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.

ACM Reference Format:

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 maps over the result to extract the head 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 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. To enable 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+ selects a set of likely type specifications as shown in Fig. 1.

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 \( \forall x \rightarrow [x] \) (which always returns the empty list) or \( \forall x \rightarrow \text{head} [x] \) (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. 2 illustrates using Hoogle+ for our running example: the programmer has specified the Haskell type signature for dedup and one example
Digging for Fold: Synthesis-Aided API Discovery for Haskell

Fig. 1. Example to Type search UI, searching for `dedup` with input was "aab", output to "ab"

of its behavior. Fig. 3 shows the list of candidate programs returned by Hoogle+ (with the correct solution at the top).

**User study.** Finally, 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 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. The core, code synthesizer uses type-guided abstraction refinement (TYGAR), described in Sec. 3.
Consider a user tasked with implementing our running example, `dedup`, who only has a test in mind for how `dedup` should behave.

**Problem.** Our user study shows that in some cases, the programmer does not know the most appropriate type of the function that they are looking to implement (Sec. 7). Specifications containing typeclass constraints are particularly tricky for beginners who may not yet have a solid grasp on which classes are needed for a given computation. For example, `dedup`’s type signature should be `Eq a => [a] -> [a]` which allows using equality checks to remove the duplicates, but the functional description of `dedup` does not explicitly state the equality requirement. Consequently, the user might search using the overly general type `a -> a` which will prove fruitless for the task at hand.

**Solution: Types from Tests.** We address the specification problem with a novel mechanism that bridges the gap between programming by-example (i.e. tests) and by-specification (i.e. types). In particular, Hoogle+ uses the provided tests to first automatically generate candidate types. In our example, as the user does not know the appropriate type for `dedup`, they search using a simple test: the input “aab” should produce the output “ab”. (Note that this input does not demonstrate the intended behavior particularly well.)

The type inference problem is itself a search-and-rank problem. Hoogle+’s inference engine uses the provided tests to infer a list of up to 10 candidate types that are presented to the user in the GUI, as shown in Fig. 1. Notice that the inferred candidates include both the “too-general” type `a -> a` as well as the “correct” type `Eq a => [a] -> [a]`.

Ideally, Hoogle+ would only list types that can actually be implemented (i.e. inhabited) by terms synthesizable from the component library. However, realizing this ideal would be impractical as it would entail running a program search for each candidate type. Instead, Hoogle+ uses an overapproximate check to show types that may be inhabited. That is, while we present the type `a -> [Char]`, we do not present the type `[Char] -> [a]` because there can be no meaningful function that can produce a polymorphic list from a string. We explain this approximation in detail in Sec. 4.4.

**2.2 Elimination**

Let’s suppose that the typeclass constraint in the “correct” candidate type `Eq a => [a] -> [a]` provides the user a clue about the functionality they are trying to achieve that nudges the user to select that particular type for synthesis, as shown in Fig. 2.

**Problem: Ambiguity.** Unfortunately even this type specification is very ambiguous: there are many candidate programs that implement the type `Eq a => [a] -> [a]` that should be quickly eliminated from consideration without troubling the user for additional specifications or hints.

---

Fig. 2. Specification UI, searching for `dedup` with its type and an example

---

Digging for Fold: Synthesis-Aided API Discovery for Haskell

Fig. 3. Some candidate programs and example table for `dedup`’s query

**Solution: Meaningful Programs.** A program is meaningful if there is some input on which the program produces output values without crashing or diverging. Hoogle+ eliminates bogus candidate results by ensuring that each program shown to the user is meaningful. For example, the program `xs -> head []` implements the given type specification for `dedup`, but is not meaningful as it will always crash regardless of the input. A programmer would likely immediately give up on any tool that returned the above as a candidate—regardless of the query!—and so it is imperative that such candidates be swiftly eliminated from consideration.

In principle, the problem of determining whether a program is meaningful is undecidable. However, in practice, we show that the SmallCheck property-based testing library [Runciman et al. 2008] can be used to efficiently sample a small space of candidate inputs to determine whether a function is meaningful. Specifically, Hoogle+ uses SmallCheck to test whether there is some input that produces an output value within a given time-bound. Given the non-strict nature of Haskell, it is common practice to write terms that produce infinite streams of values. Hoogle+ builds upon the ChasingBottoms library [Danielsson and Jansson 2004] to ensure that such terms are nevertheless considered meaningful, and hence, not eliminated from consideration.

**Problem: Duplication.** Hoogle+ displays a sequence of meaningful candidates as soon as they are found, as shown in Fig. 3. Unfortunately, there may be too many meaningful candidate to make the whole search experience useful, as many candidates re-implement the same behavior with different components. For example, one candidate 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 `init` and `tail` will be a non-empty list of identical values, e.g. `init ['a', 'a']` and `tail ['a', 'a']` are both equal to `['a']`.

**Solution: Uniqueness.** A candidate `c` is unique with respect to a set of programs if, for each program in the set, there exists a distinguishing input [Jha et al. 2010], on which `c` and the other program and
produce different results. Hoogle+ eliminates semantic duplicates from the results presented to the
programmer, by ensuring that each candidate displayed is unique with respect to the previously shown
set of programs. For example, once Hoogle+ has displayed the result \`\xs \rightarrow \init \ (\head \ (\group \ \xs))\`

it eliminates the candidate \`\xs \rightarrow \tail \ (\head \ (\group \ \xs))\` as it is not unique with respect to the
previous program.

Again, the problem of checking uniqueness is undecidable in theory, but is felled by property-
based testing in practice. As each new candidate is generated (in a streaming fashion) by Hoogle+’s
synthesizer, we use SmallCheck to search for witnesses that distinguish the candidate from each
previously shown result as detailed in Sec. 5.1. Hoogle+ only presents the new candidate to the user if
tests witnessing uniqueness can be found; otherwise the candidate is eliminated from consideration.

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 are 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 command 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 was 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 analyzing 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]. The properties evaluated are logical relations on functions, typically to ensure an invariant holds. Like QuickCheck before it [Claessen and Hughes 2000], SmallCheck tests many values to ensure that property holds
or to find a concrete counterexample where it does not. Unlike QuickCheck, SmallCheck exhaustively checks all values within a certain constructor depth, giving it the ability to find the smallest counterexample without the need for shrinking. Small counterexamples are desirable, as discussed more in Sec. 5.2.

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 "aabbaabb" → "abab", 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: "aabbaabb" → "abab" 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, [α] → [α]. 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] → [α], [α] → [Int], [α] → [α], or α → β. 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. 5.

Types. The language is equipped with a standard prenex-polymorphic type system: types T are either type variables, type constructor applications C T, 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 it contains no type variables.

Type ordering. A substitution σ = [α1 ↦ T1, ..., αn ↦ Tn] is a mapping from type variables to types that maps each αi to Ti and is identity 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′ ⊑ T, iff there exists σ such that T′ = σT. The relation ⊑ is a partial order on types. For example, type [Int] ⊑ [α] ⊑ β. This partial order induces an equivalence relation T1 ≡ T2 ≡ T1 ⊑ T2 ∧ T2 ⊑ T1 (equivalence up to variable renaming).

We say type T′ is a common generalization of a set of types T i if ∀i, T i ⊑ T′. The least common generalization (or join) of T i always exists and is unique up to ≡, so, by slight abuse of notation, we write it as a function ⊔(T i). For example, [Char] and [Int] have two common generalizations, [α] and β, and [Char] ⊔ [Int] = [α], the more specific of the two.

1Throughout this section, we write X to denote a sequence of syntactic elements X.
**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 λ-terms, and term typing, since they are entirely standard; instead we assume access to a type checking oracle Γ ⊢ e :: T, which decides whether term e checks against type T in Γ and a type inference oracle Γ ⊢ e → T, which computes the most specific type T such that Γ ⊢ 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: Γ ⊢ v → t → T1 → T2 where Γ ⊢ v → T1 and Γ ⊢ t → T2. 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 (Γ ⊢ t ∈ T), if Γ ⊢ t → T′ and T′ ⊑ 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 Γ ⊢ [1,1,1] → [1] → [Int] → [Int], witnesses the type [α] → [α].

### 4.2 From Tests to Types

Fig. 6 presents an overview of our TestToType inference algorithm. The algorithm takes as input an environment Γ (the component library) and a test suite ġ, and returns a sequence of likely type specifications Γ. Which properties need to hold of Γ? Assume that the user’s intended program is e*: and its most general type is T*: (Γ ⊢ e* → T*). Then T* is the best type specification for synthesizing e*: although any T ⊑ T* might yield the desired program since Γ ⊢ e* :: T necessarily holds, there might be many more programs e such that Γ ⊢ e :: T compared to Γ ⊢ 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 Γ such that T* is likely to occur early in this sequence.

Towards this goal TestToType proceeds in three steps. First, it uses the inference oracle to obtain the concrete types Γ of the tests. Next, it uses the function AntiUnifyAll to compute Γ, the least common generalization of Γ, that maybe be inhabited by relevant programs, as determined by the function Inhabited. Note that each T ∈ G is witnessed by every test t in the input test suite: this is because Γ ⊢ t ∈ Γ by the definition of least common generalization, and Γ ⊑ 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. 7 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...
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 \( \text{Int} \rightarrow \alpha \) is uninteresting because there is no way to construct a value of arbitrary type \( \alpha \), while the type \( \alpha \rightarrow \beta \rightarrow \beta \) 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 inhabited by degenerate terms; if the analysis says "yes", the type might still be uninhabited depending on the component library. The function \text{INHABITED} in Fig. 8 implements this analysis. This function deems a type inhabited if its return type is \text{reachable} and each of its argument types is \text{relevant}.

4.4 Type Filtering

Although the least common generalization \( T_\cup \) 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 \( T_\cup \) \( \{ T \mid T_\cup \subseteq T \} \), 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 \( \text{Int} \rightarrow \alpha \) is uninteresting because there is no way to construct a value of arbitrary type \( \alpha \), while the type \( \alpha \rightarrow \beta \rightarrow \beta \) 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 inhabited by degenerate terms; if the analysis says "yes", the type might still be uninhabited depending on the component library. The function \text{INHABITED} in Fig. 8 implements this analysis. This function deems a type inhabited if its return type is \text{reachable} and each of its argument types is \text{relevant}.
As a final step, the function TopK returns the k highest ranked candidate types (in our implementation k = 10). Our ranking approximates the likelihood a candidate type was 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.

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 \([\text{Int}] \rightarrow [\text{Int}]\). This heuristic penalizes abstracting this type into \(\alpha \rightarrow \alpha\) or \(\alpha\), 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 $[\text{Char}]$ and do not penalize abstracting it into $\alpha$, 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 $[\text{Int}] \to [\text{Int}]$ example, the generalization $[\alpha] \to [\alpha]$ has higher rank than $[\alpha] \to [\text{Int}]$ because the former abstracts both occurrences of $\text{Int}$ into $\alpha$. 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 Hooble+ is capable of producing. To implement this heuristic, we build an inverse substitution between the anti-unification result $\Gamma_\perp$ 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 $[\text{Int}\mapsto\alpha]$, whereas for $[\alpha] \to [\text{Int}]$ it is $[\text{Int}\mapsto(\alpha,\text{Int})]$, so the former is ranked higher (note that we keep the identity mapping $\text{Int}\mapsto\text{Int}$ in the substitution, unless all occurrences of $\text{Int}$ were replaced).

Our third heuristic is to prioritize general types over specific types. In our example, $[\alpha] \to [\alpha]$ has higher rank than $[\text{Int}] \to [\text{Int}]$ 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 $[\alpha] \to \text{Int}$ (e.g. list length), $[\text{Int}] \to \text{Int}$ (e.g. sum of the elements), or even $[\text{Char}] \to \text{Int}$ (e.g. number of spaces). Note that we cannot assume that the $T_i$ type for this (singleton) test suite is $[\alpha] \to \text{Int}$ with $\alpha$ 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 $\text{Nothing}$ 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 applyN times from our user study (see Sec. 7 for details): $(\lambda x \to x + x) \to "s" \to 2 \to "ssss"$. Here the first argument has polymorphic type $[\alpha] \to [\alpha]$, and, perhaps counter-intuitively, the test does not actually constrain $\alpha$ to be $\text{Char}$.

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: $T ::= \ldots [?\alpha]$. The wildcards are introduced by the inference oracle for tests $\Gamma \vdash t \Rightarrow T$, when tests contain polymorphic values. For example, we infer the type $?\alpha \to \text{Int}$ for $[] \to 0$. Unlike regular type variables $\alpha$, 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 $\text{AntiUnify}$. The join of two types is now not a single type but a set of types, for each possible instantiation of the wildcard. Our algorithm for computing the join efficiently (without enumerating infinitely many potential instantiations) is deferred to Appendix A. For example, the join of two types $([?\alpha] \to [?\alpha]) \to [\text{Char}] \to \text{Int} \to [\text{Char}]$ and $(\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \to \text{Int}$ from the higher-order example above, is a pair of types $(?\beta \to ?\beta) \to ?\beta \to ?\beta \to ?\beta$ and $(\text{Int} \to \text{Int}) \to ?\beta \to ?\beta \to ?\beta$. The first result comes from instantiating $?\alpha \mapsto \text{Char}$ and the second one from $?\alpha \mapsto \text{Int}$; importantly, any other instantiation would lead to a type that is more general than either of the two. After the join has been computed, 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.

we say that a program is meaningful if we check that both (partially) printed, then the program is still deemed meaningful. Formally, we extend The elimination

Type classes are a popular feature of the Haskell type system [Wadler and Blott 1989], and we Hoogle+

Unique Programs.

head []

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 differing comprehension goals.

5.1 Elimination

The elimination procedure takes as input a set \( \mathbb{C} \) of candidate programs found by the synthesizer and eliminates some candidates to return a subset \( \mathbb{P} \subseteq \mathbb{C} \) only comprising candidates that are meaningful and unique.

**Meaningful Programs.** A candidate program is meaningful if there exist input values on which the candidate terminates and produces an output value within some timeout. That is a candidate is not meaningful if for every input, the candidate either evaluates to an exception (written as \( \bot \)) or times out without producing a result. Formally, we denote result of a program \( P \) over input \( I \) as \([P](I)\), and we say that a program is meaningful if \( \exists I. [P](I) \neq \bot \). For example, the well-typed candidate \( \\lambda x \rightarrow \text{head } [\ ] \) is not meaningful as it yields \( \bot \) regardless of the input bound to \( x \).

**Testing Meaningfulness.** Hoogle+ tests whether a candidate is meaningful by invoking SMALLCHECK to enumerate all the values of the given input type up to a given size and then running the candidate on the enumerated inputs to see if they successfully produce a value within a given timeout. Thus, candidates like \( \\lambda x \rightarrow \text{head } [\ ] \) that yield \( \bot \) for all inputs enumerated by SMALLCHECK are quickly deemed not meaningful, and hence, eliminated.

**Lazy Candidates Can Be Meaningful.** However, Hoogle+ has to take special care not to eliminate Haskell functions that can lazily operate over infinite data structures. For example, consider the candidate \( \\lambda x \rightarrow \text{repeat } x \) which returns an infinite sequence of \( x \) values. This candidate is not useless, but it does not complete execution within any timeout.

Instead, we test such candidates using the method introduced by [Danielsson and Jansson 2004] to test programs operating on infinite data, which is, to (a) execute the program for a finite execution depth and then (b) print the results of the work done during that finite execution. If the partial work can be (partially) printed, then the program is still deemed meaningful. Formally, we extend \([P](I)\) to denote the approximated execution result of a program \( P \) over input \( I \). For example, to test the above repeat candidate, for a depth of 3 Hoogle+ uses SMALLCHECK to generate the input 1 and then evaluates the term: \( \text{approxShow } 3 \ \text{(repeat } 1) \) which returns the result \"[1,1,1,\_\_\_]\" (i.e. the list with a prefix of three copies of 1). As the above value is not \( \bot \), Hoogle+ deems the candidate to be meaningful.

**Unique Programs.** Hoogle+ ensures that the returned candidates do not contain duplicates which have the same input-output semantics even though they may syntactically be different. We say 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 $\overline{P}$ if for each $P' \in \overline{P}$ we have $P \neq P'$, i.e. if for each $P'$ there exists some input $I$ such that $[P](I) \neq [P'](I)$.

**Testing Uniqueness.** To improve interactivity, we want to present the candidates one-by-one, as soon as they are found. Hence, HOOgle+ uses SMALLCHECK to test whether each new candidate $P$ is unique with respect to the programs $P_1, \ldots, P_k$ that were previously shown to the user. To do so, we query SMALLCHECK to check if

$$\exists I_1, [P](I_1) \neq [P_1](I_1) \land \ldots \land \exists I_k, [P](I_k) \neq [P_k](I_k)$$

That is, we search for $k$ inputs $I_1, \ldots, I_k$ that respectively witness how the the new candidate $P$ is not equivalent to each of the previously shown $P_i$ for $i \in 1..k$. If the above search fails—that there is some $P_j$ for which a distinguishing input cannot be found—then the candidate is considered to be equivalent to $P_j$ and hence not unique, and so is eliminated from the user’s consideration. Note that we do not require a single input $I$ on which $P$ differs from each of $P_1, \ldots, P_k$ as that would be too restrictive and would not allow us to eliminate enough programs.

**Examples of Unique Programs.** Our uniqueness test yielded some interesting results. Consider one of our test queries `applyNTimes :: (a -> a) -> Int -> a -> a` which composes $n$ copies of function $f$ and applies it to $x$. HOOgle+ synthesized the candidate $\backslash f \ n \ x \rightarrow (\text{iterate} \ f \ x) \!! \ n$. This was different, and far shorter, than the solution we expected to find: `foldr \$ \ x \ (\text{replicate} \ n \ f)`. In fact, the two terms above behave identically when $n$ is non-negative, but when $n$ is negative, the synthesized solution would crash but the expected one does not and hence, they are in fact, not equivalent. 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. Our expected solution was $\backslash p \rightarrow (\text{fst} \ p) \$ \ (\text{snd} \ p)$ but we found that the uniqueness test eliminated a seemingly unrelated solution, $\backslash p \rightarrow \text{uncurry} \ \text{id} \ p$ which, at first seemed like a bug. However, after some examination, we found that these two candidates indeed have the same behavior, despite their dissimilarity.

**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 the program is meaningful by finding at least one success example $(I_{\text{succ}}, O_{\text{succ}})$ where $O_{\text{succ}} \neq \bot$. If $P$ was not a total function, then in the course of determining meaningfulness HOOgle+ may also have found a failure example $(I_{\text{fail}}, \bot)$. 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 $(I_j, O_j)$ that differentiate $P$ from its predecessors $\overline{F}_j$, in that for each $j$, 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—and vice versa.

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 all inputs are guaranteed to be different than those used to show meaningfulness and uniqueness.
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]. We filter this suite to include only types for which a user could input a value from a string (e.g., Int because “7” is readily read to an Int). This filtering ensures we only evaluate benchmarks a user could reasonably encounter. This filtering maintains all queries except those involving the ByteString type, a type requiring 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. 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 from user-provided tests 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 generated 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 was at that rank. Cells with darker colors represent a larger percentage.

For the most part, Hoogle+ ranks the correct solution first, across the board. The few exceptions are seen at the bottom right of the chart, in runs with four type variables but 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 provide less information about how to generalize the concrete types. Fewer tests also means more options for generalization to overwhelm the heuristic ranking function, leading to more “no answer” results.

Generally, the number of generalizations increases as the number of type variables increases or the number of tests decreases. Consequently, benchmarks with more type variables are harder to solve: there is a higher higher probability that all test inputs for some argument will be drawn from the same concrete type which yields a concrete type in the result of anti-unification. This thereby places a high burden on type generalization and ranking.
To demonstrate the above phenomenon, we study the effect of the number of type variables on the number of pre- and post-filter types. Fig. 9 (Right) shows the minimum and maximum numbers of possible type generalizations before and after the type filtering as well as median ranks over six runs. Specifically, 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 elimination of irrelevant candidates returned by TYGAR.

In theory, evaluating elimination would test two directions of error: (i) keeping an uninteresting program, and (ii) ruling out an interesting program. However, since our algorithm always rules out a program unless there is a concrete witness that it is meaningful and unique, errors of the first kind cannot occur. We therefore focus our evaluation on errors of the second kind.

**Experimental setup.** To evaluate the accuracy of 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 by 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 by timeout. The results are shown in Fig. 10.

Loss by 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 example depth. For instance, two solutions for the query flatten ([[[a]]] -> [a]) are \(\mathcal{X}_S \rightarrow \text{concat} (\text{init} \mathcal{X}_S)\) and \(\mathcal{X}_S \rightarrow \text{concat} (\text{concat} \mathcal{X}_S)\), with a distinguishing input \(\mathcal{X}_S = [[[[]]], [[[0]]]]\); this input, however, lies at depth of 5, and hence is not generated.

Loss by Timeout. 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 timeout loss rate of no more than 10%.

We also carefully examined the benchmarks with high timeout 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 timeout losses.

Tradeoff. There is a tradeoff between reducing timeout losses and reducing misclassifications as the former requires more efficient candidate elimination, while the latter requires searching exhaustively up to to larger depths. We experimented with different timeouts and depth limits in finding differentiating examples. Generally, the differences between changing these limits are not significant for most of the benchmarks. It only shows obvious differences for benchmarks with a high rate of loss by misclassification or timeout. For example, in the aforementioned flatten benchmark, more misclassified candidates are identified to be meaningful after we increase the depth limit to 6. Meanwhile, the side effect of adopting this strategy is more loss by timeout.

7 USER STUDY
We focus on the technical and human aspects of program search and evaluation. We sought to answer questions about the utility and usability of our tool with 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 will lead to greater productivity, as many mundane programming tasks boil down to program search problems.
- **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 input-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 listed using Hoogle as
their number one search engine of choice. 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 for Haskell code
search, 121 listed searching by type and 107 listed searching by name as one of their search engines
of choice. 127 users listed Hoogle as their first or second search engine of choice for Haskell code.

The next most popular search engine, Google, was reported by 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. Our participants represented 22 participants from 11 academic
institutions and 8 industry programmers from 7 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 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. We list for each task its name,
type and expected solution. Their full description as provided to users can be found in Appendix B:

(0) Training - concatNTimes :: Int → [a] → [a]. This program concatenates its second argu-
ment n times to itself. This task was intended to be simple to solve with no express challenges.
Solution:\i xs → concat (replicate i xs)
Digging for Fold: Synthesis-Aided API Discovery for Haskell

(1) Task A - \texttt{firstJust :: a \rightarrow [Maybe a] \rightarrow a}, gets the first \texttt{Just} from the list with a fallback, default value. This task requires composing one common and two uncommon components.

Solution: \texttt{\textproc{\textbackslash def \textit{xs} \rightarrow fromMaybe \textit{def} (\textit{listToMaybe} (\textit{catMaybes} \textit{xs}))}}

(2) Task B - \texttt{dedup :: Eq \textit{a} \Rightarrow \{[\textit{a}]\} \rightarrow \{[\textit{a}]\}}, our running example removes adjacent duplicates from its input. This task challenges participants to consider and produce a typeclass constraint.

Solution: \texttt{\textproc{\textbackslash xs \rightarrow map \textit{head} (\textit{group} \textit{xs})}}

(3) Task C - \texttt{applyNTimes :: (\textit{a} \rightarrow \textit{a}) \rightarrow \textit{Int} \rightarrow \textit{a} \rightarrow \textit{a}}, applies its function argument \textit{n} times to its last argument. This task requires thinking about combining higher-order functions.

Solution: \texttt{\textproc{\textbackslash f \textit{i} \textit{x} \rightarrow (iterate \textit{f} \textit{x}) \textprop{!!} \textit{i}}}

(4) Task D - \texttt{inverseMap :: \{[\textit{a} \rightarrow \textit{b}]\} \rightarrow \textit{a} \rightarrow \{[\textit{b}]\}}, applies each element of its list of functions to its second argument. Like task C, this also requires considering higher-order functions.

Solution: \texttt{\textproc{\textbackslash fs \textit{x} \rightarrow \textit{zipWith} (\$) fs (\textit{repeat} \textit{x})}}

\textbf{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. The first two tasks were completed under our control workflow followed by a short questionnaire. In the second half of the study—the treatment phase—participants used Hoogle+ with guidance for a short training task, also listed above, to become familiar with it. They then completed the next two tasks with Hoogle+ followed by another questionnaire. Each task was time limited to 8 minutes to ensure the whole study would fit within one hour.

\textbf{Control}. In the control segment of the experiment, users were provided with an online GHCi session\footnote{https://repl.it/languages/haskell} 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 \texttt{:\textit{t}--\textit{i}}--which prints the type of an expression--\texttt{:\textit{i} or :\textit{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.

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.

\textbf{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.

\textbf{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+;
Fig. 11. Comparison of time to complete, with and without participant timeouts. 11c shows completion improvements. An asterisk next to a task indicates a statistically significant change.

(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 user’s 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. 11.

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. 11c 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+ ($p = .009$), 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. 11a.

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. 11b. 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 less 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. 12. 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. 13.

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. All tasks required reasoning about the intermediate type of function compositions, either by coming up with that type in the control setting, or by reading that type in the treatment setting. In our experience, about half of the participants did not know what the intermediate types should be in a solution *a priori*—they relied on clues from the provided example or from code snippets on screen.

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 the heaviest decomposition,

and again reinforces the possibility that deconstruction by type is harder than a functional division into subtasks that synthesizers sometimes rely on.

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. 13, 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 `St` 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 those that inhabit a given query type [Urzyczyn 1997], which can be done via proof search [Augustsson 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.** Though 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. While Hoogle+ is also iterative in the structure of its session, its setting is different: it performs component-based synthesis, in a functional language with varied types. Also, in its role to users, Hoogle+ is a search tool, a far better defined scenario than either work that consider “synthesizers” in general.

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 (though they are sometimes programming-capable), and the synthesis domain is far from general. Unlike these, Hoogle+ is a tool for 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 handling 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 them 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 input generation focuses on validating a particular program or finding bugs in it. Our key observation is that these ideas in general, and the SMALLCHECK library [Runciman et al. 2008] in particular, are crucial for making synthesizers usable. Hoogle+ uses random input generation to filter its candidate program list and to provide usage scenarios to demonstrate the semantics of the resulting 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 fast symbolic 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, focusing on end-to-end usability of program synthesis. We present a novel mechanism to infer plausible candidate types from tests alone and filter them with an over-approximation to enable rapid synthesis. We further presented a candidate program filtering strategy and evaluated it be largely effective at removing always-crashing or duplicate candidates. We aid user comprehension by generating examples that support different comprehension goals; we generate examples to demonstrate meaningfulness, uniqueness, and functionality. Finally we ran a user study on Hoogle+ with 30 participants comparing their performance to the Hoogle search engine. We find users equipped with Hoogle+ both 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


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


The advanced anti-unification algorithm is presented in Fig. 14.

Example. concatNTimes 2 "abc" = "abcabc"

Example. concatNTimes 2 "abc" = "abcabc"

**A ANTI-UNIFICATION WITH WILDCARDS**

The advanced anti-unification algorithm is presented in Fig. 14.

**B TASK CARDS**

**B.1 Training Card**

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"

**B.2 Task A Card**

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

**B.3 Task B Card**

Description. Function dedupe takes one input, a list. It returns the list with any adjacent duplicate values removed.

Example. dedupe "aaabbab" = "abab"
B.4 Task C Card

**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.