Digging for Fold: Synthesis-Aided API Discovery for Haskell

MICHAEL B. JAMES, UC San Diego, USA
ZHENG GUO, UC San Diego, USA
ZITENG WANG, UC San Diego, USA
SHIVANI DOSHI, UC San Diego, USA
HILA PELEG, UC San Diego, USA
RANJIT JHALA, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

We present Hoogle+, a web-based API discovery tool for Haskell. A Hoogle+ user can specify a programming task using either a type, a set of input-output tests, or both. Given a specification, the tool returns a list of matching programs composed from functions in popular Haskell libraries, and annotated with automatically-generated examples of their behavior. These features of Hoogle+ are powered by three novel techniques. First, to enable efficient type-directed synthesis from tests only, we develop an algorithm that *infers likely type specifications from tests*. Second, to return high-quality programs even with ambiguous specifications, we develop a technique that automatically *eliminates meaningless and repetitive synthesis results*. Finally, we show how to extend this elimination technique to automatically *generate informative inputs* that can be used to demonstrate program behavior to the user. To evaluate the effectiveness of Hoogle+ compared with traditional API search techniques, we perform a user study with 30 participants of varying Haskell proficiency. The study shows that programmers equipped with Hoogle+ generally solve tasks faster and were able to solve 50% more tasks overall.

CCS Concepts: • **Human-centered computing** → **Interactive systems and tools**; • **Theory of computation** → **Automated reasoning**; **Type theory**.

Additional Key Words and Phrases: Program Synthesis, Type Inference, Human-Computer Interaction

ACM Reference Format:
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

1 INTRODUCTION

Consider the task of implementing a function `dedup` that eliminates adjacent duplicate elements from a list (*e.g.* `dedup [1,1,2,2,1] = [1,2,1]`). In a functional language like Haskell, this task can be accomplished without explicit recursion, simply by using functions from the standard library:

```
dedup xs = map head (group xs)
```

This solution first calls `group` on the input list to split it into clusters of adjacent equal elements (*e.g.* `group [1,1,2,2,1] = [[1,1], [2,2], [1]]`), and then `map`s over the result to extract the `head`

Authors' addresses: Michael B. James, UC San Diego, USA, m3james@ucsd.edu; Zheng Guo, UC San Diego, USA, zhg069@ucsd.edu; Ziteng Wang, UC San Diego, USA, ziw329@ucsd.edu; Shivani Doshi, UC San Diego, USA, s1doshi@ucsd.edu; Hila Peleg, UC San Diego, USA, hpeleg@eng.ucsd.edu; Ranjit Jhala, UC San Diego, USA, jhala@cs.ucsd.edu; Nadia Polikarpova, UC San Diego, USA, npolikarpova@ucsd.edu.

2475-1421/2020/11-ART205
https://doi.org/10.1145/3428273

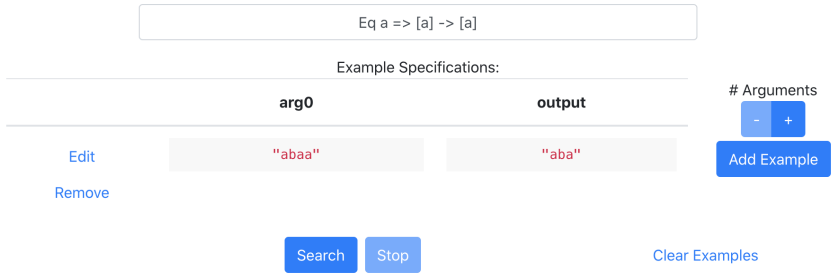


Fig. 1. In Hoogle+, the user can search for `dedup` using its type, one or more tests, or both.

of each cluster. This implementation is not only shorter than a recursive one, but also more idiomatic. But how is the programmer to *discover* this solution?

The need for such discovery is particularly acute in functional languages, whose expressive types and higher-order functions make libraries extremely versatile and compositional. As a result, discovery is especially *useful* as many computations can be expressed by gluing components from existing libraries. At the same time, discovery is especially *difficult* as library functions are very general and can be composed in myriad ways. Online help forums like StackOverflow only contain solutions for common programming tasks, and are generally less helpful outside of a handful of most popular programming languages. As an alternative, Haskell programmers often turn to the Hoogle API search engine [Mitchell 2004] to search for library functions by their type; but Hoogle only helps if there is a single library function that does the job, which is not the case for `dedup` where we must compose multiple functions into a *snippet*. Our goal is to bridge this gap and build an API discovery tool for Haskell that helps programmers find snippets like our implementation of `dedup`.

Type-Directed Component-Based Synthesis. The core technical challenge for API discovery is how to efficiently search the space of all snippets when the API library has hundreds or thousands of functions. *Component-based program synthesis* techniques [Feng et al. 2017; Guo et al. 2020; Gvero et al. 2013; Mandelin et al. 2005] tackle this challenge using a type-directed approach. In particular, our prior work on synthesis by *type-guided abstraction refinement* (TyGAR) [Guo et al. 2020] demonstrates how to efficiently perform type-directed search in the presence of polymorphism and higher-order functions, which are ubiquitous in functional languages. In this work we build upon the TyGAR search algorithm to implement an API discovery tool we dub Hoogle+.

Challenges. Although the core search algorithm behind Hoogle+ is not new, turning this algorithm into into a practical API discovery tool required overcoming three important challenges.

1. **Specification:** The first challenge is that of specification: how should the programmer communicate their intent to the synthesizer? In Haskell, *types* are a powerful and concise way to specify program behavior thanks to parametric polymorphism, which significantly restricts the space of possible implementations of a given type. Types are the preferred mode of specification for Hoogle users and moreover, TyGAR requires a type in order to perform snippet search. The flip side of expressive types is that a Haskell beginner might not immediately know the most appropriate type for the function they want to implement. Consider `dedup`: its most general type is `Eq a => [a] → [a]`; this type is polymorphic in the list element, but restricts these elements to be *equatable*, because `dedup` has to compare them for equality. When types become non-trivial, it is more natural for a user to specify their intent using *input-output tests*. Based on these observations, we design Hoogle+ to allow three different modes of intent specification: *only types*, *only tests*, or *both* (see Fig. 1). To enable

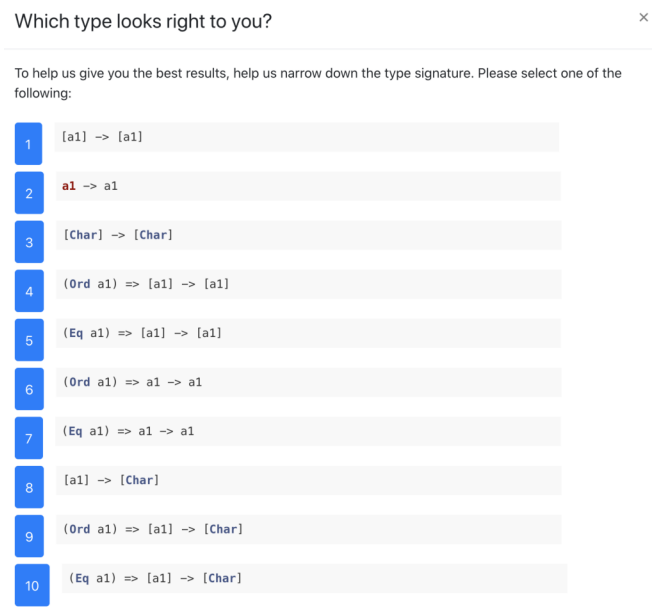


Fig. 2. Candidate types for `dedup` inferred from the test `"abaa"` ⟶ `"aba"`.

type-directed search when the user only provides tests, we develop an algorithm to *infer* types from the tests. Note that there might be many types of different levels of generality that are consistent with the tests, so Hoogle+ presents a set of *likely type specifications* to the user, as shown in Fig. 2.

2. **Elimination:** Specifications are often ambiguous, especially when the user provides the type signature alone. In this case TyGAR might return many irrelevant candidate programs. For example, searching for `dedup` by its type might generate programs like `\xs ⟶ []` (which always returns the empty list) or `\xs ⟶ head []` (which always crashes by taking the head of an empty list). Intuitively, these programs are clearly uninteresting, and we shouldn't need additional user input to eliminate them from the synthesis results. To address this challenge, we have developed an efficient heuristic for identifying uninteresting candidates using *property-based testing* [Claessen and Hughes 2000; Runciman et al. 2008].

3. **Comprehension:** Finally, once the candidate programs have been generated: how should the programmer decide which, if any, synthesis result solves their problem? To facilitate comprehension of a candidate program, Hoogle+ automatically generates several examples of its behavior as shown in Fig. 3. Unfortunately, a naive exhaustive or random generation yields many uninformative examples. We show how to address this challenge by relying, once again, on property-based testing to generate inputs with certain desirable qualities, such as examples of success and failure and examples that differentiate this candidate from the rest.

Hoogle+. We have incorporated the three techniques described above together with the TyGAR search algorithm into a web-based API discovery engine. Fig. 1 illustrates using Hoogle+ for our running example: the programmer has specified the Haskell type signature for `dedup` and one example of its behavior. Fig. 3 shows the list of candidate programs returned by Hoogle+ (with the correct solution at the top).

User study. Does synthesis-aided API discovery actually help programmers solve their tasks compared to a more traditional workflow? We evaluate this question by conducting a user study with

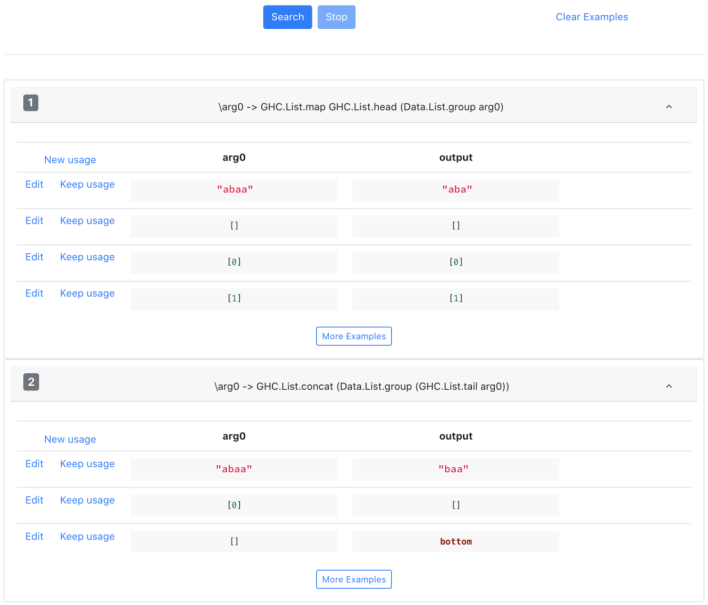


Fig. 3. Two candidate solutions for `dedup`. The behavior of each solution is illustrated with both user-provided and auto-generated examples.

30 participants of varying levels of Haskell proficiency. The participants were asked to solve various programming tasks (including `dedup`) either using Hoogle+ or using a popular code search workflow (Hoogle together with an interpreter). The study shows that Hoogle+ enables programmers to solve tasks faster and increases their success rate in finding a correct solution by more than 50%.

Contributions. In summary, this paper makes the following contributions:

(1) Hoogle+, the first practical API discovery tool for a functional language with higher-order functions and polymorphic types; the tools accepts specifications in the form of types, input-output tests, or both, and displays candidate snippets together with examples of their behavior (Sec. 2).
(2) A new algorithm that infers likely type specifications from tests (Sec. 4).
(3) A new technique for automatically eliminating uninteresting synthesis results using property-based testing (see Sec. 5.1).
(4) A new technique for automatically generating examples of program behavior using property-based testing (see Sec. 5.2).
(5) The first user study evaluating the usefulness of synthesis-aided API discovery in a functional language (Sec. 7).

2 OVERVIEW

We begin with an overview of the challenges to practical synthesis-aided API discovery and show how Hoogle+ overcomes these challenges. We postpone the description of the core synthesis engine, type-guided abstraction refinement (TyGAR), to Sec. 3.

2.1 Specification

Consider a user tasked with implementing our running example `dedup`, and assume that the user has a test in mind: on input `"abaa"` the snippet should return `"aba"`[1]. In order to make use of the TyGAR synthesis engine, however, the user also needs to provide a *type signature* for `dedup`, which the engine uses to efficiently navigate the search space.

Problem. Our user study shows that Haskell beginners often do not know the most appropriate type signature for the snippet that they are looking to implement (Sec. 7). In particular, *typeclass* constraints are particularly tricky for beginners. For example, the appropriate type specification for `dedup` is `Eq a => [a] → [a]`, where the constraint `Eq a =>` allows using equality checks on `a` to remove the duplicates, but the need for this constraint is not obvious from the task description. Consequently, the user might search using the *overly general* type `[a] → [a]`, which will prove fruitless (the `dedup` snippet does not check against this type). Alternatively, the user might search an *overly specific* type, like `[Char] → [Char]`, which will yield too many results to be useful.

Solution: Types from Tests. We address the specification problem with a novel technique that automatically *infers* likely type specifications from tests. In our running example, the user enters their intended test `"abaa" → "aba"` and leaves the type specification blank. Hoogle+ then presents the user with a list of up to 10 candidate types, as shown in Fig. 2. Notice that the correct type `Eq a => [a] → [a]` is listed in position 5. When reminded of typeclass constraints explicitly, users can often figure out which constraints they need.

Inferring likely type specifications is a difficult problem: as we show in Sec. 6.1, there can be anywhere from a dozen to a *few million* types of different levels of generality consistent with a given set of tests. To pick a few likely candidates, our inference algorithm incorporates two new mechanisms: (1) a *filtering* mechanism eliminates candidate types that cannot be inhabited by a meaningful program (*e.g.* the type `[Char] → [a]` is eliminated, because any program of this type always returns the empty list) (2) a *ranking* mechanism that prioritizes simple and general types. We describe our inference algorithm in detail in Sec. 4 and evaluate it empirically in Sec. 6.1.

2.2 Elimination

Now consider a scenario where a user is searching for `dedup` by only its type, `Eq a => [a] → [a]`.

Problem: Meaningless and Duplicate Results. Type-only specifications are often highly ambiguous: although polymorphic type signatures help narrow down the search space, there might still be too many programs that check against a given type. Fortunately, many of these programs are clearly uninteresting, and can be *eliminated* without requiring additional input from the user.

We have identified two main sources of uninteresting synthesis results. *Meaningless* programs are those that crash or diverge on every input. For example, any program that contains the subexpression `head []` is meaningless as it *always* crashes regardless of the input. The second source of uninteresting synthesis results are *duplicates*, *i.e.* semantically equivalent programs. For example, one candidate solution for `dedup` may be `\xs → init (head (group xs))` and the following one could be `\xs → tail (head (group xs))`. The two candidates syntactically differ in that they take the prefix (`init`) and suffix (`tail`) of the result of `head (group xs)`. However, they are semantically equivalent as the input to `init` and `tail` is always a non-empty list of identical values, *e.g.* `init ['a', 'a']` and `tail ['a', 'a']` are both equal to `['a']`. Our goal is to eliminate meaningless and duplicate programs from the output of Hoogle+ automatically.

Solution: Property-Based Testing. In principle, the problem of determining whether a program is meaningful or whether two programs are equivalent is undecidable. In practice, however, it turns

[1] In Haskell, a string is just a list of characters, so `dedup` can operate on strings.

out to be sufficient to test these properties on a finite set of inputs. To do so efficiently, Hoogle+ relies on the the SmallCheck property-based testing library [Runciman et al. 2008].

Specifically, for each new candidate program p generated by the synthesis back-end, Hoogle+ invokes SmallCheck to test whether there exists some input where p produces an output within a given timeout. If this check succeeds, then for each previously displayed candidate p', Hoogle+ asks SmallCheck to find a *distinguishing input* [Jha et al. 2010] for p and p', *i.e.* an input where they produce different outputs. For example, assume that Hoogle+ has displayed several results for `dedup`, including the program p' = `\xs → init (head (group xs))`, and the newly generated candidate p is `\xs → tail (head (group xs))`. This candidate passes the meaningfulness check (SmallCheck finds the input `[0]` where p returns `[]`), but fails the uniqueness check: after exhaustively searching all lists up to a given length and range of values, SmallCheck is unable to find an input where the output of p differs from the output of p'. Based on the failed uniqueness check, Hoogle+ eliminates p from the list of results presented to the user.

Haskell's laziness presents a subtle challenge for test-based elimination: in Haskell, it is common practice to write functions that produce infinite data structures, and such functions should be considered meaningful. At the same time, trying to print the output of such a function or compare two infinite outputs would lead to non-termination. Hoogle+ builds upon the `ChasingBottoms` library [Danielsson and Jansson 2004] to ensure proper handling of infinite values.

2.3 Comprehension

Hoogle+ displays a sequence of meaningful and unique candidates to the user, but how is the user to know *which* result implements their requirements? While some experienced programmers might be able to recall the behavior of the components well enough to mentally reconstruct the semantics of their composition, most users require further assistance to understand how each candidate behaves. One way forward is to show the user input-output *examples* for each candidate. However, there are two questions that must be addressed to facilitate example-based comprehension.

Problem: Comprehension Conflicts. First, *what* kind of examples is the user looking for? There is no "best" example for a candidate program as there are a range of different comprehension goals that a user might have for each candidate. They may try to *differentiate* that candidate from other similar snippets or they might be trying to understand the *functionality* of the candidate itself.

Solution: Multiple-Objective Witnesses. Hoogle+ supports multiple comprehension objectives by generating input-output examples that serve to witness different properties of the candidate, namely: (1) Meaningfulness (2) Uniqueness (3) Functionality. In Fig. 3, the first candidate's examples demonstrate these properties in order. The first example, with input-output (`"aab"`,`"ab"`), repeats the test specification. The next example, (`[]`,`[]`) witnesses that this candidate is meaningful. The subsequent example (`[0]`, `[0]`) serves a differentiating objective: the same input produces different output on candidate program #2. The last example, (`[1]`, `[1]`), demonstrates the functionality of the candidate program, more such examples are available on demand from the "More Examples" button. Note that in candidate program #2, the input `[]` demonstrates that the candidate is a partial function.

Problem: Minimality vs Interactivity. Second, *when* should examples be shown to the user? Hoogle+ could wait until we have all candidates and *then* generate examples. On the plus side, waiting would let us find fewer inputs (or even just one) to differentiate each candidate. Unfortunately, the resulting lack of interactivity could drive users away from the tool altogether.

Solution: Laziness. Instead, we stream input-output examples with every candidate, providing more examples to already-displayed candidates, which may be hidden from view until the user clicks "More Examples" to avoid cluttering the UI. In the case of `dedup`, a user might see a *usage table* as seen in Fig. 3. This table shows inputs along with their output for that candidate program. A user can edit

the input for this usage and see the new corresponding output with the "edit" button on the left. A user can input their own usage with the "New Usage" button on the top left. Finally, a user can ask for more examples explicitly from the system with the "More Examples" button.

3 BACKGROUND

Hoogle+ builds upon prior work from two sources. First, the core program synthesis algorithm comes from our own prior work on *type-guided abstraction refinement* (TyGAR) [Guo et al. 2020]. That work developed a novel search technique but we did not focus on end-to-end usability of the synthesizer. Second, we filter candidate programs with the help of exhaustive testing framework SmallCheck [Runciman et al. 2008].

3.1 Type-Guided Abstraction Refinement

In our prior work [Guo et al. 2020] we developed TyGAR, a component-based synthesis algorithm that takes as input a Haskell type and a set of library functions, and returns a list of programs of the given type, composed from the library functions. Like prior work in component-based synthesis [Feng et al. 2017; Gvero et al. 2013; Mandelin et al. 2005], TyGAR reduces the synthesis task to graph search; the challenge, however, is that in Haskell polymorphic components can infinitely explode the graph to search through. The key insight to overcome that explosion is to build a graph over *abstract types* which represent a potentially unbounded set of concrete types. We showed how to use graph reachability to search for candidate programs over those abstract types, and introduced a new algorithm that uses *proofs of untypeability* of ill-typed candidates to iteratively refine the abstraction until a well-typed result is found. TyGAR uses a *relevant* type system to ensure that every argument is used at least once in a candidate program.

Although TyGAR was able to produce a stream of well-typed candidates, our own experience during its empirical evaluation identified several shortcomings that had to be fixed in order to turn it into a practical API discovery tool. Firstly, for some type queries it returned too many uninteresting (meaningless or repetitive) programs. Secondly, it required the user to describe every programming task using its most general type, which can be challenging for beginners. Finally, it was often difficult to analyze synthesis results simply by looking at the generated code. We address these three shortcomings in present work.

3.2 SmallCheck

SmallCheck is a *property-based testing* framework for Haskell [Runciman et al. 2008]. Property-based testing takes as input a *property*, *i.e.* a Boolean Haskell function with one or more arguments, and executes this function on a set of input values in an attempt to find a *counterexample*, *i.e.* an input where the property does not hold. While the original property-based testing framework QuickCheck [Claessen and Hughes 2000] uses random input generation, its successor SmallCheck generates inputs exhaustively up to a user-provided *constructor depth*. As a result, SmallCheck always find the smallest counter-example to a property.

4 TYPE INFERENCE FROM TESTS

In this section, we detail our algorithm for inferring *likely type specifications* from tests. In a simply-typed language this is a straightforward task, since any function the user might want to synthesize has a unique, concrete type, which must coincide with the type of the test: for example, if the test is `"abaa"` ⟶ `"aba"`, the intended type specification must be `String` ⟶ `String`. In a language like Haskell, however, intended type specifications are often *polymorphic*, which poses two main challenges for type inference:

(1) ***Reconciling multiple tests.*** Consider user input with two tests: `"abaa"` ⟶ `"aba"` and `[1,1,1]` ⟶ `[1]`, whose concrete types are `[Char]` ⟶ `[Char]` and `[Int]` ⟶ `[Int]`, respectively. To reconcile these tests we must find a polymorphic type that can be instantiated into either of the two concrete types: for example, $[\alpha] \to [\alpha]$. To tackle this challenge, we build upon prior work on *anti-unification* [Plotkin 1970; Reynolds 1969].
(2) ***Generalizing from tests.*** Now consider user input with a single test `[1,1,1]` ⟶ `[1]`. The concrete type of this test is `[Int]` ⟶ `[Int]`, but this behavior can also be produced by a function with a *more general* type, such as `[Int]` ⟶ $[\alpha]$, $[\alpha]$ ⟶ `[Int]`, $[\alpha] \to [\alpha]$, or $\alpha \to \beta$. It is not obvious which one of these types would make the best specification: more general types reduce the search space and hence yield better synthesis results, but generalize too much and you will miss the intended solution. To tackle this challenge we propose a ranking heuristic to identify which generalized types are more likely to match user intent.

In Sec. 4.1–Sec. 4.5 we formalize our base algorithm for a simplified setting, where all tests have an unambiguous concrete type and the type system does not have type type classes. The base algorithm is extended to deal with ambiguous tests in Sec. 4.6 and type classes in Sec. 4.7.

4.1 Preliminaries

We formalize our base type inference algorithm for a core language defined in Fig. 4.

Types. The language is equipped with a standard prenex-polymorphic type system: types T are either type variables, type constructor applications $C\ \overline{T}$[2], or function types. Type variables are denoted with lower-case Greek letters α,β,.... For lists, we use the familiar notation $[T]$ as syntactic sugar for `List` T. All type variables are implicitly universally quantified at the top level. A type T is *concrete* if is contains no type variables.

Type ordering. A *substitution* $\sigma = [\alpha_1 \mapsto T_1, ..., \alpha_n \mapsto T_n]$ is a mapping from type variables to types that maps each α_i to T_i and is the identity mapping elsewhere. We write σT to denote the application of σ to type T, which is defined in a standard way. We say that type T is *more general* than type T' (or alternatively, that T' is *more specific* than T) written $T' \sqsubseteq T$, iff there exists σ such that $T' = \sigma T$. For example, `[Int]` $\sqsubseteq [\alpha] \sqsubseteq \beta$. The relation $\sqsubseteq$ is a partial order on types, and induces an equivalence relation $T_1 \equiv T_2 \triangleq T_1 \sqsubseteq T_2 \wedge T_2 \sqsubseteq T_1$ (equivalence up to variable renaming).

We say type T' is a *common generalization* of a set of types $\overline{T_i}$ if $\forall i. T_i \sqsubseteq T'$. The *least common generalization* (or *join*) of $\overline{T_i}$ always exists and is unique up to $\equiv$, so, by slight abuse of notation, we write it as a function $\sqcup(\overline{T_i})$. For example, `[Char]` and `[Int]` have two common generalizations, $[\alpha]$ and β, and `[Char]` $\sqcup$ `[Int]` $= [\alpha]$, the more specific of the two.

Type checking. We omit the exact syntax of terms e, apart from the fact that they include values v. Tests t are built from argument values and a result value. We also omit the definition of typing environments Γ, which hold the types of data constructors and binders for $\lambda-$ terms, and term typing, since they are entirely standard; instead we assume access to a *type checking oracle* $\Gamma \vdash e :: T$, which decides whether term e checks against type T in Γ and a *type inference oracle* $\Gamma \vdash e \Longrightarrow T$, which computes the most general type T such that $\Gamma \vdash e :: T$ holds. Our implementation uses GHC to implement both oracles.

We extend type inference to tests; in particular, for a test with arguments we infer a function type: $\Gamma \vdash v \to t \Longrightarrow T_1 \to T_2$ where $\Gamma \vdash v \Longrightarrow T_1$ and $\Gamma \vdash t \Longrightarrow T_2$. In this section we assume that all inferred test types are concrete (we relax this restriction in Sec. 4.6). We say that a test t *witnesses* a type T in Γ ($\Gamma \vdash t \in T$), if $\Gamma \vdash t \Longrightarrow T'$ and $T' \sqsubseteq T$. The intuition is that t demonstrates a possible behavior of a function of type T, if the test's type is more specific than T. For example, the test $\Gamma \vdash$ `[1,1,1]` ⟶ `[1]` $\Longrightarrow$ `[Int]` ⟶ `[Int]`, witnesses the type $[\alpha] \to [\alpha]$.

[2]Throughout this section, we write $\overline{X}$ to denote a sequence of syntactic elements X.

T	$::= \alpha \mid C\ \overline{T} \mid T \to T$	*Types*
σ	$::= [\overline{\alpha \mapsto T}]$	*Substitutions*
e	$::= v \mid \ldots$	*Terms*
t	$::= v \mid v \to t$	*Tests*
	$\Gamma \vdash e :: T$	*Type checking*
	$\Gamma \vdash e \Longrightarrow T$	*Type inference*
	$\Gamma \vdash t \in T$	*Type witnessing*

Fig. 4. Core language.

Input: Environment Γ, test suite $\overline{t_i}$
Output: Types $\overline{T_k}$ such that $\forall i,k.\Gamma \vdash t_i \in T_k$ and $\overline{T_k}$ are likely specification types

1: TestToType(Γ,$\overline{t_i}$)
2: $\quad\Gamma \vdash t_i \Longrightarrow T_i$
3: $\quad T_\sqcup :=$ AntiUnifyAll($\overline{T_i}$)
4: $\quad G := \{T \mid T_\sqcup \sqsubseteq T \wedge$ Inhabited(T)$\}$
5: $\quad$ **return** TopK(G)

Fig. 5. Type inference algorithm.

4.2 From Tests to Types

Fig. 5 presents an overview of our TestToType inference algorithm. The algorithm takes as input an environment Γ (the component library) and a test suite $\overline{t}$, and returns a sequence of likely type specifications $\overline{T}$. Which properties need to hold of $\overline{T}$? Assume that the user's intended program is e^* and its most general type is T^* ($\Gamma \vdash e^* \Longrightarrow T^*$). Then T^* is the best type specification for synthesizing e^*: although any $T \sqsubseteq T^*$ might yield the desired program since $\Gamma \vdash e^* :: T$ necessarily holds, there might be *many more* programs e such that $\Gamma \vdash e :: T$ compared to $\Gamma \vdash e :: T^*$, hence using the more specific type as the specification is likely to yield many irrelevant results and slow down the synthesis. Of course, we do not have T^* (let alone e^*) at our disposal, so informally, the goal of TestToType is to produce a sequence $\overline{T}$ such that T^* is likely is occur early in this sequence.

Towards this goal TestToType proceeds in three steps. First, it uses the inference oracle to obtain the concrete types $\overline{T_i}$ of the tests. Next, it uses the function AntiUnifyAll to compute $T_\sqcup$, the least common generalization of $\overline{T_i}$. Then, it computes G, the set of all generalizations of $T_\sqcup$ that maybe be inhabited by relevant programs, as determined by the function Inhabited. Note that each $T \in G$ is witnessed by every test t_i in the input test suite: this is because $T_i \sqsubseteq T_\sqcup$ by the definition of least common generalization, and $T_\sqcup \sqsubseteq T$. Finally, the algorithm ranks the remaining types based on a heuristic TopK. The remaining part of this section will introduce each step in detail.

4.3 Anti-Unification

Fig. 6 details the function AntiUnifyAll that computes the least common generalization of a sequence of types, using *anti-unification* [Plotkin 1970; Reynolds 1969]. This top-level function relies on a pairwise anti-unification procedure AntiUnify, which does the actual work. At a high level, AntiUnify compares the structure of two types, abstracting different substructures into fresh type variables. This function takes as input two types and returns their join, and additionally threads through an *anti-substitution* $\theta = [\overline{(T,T) \mapsto \alpha}]$—a map from pairs of types to type variables—which keeps track of the substructures that have already been abstracted.

Now, let us look at the AntiUnify algorithm closely. Lines 18- 23 handle the interesting case when the top-level structure of T_1 and T_2 is different. In particular, lines 20- 23 abstract the two dissimilar types into a freshly created type variable α and add a new entry into the anti-substitution, which maps the pair (T_1,T_2) to α. To find the least common generalization, AntiUnify does not always create a fresh variable when two types are different. If this pair of types is found in θ (lines 18- 19), then it has already been abstracted into some type variable α, so we simply reuse this variable. Other cases of AntiUnify recursively descend into type substructures, threading the anti-substitution through. For example, when anti-unifying $[\texttt{Int}] \to [\texttt{Int}]$ with $[\texttt{Char}] \to [\texttt{Char}]$, first the two argument types are anti-unified into the type $[\alpha]$ with a fresh variable, and the mapping $[(\texttt{Int},\texttt{Char}) \mapsto \alpha]$ is

Input: Sequence of concrete types $\overline{T_i}$
Output: $T_\sqcup = \sqcup(\overline{T_i})$
1: AntiUnifyAll(T)
2: **return** T
3: AntiUnifyAll($T_1;\overline{T_i}$)
4: T := AntiUnifyAll($\overline{T_i}$)
5: $T_\sqcup,\_$:= AntiUnify($T_1,T,[]$)
6: **return** $T_\sqcup$

Input: Types T_1,T_2, anti-substitution θ
Output: Type $T = T_1 \sqcup T_2$, anti-substitution θ
7: AntiUnify($T_1 \to T_1', T_2 \to T_2', \theta$)
8: T,θ := AntiUnify(T_1,T_2,θ)
9: T',θ := AntiUnify(T_1',T_2',θ)
10: **return** $T \to T',\theta$

11: AntiUnify($C\ \overline{T_i}, C\ \overline{T_i'}, \theta$)
12: **for** T_i,T_i' **do**
13: T_i,θ := AntiUnify(T_i,T_i',θ)
14: **return** $C\ \overline{T_i},\theta$

15: AntiUnify(T_1,T_2,θ)
16: **if** $T_1 = T_2$ **then**
17: **return** T_1,θ
18: **else if** $\theta[(T_1,T_2)] = \alpha$ **then**
19: **return** α,θ
20: **else**
21: α := fresh type variable
22: $\theta := \theta \cup [(T_1,T_2) \mapsto \alpha]$
23: **return** α,θ

Fig. 6. Anti-unification algorithm.

recorded in θ; this mapping is reused when anti-unifying the return types, in order to obtain the least common generalization $[\alpha] \to [\alpha]$ (rather than the more general $[\alpha] \to [\beta]$).

4.4 Type Filtering

Although the least common generalization $T_\sqcup$ computed by anti-unification reconciles the types of all tests, we also want to include more general types into the final type inference result. The challenge is that the set of all generalizations of $T_\sqcup$, $\{T \mid T_\sqcup \sqsubseteq T\}$, can contain thousands of types (see Sec. 6), and we need to pick a few that are most likely to represent the user intent. Luckily, many of these types are obviously uninteresting in the sense that they can only be inhabited by meaningless programs (*i.e.* terms that ignore their arguments, or crash / diverge on all arguments). For example, the type `Int` $\to \alpha$ is uninteresting because there is no way to construct a value of arbitrary type α, while the type $\alpha \to \beta \to \beta$ is uninteresting because there is no way to use the first argument.

To filter out uninteresting types, we define a simple analysis that computes an over-approximation of the set of inhabited types: *i.e.* if the analysis says "no", then the type can only be inhibited by degenerate terms; if the analysis says "yes", the type might still be uninhabited depending on the component library. The function Inhabited in Fig. 7 implements this analysis. This function deems a type inhabited if its return type is *reachable* and each of its argument types is *relevant*.

Return type reachability. A return type is unreachable if it contains type variables that do not occur in the argument types (see function Reachable in Fig. 7). Examples include `Int` $\to \alpha$ and [`Int`] $\to [\alpha]$. Although the latter is inhabited in the strict sense, note that all programs of this type must return the empty list regardless of the input; we consider such programs degenerate.

Argument type relevancy. An argument type is *irrelevant* if it cannot be used to compute a value of the return type (see function Relevant in Fig. 7). There are two interesting cases: type variables and functions. A type variable can be used in two different ways: (1) if it directly occurs in the return type or (2) if it can be consumed by a higher-order argument, which is itself relevant. For example, the sole argument in $\alpha \to \alpha$ can be used directly, while the second argument in $(\alpha \to \beta) \to [\alpha] \to [\beta]$ can be consumed by its first argument, to eventually produce the return type. In turn, an argument of a function type $T \to T'$ is relevant if T is reachable from the rest of the arguments and T' is relevant. For example, the first argument of $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is relevant because α is reachable (from $[\alpha]$) and β is relevant (it directly occurs in $[\beta]$).

Input: Type T
Output: May T be inhabited?

```
1: Inhabited(T)
2:     (A,R) := ArgsRet(T)
3:     reach := Reachable(A,R)
4:     rel := ⋀_{T∈A} Relevant(A,R,T)
5:     return reach ∧ rel

1: ArgsRet(T → T′)
2:     (A,R) := ArgsRet(T′)
3:     return ({T}∪A,R)
4: ArgsRet(T)
5:     return ({},T)

1: FunTypes(T → T′)
2:     return T → T′
3: FunTypes(C T̄)
4:     return ⋃FunTypes(T)
5: FunTypes(α)
6:     return ∅
```

Input: Argument types A, return type R
Output: Whether R can be computed from A

```
1: Reachable(A,R)
2:     return ⋃TVars(A) ⊇ TVars(R)
```

Input: Argument types A, return type R, type T
Output: Whether T can be used to compute R

```
1: Relevant(A,R,α)
2:     if α ∈ TVars(R) then
3:         return true
4:     for T_a ∈ A, T ∈ FunTypes(T_a) do
5:         (A′,R′) := ArgsRet(T)
6:         if Reachable(A′,α) ∧
7:             Relevant(A\T_a,R,T) then
8:             return true
9:     return false
10: Relevant(A,R,T → T′)
11:     return Reachable(A,T) ∧
12:     Relevant(A∪{T′},R,T′)
13: Relevant(A,R,T)
14:     return true
```

Fig. 7. Type filtering algorithm.

4.5 Type Ranking

As a final step, the function TopK returns the k highest ranked candidate types (in our implementation $k = 10$). Our ranking approximates the likelihood that a candidate type is the user's intended type, conditioned on the examples provided. At a high level our strategy approximating that likelihood first picks the "simplest" types given the tests, then picks the most general types. Our ranking assumes the user's tests were just informative enough, that any type structures or similarities were intentional. The function is based on lexicographic ordering of three simple heuristics.

Our *first heuristic* penalizes generalizations that abstract over a complex type: a function or a non-nullary constructor application. For example, consider possible generalizations of $[\texttt{Int}] \to [\texttt{Int}]$. This heuristic penalizes abstracting this type into $\alpha \to \alpha$ or α, because these generalizations abstract over a list constructor and a function, respectively. The intuition is that a user is unlikely to supply a complex value if it is not required to illustrate the behavior: *e.g.* it is more natural to illustrate the identity function with the test $1 \to 1$ rather than $[1,1] \to [1,1]$. As an optimization, our implementation does not generate this kind of generalizations in the first place, since in practice they never make it into top k. We make an exception for the type $[\texttt{Char}]$ and do not penalize abstracting it into α, because of the special string literal syntax, which makes values of this type appear simple.

Our *second heuristic* is to prioritize types that generalize same substructures into the same variable. Going back to the $[\texttt{Int}] \to [\texttt{Int}]$ example, the generalization $[\alpha] \to [\alpha]$ has higher rank than $[\alpha] \to [\texttt{Int}]$ because the former abstracts both occurrences of `Int` into α. We assume a-priori that simpler types, those with fewer distinct atomic types, are more likely than complex types, with more atomic types, for reusable code snippets Hoogle+ is capable of producing. To implement this heuristic, we build an inverse substitution between the anti-unification result $T_\sqcup$ and the generalized type T, and penalize T proportionally to the size of this substitution. In our example, the inverse substitution for $[\alpha] \to [\alpha]$ is $[\texttt{Int} \mapsto \alpha]$, whereas for $[\alpha] \to [\texttt{Int}]$ it is $[\texttt{Int} \mapsto \{\alpha, \texttt{Int}\}]$, so the former

Input: Types T_1, T_2, anti-substitution θ, dis-unification constraints Ω
Output: Set of types $\overline{T_i}$ such that $T_i = T_1 \sqcup T_2$, and their respective anti-substitution θ, wildcard substitution $?\sigma$, and dis-unification constraints Ω

1: AntiUnify(T,T,θ,Ω)
2: **return** $\{T,\theta,[],\Omega\}$

3: AntiUnify($T_1 \to T_1'$,$T_2 \to T_2'$,θ,Ω)
4: $T,\theta,?\sigma,\Omega \leftarrow$ AntiUnify(T_1,T_2,θ,Ω)
5: $T',\theta,?\sigma,\Omega \leftarrow$ AntiUnify($?\sigma T_1'$,$?\sigma T_2'$,θ,Ω)
6: **return** $\{T \to T',\theta,?\sigma,\Omega\}$

7: AntiUnify($C\ \overline{T_i}$,$C\ \overline{T_i'}$,θ,Ω)
8: **for** T_i,T_i' **do**
9: $T_i,\theta,?\sigma,\Omega \leftarrow$ AntiUnify($?\sigma T_i$,$?\sigma T_i'$,θ,Ω)
10: **return** $\{C\ \overline{T_i},\theta,?\sigma,\Omega\}$

11: AntiUnify($?\alpha$,T_2,θ,Ω)
12: $R :=$ Abstract($?\alpha$,T_2,θ,Ω)
13: $?\sigma := [?\alpha \mapsto T_2]$
14: **return** $R \cup \{T_2,?\sigma\theta,?\sigma,?\sigma\Omega \mid \mathsf{sat}(?\sigma\Omega)\}$

15: AntiUnify(T_1,$?\alpha$,θ,Ω)
16: ... (symmetrical)

17: AntiUnify(T_1,T_2,θ,Ω)
18: **return** Abstract(T_1,T_2,θ,Ω)

Input: Types T_1,T_2, anti-substitution θ, dis-unification constraints Ω
Output: Type variables $\overline{\alpha}$ and corresponding anti-sub θ, sub $?\sigma$, new constraints Ω

19: Abstract(T_1,T_2,θ,Ω)
20: $\alpha :=$ fresh type variable
21: $\theta' := \theta \cup [(T_1,T_2) \mapsto \alpha]$
22: $\Omega' := \Omega \cup \{(T_1,T_2) \nsim (T_1',T_2')] \mid [(T_1',T_2') \mapsto \alpha] \in \theta\}$
23: $R := \{\alpha,\theta',[],\Omega' \mid \mathsf{sat}(\Omega')\}$
24: **for** $[(T_1',T_2') \mapsto \alpha] \in \theta$
25: s.t. $\exists ?\sigma = \mathsf{mgu}((T_1',T_2'),(T_1,T_2))$
26: and $\mathsf{sat}(?\sigma\Omega)$ **do**
27: $R := R \cup \{\alpha,?\sigma\theta,?\sigma,?\sigma\Omega\}$
28: **return** R

Fig. 8. Anti-unification algorithm with wildcard type variables.

is ranked higher (note that we keep the identity mapping `Int` $\mapsto$ `Int` in the substitution, unless all occurrences of `Int` were replaced).

Our *third heuristic* is to prioritize general types over specific types. In our example, $[\alpha] \to [\alpha]$ has higher rank than $[\texttt{Int}] \to [\texttt{Int}]$ because their inverse substitutions have the same size one, but the former is more general. This heuristic easily over-generalizes: in the absence of the second heuristic, it prefers $[\alpha] \to [\beta]$ on our example. For this reason we give it the least priority.

4.6 Support for Ambiguously-Typed Tests

Our formalization so far assumed that each test t_i has a unique concrete type T_i. Unfortunately, this is not always the case: Haskell values can have polymorphic types, and using such values inside tests presents a subtle issue. The simplest example of a polymorphic value is the empty list; so, what is the type of the test $[] \to 0$? The user could have intended $[\alpha] \to$ `Int` (*e.g.* list length), $[\texttt{Int}] \to$ `Int` (*e.g.* sum of the elements), or even $[\texttt{Char}] \to$ `Int` (*e.g.* number of spaces). Note that we cannot assume that the $T_\sqcup$ type for this (singleton) test suite is $[\alpha] \to$ `Int` with α interpreted as universally quantified, because this would preclude the inference of the other two plausible type specifications. Polymorphic values are not a corner case that can simply be ignored: values like [] and `Nothing` are common enough, but things get even worse with higher-order tests, because many functions are naturally polymorphic. For example, consider the following test for the function `applyNTimes` from our user study, which applies a function to some initial value n times (see Sec. 7 for details): `(\x → x ++ x) → "s" → 2 → "ssss"`. Here the first argument has a polymorphic type $[\alpha] \to [\alpha]$, and, perhaps counter-intuitively, the test does not actually constrain α to be `Char`.

In order to support ambiguously-typed tests, we extend the syntax of types with a separate kind of type variables that we refer to as *wildcards*: $T ::= \ldots \mid ?\alpha$. The wildcards are introduced by the inference

oracle for tests $\Gamma \vdash t \Longrightarrow T$, when tests contain polymorphic values. For example, we infer the type $[?\alpha] \to \texttt{Int}$ for $[] \to 0$. Unlike regular type variables α, which are implicitly universally quantified, a wildcard stands for a concrete type a user had in mind, which is unknown to the synthesizer.

To accommodate for wildcards during type specification inference, we need to modify to the function AntiUnify. The join of two types is now not a single type but a set of types, for each possible instantiation of the wildcard. For example, consider two tests for the function `applyNTimes`: `(\x → x ++ x) → "s" → 2 → "ssss"` and `(\x → 0:x) → [1] → 3 → [0,0,0,1]` whose types are, respectively, $([?\alpha] \to [?\alpha]) \to [\texttt{Char}] \to \texttt{Int} \to [\texttt{Char}]$ and $([\texttt{Int}] \to [\texttt{Int}]) \to [\texttt{Int}] \to \texttt{Int} \to [\texttt{Int}]$. The join of these two types is a pair of types $([\beta] \to [\beta]) \to [\beta] \to \texttt{Int} \to [\beta]$ and $([\texttt{Int}] \to [\texttt{Int}]) \to [\beta] \to \texttt{Int} \to [\beta]$. The first result comes from instantiating $?\alpha \mapsto \texttt{Char}$ and the second one from $?\alpha \mapsto \texttt{Int}$; importantly, any other instantiation would lead to a type that is more general than either of the two. After computing the join, the rest of the inference algorithm proceeds by taking the union of all generalizations of each member of the join, and performs the filtering and ranking as before.

Anti-Unification with Wildcards. Our algorithm for computing the join efficiently (without enumerating infinitely many potential wildcard instantiations) is shown in Fig. 8. There are several differences between this algorithm and AntiUnify from Fig. 6. First, the algorithm returns a *set* of anti-unification results, and uses Haskell-like monadic notation to compute all combinations of results in lines 3–6 and 7–10. Second, each anti-unification result also includes a wildcard substitution $?\sigma$, which maps wildcards to types, and a set of *dis-unification constraints* Ω of the form $T_1 \not\sim T_2$. These components record the decisions made about wildcard instantiations in the current branch of the search.

The most interesting case of anti-unification is captured by the helper procedure Abstract, which abstracts two types with dissimilar top-level structure into an anti-unification variable. Because the input types T_1 and T_2 might contain wildcards, Abstract cannot decide a-priori whether to create a fresh anti-unification variable or to reuse one of the variables already in θ. Instead it tries *all of these option* in turn and discard those that conflict with the accumulated dis-unification constraints Ω. Lines 20-23 create a fresh type variable, and add to Ω the constraints that the input types T_1,T_2 must not unify with any existing key in θ; if the resulting constraints Ω are unsatisfiable (*i.e.* contain a dis-unification of equal types), then this option is discarded. Lines 24–27 instead attempt to reuse an existing mapping $[(T_1',T_2') \mapsto \alpha]$ from θ; the mapping is only considered if (T_1,T_2) unifies with the key (T_1',T_2') (mgu stands for "most general unifier"), and the result of this unification is consistent with the current Ω. Note that in the absence of wildcards, Abstract reduces exactly to the case of dissimilar types in Fig. 6: in this case, the pair (T_1,T_2) either occurs as a key in θ exactly or it does not; hence only one set of dis-unification constraints computed in lines 22 and 26 can be satisfiable, and Abstract will always return exactly one result.

4.7 Support for Type Classes

Type classes are a popular feature of the Haskell type system [Wadler and Blott 1989], and we support them by making another modification to AntiUnify. When abstracting types T_1 and T_2 into a fresh type variable, we compute the set of type classes these two types have in common and attach a corresponding type class constraint to the resulting variable. For instance, consider the anti-unification of $[\texttt{Int}] \to [\texttt{Int}]$ and $[\texttt{Bool}] \to [\texttt{Bool}]$. When the first pair (`Int`,`Bool`) is anti-unified, we check that both `Int` and `Bool` are instances of the type classes `Eq` and `Ord`. Hence, we abstract them into constrained type variable $(\texttt{Eq}\ \alpha, \texttt{Ord}\ \alpha) \Rightarrow \alpha$; collecting constraints on all variables, we compute the anti-unifier $(\texttt{Eq}\ \alpha, \texttt{Ord}\ \alpha) \Rightarrow [\alpha] \to [\alpha]$ for the top-level types.

5 TESTS FOR ELIMINATION AND COMPREHENSION

Next, we describe how we use the SmallCheck's property-based testing [Runciman et al. 2008] to eliminate undesirable candidates and produce examples that aid different comprehension goals.

5.1 Elimination

The *elimination* procedure takes as input a sequence P of candidate programs found by the synthesizer and returns a subsequence $P^* \subseteq P$ that only contains *meaningful* and *unique* programs.

Meaningful Programs. A candidate program is *meaningful* if there *exist* an input value on which the candidate terminates and produces an output value within some timeout. Formally, we denote the output of a program p on an input tuple i as $[\![p]\!](i)$, where $[\![p]\!](i) = \bot$ if p crashes or diverges on i. We say that a program is meaningless if $\forall i.\ [\![p]\!](i) = \bot$. For example, the well-typed candidate `\x → head []` is meaningless as it yields $\bot$ regardless of the input `x`.

Testing Meaningfulness. We cannot test exactly whether an arbitrary program p is meaningless for two reasons: first, the set of possible inputs can be infinite, and second, for a given input we might need to wait an unbounded amount of time to determine whether a program terminates. Instead we say that p is *likely meaningless* with respect to a finite set of inputs I and timeout T if $\forall i \in I.\ [\![p]\!]^T(i) = \bot$, where $[\![p]\!]^T$ denotes the result of executing p for at most time T. Hoogle+ tests whether a candidate is likely meaningless by invoking SmallCheck to enumerate all the values of a given input type up to a given constructor depth, and then running the candidate on the enumerated inputs to check if they successfully produce an output within a given timeout. Thus, candidates like `\x → head []` that yield $\bot$ for all inputs are deemed meaningless and eliminated. Because the check is approximate, Hoogle+ might erroneously eliminate a meaningful program if it requires large inputs to produce an output. Our empirical evaluation shows (Sec. 6), that this happens very rarely in practice.

Lazy Candidates Can be Meaningful. In a lazy language like Haskell, determining whether a given program output $[\![p]\!]^T(i)$ is $\bot$ is actually non-trivial. Generally, Hoogle+ has to *force* program evaluation, for example, by *printing* the output (*i.e.* converting it to string). While doing so, however, Hoogle+ has to take special care not to eliminate programs that return infinite data structures. For example, consider the candidate `\x → repeat x` which returns an infinite list of `x` values. This candidate should be deemed meaningful, since it is common practice to produce infinite data structures that can be consumed lazily. Printing the output of this program, however, leads to a non-terminating execution, and hence by default the program is deemed meaningless.

To overcome this challenge, we use a special function `approxShow` introduced in [Danielsson and Jansson 2004], which prints an execution result only up to a finite given depth. If the result can be partially printed, the program is deemed meaningful. In the example above we invoke `approxShow 3 (repeat 1)` to print the result of `repeat 1` up to depth 3, which yields `"[1,1,_"`. As this value is not $\bot$, Hoogle+ deems the candidate to be meaningful.

Unique Programs. We say that a candidate p is *observationally equivalent* to another candidate p', written $p \equiv p'$ if $\forall i.\ [\![p]\!](i) = [\![p']\!](i)$, *i.e.* if p and p' return the same results for all inputs i. We say a candidate p is *unique* with respect to a set of candidates P' if for each $p' \in P'$ we have $p \not\equiv p'$, *i.e.* if for each p' there exists some *distinguishing input* i such that $[\![p]\!](i) \neq [\![p']\!](i)$.

Testing Uniqueness. Just like meaningfulness, uniqueness is impossible to check exactly. Instead, we say that p is a *likely duplicate* with respect to P' and relative to an input set I and timeout T, if $\exists p' \in P'. \forall i \in I. [\![p]\!]^T(i) = [\![p']\!]^T(i)$, *i.e.* there exists a program p' such that on any input from I either both programs return the same value or they both fail (crash or execute longer than T).

Hoogle+ presents the candidates to the user one-by-one, as soon as they are found. For each new candidate p, Hoogle+ uses SmallCheck to test whether it is a likely duplicate with respect to the

programs p_1,...,p_k that were previously shown to the user. Because the check is approximate, HOOGLE+ might accidentally eliminate a unique program if the distinguishing input required to differentiate it from every previous program is large; again, our study show that this rarely happens in practice.

Examples of Unique Programs. Our uniqueness test yielded some interesting results. Consider a queries `applyNTimes :: (a → a) → Int → a → a` from our user study, which composes n copies of a given function and applies it to an initial value. To our surprise, HOOGLE+ synthesized two candidate solutions, which at the first glance appeared equivalent: `\f n x → (iterate f x) !! n` and `\f n x → foldr ($) x (replicate n f)`. Closer examination revealed, that in fact, the two terms above behave identically when `n` is non-negative, but when `n` is negative, the former solution crashes while the latter returns `x`. On the other hand, consider the result found by the query `applyPair :: (a → b, a) → b` which applies the first element in the pair to the second element. HOOGLE+ returned the expected solution `\p → (fst p) $ (snd p)` but we found that the uniqueness test eliminated a seemingly different solution, `\p → uncurry id p`. Upon closer examination, however, we found that these two candidates indeed have the same behavior.

5.2 Comprehension

Often, the best way to understand a piece of code is to run it on some inputs, observe the outputs and then build a mental model relating the two. However, to understand a new piece of code, one does not typically run arbitrarily (randomly) chosen inputs. Instead, we can often discern patterns from small, carefully chosen inputs, that may be crafted to demonstrate some difference between the program under study and another candidate.

Examples for Comprehension. An *example* for a program p is a pair of input and output values (i,o) where $o = [\![p]\!](i)$. Motivated by the above observations, HOOGLE+ generates three kinds of examples to comprehend the synthesized programs more easily, deeply, and rapidly.

(1) ***Meaningfulness*** : HOOGLE+ determines that the program is meaningful by finding at least one *success* example (i_{succ},o_{succ}) where $o_{succ} \neq \bot$ If p is a partial function, then in the course of determining meaningfulness HOOGLE+ may also have found a failure example $(i_{fail},\bot)$. Both the success and failure examples are shown to the user to help with comprehension.
(2) ***Uniqueness*** : Additionally, HOOGLE+ only shows programs that are unique with respect to all previously shown candidates. This is established by a set of *uniqueness* examples $\overline{(i_j,o_j)}$ that differentiate p from its predecessors P', in that for each $p'_j \in P'$, we have $o_j \neq [\![p_j]\!](i_j)$. Thus, each of these uniqueness examples are also shown to help the user understand how the candidate p is different than the other p'_j candidates.
(3) ***Functionality*** : Finally, sometimes the user wants other examples that illustrate the functionality of the candidate. Hence, HOOGLE+ generates a set of functionality examples where each new input is different from *all* previously generated inputs.

6 EMPIRICAL EVALUATION

In this section, we empirically evaluate the effectiveness of type inference from tests and elimination.

Benchmarks. In all experiments, we use the component library and benchmark suite used by Guo et al. [2020]. Each of these benchmarks is a type-only query. We exclude one benchmark, which contains the type `ByteString`, since it is impossible to provide a test value for this type in HOOGLE+ (this type requires a special function call to convert from a string). To the remaining set of queries we add the tasks from our user study (Sec. 7), arriving at a total of 45 benchmarks.

6.1 Type Inference From Tests

We evaluate the quality of the type inference algorithm on two sets of inputs: tests written by participants in our user study, and tests generated randomly by QuickCheck.

User-Provided Tests. Our first experiment evaluates the accuracy rate of type inference algorithm on real user data. For this purpose, we consider the five user study tasks, for which the correct type is defined in the study definition (see Sec. 7.1). We collected 76 type inference queries for these tasks out of the logs of searches performed by users in the course of the user study, after ruling out ill-formed searches (e.g., syntactically incorrect examples). We ran the type inference algorithm on these queries. In 39 queries the correct answer is ranked first, in 4 queries it is ranked second, and in one query it is ranked third. The median rank of all queries is 1. For only 5 out of 76 queries the correct result does not appear in the top 10. This shows that our algorithm infers correct types from user-provided tests.

Randomly Generated Tests. While Hoogle+ effectively infers types for tasks from our user study, this only accounts for 5 out of 45 benchmarks. Thus, we perform a second experiment to determine whether our inference algorithm generalizes to *other* programming tasks. Recall that each of our benchmarks is a type-only query. In our second experiment, we use QuickCheck [Claessen and Hughes 2000] to generate random input-output examples as follows. First, if the query has type parameters, we randomly instantiate them using a fixed set of base types (*e.g.* `Int`, `Char`, *etc*), to get a randomly generated monomorphic instantiation. Next, we invoke QuickCheck on the instance to generate values for the inputs and outputs of the signature to get a concrete test for the original type query. We evaluate our inference algorithm by running it on one, two, or three randomly generate tests and measuring the rank at which the "correct" signature (*i.e.* original type query) appears in the inference results. We report average results over six runs to reduce the uncertainty of random example generation. The results of this experiment are summarized in Fig. 9 (Left).

Results. The heat map is sectioned by the number of type variables in benchmark queries, and each cell of the heat map shows the percentage of benchmarks (of that number of type variables) where the correct result appears at that rank. Cells with darker colors represent a larger percentage.

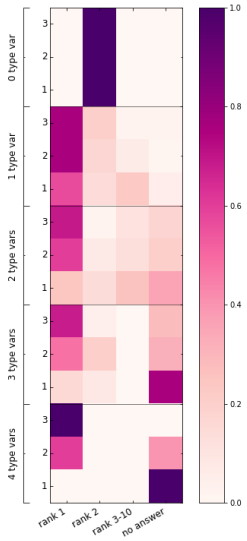
For the most part, Hoogle+ ranks the correct solution first or second, across the board. The few exceptions are seen at the bottom right of the chart, in runs with four type variables and only one test, making it hard to get the correct generalization from single concrete type. Within a given number of type variables, the rank of the correct type worsens as the number of tests decreases. This is as expected as fewer tests and more type variables lead to a larger set of possible generalizations, which makes it harder to identify the correct ones.

To confirm this intuition, we study the effect of the number of type variables and tests on the number of generalization, before and after filtering. Fig. 9 (Right) shows the minimum and maximum numbers of generalizations as well as median ranks over six runs. As we can see, the maximum value of type generalizations may reach hundreds of thousands or even millions in the case of many type variables when few test inputs are provided. However, our inference algorithm still produces the correct solution at a high rank: it has a median rank 1 or 2 in 12 out of 15 cases.

The difference in pre- and post-filtering generalizations shows that our filtering algorithm is highly effective, usually reducing the number of generalizations by at least an order of magnitude. This drastically reduced search space is a big step toward selecting the correct program. Of course, there is room for improvement as in a few cases (*e.g.* 3 type variables and 1 test) we still fail to provide a good answer.

6.2 Elimination

Next, we evaluate our test-based technique for eliminating irrelevant program candidates returned by TyGAR. Recall that test-based elimination can produce *false negatives*, *i.e.* erroneously eliminate a meaningful and unique program because it did not search large enough inputs or did not wait



# vars	# tests	median rank	pre-filter generalizations min	pre-filter generalizations max	post-filter generalizations min	post-filter generalizations max
0	3	2	45	2,216	7	413
0	2	2	45	2,216	7	413
0	1	2	45	2,216	7	413
1	3	1	9	124,176	3	37,703
1	2	1	6	3,329,455	3	779,819
1	1	1	27	3,329,455	9	779,819
2	3	1	9	40,728	6	5,071
2	2	1	9	141,127	6	31,832
2	1	6	27	407,759	18	37,778
3	3	1	2,430	546,966	496	71,853
3	2	2	2,430	969,220	54	97,433
3	1	-	35,964	8,242,171	4,846	589,284
4	3	1	243,000	818,262	38,520	77,514
4	2	1	414,558	2,276,910	27,759	202,652
4	1	-	3,314,300	81,100,863	168,838	5,939,433

Fig. 9. (Left) Type inference results on random generated tests. We compare the results between different number of type variables in the expected type and different number of tests. The x-axis is different ranks. The y-axis is grouped by number of type variables, and each group has three rows corresponding to evaluation on 3 to 1 tests. Each cell shows the row-wise percent of benchmarks. Darker colors mean more benchmarks have correct answers of that rank among those with the same number of vars and tests. (Right) Type inference counts on random generated tests. We report the median rank of correct solutions, min and max counts of type generalizations before and after filtering. '-' in ranks means no correct answer found in the top 10 results.

long enough for the program to terminate. On the other hand, elimination cannot produce *false positives*: if it deems a program meaningful and unique, this is always accurate. In the rest of the section we evaluate both the *importance* of elimination (the number of true negatives) and its *recall* (the proportion of false negatives).

Experimental setup. To evaluate the elimination strategy in Hoogle+, we ran three experiments on the 45 benchmarks in our suite:

(1) TyGAR-180: we ran TyGAR with a 180-second timeout per benchmark
(2) HP-180: we ran Hoogle+ with a 180-second timeout per benchmark
(3) HP-360: we ran Hoogle+ with a 360-second timeout per benchmark

We then manually labeled all meaningless results in TyGAR-180 and partitioned the rest into semantic equivalence classes. Next, we compared results in HP-180 to our labeled set, expecting one representative from each meaningful equivalence class to remain. We observed some differences between HP-180 and the labeled set; we refer to these mistakenly discarded programs as *loss due to misclassification*. Finally, we compared results in HP-180 with those in HP-360, detecting programs that are missing simply because they take too long to generate with elimination; we refer to this as *loss due to testing overhead*. The results are shown in Fig. 10.

The graph shows that the number of true negatives is often high, sometimes an order of magnitude higher than the number of true positives. Hence we conclude that elimination is important: without it, the user is likely to be overwhelmed with meaningless and redundant programs when searching with a type-only specification.

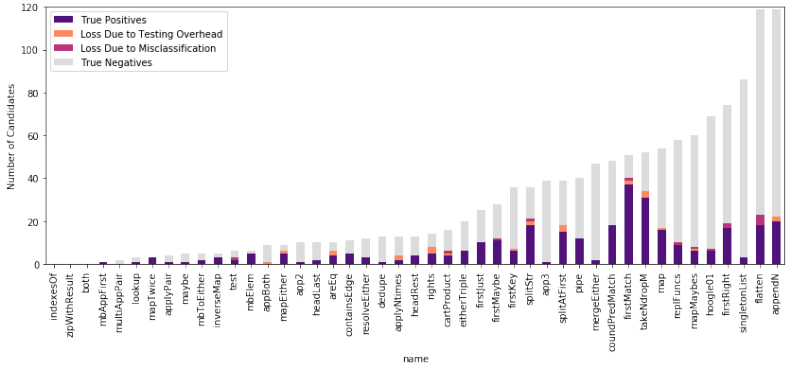


Fig. 10. Elimination results on the benchmark suite. For each benchmark we report the number of true positives (interesting programs that are reported), true negatives (uninteresting programs that are eliminated), and false negatives (programs that are mistakenly eliminated or never generated due to testing overhead).

Loss due to Misclassification. Programs lost by misclassification are programs where no witness to their meaningfulness or uniqueness was found. When looking for a witness, we only enumerate examples up to a certain constructor depth (in this experiment we used depth 3) within a timeout of 4 seconds. When the witness is outside this range, Hoogle+ will misclassify the program as uninteresting.

Our results show that misclassification is infrequent. In benchmarks with relatively high misclassification rates (*e.g.* `flatten`) it is caused by the complexity of input types, which in turn requires large constructor depths to generate interesting inputs. For instance, two solutions for the query `flatten :: [[[a]]] → [a]` are `\xs → concat (init xs)` and `\xs → concat (concat xs)`, with a distinguishing input `xs = [[[]], [[0]]]`; this input, however, lies at constructor depth 5, and hence is not generated.

Loss due to Testing Overhead. It takes extra time for Hoogle+ to test meaningfulness and uniqueness for each candidate, which in turn takes away from the time to perform the TyGAR candidate search. This means TyGAR may find fewer results than before. Most benchmarks have a testing overhead loss rate of no more than 10%.

We also carefully examined the benchmarks with high testing overhead loss rates (*e.g.* `takeNdropM`), and found that they all have a large number of displayed candidates found by the synthesizer. This means it takes more time to establish uniqueness for each *new* candidate as we must find inputs that distinguish the candidate from *each* previously displayed one. This delay in uniqueness check prevents the synthesizer from enumerating more candidates yielding testing overhead losses.

Trade-off. There is a trade-off between misclassification rate and testing overhead: increasing constructor depth and testing timeout makes test-based elimination more precise and thus decreases loss due to misclassification; at the same time, this increases testing overhead and the associated loss. We experimented with different timeouts and depth limits, and found that changing these parameters had no significant effect on most benchmarks.

7 USER STUDY

We conducted a user study that sought to answer questions about the utility and usability of our tool. We focused on the following research questions:

- **RQ1**: *Does synthesis help programmers solve program search tasks compared to traditional methods?* We believe that better performance on search tasks leads to greater productivity, as many mundane programming tasks boil down to snippet search.
- **RQ2**: *How do functional programmers express their intent in the synthesizer?* What styles of input do these programmers use to guide their search with a tool? Do they prefer to search with types, tests, or a mix?
- **RQ3**: *How do functional programmers interpret the results they receive from the synthesizer?* Users have several methods to understand a candidate presented to them. Hoogle+ provides documentation, automatically generated examples, user provided examples, and the code itself. Out of this wealth of information, what did programmers find useful in understanding the programs they are looking at and making decisions about candidate programs?

Choosing a control. In order to better understand the way Haskell programmers search for code today, we performed an initial information-gathering survey on the way Haskell programmers search for code. We surveyed 151 people online. Of those respondents who use Haskell in varied settings (47% industry, 48% academic, 54% open source are the top three) and with different levels of experience (12% less than one year, 29% 1-6 years, and the remainder over 7 years), 84 users listed Hoogle as their first engine of choice and further 27 as their second engine of choice for Haskell code. Hoogle permits searching for a library function by either a type signature or by its name. Of those who listed Hoogle as one of their top choices, 121 listed searching by type and 107 listed searching by name as one of their preferred search modalities. The next most popular search engine, Google, was reported by only 37 users as their their top choice. We therefore assess the utility of our method compared to the most frequently used alternative, and choose searching with Hoogle as our control.

7.1 Study Design

Recruitment. The Haskell community is scattered in small pockets around the world. We planned our study to work remotely to sample from the broad community. We recruited 30 participants (6 female, 24 male) via Twitter, Reddit, university lab mailing lists, and mailing lists devoted to functional programming or specifically Haskell. 22 participants were from academia (11 different institutions) and 8 were from industry (7 different companies). We asked participants to self-identify with same experience classification from our exploratory survey, and did not admit into the experiment users who have never used Haskell regularly. Of those categories, we had 12 participants new to Haskell, 10 intermediate-level users, and 8 expert users. The participants were paid for their time.

Task Selection. We selected our tasks to test different aspects of Haskell that programmers must keep in mind when searching for program snippets. We created two tasks that require using a higher-order function, while the other two tasks do not need a function as an argument. Their full description as provided to users can be found in Fig. 11:

(0) Training - `concatNTimes :: Int → [a] → [a]`. This program concatenates its second argument `n` times to itself. This task was intended to be simple with no express challenges.
Solution: `\i xs → concat (replicate i xs)`

(1) Task A - `firstJust :: a → [Maybe a] → a`, gets the first `Just` from the list with a fallback, default value. This task is challenging as it requires composing three uncommon components.
Solution: `\def xs → fromMaybe def (listToMaybe (catMaybes xs))`

(2) Task B - `dedup :: Eq a => [a] → [a]`, our running example removes adjacent duplicates from its input. This task challenges participants to consider and produce a typeclass constraint.
Solution: `\xs → map head (group xs)`

Training	***Description.*** Function `concatNTimes` takes two inputs, a natural number n and a list xs. It concatenates xs n times to itself. ***Example.*** `concatNTimes 2 "abc" = "abcabc"`
Task A	***Description.*** Function `firstJust` takes two arguments: a list of Maybe a's and a default value. It returns the first element from the list that is a Just or the default, if no such element exists. ***Example.*** `firstJust 0 [Nothing, Just 1] = 1`
Task B	***Description.*** Function `dedupe` takes one input, a list. It returns the list with any adjacent duplicate values removed. ***Example.*** `dedupe "aaabbab" = "abab"`
Task C	***Description.*** Function `applyNTimes` takes three arguments: a one-argument function f, a natural number n, and an initial value x. It applies f to x, n times, setting up a pipeline of function applications. ***Example.*** `applyNTimes (\x → x ++ x) 3 "f-" = "f-f-f-f-f-f-f-f-"`
Task D	***Description.*** Function `inverseMap` takes two inputs, a list of functions fs and an input x. It applies each element of fs to x and returns a list of those results. ***Example.*** `inverseMap [(\x → x + 2), (\x → x * 2)] 5 = [7, 10]`

Fig. 11. The task names and descriptions provided to users in our study.

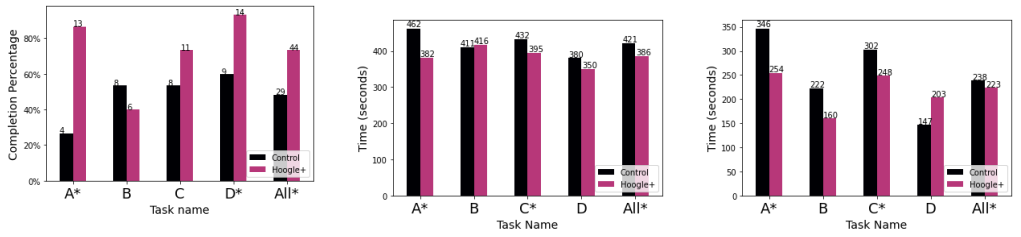
(3) Task C - `applyNTimes :: (a → a) → Int → a → a`, applies its function argument `n` times to its last argument. This task requires thinking about combining higher-order functions. Solution: `\f i x → (iterate f x) !! i` or `\f n x → foldr ($) x (replicate n f)`
(4) Task D - `inverseMap :: [a → b] → a → [b]`, applies each element of its list of functions to its second argument. Like task C, this also requires considering higher-order functions. Solution: `\fs x → zipWith ($) fs (repeat x)` or `\fs x → map ($ x) fs`

Procedure. Each participant was asked to complete four short program search tasks, listed above. Each of the four tasks had a high level, English-language description of the desired result, along with one example to characterize the expected results of that program, as shown in Fig. 11. The first two tasks were completed under our control workflow, and the next two tasks—under the treatment workflow with Hoogle+. Each half of the study opened with a training task to allow the participant some time to familiarize themselves with the workflow; each half closed with a short questionnaire. Each task was time limited to 8 minutes to ensure the whole study would fit within one hour.

Control. In the control segment of the experiment, users were provided with an online GHCi session[3] and the Hoogle search engine, which they were permitted to search by name or by type. The GHCi session was pre-seeded with all the same function and modules that Hoogle+ had at its disposal. Users were instructed to solve the task with a composition of existing library functions.

The purpose of the interpreter was to compose the different components of the solution. We therefore imposed several restrictions to focus users on program search: 1) Participants could not invoke GHCi's type informational features on library functions such as `:t`–which prints the type of an expression– `:i` or `:browse`–which give further information on a type or module; 2) they could not import any additional modules, and 3) they could only invoke GHCi to execute a (partial) solution on an example input or to inquire about the type of their (partial) solution. Additionally, users were not allowed to use control structures, recursion, or pattern matching in their solution to ensure a component-based answer.

[3] https://repl.it/languages/haskell



(a) Task completion percentage. Data labels are absolute completions.

(b) Task time averages (seconds), including timeouts

(c) Task time averages (seconds), without timed out sessions

Fig. 12. Comparison of time to complete, with and without participant timeouts. 12a shows completion improvements. An asterisks next to a task indicates a statistically significant change.

Users could follow any links on the Hoogle website, but were forbidden from making an open-internet search (*e.g.* Google or Stackoverflow). Participants were given a training task to familiarize themselves with these interaction restrictions.

Treatment. Users were presented with our tool, as presented in Sec. 2–they did not have access to the GHCi or to Hoogle. Users were trained with the same training task as in the first half of the study.

Experiment groups. Every participant in our study executed the control setting, followed by the treatment (within-subjects). In order to collect data on all tasks, we assigned users to one of two groups, rotating which tasks are control tasks and which are treatment tasks. Note that we did not additionally randomize the order of the tasks, since our control setting is similar to users' regular workflow, so there is no need to isolate knowledge transfer from it.

The tasks were grouped together: task group 1, task A and task C; and task group 2, task B and task D. We grouped the tasks as A/C and B/D to ensure that each group would have one higher-order query and one first-order query. The study groups are then:

(1) Task group 1 in control, then task group 2 with Hoogle+;
(2) Task group 2 in control, then task group 1 with Hoogle+.

Users were randomly assigned into one of the two study groups, while preserving an equal distribution of experience between the groups. Each group had: 6 with less than 1 year, 5 with 1-6 years experience, and 4 with 7+ years experience.

7.2 Results

We present the results relevant to each research question separately. In the remainder of this section, we set the threshold for statistical significance at $p < 0.1$.

7.2.1 RQ1: Does synthesis help programmers with program search tasks? For each of the four tasks in a session we measured the time until the user completed the task, and whether the task was completed or timed out (8 minutes). The results are shown in Fig. 12.

Completion rates. Of the 60 tasks attempted with each tool, 29 were completed with Hoogle and 44 with Hoogle+, a 51% increase in completion rate with Hoogle+. Fig. 12a shows the breakdown by task.

In a per-task breakdown, completion rates of users improved for tasks A, C, and D. We evaluated the change in the number of completed sessions with a Fishers-Exact test, and found the change to be statistically significant for the overall increase in completed sessions with Hoogle+ ($p = .009$),

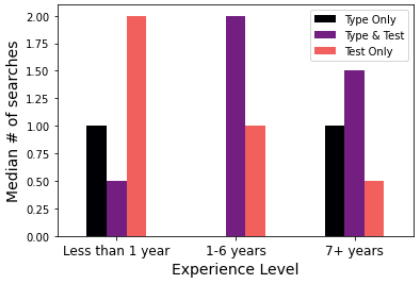


Fig. 13. Median search modality across all four tasks, by experience level

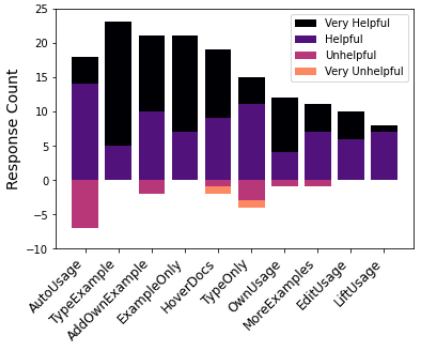


Fig. 14. Breakdown of feature perception, ordered from most used to least used.

and for tasks A ($p = .003$), and D ($p = .080$). While more users completed task C with Hoogle+ than in the control setting, this change is not statistically significant ($p = .5$).

In task B there is virtually no association between the setting used and completions ($p = .715$), and the low completion rate seems to be more influenced by the difficulty of producing the typeclass constraint in the searched type.

Completion time. Hoogle+ improved the average time to complete a task by 35 seconds. Average times are shown in Fig. 12b.

Since tasks vary in components and difficulty, we also examine the data per-task. The improvement is preserved in tasks A, C, and D. We evaluated the change in time-to-complete with a Mann-Whitney U-test, and found the change statistically significant for tasks A ($p = .0003$) and C ($p = .051$), but neither the improvement in task D ($p = .354$) nor the 5 second increase in task B ($p = .460$) are statistically significant. The tool overall enjoys statistical significance over control ($p = .004$).

Additionally, we examined only the times to complete when the user did not time out, shown in Fig. 12c. This allow us to take a closer look at how much help was Hoogle+ when it does help. While the aggregate difference is smaller, a mere 15s improvement, we notice that in the individual tasks, differences are intensified. We also notice that for two tasks, B and D, the trend has reversed itself: users who were helped by Hoogle+ completed task B, on average, a full minute faster, and task D almost a minute slower. Even still considering only those who completed their task, we do not find these differences statistically significant in task B ($p = .165$) or task D ($p = .386$). We observe similar significance for the remaining tasks (task A: $p = .0002$, task C: $p = .060$, overall: $p = .005$).

We conjecture that task B required familiarity with typeclasses, so for those unfamiliar with the feature, Hoogle+ could not help them; and, those with that knowledge could fly. Further, task D's expected solution may have been obvious to some and could easily write it out in control; yet, those in the treatment setting had to coax Hoogle+ to generate the right candidate with enough examples.

Correctness. We logged the final solutions presented if users did not time out. Between both control and treatment, across 120 recorded tasks, and 73 total completions, only one participant concluded with an incorrect solution, for task A, using Hoogle+. While this falls entirely within the margin of error, we do discuss the particulars of the session further in the next subsection.

Overall, we see that Hoogle+ greatly improves completion rates over the control setting, as well as a modestly improving the time to result. Therefore **we answer RQ1 in the affirmative.**

7.2.2 RQ2: How do functional programmers express their intent in the synthesizer? We logged user searches made in the course of the experiment, and analyzed the style of Hoogle+ searches users

made. Hoogle+ permits 3 kinds of searches: (1) *type-only* search, leaving the test part of the specification empty, (2) *test-only* search, then using a type that Hoogle+ suggested, and (3) *type-and-test* search. Users made a total of 115 searches across all Hoogle+ sessions, with users making on average a little fewer than 2 searches per task. Only 22 searches were type-only, leaving 93 searches involving at least one test.

The style of search varied greater by experience level than by task. The breakdown of these searches is shown in Fig. 13. Experts relied on tests the least, making a median of 0.5 test-only searches across all tasks, while inexperienced Haskell users made a median of 2 test-only queries. Despite our pre-study survey discovering that searching Hoogle by type was the most popular way to query for a component, searching by type-only in our synthesis setting was uniformly the least popular mode.

Test Provenance. Tests were an important part of how participants made their searches. We note where these tests came from. Task descriptions included one ready-made test. Of the tests used in searches, 46 were directly from the task; 63 were original to the participant (though some were closely based on the task or what was on screen); only 2 tests came from examples provided by Hoogle+.

To answer RQ2: across the board, users searched by type *the least* during their Hoogle+ sessions. While beginners preferred test-only searches significantly, **tests were overwhelmingly part of user searches.** Additionally, users have a strong preference for **providing their own tests**.

7.2.3 RQ3: How useful are Hoogle+ features in interpreting results? We asked users to fill out a questionnaire after completing the tasks to assess what parts of Hoogle+ they used and what they found most helpful. The ratings of Hoogle+ features by users who used them (i.e., did not mark "did not use" in the survey) appear in Fig. 14.

In general, users found Hoogle+ features to be helpful or very helpful. The only features rated very unhelpful by any user were the documentation available when hovering on a component and the type-only search, which, as seen earlier, was also the least used of all search options. The users dissatisfied with the documentation liked the idea but indicated they wanted a different experience around reading the documentation inline.

The less-used features of Hoogle+, editing and lifting a usage, were used by participants who needed their functionality, so it is not surprising they also found them helpful. A non-negligible number of users found auto-generated examples unhelpful, which we will discuss in the next subsection.

To answer RQ3, with the exception of the auto-generated examples, **Hoogle+ features are useful to users in interpreting results**.

7.3 Discussion

Overall effect on aid. Overall, the effect of Hoogle+ on user performance was very encouraging, and feedback from participants was positive. In fact, one user said they felt they didn't really solve the task–the tool did–and that it felt like cheating at programming!

The four different tasks tested in our experiment are varied and stress different parts of a participant's Haskell knowledge. Task B required knowing or picking a typeclass, and task A involved lesser-known components in the `Data.Maybe` library. The control setting required the participants to come up with intermediate types for function compositions, in order to search the individual functions by type on Hoogle. In our experience, about half of the participants did not know what the intermediate types should be in a solution *a priori*.

We observed that the task either immediately made sense to participants or they struggled with it. In the data, we see a clear bimodal performance curve in both control and treatment, between those who "got it" and those who timed out or almost timed out. Task D is the most extreme example of this, causing the time to completion of those who finished the task in the control setting to be extremely fast (e.g., one participant solved the task in under a minute, saying they encountered

a similar problem in their work). Still, more participants could solve task D with Hoogle+ than without, showing that its value is in the cases that don't immediately click.

In tasks A and C the effect of using Hoogle+ was most significant, both in completion and in the change in times. We believe that these two tasks were particularly hard to break down into intermediate types and component-searches and this played to our tool's strengths. A basic assumption of human-in-the-loop synthesis is often that the programmer is capable of helping the synthesizer break down the task. It is possible that in a functional setting, this assumption does not hold.

Barriers. We asked users about barriers to solving their tasks after both the control portion and the treatment portion of the study.

After the control setting, several users expressed feeling daunted by the task of coming up with the right intermediate types and searching for the right function that contains what they need. This ties in to the significantly slower control times in the tasks A and C that require uncommon components and complex, higher-order types, respectively.

Additionally, Hoogle users had frustrations about the tool itself. Results often contain cruft from domain specific libraries that are usually not the function or direction intended. One user explicitly named as a barrier the need to browse a large set of results on a simple search. Several users mentioned vaguely remembering the necessary function, and having to search Hoogle to recall the order of arguments or the precise name of the function, but Hoogle doesn't permit searches by documentation.

The most frequent barrier to Hoogle+ users were slow synthesis times. Specifically, the lack of indication if a search that was taking long would yield results or wait and then return nothing. Additionally, users expressed the need for messaging suggesting actions to the user when no results came up.

Several users mentioned difficulty in understanding what the candidate functions were actually doing, because any example provided was only shown as an end-to-end execution. One participant suggested drilling down into a candidate's execution on an example would help.

These point to experience and design improvements that are needed for Hoogle+ to become an effective production tool, but are not insurmountable.

Search Style. In our observations, we found that participants would fall back to example-only searches when they were at a loss for the right type (mostly with the most novice participants), or when they wanted to let the tool do more work for them. One participant made the observation that, "the point of a tool is to take the thinking out".

Task B is a particularly interesting case: only two participants searched with types-alone, the fewest of any task. Perhaps most users could tell the function's type signature `Eq a => [a] → [a]` is very underspecified– that it says very little about what should happen to inputs– and so included at least one test with their search. This highlights the occasional shortcomings of types as specifications, ones that are mitigated by allowing tests in the search specification.

Auto-generated Examples. As shown in Fig. 14, the auto-generated usage examples for candidate programs were the Hoogle+ feature users were least satisfied with. We observed that this stemmed mainly from user expectation of usage examples did not entirely aligning with the criteria for example generation (Sec. 5.2). Specifically, users did not need differentiation between the candidates as much as they wanted usages to explore the functionality of the current program they are investigating.

Users who did try to understand the candidates via the generated examples wished for a greater diversity of examples. Those who did ask for more examples tended to ask for *many* more examples, 6.7 more, on average, with some clicking the button up to 17 times (between both tasks). This shows that these users were hoping for the system to help them better understand their candidates.

This is perhaps best illustrated by the only incorrect result out of 60 Hoogle+ tasks performed. The user, a Haskell novice, made use of test-only searches but selected a type too specific for the task. They then selected a candidate that appeared to fit the task description but would crash on

inputs they never tested. The user *did investigate* the candidate by asking for more examples and editing existing ones; however, the user did not attempt any complex inputs. This user's experience demonstrates room for improvement in our example generation—better aligning its goals with the needs of users, and producing a greater variety of examples.

7.4 Threats to Validity

We selected tasks and components to operate over several common, built-in libraries. Most participants were familiar with many functions but had to limit themselves to the subset we permitted in the control setting. This introduced "unintentional complexity" as one expert user aptly put. We attempted to mitigate this with a training task in the control setting to familiarize users with our restrictions.

We gave participants only 8 minutes to complete each task. This short time limit is lab induced, and some participants reported a sort of test-anxiety that may have affected their performance. Anecdotally, many participants were close to completing the task in both control and experiment after timing out. Since the data is right-censored, times over eight minutes are only known to be over eight minutes, which may make generalizing the results for more complex tasks incorrect.

8 RELATED WORK

Component-based Synthesis. Modern IDEs support code-completion based on matching common prefixes of names (*e.g.* completing `Str` into `String`), or by using the context to narrow the candidates to well-typed completions [Perelman et al. 2012]. Type-based search engines like Hoogle [Mitchell 2004] generalize the above to find type isomorphisms [Di Cosmo 1993] *i.e. single* components whose signature match the query. In contrast, our goal is to find *combinations* of components that implement some higher-level task. When the task is specified as a type, the problem of search reduces to that of *type inhabitation*, *i.e.* finding terms that inhabit a given query type [Urzyczyn 1997]. One approach to type inhabitation is proof search [Augusstson 2005; Heineman et al. 2016; Norell 2008], which can be difficult to scale up to large component libraries. Prospector [Mandelin et al. 2005] introduces a scalable graph-based inhabitation algorithm where the components are unary functions, SyPet [Feng et al. 2017] uses Petri-nets to generalize graph-based methods to multiple argument functions, and TyGAR [Guo et al. 2020] shows how to further extend SyPet's search to polymorphic components using the idea of *succinct* type-abstractions introduced by InSynth [Gvero et al. 2013]. However, all of these require type-based queries which can be problematic for non-experts, and do not consider the question of end-to-end usability.

User Interaction in Program Synthesis. Although program synthesis is supposed to serve a user, few papers focus on the user's role in the synthesis loop. Le et al. [2017] and Peleg et al. [2018] highlight two models of iterative synthesis, the first driven by the synthesizer and the second by the user. Our work is in a different setting: API discovery for functional languages.

Several domain-specific synthesizers [Chasins et al. 2018; Chugh et al. 2016; Drosos et al. 2020] give end-users and data scientists access to synthesis to automate some of their work. These tools were evaluated against users' alternative workflow, but their users are not programmers, and the synthesis domain is far from general. Unlike these, Hoogle+ is a tool for (functional) programmers and allows users to search for general Haskell code.

Filtering and ranking synthesis results. Ranking and returning multiple results are two common approach to handing ambiguous specifications in program synthesis; the two often—but not always—go hand-in-hand. The FlashX tool family [Gulwani 2011; Polozov and Gulwani 2015] uses a ranking function to select a single, most likely program from programs that satisfy all user-provided examples, exploring both hand-crafted [Gulwani 2011] and learned [Singh and Gulwani 2015] ranking functions. Recent work on synthesizing lenses [Miltner et al. 2019] proposed a novel approach to semantic

ranking based on information theory. Unlike PBE tools that use ranking to select a single result, code completion tools [Gvero et al. 2013; Raychev et al. 2014] typically present a ranked list of results to the user, and most commonly rely on learned statistical models and syntactic features. Like these tolls, Hoogle+ offers the users several ranked candidates, both of synthesis results and of inferred types.

Synthesizers also need to filter their results to discard irrelevant programs. Sypet [Feng et al. 2017] uses Petri-nets to only return programs that use all available arguments. Hoogle+ extends this filtering: it filters TyGAR results after they are constructed, and uses more extensive criteria.

Test input generation. The extensive literature on automating testing focuses on finding bugs in manually written code. Our key observation is that these ideas in general, and property-based testing in particular, can be re-purposed for example-based elimination and comprehension in program synthesis. Hoogle+ uses the SmallCheck library [Runciman et al. 2008] to filter its candidate program list and to provide examples to demonstrate the semantics of synthesized programs.

Inferring Types from Examples. A key innovation of Hoogle+ is to allow users to specify their queries via tests that are then translated into types, enabling efficient search. Prior work on the problem of inferring types from tests has a very different context: inferring type annotations for dynamically typed languages. E.g., Chugh et al. [2011] infer types from run-time logs, An et al. [2011] instrument Ruby programs to track how each variable is *used* to then build a constraint system that is solved to infer method types, and Bonnaire-Sergeant [2019] show how to generalize execution-based guided type-recovery to handle ad-hoc recursive datatypes as found in Clojure programs. All these differ from our approach in several ways. First, the different setting: when discovering type annotations, they have program execution traces to help guide type inference. Second, all infer monomorphic types, while our goal is to infer polymorphic signatures greatly narrowing the synthesizer search space.

9 CONCLUSION

In this work we presented Hoogle+, a component-based synthesizer for Haskell that focuses on end-to-end usability of program synthesis. Hoogle+ extends a core type-driven synthesis engine TyGAR in three major ways. First, we present a novel mechanism to infer likely polymorphic type signatures from tests, which helps beginners, who are not yet fully comfortable with the Haskell type system. Second, we show how to leverage property-based testing to eliminate meaningless and repetitive synthesis results, without asking the user for additional input. Finally, we again rely on property-based testing to automatically generate examples that demonstrate the behavior of synthesized programs.

To evaluate the usefulness of Hoogle+ relative to a traditional code search workflow, we conducted a user study with 30 participants, comparing their performance on solving simple programming tasks with the Hoogle search engine *vs.* Hoogle+. We find that users equipped with Hoogle+ perform their search tasks faster are able to complete 50% more tasks.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their feedback on the draft of this paper. This work was supported by the National Science Foundation under Grants No. 1943623 and 1911149.

REFERENCES

Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for ruby. In *POPL. Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 459–472.

Lennart Augusstson. 2005. Djinn. https://github.com/augustss/djinn.

Ambrose Bonnaire-Sergeant. 2019. *Typed Clojure in Theory and Practice.* Ph.D. Dissertation. Indiana University, Bloomington.

Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *UIST 2018*. 963–975.

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *PLDI '16 (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354.

Ravi Chugh, Sorin Lerner, and Ranjit Jhala. 2011. Type Inference with Run-time Logs. In *Workshop on Scripts to Programs (STOP)*.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00 (ICFP '00)*. Association for Computing Machinery, 268–279.

Nils Anders Danielsson and Patrik Jansson. 2004. Chasing Bottoms. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Dexter Kozen (Ed.). Springer, 85–109.

Roberto Di Cosmo. 1993. Deciding Type isomorphisms in a type assignment framework. *Journal of Functional Programming* 3, 3 (1993), 485–525. http://www.dicosmo.org/Articles/JFP93.pdf Special Issue on ML.

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *CHI 2020*. Association for Computing Machinery, New York, NY, USA, 1–12.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL 2017*.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330.

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.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI 2013*.

George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. 2016. A Long and Winding Road Towards Modular Synthesis. In *ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. 303–317.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE '10*, Vol. 1. ACM Press, 215. http://portal.acm.org/citation.cfm?doid=1806799.1806833

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *CoRR* abs/1703.03539 (2017). arXiv:1703.03539 http://arxiv.org/abs/1703.03539

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI 2005*.

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proc. ACM Program. Lang.* 3, ICFP 2019, Article Article 95 (July 2019), 28 pages.

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

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. 230–266.

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *ICSE 2018*. ACM, 1114–1124.

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *PLDI '12, Beijing, China - June 11 - 16, 2012*. 275–286.

Gordon Plotkin. 1970. *Lattice Theoretic Properties of Subsumption*. Edinburgh University, Department of Machine Intelligence and Perception. https://books.google.com/books?id=2p09cgAACAAJ

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428.

John C. Reynolds. 1969. Transformational systems and the algebraic structure of atomic for-mulas.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell Symposium 2008 (Haskell '08)*. Association for Computing Machinery, 37–48.

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV - 27th International Conference, 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 398–414.

Pawel Urzyczyn. 1997. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*. 373–389.

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL 1989, Austin, Texas, USA, January 11-13, 1989*. 60–76.