




Hooglex: Constants and λ -abstractions in Petri-net-based Synthesis using Symbolic Execution

Henrique Botelho Guerra  

INESC-ID and IST, University of Lisbon, Portugal

João F. Ferreira   

INESC-ID and IST, University of Lisbon, Portugal

João Costa Seco   

NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

Abstract

Type-directed component-based program synthesis is the task of automatically building a function with applications of available components and whose type matches a given goal type. Existing approaches to component-based synthesis, based on classical proof search, cannot deal with large sets of components. Recently, HOOGLE+, a component-based synthesizer for Haskell, overcomes this issue by reducing the search problem to a Petri-net reachability problem. However, HOOGLE+ cannot synthesize constants nor λ -abstractions, which limits the problems that it can solve.

We present HOOGLE*, an extension to HOOGLE+ that brings constants and λ -abstractions to the search space, in two independent steps. First, we introduce the notion of *wildcard* component, a component that matches all types. This enables the algorithm to produce *incomplete functions*, i.e., functions containing occurrences of the wildcard component. Second, we complete those functions, by replacing each occurrence with constants or custom-defined λ -abstractions. We have chosen to find constants by means of an inference algorithm: we present a new unification algorithm based on symbolic execution that uses the input-output examples supplied by the user to compute substitutions for the occurrences of the wildcard.

When compared to HOOGLE+, HOOGLE* can solve more kinds of problems, especially problems that require the generation of constants and λ -abstractions, without performance degradation.

2012 ACM Subject Classification Software and its engineering \rightarrow Automatic programming; Theory of computation \rightarrow Automated reasoning

Keywords and phrases Type-directed, component-based, program synthesis, symbolic execution, unification, Haskell

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.4

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.9.2.20>

Funding FCT UIDB/04516/2020, FCT UIDB/50021/2020, and ANI Lisboa-01-0247-Feder-045917.

Acknowledgements We want to thank to the anonymous reviewers, for the constructive feedback.

1 Introduction

Program synthesis is the task of automatically building a program that fulfills a specification supplied by the user [12]. Specifications can vary from examples [31], sketches [36], to ontologies [4] and types [13]. In type-guided component-based program synthesis, users provide the type of the function to synthesize (the *query type*), and optionally, input-output examples. Each solution is composed of applications of functions from a given *component set*. A recent example is HOOGLE+ [14, 20], a state-of-the-art synthesizer for the Haskell programming language that successfully solves several real-world problems with large component sets. For example, given the query type $(a \rightarrow b) \rightarrow [a] \rightarrow b$, it can



© Henrique Botelho Guerra, João F. Ferreira, and João Costa Seco;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 4; pp. 4:1–4:28



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



synthesize the function `\x1 x2 -> x1 (GHC.List.head x2)`. Multiple solutions are filtered using input-output examples. Unlike most approaches to component-based synthesis, which are based on classical proof search, HOOGLE+ can deal with large sets of components, because it reduces synthesis to a Petri-net reachability problem, following the approach of SYPET [7], a component-based synthesizer for Java.

Challenges for constants and λ -abstractions. Despite the benefits of Petri-net-based approaches, they exclude constants and custom λ -abstractions from the search space, because Petri nets only synthesize solutions whose terms belong to the component set, and it is impossible to insert all constants and custom λ -abstractions in a finite set. We found several problems in StackOverflow that HOOGLE+ cannot solve because the solutions require constants or λ -abstractions to be synthesized. So, bringing both classes of terms to the search space will allow HOOGLE+ to solve more problems, making life easier for Haskell programmers.

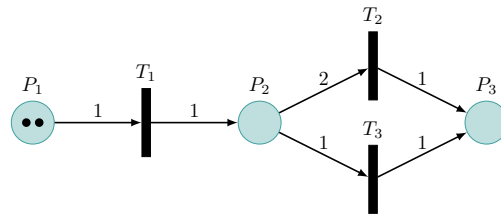
Motivating example. As an example, suppose that we want to append the constant 0 to a list. We provide to HOOGLE+ the query type `[Int] -> [Int]` together with the example that maps the input `[1]`¹ to the output `[1, 0]`. A solution is as simple as `\x1 -> x1 ++ [0]`, however, HOOGLE+ is not able to synthesize it, because it requires the constant `[0]` to be synthesized. The same happens with custom λ -abstractions. Suppose that we want to map each element of a list of integers to its square. For example, given `[1, 2, 3]`, the output should be `[1, 4, 9]`. The query type is `[Int] -> [Int]`, and a solution is `\list -> map (\x -> x * x) list`. However, this solution cannot be synthesized by HOOGLE+ as λ -abstractions do not belong to the search space.

Our approach. In this work we propose and evaluate a solution to bring constants and λ -abstractions to the search space of HOOGLE+, following two independent steps. First, we add to the component set the *wildcard component*, a component that matches all types. The Petri net is then allowed to synthesize *incomplete functions*: functions that use that component, such as `\list -> map wildcard list`. In this example, the wildcard component appears where a function is expected; however, in general, it could appear in place of an integer, string, or any other type. The second step is to replace the occurrences of the wildcard component with constants or λ -abstractions. When the wildcard occurs in place of a constant, we use a unification algorithm based on symbolic execution, that uses the input-output examples provided by the user to infer the constant. When the wildcard occurs in place of a function, we synthesize a λ -abstraction, using a faster, bespoke synthesizer.

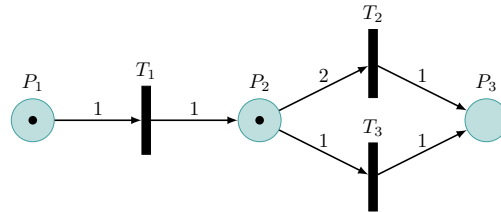
Contributions. In summary, our contributions are:

1. We develop a new unification algorithm for a subset of Haskell, that we use to replace the occurrences of the wildcard component with constants. The algorithm is generic enough for other uses, as explained in Section 8.
2. We present HOOGLE*, an extension of HOOGLE+, that synthesizes functions with constants and λ -abstractions. As shown in Section 5, it solves several problems that cannot be solved by the original HOOGLE+.

¹ Haskell list notation is represented in italic font, to avoid confusion with citations.



■ **Figure 1** A Petri net with 3 places, 3 transitions, and 6 edges. In Feng et al. [7].



■ **Figure 2** The Petri net of Figure 1 after T_1 has fired. In Feng et al. [7].

Document structure. Section 2 presents the background: Petri nets and HOOGLÉ+; Section 3 presents the unification algorithm; Section 4 describes the extension made to HOOGLÉ+; Section 5 evaluates HOOGLÉ*, by comparing it to HOOGLÉ+; Section 6 discusses the related work; Section 7 discusses the limitations of this work; and Section 8 summarizes the lessons learned and the future work.

2 Background

In this section, we describe Petri nets, HOOGLÉ+, and SYPET.

2.1 Petri nets

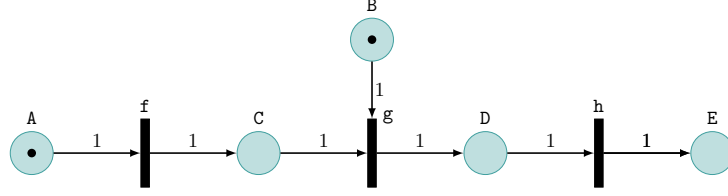
Petri nets are used by HOOGLÉ+ to represent the search space. In this section, we define relevant concepts of Petri nets and present examples.

► **Definition 1.** A *Petri net* is a tuple (P, T, E, W) , where P is the set of places, T is the set of transitions, $E \subseteq (P \times T) \cup (T \times P)$ is the set of edges between places and transitions and between transitions and places, and $W : E \rightarrow \mathbb{N}_0$ is a function that maps each edge to its weight. Each place in a Petri net can have zero or more tokens. A **marking** (also known as configuration) M of a Petri net $N = (P, T, E, W)$ is a function $P \rightarrow \mathbb{N}_0$ that maps each place to its number of tokens.

We represent places by drawing circles and transitions by drawing narrow rectangles. Edges are represented by arrows, and each natural number we write near each edge is its weight. For example, consider the Petri net in Figure 1. The places are P_1 , P_2 and P_3 , and the transitions are T_1 , T_2 and T_3 . The edge (P_2, T_2) has weight 2, and all the other edges have weight 1. The place P_1 has 2 tokens, whereas the remaining places have no tokens.

We explain next how transitions can change the marking of a Petri net, defining the concepts of enabled transition and firing a transition.

► **Definition 2.** We say that the transition t is **enabled** if and only if for each place p with an edge to t , the number of tokens is at least the weight of the edge (p, t) . We say that **firing an enabled transition** is changing the marking of the Petri net, consuming a certain number of tokens from each place that has an edge to the transition, and producing a certain number of tokens in each place with an edge from the transition, according to the weight of each edge.



■ **Figure 3** A Petri net for the component set $f:A \rightarrow C$, $g:B \rightarrow C \rightarrow D$, $h:D \rightarrow E$. The initial marking for the query type $A \rightarrow B \rightarrow E$.

In the example of Figure 1, T_1 is the only enabled transition. Firing T_1 produces the marking of Figure 2: one token was consumed in P_1 ($W(P_1, T_1) = 1$), and one token was produced P_2 ($W(T_1, P_2) = 1$).

The last concept about Petri nets that we introduce is a decision problem called reachability.

► **Definition 3.** Given a Petri net $N = (P, T, E, W)$, with marking M , and new marking M' , the **reachability problem for Petri nets** consists of assessing whether it is possible to reach M' starting at M and by firing a certain sequence of transitions. We say that M is the **initial marking**, M' is the **target marking**, and the **trace** is the sequence of fired transitions.

For example, consider the Petri net and marking M shown in Figure 1. A marking $\{P_1 \mapsto 0, P_2 \mapsto 0, P_3 \mapsto 1\}$ is reachable from M . A trace is $\langle T_1, T_1, T_2 \rangle$.

2.2 SyPet, Hoogle⁺ and Synthesis via Petri-net reachability

SYPET [7] is a scalable component-based synthesizer for Java and deals with large component sets by reducing the problem to a Petri-net reachability problem. HOOGLE⁺ [14] adapts this idea to the Haskell programming language, extending the approach to deal with parametric polymorphism, high-order functions, and typeclasses. In its latest version, it support input-output examples [20]. In this section, we explain how Petri nets can be used for synthesis, as well as an overview of SYPET, and the changes introduced by HOOGLE⁺.

Petri net construction. Generally speaking, SYPET starts with building a Petri net that models the component set, and then solves the reachability problem, using the resulting trace to synthesize functions. Given a component set C and a query type t , SYPET constructs the Petri net $N = (P, T, E', W)$, and the initial marking as follows.

1. The places in P are the parameter types and return types of the components in C .
2. The transitions in T are the components of C , i.e., $T = C$.
3. For each type t' , and component c , we have $(t', c) \in E$ with $W((t', c)) = m$ if and only if t' is the type of $m > 0$ parameters of c .
4. For each type t' , and component c , we have edge (c, t') if and only if t' is the return type of c .
5. For each place p that is the type of a parameter of the query type, we add a clone transition k where $W(p, k) = 1$ and $W(k, p) = 0$.
6. For each parameter type of the query type we put as many tokens as the number of arguments of the given type.

Figure 3 shows an example of a Petri net that models a synthesis problem.

Synthesizing a function from a trace. Once the Petri net is built, we solve a reachability problem, where the target marking has a single token in the place that represents the return type of the query type. Then, the resulting trace is used to synthesize the desired function. For instance, in Figure 3, the target marking would have a token in place E , and a trace is $\langle f, g, h \rangle$, from which the function $\backslash \text{arg1 arg2} \rightarrow h (g (f \text{ arg1}) \text{ arg2})$ is synthesized. However, synthesizing a function from a trace is not trivial, because a trace cannot distinguish between different tokens in the same place, and no notion of order of incoming edges is maintained. So, multiple functions may arise from a single trace. We do not explain how SYPET performs the reachability analysis and synthesis from traces, as it is not necessary to follow the rest of this paper; for more details, see Feng et al. [7].

Hoogle+. So far we have discussed the algorithm of SYPET, which only supports monomorphic types. However, most functions from the Haskell libraries have polymorphic types. Thus, HOOGLÉ+ [14] has to deal with polymorphic types, which introduce the following challenges, if we represent all the monomorphic types in the Petri net: there is no limit to the set of types that may arise (for example $[\text{Char}]$, $[[\text{Char}]]$, $[[[\text{Char}]]]$, etc.), and some components, such as $\text{id} :: a \rightarrow a$, create a transition for each place. Representing monomorphic types, even if we bound the set of types, leads to an intractable Petri net, so HOOGLÉ+ uses abstract types, representing sets of concrete, monomorphic types. For example, the abstract type τ is the set of all existing types, whereas $\text{Maybe } \tau = \{\text{Maybe } t : t \in \text{Type}\}$. The algorithm starts with the most abstract Petri net, containing only the place τ , which leads to ill-typed programs. Then, the type errors are used to refine the Petri net, introducing more concrete types. For more details, see Guo et al. [14].

3 Unification via Symbolic Execution

Petri nets allow the generation of functions with occurrences of the wildcard component. Our goal is to use HOOGLÉ+ to generate functions that may contain wildcards and then replace each wildcard occurrence with a constant or a custom λ -abstraction, matching the set of given input-output examples. For this purpose, we use a unification algorithm that, given a source expression with symbolic variables, and a target grounded expression, computes a substitution for the symbolic variables so that the first expression evaluates to the second expression. When a solution is found by HOOGLÉ+, we replace the occurrences of the wildcard with symbolic variables in the synthesized function, the parameters with the input of the input-output example, and unify the resulting expression with the output of the input-output example. Consider the example from Section 1, where constant $[0]$ is appended to the input list. The query type is $[\text{Int}] \rightarrow [\text{Int}]$, and an example maps the input $[1]$ to the output $[1, 0]$. The Petri net will generate $\backslash x1 \rightarrow x1 ++ \text{wildcard}$. We then replace the occurrences of *wildcard* with fresh symbols and unify $[1] ++ s1$ with $[1, 0]$, where the algorithm substitutes the symbolic variable $s1$ with the constant $[0]$, by inspecting the definition of $(++)$.

Requirements. The unification algorithm has the following requirements:

- The input of the unification algorithm is the function synthesized by the Petri net, as well as the input-output examples. As the algorithm inspects the definitions of the synthesized function, and of the applied components, it has to support enough language constructs to encode the component set of HOOGLÉ+, such as the case construct, algebraic data types, integers, or, at least naturals, ad-hoc polymorphism, function application, and λ -abstractions. Additionally, it has to support symbols both in place of constants and in place of functions.

e	$::=$	x	<i>(variable)</i>	
		$ $	s	<i>(symbolic variable)</i>
		$ $	$\lambda \bar{x}.e$	<i>(λ-abstraction)</i>
		$ $	$\mu \{\bar{e}\}$	<i>(polymorphic λ-abstraction)</i>
		$ $	$K \bar{e}$	<i>(data constructor)</i>
		$ $	case e of $\{\bar{a}\}$	<i>(case)</i>
		$ $	$e \bar{e}$	<i>(application)</i>
a	$::=$	$K \bar{x} \rightarrow e$	<i>(case alternative)</i>	

■ **Figure 4** Grammar of the supported language, λ_U .

- The unification algorithm does not need to support the occurrence of symbols, nor the application of functions or case constructs in the target expression, because this expression is always the output of an input-output example, simplifying the algorithm.

3.1 Syntax

Figure 4 presents the grammar of the language supported by the unification algorithm, which we call λ_U . Now, we discuss each construct, and present Example 4 and Example 5.

Variables play the same role as in λ -calculus, and are represented by x, x_i .

Symbolic variables denote unknown expressions and are represented by s, s_i . Symbolic variables can occur in place of functions or constants.

Abstractions have a sequence of variables (the parameters) and an expression that defines the abstraction. For example, the identity function can be encoded as $\lambda x . x$.

Polymorphic abstractions allow us to encode ad-hoc polymorphism, present in Haskell through typeclasses: each type provides an implementation for a given operation, which are chosen depending on the types of the arguments [33]. In λ_U , a polymorphic abstraction consists of a set of λ -abstractions, each one being a monomorphic variant.

Data constructors have a name and a sequence of arguments. As we do not have literals, this is the only way to represent data (natural numbers are represented using Peano numbers², such as $S (S Z)$), lists are represented using constructors *Cons* and *Nil*).

Case expressions have an expression (the *scrutinee*) and a sequence of alternatives. Each alternative has the name of a data constructor, a sequence of variables (one variable per constructor argument), and an expression. A case expression is of the form: **case** x **of** $\{Cons\ x_1\ x_2 \rightarrow Just\ x_1; Nil \rightarrow Nothing\}$. There are two differences with relation to case expressions in Haskell: we do not support guards and our alternatives only support variables after the data constructor (Haskell allows patterns such as **Just True**).

Applications have an expression and a sequence of expressions (the arguments). For example, $e\ e_1\ e_2$ denotes the application of e to the arguments e_1 and e_2 . Currying is not supported natively, and requires a specific encoding, as explained in Section 7.

► **Example 4.** Function map, applying a function f to a list l , can be encoded in λ_U as:

$$map = \lambda f\ l . \mathbf{case}\ l\ \mathbf{of}\ \{Nil \rightarrow Nil; Cons\ h\ t \rightarrow Cons\ (f\ h)\ (map\ f\ t)\}$$

² To be concise, we may write Arabic numbers as an abbreviation of the Peano representation.

► **Example 5.** We define the polymorphic abstraction eq , similar to the function `Data.Eq.(==)` from the Haskell standard library, which has two arguments, and returns *True* if and only if the arguments are equal. We define two versions: one for naturals and another for booleans: $eq = \mu \{eqN, eqB\}$.

$$\begin{aligned}
 eqN &= \lambda x_1 x_2 . \mathbf{case} x_1 \mathbf{of} \{Z \rightarrow \mathbf{case} x_2 \mathbf{of} \{Z \rightarrow True; S x_3 \rightarrow False\}; \\
 &\quad S x_3 \rightarrow \mathbf{case} x_2 \mathbf{of} \{Z \rightarrow False; S x_4 \rightarrow eq x_3 x_4\}\} \\
 eqB &= \lambda x_1 x_2 . \mathbf{case} x_1 \mathbf{of} \{False \rightarrow \mathbf{case} x_2 \mathbf{of} \{False \rightarrow True; True \rightarrow False\}; \\
 &\quad True \rightarrow \mathbf{case} x_2 \mathbf{of} \{False \rightarrow False; True \rightarrow True\}\}
 \end{aligned}$$

3.2 Inference rules

Now, we explain how the algorithm works by providing inference rules. First, we define the concept of map of substitutions, in Definition 6, and then, in Definition 7, we introduce a judgement that establishes the result of unifying two expressions. The inference rules can be used to derive judgements, and we provide different rules for different combinations of source and target expressions.

► **Definition 6.** A *map of substitutions* Σ is a mapping from symbolic variables to expressions (or applications of symbolic variables, when they occur in place of functions, as explained in Section 3.2.5). $\Sigma(s)$ denotes the value of s in map Σ , while $\Sigma[s \mapsto e]$ denotes the map Σ updated with the substitution of s with e . We write $[\]$ to denote the empty map, and $\text{dom}(\Sigma)$ to denote the set of symbolic variables that are substituted in Σ .

► **Definition 7.** The *judgement* $\Sigma \vdash e_{src} \equiv e_{tgt} \triangleright \Sigma'$, defined by the rules in Figure 5, denotes the relation where Σ' is the map of substitutions that results from unifying e_{src} and e_{tgt} , given the initial map Σ .

The rules shown in Figure 5 guarantee that for all resulting substitutions Σ' , we have $\text{eval}(\Sigma', e_{src}) = \text{eval}(\Sigma', e_{tgt})$, with $\Sigma \subseteq \Sigma'$. We validated this experimentally (a formal proof is left for future work). The evaluation function is defined in Algorithm 1. We need to use two maps Σ and Σ' because the first map represents the substitutions computed so far, and the second map represents the first one, eventually updated with new substitutions so that a substitution computed before is not discarded. So, we have $\Sigma \subseteq \Sigma'$. We will return to this topic when we address the rule for unifying data constructors, in Section 3.2.2. In the rest of this section, we present the syntax-directed rule system, which is summarized in Figure 5.

3.2.1 Unifying symbolic variables with expressions

We start with the simplest case: $\Sigma \vdash s \equiv e \triangleright \Sigma'$, in which the source expression is a symbolic variable s , and the target expression is any expression e . In this case, adding the substitution of s for e to the input map solves the problem, if s is not already assigned in the substitutions map computed so far, which is Σ (rule *SNAL*). When the symbolic variable s is already substituted in Σ , we try to unify e with $\Sigma(s)$ and add the new substitutions to the initial map (rule *SAL*).

► **Example 8.** We derive $[s_1 \mapsto s_2] \vdash s_1 \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False]$.

$$\frac{s_1 \in [s_1 \mapsto s_2] \quad \frac{s_2 \notin \text{dom}([s_1 \mapsto s_2]) \quad [s_1 \mapsto s_2] \vdash [s_1 \mapsto s_2](s_1) \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False] \quad \text{SNAL}}{[s_1 \mapsto s_2] \vdash [s_1 \mapsto s_2, s_2 \mapsto False] \quad \text{SAL}}}{[s_1 \mapsto s_2] \vdash s_1 \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False]}$$

s_1 cannot be assigned to *False* because it is already assigned to s_2 . So, we unified $\Sigma(s_1)$ with *False*.

Algorithm 1 Function eval.

$$\begin{aligned}
 \text{eval}(\Sigma, \lambda \bar{x} . b) &= \lambda \bar{x} . b & \text{eval}(\Sigma, \mu \{\overline{opts}\}) &= \mu \{\overline{opts}\} \\
 \\
 \text{eval}(\Sigma, s) &= \begin{cases} s & \text{if } s \notin \text{dom}(\Sigma) \\ \text{eval}(\Sigma, e) & \text{if } \Sigma(s) = e \end{cases} \\
 \\
 \text{eval}(\Sigma, K) &= K \\
 \text{eval}(\Sigma, K e_1 \dots e_k) &= K \text{ eval}(\Sigma, e_1) \dots \text{eval}(\Sigma, e_k) \\
 \\
 \text{eval}(\Sigma, f e_1 \dots e_k) &= \begin{cases} \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = \lambda x_1 \dots x_k . b \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ \\ \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = \mu \{\overline{opts}\}, \\ \lambda x_1 \dots x_k . b \in \overline{opts}, \text{ and} \\ \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) \neq \text{error} \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ \\ \text{eval}(\Sigma, \Sigma(s e'_1 \dots e'_k)) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = s \\ s e'_1 \dots e'_k \in \text{dom}(\Sigma) \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ \\ s e'_1 \dots e'_k & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = s \\ s e'_1 \dots e'_k \notin \text{dom}(\Sigma) \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \end{cases} \\
 \\
 \text{eval}(\Sigma, \text{case } scr \text{ of } \{\overline{alts}\}) &= \text{eval}(\Sigma, b\{e_1/x_1, \dots, e_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, scr) = K e_1 \dots e_k \\ \text{and } K x_1 \dots x_k \rightarrow b \in \overline{alts} \end{array} \\
 \text{eval}(\Sigma, e) &= \text{error} & \text{otherwise}
 \end{aligned}$$

We have said that the target expression should not contain symbols. However, internally, we need support for symbols in the target expression, due to the rule for case constructors, which is presented in Section 3.2.3. So, there are two more rules: *SNAR* and *SAR*, similar to the *SNAL* and *SAL*, with the difference that the symbolic variable is now the target expression.

3.2.2 Unifying data constructors

The rule *DC* is applied when both expressions are data constructors, with the same data constructor and the same number of arguments, and unifies each argument of the source data constructor with the corresponding argument in the target data constructor.

► **Example 9.** We derive $[] \vdash \text{Pair } s_1 s_2 \equiv \text{Pair } \text{True } \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]$.

$$\frac{\frac{s_1 \notin \text{dom}([])}{[] \vdash s_1 \equiv \text{True} \triangleright [s_1 \mapsto \text{True}]} \quad \frac{s_2 \notin \text{dom}([s_1 \mapsto \text{True}])}{[s_1 \mapsto \text{True}] \vdash s_2 \equiv \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]}{[] \vdash \text{Pair } s_1 s_2 \equiv \text{Pair } \text{True } \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]}$$

where the two top rules are *SNAL*, and the bottom rule is *DC*. Both expressions are data constructors with two arguments, and the constructor is the same, *Pair*. So, we unify s_1 with *True*, and the result is passed to the unification of s_2 with *Nil*. An important aspect is

$$\begin{array}{c}
\frac{s \notin \text{dom}(\Sigma)}{\Sigma \vdash s \equiv e \triangleright \Sigma[s \mapsto e]} \text{SNAL} \qquad \frac{e'_i = \text{eval}(\Sigma, e_i), i = 1, 2, \dots, k}{\Sigma \vdash s e_1 \dots e_k \equiv \text{dst} \triangleright \Sigma[s e'_1 \dots e'_k \mapsto \text{dst}]} \text{ASNA} \\
\\
\frac{s \notin \text{dom}(\Sigma)}{\Sigma \vdash e \equiv s \triangleright \Sigma[s \mapsto e]} \text{SNAR} \qquad \frac{e'_i = \text{eval}(\Sigma, e_i), i = 1, 2, \dots, k}{\Sigma \vdash s e_1 \dots e_k \equiv \text{dst} \triangleright \Sigma'} \text{ASA} \\
\\
\frac{s \in \text{dom}(\Sigma) \quad \Sigma \vdash \Sigma(s) \equiv e \triangleright \Sigma'}{\Sigma \vdash s \equiv e \triangleright \Sigma'} \text{SAL} \qquad \frac{s \in \text{dom}(\Sigma) \quad \Sigma \vdash e \equiv \Sigma(s) \triangleright \Sigma'}{\Sigma \vdash e \equiv s \triangleright \Sigma'} \text{SAR} \\
\\
\frac{\Sigma_0 \vdash e_1 \equiv f_1 \triangleright \Sigma_1 \quad \dots \quad \Sigma_{k-1} \vdash e_k \equiv f_k \triangleright \Sigma_k}{\Sigma_0 \vdash K e_1 \dots e_k \equiv K f_1 \dots f_k \triangleright \Sigma_k} \text{DC} \qquad \frac{\lambda x_1 \dots x_k . b \in \overline{\text{opts}}}{\Sigma \vdash (\lambda x_1 \dots x_k . b) e_1 \dots e_k \equiv e' \triangleright \Sigma'} \text{AP} \\
\\
\frac{\Sigma \vdash b\{s_1/x_1, \dots, s_k/x_k\} \equiv e' \triangleright \Sigma_0 \quad s_1, \dots, s_k \text{ fresh} \quad \Sigma_0 \vdash e_1 \equiv s_1 \triangleright \Sigma_1 \quad \dots \quad \Sigma_{k-1} \vdash e_k \equiv s_k \triangleright \Sigma_k}{\Sigma \vdash (\lambda x_1 \dots x_k . b) e_1 \dots e_k \equiv e' \triangleright \Sigma_k} \text{AL} \qquad \frac{K \bar{x} \rightarrow b \in \bar{a} \quad \Sigma \vdash \text{scr} \equiv K \bar{s} \triangleright \Sigma', \bar{s} \text{ fresh} \quad \Sigma' \vdash b\{\bar{s}/\bar{x}\} \equiv e \triangleright \Sigma''}{\Sigma \vdash \mathbf{case} \text{scr} \mathbf{of} \{\bar{a}\} \equiv e \triangleright \Sigma''} \text{C}
\end{array}$$

■ **Figure 5** Syntax-directed rule system that defines the judgement $\Sigma \vdash e_{src} \equiv e_{tgt} \triangleright \Sigma'$.

that the map that results from unifying e_1 with f_1 is passed as input to unify e_2 with f_2 , and so on, to preserve all substitutions and to avoid contradictions, which is illustrated in Example 10.

► **Example 10.** We want to unify *Pair s s* with *Pair True False*. Both expressions have the same data constructor and the same number of arguments, so let us apply the *DC* rule. First, we have to derive $[\] \vdash s \equiv \text{True} \triangleright [s \mapsto \text{True}]$, by using the rule *SNAL*. Second, we have to derive $[s \mapsto \text{True}] \vdash s \equiv \text{False} \triangleright \Sigma'$, for a certain Σ' . However, this is impossible because we are trying to unify s with *False*, and the unification of the first argument substituted s with *True*.

3.2.3 Unifying case expressions with expressions

When the source expression is a case construct, the rule *C* chooses a case alternative such that the *scrutinee* (*scr*) unifies with the selected data constructor. For example, if the case expression is **case** *expr* **of** $\{\text{Nothing} \rightarrow \text{False}, \text{Just } x \rightarrow x\}$, and supposing that we have chosen the first alternative, we have to ensure that *expr* unifies with *Nothing*. When the

data constructor has arguments (for instance, *Just* has one argument), we generate fresh symbols (which introduces the need to support symbolic variables in the target expression). This would be the case if we selected the second alternative: we would unify *expr* with *Just s*, where *s* is a fresh symbol. Finally, we unify the body of the alternative, *b*, with the target expression, substituting the variables of the alternative with the fresh symbols ($b\{\bar{s}/\bar{x}\}$). In rule *C* of Figure 5, \bar{s} denotes the sequence of fresh symbols, \bar{x} denotes the sequence of arguments of the data constructor, and $b\{\bar{s}/\bar{x}\}$ denotes the expression *b* in which each occurrence of $x_i \in \bar{x}$ is replaced with the corresponding symbol $s_i \in \bar{s}$.

► **Example 11.** We derive

$$[] \vdash \mathbf{case\ } s \mathbf{ of\ } \{Nothing \rightarrow False, Just\ x \rightarrow x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]$$

where s_1 is a fresh symbol, and *A* and *B* are derivation subtrees. The derivation applies *C*:

$$\frac{A \quad B \quad (Just\ x \rightarrow x) \in \{Nothing \rightarrow False, Just\ x \rightarrow x\}}{[] \vdash \mathbf{case\ } s \mathbf{ of\ } \{Nothing \rightarrow False, Just\ x \rightarrow x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]}$$

where *A* abbreviates

$$\frac{s \notin \text{dom}([]) }{[] \vdash s \equiv Just\ s_1 \triangleright [s \mapsto Just\ s_1], s_1 \text{ fresh}} \text{SNAL}$$

and *B* abbreviates

$$\frac{s_1 \notin \text{dom}([s \mapsto Just\ s_1]) }{[s \mapsto Just\ s_1] \vdash x\{s_1/x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]} \text{SNAL}$$

We choose the alternative *Just x → x*, unify *s* with *Just* instantiated with a fresh symbolic variable s_1 and then unify the body with the target expression. Note that the symbolic variable *s* is substituted with *Just s₁* and s_1 is substituted with *True*, so we have to evaluate *s*. We have $\text{eval}([s \mapsto Just\ s_1, s_1 \mapsto True], s) = Just\ True$, and substituting *s* with *Just True* solves the unification problem.

3.2.4 Unifying λ -abstraction applications with expressions

When the source expression is an application of a λ -abstraction, and the number of arguments is the same as the number of parameters, we apply *AL*, replacing the arguments of the application with fresh symbols in *b* and unifying this result, $b\{s_1/x_1, \dots, s_k/x_k\}$, with the target expression. The idea is to propagate the target expression to the arguments of the application: the unification will compute substitutions for the fresh symbols, and then we unify each argument with the corresponding fresh symbol.

► **Example 12.** We derive $[] \vdash (\lambda\ x\ y.\ x)\ s\ F \equiv T \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]$, where s_1 and s_2 are fresh symbols, and *A*, *B* and *C* are derivation subtrees.

$$\frac{A \quad B \quad C}{[] \vdash (\lambda\ x\ y.\ x)\ s\ F \equiv T \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]} \text{AL}$$

where *A*, *B*, and *C* abbreviate respectively

$$\frac{s_1 \notin \text{dom}([]) }{[] \vdash x\{s_1/x, s_2/y\} \equiv T \triangleright [s_1 \mapsto T], s_1, s_2 \text{ fresh}} \text{SNAL}$$

$$\frac{s \notin \text{dom}([s_1 \mapsto T])}{[s_1 \mapsto T] \vdash s \equiv s_1 \triangleright [s_1 \mapsto T, s \mapsto s_1]} \text{SNAL}$$

$$\frac{s_2 \notin \text{dom}([s_1 \mapsto T, s \mapsto s_1])}{[s_1 \mapsto T, s \mapsto s_1] \vdash F \equiv s_2 \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]} \text{SNAR}$$

The λ -abstraction has two parameters, and so we generate two symbols: s_1 and s_2 . The body of the λ -abstraction is x , and $x\{s_1/x, s_2/y\} = s_1$, which is unified with the target expression. Finally, we unify the arguments s and F with s_1 and s_2 .

3.2.5 Unifying applications of symbols with expressions

As stated in Section 3.1, a symbolic variable can occur in place of a function, however, instead of assigning it directly to expressions, which is out of the scope of this work, we assign applications of symbolic variables to expressions. This allows the generation of input-output examples for unknown functions and the detection of contradictions, which is relevant for HOOGL \star , as described in Section 4. For instance, unifying $\text{map } s \text{ (Cons 1 (Cons 2 Nil))}$ with $\text{Cons 2 (Cons 3 Nil)}$ generates examples for the unknown function s : assigns s 1 to 2 and s 2 to 3. On the other hand, it will be impossible to unify $\text{map } s \text{ (Cons 1 (Cons 1 Nil))}$ with $\text{(Cons 1 (Cons 2 Nil))}$, because s 1 will be assigned to 1, and then to 2, which generates a contradiction.

We have two rules *ASNA* and *ASA*, very similar to the rules for symbols, shown in Section 3.2.1. We need to store the arguments $\bar{e} = e_1 e_2 \dots e_k$ in a form as reduced as possible (using *eval*) because we need to compare each argument for equality³, to check if an application is already assigned as in the following example.

► **Example 13.** We derive $[s \ 0 \mapsto 2, s \ 1 \mapsto 3] \vdash s \ ((\lambda x . 0) \ 1) \equiv 2 \triangleright [s \ 0 \mapsto 2, s \ 1 \mapsto 3]$.

$$\frac{0 = \text{eval}(\Sigma, s \ ((\lambda x . 0) \ 1)) \quad \Sigma(s \ 0) = 2 \quad \Sigma \vdash 2 \equiv 2 \triangleright \Sigma}{\Sigma \vdash s \ ((\lambda x . 0) \ 1) \equiv 2 \triangleright \Sigma} \text{ASA}$$

where $\Sigma = [s \ 0 \mapsto 2, s \ 1 \mapsto 3]$. We omit the derivation of $\Sigma \vdash 2 \equiv 2 \triangleright \Sigma$. This example shows the importance of evaluating the arguments before updates and lookups to the map of substitutions. Indeed, the application $s \ ((\lambda x . 0) \ 1)$ is not substituted in Σ , but $(\lambda x . 0) \ 1$ evaluates to 0, and $s \ 0$ is already substituted in Σ .

3.2.6 Unifying applications of polymorphic abstractions with expressions

When the source expression is an application of a polymorphic abstraction, we apply *AP*, which chooses a λ -abstraction from $\overline{\text{opts}}$ and then unifies the application of this λ -abstraction to the provided arguments with the target expression. However, we cannot just choose any λ -abstraction. For instance, if the arguments are lists, we cannot choose a function that expects booleans, as the unification will fail.

► **Example 14.** We derive that $[] \vdash \text{eq } s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]$ where polymorphic abstraction *eq* was defined in Example 5.

$$\frac{\text{eq}B \in \overline{\text{opts}} \quad [] \vdash \text{eq}B \ s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]}{[] \vdash \text{eq } s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]}$$

³ Rule *ASA* can be applied only if the arguments are comparable for equality, which require the arguments not to be abstractions.

where rule AP is applied, $\overline{opts} = \{eqN, eqB\}$ and the derivation of the second hypothesis is omitted. When applying rule AP , we cannot choose the λ -abstraction eqN , because the derivation of $[\] \vdash eqN \ s \ False \equiv True \triangleright [s \mapsto s_1, s_1 \mapsto False, s_2 \mapsto False]$ gets stuck, as the case expressions of eqN do not contain alternatives matching $True$ and $False$.

3.3 Lazyness

The inference rules are lazy. For instance, consider the application $(\lambda x y . x) e_1 e_2$ being unified with $target$. We first unify the body with arguments replaced with symbols $x\{s_1/x, s_2/y\}$ with $target$, and then unify e_1 with s_1 and e_2 with s_2 . As y is not used in the abstraction, s_2 will not be assigned, so the unification of e_2 with s_2 will simply assign the symbol to the expression, and e_2 is not reduced. Thus, in principle, the algorithm supports computations with conceptually infinite structures [18]. For instance, it can unify `take sym1 (repeat sym2)` with `[3, 3]`, replacing `sym1` with 2 and `sym2` with 3.

3.4 Implementation

The unification algorithm was implemented in Haskell and performs a depth-first search: it tries to apply the rules and backtracks if it fails. The heart of symbolic execution and backtracking in the algorithm is the rule for the case construct: it attempts each alternative until it succeeds. To prevent the algorithm from running forever, we limit the depth of the DFS. The algorithm returns a substitution if found; `Mismatch`, if no substitution was found after trying all the possible rules (never reaching the limit for rules); or `DepthReached` if the maximum number of rules was reached in at least one path and no solution was found. Although the algorithm implements a search, it is very fast in practice. We conjecture that this is because the branching factor is reduced. Functions that work with lists typically have case expressions with no more than two cases (*Nil* and *Cons*), and the *Nil* case tends to be a base case (a leaf, in the search tree).

4 Extension to Hoogle \star : Hoogle \star

In this section, we describe the implementation of `HOOGLE \star` ⁴, by explaining the two independent steps presented in Section 1. We start with the modifications to introduce the wildcard component in Section 4.1, and, in Section 4.2, we address how the occurrences of the wildcard component are replaced.

4.1 The wildcard component

The first step is to add a component that matches all types so that it can occur where an integer, a list, a function, etc., is expected. `HOOGLE+` requires the name and the type of each component, so we provided the name *wildcard* associated with the type `a`, which matches all types. With this extension, `HOOGLE+` can synthesize functions containing the wildcard such as `\arg1 -> map wildcard arg1`, in which the wildcard occurs in place of a function (the first parameter of `GHC.List.map` is a function) or `\arg1 -> arg1 ++ wildcard`, in which the wildcard occurs in place of a list (both parameters of `GHC.List.++` are lists).

⁴ The `HOOGLE \star` repository is available at https://github.com/sr-lab/hoogle_plus.

The component set of HOOGLE+ contains four constants (\square , True, False, and Nothing), which are not required in HOOGLE* because the unification algorithm is able to generate them. So, these constants are not present in the default component set of HOOGLE*.

4.2 Replacing occurrences of the wildcard component

■ **Algorithm 2** Overview of HOOGLE*.

```

1: procedure HOOGLE*(components, query, N, examples)
2:   parsedExamples  $\leftarrow$  parse examples to  $\lambda_U$ 
3:   petri  $\leftarrow$  build the Petri Net with components  $\cup$  {("wildcard", "a")}
4:   for  $i = 1, \dots, N$  do
5:     synth  $\leftarrow$  synthesize a function for query using petri
6:     if synth has wildcards then
7:       completions  $\leftarrow$  COMPLETE(synth, parsedExamples, components, query)
8:       PRINT(comp) for  $comp \in completions$ 
9:     else if synth respects examples then
10:      PRINT(synth)

```

Algorithm 2 shows an overview of HOOGLE*. It takes four parameters: *components*, the component set; *query*, the query type; *N*, the number of functions the Petri net should synthesize; and *examples*, the input-output examples. HOOGLE* starts by parsing each input-output example to a pair $(x_{i1} \dots x_{ik}, y_i)$, containing a sequence of k inputs and output in λ_U , and the Petri net is built, considering the wildcard component. Then the Petri net synthesizes N functions. The function COMPLETE then tries to replace the wildcards (Algorithm 3).

4.2.1 The Complete function

COMPLETE takes four parameters: f , the function generated by the Petri net, expressed in the Hoogle+ grammar; *examples*, the examples expressed in λ_U ; *components*, the component set; and *type*, the query type. It has three main steps, presented afterward.

Step 1: Convert to λ_U . COMPLETE starts by converting the function generated by the Petri net to λ_U , where each wildcard is replaced with a fresh symbolic variable. The variable f' denotes the function in λ_U , and *symbols* denotes the array of generated symbols.

Step 2: Unify. The next step is to call the unification algorithm with the examples. For each example, the application of f' to the inputs is unified with the output, which requires as many calls to the unification algorithm as the number of examples. The result of all unifications, Σ , contains symbolic variables assigned to constants for the wildcards occurring in place of constants, and input-output examples for the wildcards occurring in place of a function⁵.

Note that Σ respects all the examples because the unification of each example uses the result of the unification of the previous example.

⁵ It is not guaranteed that each symbol is assigned in Σ , which can happen if its value does not impact the result of the unification. For instance, unifying *head* (*Cons* s_1 s_2) with *target* assigns s_1 to *target*, but does not assign s_2 . In this case, the incomplete function is rejected.

■ **Algorithm 3** Function COMPLETE.

```

1: procedure COMPLETE( $f, examples, components, type$ )
2:    $f', symbols \leftarrow$  convert  $f$  to  $\lambda_U$ , replace wildcards with fresh symbols, and return it
3:    $\Sigma \leftarrow []$  ▷ unify pairs of applications to outputs of examples
4:   for  $((x_1, \dots, x_k), y) \in examples$  do
5:      $\Sigma \leftarrow$  UNIFY( $\Sigma, f' x_1 \dots x_k, y$ )
6:     if  $\Sigma = \text{ERROR}$  then
7:       return Error
8:    $p \leftarrow$  the length of  $symbols$  ▷ i.e., the number of wildcards to replace
9:   for  $i = 1, \dots, p$  do
10:    if  $\exists \bar{e} : symbols[i] \bar{e} \in \text{dom}(\Sigma)$  then ▷ if  $symbols[i]$  denotes a function
11:       $wildcardType \leftarrow$  get the type of the wildcard  $i$  in function  $f$ 
12:       $\lambda \leftarrow$  SYNTH-LAMBDA( $wildcardType, f', examples, components, symbols[i]$ )
13:       $fill[i] \leftarrow \lambda$ 
14:    else if  $symbols[i] \in \text{dom}(\Sigma)$  then ▷ if  $symbols[i]$  denotes a constant
15:       $val \leftarrow$  eval( $\Sigma, symbols[i]$ )
16:       $fill[i] \leftarrow$  convert  $val$  to Haskell notation
17:    $res \leftarrow \{\}$ 
18:   for  $(e_1, \dots, e_p) \in fill[1] \times \dots \times fill[p]$  do ▷ all combinations of expressions
19:      $res \leftarrow res \cup f'\{e_1/symbols[1], \dots, e_p/symbols[p]\}$  ▷ replace wildcards
20:   return  $res$  in the grammar of HOOGLE+

```

Step 3: Replacing wildcards. After the unification, each symbolic variable is replaced with a constant or a λ -abstraction. Each iteration of the loop starting at line 9 replaces a symbol, assigning the replacement (or replacements, if there is more than one alternative) to the corresponding entry in the array $fill$. It starts with a lookup in Σ to determine the type of the expected term:

- If there is an application of the symbol, $symbols[i]$, in Σ , the corresponding wildcard must be replaced with a function. In this case, SYNTH-LAMBDA (Algorithm 4, Section 4.2.2), is called to synthesize λ -abstractions, with the data type of the function, $wildcardType$. Note that this function returns a set of λ -abstractions, because it may find more than one function that has the specified type and respects the examples.
- If the symbol s is itself assigned in Σ , the corresponding wildcard should be replaced with a constant. In this case, $\Sigma(s)$ is the expression that replaces the wildcard. However, this expression must be evaluated, because it may contain occurrences of other symbols, as explained in Section 3.2.3. Additionally, we replace Peano numbers with Arabic numbers and *Cons/Nil* lists with Haskell-syntax lists.

At the end of the loop that starts in line 9, $fill$ has one entry for each wildcard, each one containing a set of alternative replacements. Then, in the loop starting at line 18 we compute all the combinations of replacements for each wildcard, through a cartesian product, and, for each combination, we replace the wildcards in f' , and add the resulting expression to res .

► **Example 15.** Recall the first example of Section 1. The Petri net is able to synthesize $\backslash x1 \rightarrow x1 ++ wildcard$. We start by converting both the generated function and the input-output examples to λ_U . The function corresponds to $\lambda x_1 . (++) x1 s_1$, the input of the example becomes $Cons\ 1\ Nil$, and the output becomes $Cons\ 1\ (Cons\ 0\ Nil)$. The next step is to unify $(\lambda x_1 . (++) x1 s_1)(Cons\ 1\ Nil)$ with $Cons\ 1\ (Cons\ 0\ Nil)$, to compute Σ . As a result, s_1 is assigned itself in Σ to an expression, so the expected term is a constant. We have $\text{eval}(\Sigma, s_1) = Cons\ 0\ Nil$, that corresponds to $[0]$ in Haskell notation. Finally, the wildcard is replaced with $[0]$, which produces the function $\backslash x1 \rightarrow x1 ++ [0]$.

Note the importance of having the unification algorithm supporting symbols in place of functions, described in detail in Section 3.2.5. On the one hand, it allows replacing multiple function wildcards, because they are replaced one at a time. For instance, suppose that there are two function wildcards to replace. When replacing the first wildcard, we pick a λ -abstraction and then apply the unification algorithm to the original function, with the new λ -abstraction replaced. In this case, the second symbol remains, but the unification algorithm is able to validate if the first wildcard is replaced correctly. If the algorithm could not support symbolic functions, we would have to pick all the λ -abstractions at once and try each possible combination, which would lead to a combinatorial explosion. On the other hand, supporting symbolic functions allows for saving time, because the algorithm can detect if no λ -abstraction can fill a specific wildcard. In those cases, HOOGLE \star does not waste time calling SYNTH-LAMBDA. Section 4.2.2 discusses in detail SYNTH-LAMBDA.

4.2.2 The synthesizer for λ -abstractions

■ **Algorithm 4** Function SYNTH-LAMBDA.

```

1: procedure SYNTH-LAMBDA(type, originalFun, examples, comps, symbol)
2:   lamComps  $\leftarrow$  remove Data.ByteString and high-order components from comps
3:    $y_i \leftarrow$  the i-th parameter of originalFun for  $i = 1, \dots, n$ 
4:    $x_i \leftarrow$  the i-th parameter of the  $\lambda$ -abstraction, for  $i = 1, \dots, k$ 
5:   leafs  $\leftarrow$   $\{y_1, \dots, y_n, x_1, \dots, x_k, s\}$ , s fresh
6:   exprs  $\leftarrow$  SYNTH-EXPR(type, leafs, lamComps, 0)
7:   res  $\leftarrow$   $\{\}$ 
8:   for  $e \in$  exprs do
9:     lambda  $\leftarrow$   $\lambda x_1 \dots x_k . e$ 
10:    lambda'  $\leftarrow$  REPLACE-SYMBOLS(originalFun, lambda, examples, symbol)
11:    if lambda'  $\neq$  ERROR and the type of lambda' matches type then
12:      res  $\leftarrow$  res  $\cup$   $\{lambda'\}$ 
13:  return res  $\triangleright$  a list of many  $\lambda$ -abstractions with the specified type

```

The function SYNTH-LAMBDA computes a λ -abstraction that respects the input-output examples and has the specified type. We could call HOOGLE \star recursively, but we conjecture that it would lead to performance degradations, and a simpler, faster synthesizer is enough to synthesize λ -abstractions. On the one hand, it may be needed to synthesize λ -abstractions several times during a single HOOGLE+ query, but the paper that presented HOOGLE+ [14] has shown that for many problems, HOOGLE+ may take several seconds. Note that each query may require an unbounded number of synthesis of λ -abstractions because each one of the N incomplete functions may have an arbitrary number of wildcards in place of functions. On the other hand, from our experience with the Haskell programming language, λ -abstractions are simpler than other portions of code and use fewer components. So, we decided to build a faster, bespoke synthesizer that corresponds to the function SYNTH-LAMBDA.

Search space. The search space of SYNTH-LAMBDA is composed of applications of components from HOOGLE \star . To guarantee a faster synthesis, we exclude high-order functions, as well as the module **Data.ByteString** (that seems less common in λ -abstractions, from our experience as Haskell programmers), which leaves 54 popular components. The arguments of the applications can be the parameters of the λ -abstraction, parameters of the original function, symbols (to replace using the unification algorithm), and applications of components, with at

Algorithm 5 Function SYNTH-EXPR.

```

1: procedure SYNTH-EXPR(type, leafs, components, level)
2:   exprs  $\leftarrow$  {}
3:   for  $l \in \text{leafs}$  s.t. the type of  $l$  matches type do
4:     exprs  $\leftarrow$  exprs  $\cup$  { $l$ }
5:   if level < 2 then
6:     for comp  $\in$  components s.t. the return type of comp matches type do
7:       sign  $\leftarrow$  the signature of comp, with type variables replaced s.t. the return
       type matches type
8:       prms  $\leftarrow$  extract the types of the parameters from sign
9:       p  $\leftarrow$  the number of parameters of comp
10:      args[ $i$ ]  $\leftarrow$  SYNTH-EXPR(prms[ $i$ ], leafs, components, level + 1) for  $i = 1, \dots, p$ 
11:      for  $(e_1, \dots, e_p) \in \text{args}[1] \times \dots \times \text{args}[p]$  do  $\triangleright$  all combinations of expressions
12:        exprs  $\leftarrow$  exprs  $\cup$  {comp  $e_1 \dots e_p$ }
13:   return exprs  $\triangleright$  a list of many expressions with the specified type

```

Algorithm 6 Function REPLACE-SYMBOLS.

```

1: procedure REPLACE-SYMBOLS(originalFun, lambda, examples, symbol)
2:   originalFun  $\leftarrow$  originalFun{lambda/symbol}
3:    $\Sigma \leftarrow []$ 
4:   for  $((x_1, \dots, x_k), y) \in \text{examples}$  do
5:      $\Sigma \leftarrow \text{UNIFY}(\Sigma, \text{originalFun } x_1 \dots x_k, y)$ 
6:     if  $\Sigma = \text{ERROR}$  then
7:       return ERROR
8:   for each symbolic variable s in lambda do
9:     val  $\leftarrow$  eval( $\Sigma, s$ )
10:    converted  $\leftarrow$  convert val to Haskell notation
11:    lambda  $\leftarrow$  lambda{converted/s}
12:   return lambda

```

most two levels, for performance reasons (e.g., in $\backslash x \rightarrow f (g x) (h x)$, the arguments of g and h cannot be applications, only variables, and constants). For example, if the Petri net synthesizes $\backslash \text{arg1} \rightarrow \text{map } \textit{wildcard} \text{ arg1}$, for the query type $[\text{Int}] \rightarrow [\text{Int}]$, the wildcard may be replaced with $\backslash x1 \rightarrow x1 + 2$, $\backslash x1 \rightarrow x1 * (\text{length } \text{arg1})$, etc., depending on the input-output examples provided by the user.

Implementation. SYNTH-LAMBDA (Algorithm 4), takes five parameters: *type*, the signature of the function to synthesize; *originalFun*, the original function generated by the Petri net, but with wildcards replaced with fresh symbols; *examples*, the input-output examples of the original function; *comps*, the component set; and *symbol*, the symbolic variable that the new λ -abstraction should replace in *originalFun*. It follows a generate-and-test approach: SYNTH-EXPR does a type-guided enumeration of λ -abstractions, and then REPLACE-SYMBOLS tests the original function with each new λ -abstraction in place of the corresponding symbol, and replace symbols if any. Finally, we check that the λ -abstraction has the specified type, because SYNTH-EXPR does not perform a *full* type-checking, and only uses types to prune the search, thus can return ill-typed expressions. At this stage, type classes are ignored, and we leave for future work the analysis of their impact on the algorithm and its performance.

SYNTH-EXPR (Algorithm 5) takes four parameters: *type*, the type of the expression to synthesize; *leafs*, the set that contains the parameters of the new λ -abstraction (x_1, \dots, x_n), the parameters of the original function (y_1, \dots, y_k), and a fresh symbol; *components*, the component set of HOOGLE \star excluding high-order functions and the `Data.ByteString` module; and the depth of applications, *level*. If *level* is equal or greater than 2, SYNTH-EXPR only returns the *leafs* whose type matches *type*, to ensure that the maximum level is not exceeded. Otherwise, it returns also the application of components whose return type matches *type*, and the arguments are synthesized by calling SYNTH-EXPR recursively. Note that we may have to replace type variables, which we do in line 7. For instance, if the component has type `a -> a` and *type* is `Int`, we replace `a` with `Int`.

REPLACE-SYMBOLS (Algorithm 6) takes four parameters: the original Hoogle+ function, *originalFun*; the new λ -abstraction, *lambda*; the input-output examples, *examples*; and the symbolic variable that *lambda* replaces. It starts by replacing *symbol* with *lambda* in the original function. Then, the unification algorithm is used as in the COMPLETE function: for each example, we unify the application of *originalFun* to the inputs of the example with the expected output. Finally, each symbol that belongs to λ is replaced with a *lookup* in Σ , as COMPLETE does. Note that every symbol in the new λ -abstraction must be a constant because the component set excludes high-order functions.

Example 16 illustrates the synthesis of wildcards in place of functions.

► **Example 16.** Recall the second example of Section 1. The Petri net is able to synthesize `\x1 -> map wildcard x1`, which corresponds to $\lambda x_1 . map\ s\ x_1$ in λ_U . The example is converted to λ_U , and the unification is performed, assigning applications of *s* (for instance, *s* 1 to 1), which informs that the expected term is a λ -abstraction. Then, we call SYNTH-LAMBDA, where *type* is `Int -> Int`, *originalFun* is $\lambda x_1 . map\ s\ x_1$, *examples* is $\{([1, 2, 3]), [1, 4, 9]\}$ and *symbol* is *s*. The *leafs* are the parameter of the original function (x_1), the parameter of the new λ -abstraction (y_1), and a fresh symbol. One of the generated λ -abstractions can be `\y1 -> (GHC.Num.*) y1 y1`. Then REPLACE-SYMBOLS unifies `(\x1 -> map (\y1 -> y1 * y1) x1) [1, 2, 3]` with `[1, 4, 9]`, which succeeds. Finally, we check that the function has type `Int -> Int`.

5 Evaluation

In this chapter we empirically evaluate HOOGLE \star , answering the following research questions:

RQ1 Can HOOGLE \star solve all the problems that HOOGLE+ solves, without performance degradation?

RQ2 Can HOOGLE \star solve more problems than HOOGLE+?

5.1 Evaluation Design

Benchmarks. We use two different sets of benchmarks. To answer RQ1, we use the first set of 44 benchmarks (Table 1) from the original paper of HOOGLE+ [14], which consists of only query types. To answer RQ2, we use the second set of 26 benchmarks (Table 2), which consists of a query types and input-output examples and requires the generation of constants or λ -abstractions⁶.

⁶ Most of those benchmarks were adapted from questions in StackOverflow because we could not use them directly (e.g., if a question used floats, we changed, when possible, to integers). We systematically searched StackOverflow for Haskell problems, excluding the ones that did not require the generation of

■ **Table 1** First set of 44 benchmarks (from HOOGLE+ [14]).

#	Problem	Query
1	firstRight	[Either a b] -> Either a b
2	firstKey	[(a, b)] -> a
3	flatten	[[[a]]] -> [a]
4	repl-funcs	(a -> b) -> Int -> [a -> b]
5	containsEdge	Int -> (Int, Int) -> Bool
6	multiApp	(a -> b -> c) -> (a -> b) -> a -> c
7	appendN	Int -> [a] -> [a]
8	pipe	[a -> a] -> (a -> a)
9	intToBS	Int64 -> ByteString
10	cartProduct	[a] -> [b] -> [(a, b)]
11	applyNtimes	(a -> a) -> a -> Int -> a
12	firstMatch	[a] -> (a -> Bool) -> a
13	mbElem	Eq a => a -> [a] -> Maybe a
14	mapEither	(a -> Either b c) -> [a] -> ([b], [c])
15	hoogle01	(a -> b) -> [a] -> b
16	zipWithResult	(a -> b) -> [a] -> [(a, b)]
17	splitStr	String -> Char -> String
18	lookup	[(a, b)] -> a -> b
19	fromFirstMaybes	a -> [Maybe a] -> a
20	map	(a -> b) -> [a] -> [b]
21	maybe	Maybe a -> a -> Maybe a
22	rights	[Either a b] -> Either a [b]
23	mbAppFirst	b -> (a -> b) -> [a] -> b
24	mergeEither	Either a (Either a b) -> Either a b
25	test	Bool -> a -> Maybe a
26	multiAppPair	(a -> b, a -> c) -> a -> (b, c)
27	splitAtFirst	a -> [a] -> ([a], [a])
28	2partApp	(a->b)->(b->c)->[a]->[c]
29	areEq	Eq a => a -> a -> Maybe a
30	eitherTriple	Either a b -> Either a b -> Either a b
31	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b
32	head-rest	[a] -> (a, [a])
33	appBoth	(a -> b) -> (a -> c) -> a -> (b, c)
34	applyPair	(a -> b, a) -> b
35	resolveEither	Either a b -> (a->b) -> b
36	head-tail	[a] -> (a,a)
37	indexesOf	([(a,Int)] -> [(a,Int)]) -> [a] -> [Int] -> [Int]
38	app3	(a -> b -> c -> d) -> a -> c -> b -> d
39	both	(a -> b) -> (a, a) -> (b, b)
40	takeNdropM	Int -> Int -> [a] -> ([a], [a])
41	firstMaybe	[Maybe a] -> a
42	mbToEither	Maybe a -> b -> Either a b
43	pred-match	[a] -> (a -> Bool) -> Int
44	singleList	Int -> [Int]

Experiments. We compare HOOGLE* to the original HOOGLE+ as described below, giving each benchmark a timeout of 60 seconds, in the first set of benchmarks, and 90 seconds in the second set.

1. We run both HOOGLE+ (twice, with and without the constants `True`, `False`, `Nothing` and `[]`) and HOOGLE* on the 44 original benchmarks measuring the number of synthesized solutions, and the time taken to synthesize the first solution. For each benchmark, we

constants and λ -abstractions, and problems exercising the same capabilities. We also excluded problems that could not be solved by Hoogle+, for other reasons than the absence of constants and λ -abstractions. To diversify the components used we searched for questions using specific components. No problem that we have excluded would be solved by Hoogle+.

■ **Table 2** Second set of 26 benchmarks.

#	Problem	Query	Examples
1	mapAdd	[Int] -> [Int]	[[[1, 2, 3], [2, 3, 4]]]
2	mapSquare	[Int] -> [Int]	[[[1, 2]], [1, 4]]
3	appendConst	[Int] -> [Int]	[[[1, 2, 3], [1, 2, 3, 1000]]]
4	filterDiff	[Int] -> [Int]	[[[1, 2, 3], [1, 3]]]
5	getFirstOnes	[Int] -> [Int]	[[[1, 1, 0, 1, 2]], [1, 1]]
6	removeFirstOnes	[Int] -> [Int]	[[[1, 1, 0, 0, 1, 2]], [0, 0, 1, 2]]
7	listIntersect	[Int] -> [Int] -> [Int]	[[[0, 2, 4], [2, 4, 6]], [2, 4]]
8	indexConst	[a] -> a	[[[1, 2, 0, 3, 0, 1]], 3] [[2, 3, 4], True), [[2, 1, 4], False]]
9	allGreaterThan	[Int] -> Bool	[[[0, 0, 4, 4, 3]], [4, 3]]
10	dropConst	[Int] -> [Int]	[[[2, 0, 1, 3]], [2, 3]]
11	filterGreaterThan	[Int] -> [Int]	[[[1, 2], (2, 2), (3, 0)], [(2, 2)]]
12	filterPairs	[(Int, Int)] -> [(Int, Int)]	[[[1, 2, 1, 3, 4, 4]], [1, 1]]
13	filterEq	[Int] -> [Int]	[[1], [1, 1]]
14	replicateConst	Int -> [Int]	[[[1, 2, 3], [3, 4, 5]], [4, 6, 8]]
15	addElemsTwoLists	[Int] -> [Int] -> [Int]	[[[1, 3, 1]], 11]
16	sumSquares	[Int] -> Int	[[[1, 3, 2]], [1, 2]]
17	removeMax	[Int] -> [Int]	[[True, True], False), [[False, False], True), [[True, False], True), [[False, True], True)] [[[False, False], True), [[True, False], False), [[True], True)] [[[1, 3]], [[3, 1]]] [[[Nothing, Just 1]], False), [[Just 0, Just 0]], True), [[[Just 0, Nothing]], False]] [[[[True, True), (False, False)], False), ([[True, True), (False, False), (True, True)], False), [[True, True), (True, True)], True), ([[False, False]], False)]
18	nandPair	(Bool, Bool) -> Bool	[[[1, 2], 3]]
19	allEqBool	[Bool] -> Bool	[[[Nothing, Just 1]], False), [[Just 0, Just 0]], True), [[[True, True), (False, False)], False), ([[True, True), (False, False), (True, True)], False), [[True, True), (True, True)], True), ([[False, False]], False)]
20	mapReverse	[a] -> [[a]]	[[[1, 2], 3]]
21	allJust	[Maybe a] -> Bool	[[[1, 2], 3]] [[[Nothing, Just 1]], False), [[Just 0, Just 0]], True), [[[True, True), (False, False)], False), ([[True, True), (False, False), (True, True)], False), [[True, True), (True, True)], True), ([[False, False]], False)]
22	andListPairs	[(Bool, Bool)] -> Bool	[[[1, 2], 3]]
23	sumPairEntries	(Int, Int) -> Int	[[[1, 2], 3]]
24	filterPairsTyClass	(Eq a) => [(a, a)] -> [(a, a)]	[[[1, 2], (2, 2), (3, 0)], [(2, 2)]]
25	mapAddFloat	[Float] -> [Float]	[[[1, 2, 3], [1.5, 2.5, 3.5]]]
26	mapAddLarge	[Int] -> [Int]	[[[100, 200, 300]], [120, 220, 320]]]

ask both synthesizers to synthesize at most 10 solutions (parameter N in Algorithm 2). The goal is to understand the impact of the addition of the wildcard component, and the removal of the constant components.

2. We run both HOOGLER \star and the version of HOOGLER $+$ that supports examples[20], on the 26 benchmarks, measuring also the time consumed replacing the occurrences of the wildcard, and we ask both synthesizers to synthesize at most 35 solutions⁷.

Experimental setup. We run the experiments on a laptop with an AMD 5600G, running at 3.9 GHz, with 6 cores and 16 GB of RAM. All the versions of HOOGLER $+$ and HOOGLER \star use only two cores. The operating system is Ubuntu 22.04.2 LTS, the version of stack is 2.9.1, and

⁷ This is a higher value than in the previous step, because many of the N functions synthesized by the Petri net may be rejected due to the test of input-output examples, and with a lower value of N , both synthesizers ended the search before the timeout, without finding any solution.

the version of GHC is 8.4.4. The component set used by both HOOGLE+ versions contains the following modules: `Data.Bool`, `Data.ByteString.Builder`, `Data.ByteString.Lazy`, `Data.Either`, `Data.Eq`, `Data.Function`, `Data.Int`, `Data.Maybe`, `Data.Ord`, `Data.Tuple`, `GHC.Char`, `GHC.List`, and `Text.Show`. The total number of components is 297. The component set used by HOOGLE \star is the same, except that we removed the constants `Data.Bool.True`, `Data.Bool.False`, `Data.Maybe.Nothing` and `[]`, and we added the wildcard component. Those constants can be synthesized by the unification algorithm, so the component set does not need to contain them.

5.2 Results

In this section, we discuss the results of the experiments performed to answer the research questions stated at the beginning of the Section 5.

Results of the first set of benchmarks. The results of the first set of benchmarks are presented in Table 3, which shows that HOOGLE \star solves two problems that HOOGLE+ could not solve (benchmarks 6 and 9), and HOOGLE+ one problem that HOOGLE \star could not solve (benchmark 35). We have not found significant differences in the synthesized solutions, and in most benchmarks, there are solutions in common.

HOOGLE \star tends to be faster at synthesizing the first solution and synthesizes more solutions. On average, HOOGLE \star synthesizes 2.95 solutions per benchmark and takes 3.92 seconds to synthesize the first solution. HOOGLE+ synthesizes 2.41 solutions and takes 5.58 seconds. This can be explained by the removal of the four constants: on average, HOOGLE+ without constants takes 3.37 seconds to synthesize the first solution, so, in average, it is faster than HOOGLE+ with constants and HOOGLE \star . Indeed, the removal of the constants leads to a smaller component set, however, the reason for that is not the number of components that were removed, but the kind of components. Note that in the Petri net encoding, constants correspond to nullary transitions, i.e., transitions that do not need tokens to fire, so they can fire at any moment, leading to a higher branching factor. Thus, the removal of a constant should have more impact than the removal of a function.

Results of the second set of benchmarks. The results of the second set of benchmarks are shown in Table 4. HOOGLE \star solves 22 out of 26 benchmarks, whereas HOOGLE+ solves only 3 (benchmarks 50, 52, and 62), which are all solved by HOOGLE \star . This happens because most benchmarks require constants and λ -abstractions to be synthesized, which HOOGLE+ is not able to do. The authors of HOOGLE+ [14] argue that the absence of λ -abstractions does not impact the completeness of the method, because terms with λ -abstractions can be replaced with a term in point-free style, using the combinators S, K and I. However, this requires adding a nullary version of each component to the component set, which the authors consider infeasible, and in practice, only a small subset is added. The component sets of each version of HOOGLE+ used in our evaluation contain the combinators S, K, and I (module `Data.Function`), but it was not enough to solve the problems that require λ -abstractions.

In the benchmarks that require the synthesis of constants, the time spent completing the functions is always lower than 20% of the total time. However, in the benchmarks that require the synthesis of λ -abstractions, the time spent completing the wildcards can reach more than 50% of the total time, as happens in benchmarks 53, 59, and 60. HOOGLE \star cannot solve benchmark 51, whose solution is `\arg1 arg2 -> filter (\x1 -> x1 'elem' arg2) arg1`, because it requires the Petri net to synthesize the incomplete function `\arg1 arg2 -> filter wildcard arg1`, which does not use `arg2`, and the Petri net always synthesizes functions that use all the parameters. It also fails to solve benchmark 68, which is very similar to 56, with the difference that

■ **Table 3** Results of the first set of benchmarks. For both synthesizers, we show the number of solutions and the total time to synthesize the first solution, in seconds, or - if no solution was produced within the timeout of 60 seconds.

#	Benchmark	Hoogle+		Hoogle+, no consts.		Hoogle*	
		Time (s)	Sols.	Time (s)	Sols.	Time (s)	Sols.
1	firstRight	0.56	5	0.53	5	0.47	6
2	firstKey	2.32	4	1.41	2	1.21	2
3	flatten	1.09	9	6.10	9	0.93	9
4	repl-funcs	0.81	2	0.57	2	0.5	5
5	containsEdge	0.92	2	0.82	1	0.66	1
6	multiApp	-	0	-	0	1.73	2
7	appendN	0.6	10	0.54	10	0.48	10
8	pipe	6.99	4	7.48	2	7.41	2
9	intToBS	-	0	-	0	0.66	6
10	cartProduct	20.08	1	3.97	1	1.43	1
11	applyNtimes	4.88	2	5.06	3	5.15	6
12	firstMatch	0.97	5	1.03	5	1.23	6
13	mbElem	-	0	-	0	-	0
14	mapEither	2.28	1	7.42	1	3.07	1
15	hoogle01	0.68	4	0.66	4	0.61	9
16	zipWithResult	-	0	-	0	-	0
17	splitStr	0.58	5	0.54	4	0.5	9
18	lookup	-	0	-	0	-	0
19	fromFirstMaybes	2.17	3	2.03	5	1.37	2
20	map	0.78	5	0.81	5	0.54	7
21	maybe	0.68	1	0.69	1	0.51	1
22	rights	30.41	1	16.18	1	6.64	1
23	mbAppFirst	1.31	1	0.98	1	0.85	1
24	mergeEither	-	0	-	0	-	0
25	test	10.68	2	9.23	1	12.98	1
26	multiAppPair	-	0	-	0	-	0
27	splitAtFirst	1.01	5	0.79	1	0.72	3
28	2partApp	2.08	5	4.08	3	2.64	3
29	areEq	-	0	-	0	-	0
30	eitherTriple	-	0	-	0	-	0
31	mapMaybes	0.72	5	0.70	6	0.57	9
32	head-rest	3.79	3	8.67	3	2.36	3
33	appBoth	1.82	1	4.56	1	1.6	1
34	applyPair	1.68	1	1.98	1	3.82	1
35	resolveEither	42.49	1	-	0	-	0
36	head-tail	9.69	2	10.37	3	11.03	2
37	indexesOf	22.38	1	-	0	54.35	1
38	app3	0.59	1	0.86	1	0.52	7
39	both	-	0	-	0	-	0
40	takeNdropM	-	0	-	0	-	0
41	firstMaybe	1.71	6	1.41	8	1.31	4
42	mbToEither	-	0	-	0	-	0
43	pred-match	1.02	4	0.97	4	0.9	4
44	singleList	0.66	4	0.61	3	0.51	4
average		5.58	2.41	3.37	2.20	3.92	2.95

the query type has a typeclass constraint, instead of a monomorphic type. The solution is `\arg1 -> filter (\p -> fst p == snd p) arg1`, however, the Petri net does not synthesize the incomplete function `\arg1 -> filter wildcard arg1` within the timeout (whereas it synthesizes when the type is monomorphic). Benchmark 69 uses real numbers, that are not supported by the unification algorithm, and benchmark 70 contains input-output examples with large constants, leading the unification to reach the maximum depth before finding valid assignments. Comparing the solutions synthesized for the benchmarks that HOOGLE+ solves, the solutions of HOOGLE* are simpler, using fewer components. For instance, in benchmark 52, HOOGLE+ synthesizes `\arg0 -> last (init (init arg0))`, whereas HOOGLE* synthesizes `\arg1 -> (!!)` `arg1 3`.

■ **Table 4** Results of the second set of benchmarks. This table shows, for both synthesizers, the time elapsed to synthesize the first solution, in seconds, as well as the number of solutions, and the time spent replacing symbols until the first solution is completed. The timeout is 90 seconds.

#	Benchmark	Hoogle+ with examples		Hoogle \star		
		Time (s)	Sols.	Time (s)	Unify (s)	Sols.
45	mapAdd	-	0	8.07	0.66	11
46	mapSquare	-	0	7.99	0.61	11
47	appendConst	-	0	4.14	0.49	1
48	filterDiff	-	0	14.26	6.04	10
49	getFirstOnes	-	0	1.89	0.16	21
50	removeFirstOnes	2.55	1	1.49	0.18	22
51	listIntersect	-	0	-	-	0
52	indexConst	3.94	1	1.16	0.18	1
53	allGreaterThan	-	0	21.45	15.42	25
54	dropConst	-	0	1.81	0.18	9
55	filterGreaterThan	-	0	15.14	6.48	10
56	filterPairs	-	0	2.26	0.19	6
57	filterEq	-	0	24.77	11.5	12
58	replications	-	0	1.18	0.19	14
59	addElemsTwoLists	-	0	74.56	66.52	10
60	sumSquares	-	0	25.58	20.22	10
61	removeMax	-	0	14.27	5.96	10
62	nandPair	30.95	4	8.91	3.84	10
63	allEqBool	-	0	7.33	1.63	20
64	mapReverse	-	0	6.0	0.73	10
65	allJust	-	0	17.32	1.14	8
66	andListPairs	-	0	7.9	1.05	20
67	sumPairEntries	-	0	8.01	0.67	27
68	filterPairsTyClass	-	0	-	-	0
69	mapAddFloat	-	0	-	-	0
70	mapAddLarge	-	0	-	-	0
average		12.48	0.23	12.52	6.55	10.70

5.3 Answers to Research Questions

Given the results discussed in Section 5.2, we answer the two research questions as follows:

RQ1 The addition of the wildcard component did not lead to performance degradations.

Instead, the removal of constants resulted in performance improvements. From the original HOOGLE+ benchmarks, there is a single benchmark that HOOGLE+ solves and HOOGLE \star cannot solve within the timeout, but it solves two that HOOGLE+ does not solve.

RQ2 HOOGLE \star can solve many more new problems than HOOGLE+, especially when constants or λ -abstractions are required, which makes it able to solve new classes of problems.

We also found that in the cases that both synthesizers produce solutions, the solutions of HOOGLE \star are simpler, since they use fewer components.

6 Related Work

In this section, we compare our work to other research in program synthesis, unification, and symbolic execution. Most of the related work has been already presented in the HOOGLE+ original paper, so our focus is the work apart from this one.

6.1 Program Synthesis

Hoogle+ related work summary. The subjects most directly related to HOOGLE+ are type inhabitation and graph reachability. However, most of the related work on type inhabitation is based on classical proof search, such as AGDA [30], or produce solutions

that do not use all the arguments, such as DJINN [1]. In turn, the related work on graph reachability only supports functions with a single parameter, such as PROSPECTOR [25], or does not support polymorphism, such as SYPET [7]. When compared to other API search tools, such as HOOGLER [26], HOOGLER+ is able to synthesize applications of multiple components. Using statistical methods to improve the search, such as SLANG [34], the authors of HOOGLER+ conjecture that it is not effective in functional languages, due to the “high degree of compositionality”. There are also approaches to scalable proof search; however, the search space is restricted to names of parameters, functions, or fields [32], or does not support polymorphism, such as INSYNTH [15].

Synthesis from sketches. The idea of completing programs with holes, also known as sketches, has already been used in SKETCH [36] and ROSETTE [37], in which SAT/SMT solvers infer integer constants. However, in our work, a hole can be replaced with an expression of any algebraic type, or λ -abstractions. More recently, SMYTH [24], an evaluator-based program synthesizer, replaces holes with any expression, including case expressions, by performing a search guided by input-output examples. However, it inherits scalability issues from MYTH [31], the base of SMYTH, and the authors consider that HOOGLER+ “might also be incorporated into our approach in future work”. SCRIBER [27] extends the approach of SMYTH, with example propagation, and can solve more problems than SMYTH. However, we conjecture that the scalability issues remain, as the evaluation uses specific component sets for each test of at most 10 components [28], whereas the component set of HOOGLER+ has 291 components. GHC, a Haskell compiler, supports programs with missing expressions, suggesting valid fits [9]. However, constants are excluded (apart from already defined constants, such as `True`) and λ -abstractions. PROPR [10] uses this GHC feature to replace faulty sub-expressions on Haskell programs, and suggest constants, that, however, are limited to the ones contained in the program to repair.

Component-based synthesis. Apart from the related work of HOOGLER+, PETSYS [38] performs a top-down enumerative search, instead of using a Petri net encoding. Its evaluation shows that, at least with 130 components, its performance is comparable to HOOGLER+. However, it does not synthesize constants. HECTARE [23], a new synthesizer for Haskell that uses a new graph data structure to represent the search space, has shown to be faster than HOOGLER+, but it does not support constants nor λ -abstractions.

6.2 Unification and Symbolic Execution

***E*-Unification.** Unification is a process that, given two expressions, tries to replace the symbols in both expressions, such that the resulting expressions are syntactically equal [2]. In our case, the goal is to make two expressions equal after evaluation. This leads us to *E*-Unification, in which the equality of terms is established by a set of equations *E*: two terms *s*, *t* are equal if and only if $s \approx t \in E$ [35]. There are several approaches to solving *E*-Unification [8, 6], but we have not found any formulation that could be directly applied to our context. The same can be stated about Huet’s algorithm [19], which solves the unification problem for typed λ -calculus, from which Haskell’s Core language is an extension [21].

Symbolic execution. Symbolic execution tools explore multiple paths of a program to find counterexamples for a given property [3] and the unification problem discussed in this article can be reduced to finding a counterexample for $e_{src} \neq e_{tgt}$. G2 [16] and G2Q [17] are two symbolic execution tools for Haskell, but they do not support symbolic variables in place of

functions, which is required for HOOGLE \star . NEBULA [22], built on top of G2, supports symbols in place of functions, and treats applications of symbolic variables in a way similar to our approach: it replaces the application with a fresh symbol denoting the return value. However, it does not fully evaluate the arguments, so it may treat two equivalent calls as different calls. NEBULA can prove the equivalence of Haskell programs, by combining symbolic execution and coinduction, whereas our algorithm only finds assignments to symbolic variables. However, it is one order of magnitude faster, which makes the difference in the performance of HOOGLE \star . SCV [29] uses symbolic execution to validate software contracts in Racket programs and supports symbols in place of functions, but instead of assigning applications of functions to expressions, it generates candidate functions. However, while our algorithm supports infinite structures, SCV does not, since Racket is a strict language.

7 Limitations

Using polymorphic abstractions, instead of the standard way of implementing typeclasses, dictionary passing [33], simplifies the algorithm, especially when the term that determines the version of the polymorphic function is a symbol. But, as a drawback, this approach requires that each time a new monomorphic variant of a polymorphic operation is provided, the existing code must be edited (the new implementation must be added to each occurrence of the corresponding polymorphic abstraction). However, since the component set is not expected to change, this does not impact the usage of HOOGLE \star .

Currying is not supported for practical reasons. Whenever a curried application is translated to λ_U , we need to replace it with a λ -abstraction: supposing that f takes n arguments, we rewrite $f e_1 \dots e_m$ as $\lambda x_{m+1} \dots x_n . f e_1 \dots e_m x_{m+1} \dots x_n$ (with $m < n$). Also, for practical reasons, data constructors are not treated as the left side of abstractions, which means that a data constructor cannot be used as a function directly.

Data is represented by data constructors, which simplifies the algorithm, because all operations can be written in λ_U and each value can be built incrementally, by choosing a branch of each case expression. For instance, if we had to use the constant representation of integers, the implementation of operators such as integer comparison could not be expressed in λ_U , and expressions such as $n1 \leq n2$ would have to be processed by an SMT solver. A drawback of this representation is that real numbers are not supported (benchmark 69), and, in some specific cases, large integers may lead to an intractable search (if it is required to iterate the whole structure). Unifying a symbol with a large number, which is the case of benchmark 47, simply requires the application of *SNAL* or *SAL*; however, unifying $s + 1$ with N (similar to what happens in benchmark 70) requires a depth greater than N , which, in the context of complex problems with large branching factors, may become intractable.

Allow unused parameters. The Petri net does not synthesize functions that do not use all parameters, but the wildcards could be replaced with expressions using the remaining parameters. For instance, HOOGLE \star cannot synthesize `\xs n -> filter (\x -> x < n) xs`, because `\xs n -> filter wildcard xs` does not use the parameter `n`.

Queries with typeclass constraints are not solved, as in benchmark 68, because the Petri net becomes significantly slower when there are typeclass constraints (typeclass constraints are treated as extra arguments of the type query).

Completeness, normal form, and soundness. We do not have a definition of normal form for the terms of λ_U , nor proofs of completeness and of the guarantees of the inference rules, stated in Section 3.2.

8 Conclusion

In this work we developed a unification algorithm for a subset of the Haskell programming language and extended HOOGLE+, which can now synthesize constants and λ -abstractions.

Unification algorithm. To evaluate HOOGLE*, we have encoded 92 functions from the Haskell standard library⁸ in λ_U , and our algorithm successfully replaced the occurrences of the wildcard component for constants. But it has other applications; for instance, it can be used to compute inverses (by unifying $f\ s_1 \dots s_k$ with the output, $s_1 \dots s_k$ will be assigned to the values of the arguments), or for software testing and verification, finding counterexamples (for instance, if a function f is expected to always return a positive number, we can unify the application of f to symbolic variables with 0, to search for inputs that eventually make the function return 0).

Hoogle*. HOOGLE* can solve more problems than the original HOOGLE+ as it successfully synthesizes constants and λ -abstractions, without performance degradation. As explained in Section 6, existing synthesizers do not synthesize constants and λ -abstractions, or do not have the scalability that Petri nets give to HOOGLE+. HOOGLE* can generate constants and λ -abstractions while maintaining the scalability of Petri nets. Although we extended HOOGLE+, the contributions are not exclusive to this synthesizer, as they can be applied to other Petri-net synthesizers, such as SYPET. As a program synthesizer, it can impact science and industry in different ways: discovering new algorithms, allowing end users to build programs, improving teaching or assisting programmers [11, 5].

Future work. The main lines of future work are: supporting the representation of real numbers, as well as large integers; allowing the Petri net to synthesize functions that do not use all parameters; improving the synthesis of queries involving typeclass constraints; providing notions of completeness, normal forms, and a proof of the guarantees claimed in Section 3.2; and incorporating typeclasses in the type-checker of SYNTH-EXPR.

References

- 1 Lenart Augustsson. Djinn. URL: <https://github.com/augustss/djinn>.
- 2 Franz Baader. Unification theory. In Klaus U. Schulz, editor, *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings*, volume 572 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1990. doi:10.1007/3-540-55124-7_5.
- 3 Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018. doi:10.1145/3182657.
- 4 João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. Derivations with holes for concept-based program synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, pages 63–79, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563835.3567658.

⁸ From the modules `Data.Maybe`, `Data.Either`, `Data.Bool`, `GHC.List`, `Data.Ord` and `GHC.Num`.

- 5 Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017. doi:10.1098/rsta.2015.0403.
- 6 Daniel J. Dougherty and Patricia Johann. An improved general e-unification method. *J. Symb. Comput.*, 14(4):303–320, 1992. doi:10.1016/0747-7171(92)90010-2.
- 7 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017. doi:10.1145/3009837.3009851.
- 8 Jean H. Gallier and Wayne Snyder. A general complete E -unification procedure. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 1987. doi:10.1007/3-540-17220-3_19.
- 9 Matthías Páll Gissurarson. Suggesting valid hole fits for typed-holes (experience report). In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 179–185. ACM, 2018. doi:10.1145/3242744.3242760.
- 10 Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. PROPR: property-based automatic program repair. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1768–1780. ACM, 2022. doi:10.1145/3510003.3510620.
- 11 Sumit Gulwani. Dimensions in program synthesis. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24. ACM, 2010. doi:10.1145/1836089.1836091.
- 12 Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi:10.1561/2500000010.
- 13 Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022. doi:10.1145/3519939.3523450.
- 14 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020. doi:10.1145/3371080.
- 15 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013. doi:10.1145/2491956.2462192.
- 16 William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 411–424. ACM, 2019. doi:10.1145/3314221.3314618.
- 17 William T. Hallahan, Anton Xue, and Ruzica Piskac. G2Q: haskell constraint solving. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 44–57. ACM, 2019. doi:10.1145/3331545.3342590.
- 18 Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *ACM SIGPLAN Notices*, 27(5):1, 1992. doi:10.1145/130697.130698.

- 19 Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975. doi:10.1016/0304-3975(75)90011-0.
- 20 Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020. doi:10.1145/3428273.
- 21 SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.
- 22 John C. Kolesar, Ruzica Piskac, and William T. Hallahan. Checking equivalence in a non-strict language. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1469–1496, 2022. doi:10.1145/3563340.
- 23 James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. Searching entangled program spaces. *Proc. ACM Program. Lang.*, 6(ICFP):23–51, 2022. doi:10.1145/3547622.
- 24 Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020. doi:10.1145/3408991.
- 25 David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 48–61. ACM, 2005. doi:10.1145/1065010.1065018.
- 26 Neil Mitchel. Hoogle. URL: <https://hoogle.haskell.org/>.
- 27 Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. Program synthesis using example propagation. *CoRR*, abs/2210.13873, 2022. doi:10.48550/arXiv.2210.13873.
- 28 Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. Scrybe. <https://github.com/NiekM/scrybe>, 2022.
- 29 Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program.*, 27:e3, 2017. doi:10.1017/S0956796816000216.
- 30 Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijten, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.
- 31 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015. doi:10.1145/2737924.2738007.
- 32 Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286. ACM, 2012. doi:10.1145/2254064.2254098.
- 33 John Peterson and Mark P. Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236. ACM, 1993. doi:10.1145/155090.155112.
- 34 Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 419–428. ACM, 2014. doi:10.1145/2594291.2594321.
- 35 Jörg H. Siekmann. Unification theory. *J. Symb. Comput.*, 7(3/4):207–274, 1989. doi:10.1016/S0747-7171(89)80012-4.

- 36 Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013. doi:10.1007/s10009-012-0249-7.
- 37 Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013. doi:10.1145/2509578.2509586.
- 38 Darya Verzhbinsky and Daniel Wang. Petsy: Polymorphic enumerative type-guided synthesis. *POPL 2021 Student Research Competition*, 2021.