our day-to-day programming work.

Weaknesses

The paper uses a number of heuristics in its core stages, such as ranking type signatures and eliminating unhelpful programs returned by TyGAR. Although these heuristics are based on reasonable assumptions and prove to be effective in benchmarking and user studies, it still lacks the strong theoretical guarantee that is one of the most important strengths of program synthesis research.

Moreover, the user study appears to be too simplistic to be truly convincing about the effectiveness of Hoogle+. In particular, both the number of participants (30) and the number of test cases (4) are small, and the test cases are quite basic, involving only the List and Maybe monads. Therefore, it is difficult to see how much the tool can help in terms of more complex programs involving the numerous features of Haskell's rich type class, such as typeclass, which is presented as one of the paper's central features.

The user study suffers from right-censoring; we cannot gather nuanced information from the cases where the subjects failed to complete the task in the given time limit because they were not allowed to continue working on the problem after the time was up.

Citation

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. *Digging for fold: synthesis-aided API discovery for Haskell*. Proc. ACM Program. Lang. 4, OOPSLA, Article 205 (November 2020), 27 pages.
https://doi.org/10.1145/3428273