Program Synthesis Using Example Propagation

Niek Mulleners[1(✉)], Johan Jeuring[1], and Bastiaan Heeren[2]

[1] Utrecht University, Utrecht, The Netherlands
{n.mulleners,j.t.jeuring}@uu.nl
[2] Open University of The Netherlands, Heerlen, The Netherlands
bastiaan.heeren@ou.nl

Abstract. We present Scrybe, an example-based synthesis tool for a statically-typed functional programming language, which combines top-down deductive reasoning in the style of λ^2 with Smyth-style live bidirectional evaluation. During synthesis, example constraints are propagated through sketches to prune and guide the search. This enables Scrybe to make more effective use of functions provided in the context. To evaluate our tool, it is run on the combined, largely disjoint, benchmarks of λ^2 and Myth. Scrybe is able to synthesize most of the combined benchmark tasks.

Keywords: Program synthesis · Constraint propagation · Input-Output examples · Functional programming

1 Introduction

Type-and-example-driven program synthesis is the process of automatically generating a program that adheres to a type and a set of input-output examples. The general idea is that the space of type-correct programs is enumerated, evaluating each program against the input-output examples until a program is found that does not result in a counterexample. Recent work in this field has aimed to make the enumeration of programs more efficient, using various pruning techniques and other optimizations. Hoogle+ [5] and Hectare [7] explore efficient data structures to represent the search space. Smith and Albarghouthi [12] describe how synthesis procedures can be adapted to only consider programs in normal form. MagicHaskeller [6] and RESL [11] filter out programs that evaluate to the same result. Instead of only using input-output examples for the verification of generated programs, Myth [4,10], Smyth [8], and λ^2 [3] use input-output examples during pruning, by eagerly checking incomplete programs for counterexamples using constraint propagation.

Constraint Propagation. Top-down synthesis incrementally builds up a sketch, a program that may contain holes (denoted by ●). Holes may be annotated with constraints, e.g. type constraints. During synthesis, holes are filled

M. Hanus and D. Inclezan (Eds.): PADL 2023, LNCS 13880, pp. 20–36, 2023.
https://doi.org/10.1007/978-3-031-24841-2_2

with new sketches (possibly containing more holes) until no holes are left. For example, for type-directed synthesis, let us start from a single hole $\bullet_0$ annotated with a type constraint:

$$\bullet_0 :: \mathit{List\ Nat} \to \mathit{List\ Nat}$$

We may fill $\bullet_0$ using the function $\mathsf{map} :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b$, which applies a function to the elements of a list. This introduces a new hole $\bullet_1$, with a new type constraint:

$$\bullet_0 :: \mathit{List\ Nat} \to \mathit{List\ Nat} \quad \xrightarrow{\mathsf{map}} \quad \mathsf{map}\ (\bullet_1 :: \mathit{Nat} \to \mathit{Nat})$$

We say that the constraint on $\bullet_0$ is propagated through map to the hole $\bullet_1$. Note that type information is preserved: the type constraint on $\bullet_0$ is satisfied exactly if the type constraint on $\bullet_1$ is satisfied. We say that the hole filling $\boxed{\bullet_0 \mapsto \mathsf{map}\ \bullet_1}$ refines the sketch with regards to its type constraint.

A similar approach is possible for example constraints, which partially specify the behavior of a function using input-output pairs. For example, we may further specify hole $\bullet_0$, to try and synthesize a program that doubles each value in a list:[1]

$$\bullet_0 \vDash \{\, [0,1,2] \mapsto [0,2,4] \,\}$$

Now, when introducing map, we expect its argument $\bullet_1$ to have three example constraints, representing the doubling of a natural number:

$$\bullet_0 \vDash \{\, [0,1,2] \mapsto [0,2,4] \,\} \quad \xrightarrow{\mathsf{map}} \quad \mathsf{map}\ \left(\bullet_1 \vDash \left\{ \begin{array}{l} 0 \mapsto 0 \\ 1 \mapsto 2 \\ 2 \mapsto 4 \end{array} \right\}\right)$$

Similar to type constraints, we want example constraints to be correctly propagated through each hole filling, such that example information is preserved. Unlike with type constraints, which are propagated through hole fillings using type checking/inference, it is not obvious how to propagate example constraints through arbitrary functions. Typically, synthesizers define propagation of example constraints for a hand-picked set of functions and language constructs. Feser et al. [3] define example propagation for a set of combinators, including map and foldr, for their synthesizer λ^2. Limited to this set of combinators, λ^2 excels at composition but lacks in generality. Myth [4,10], and by extension Smyth [8], takes a more general approach, in exchange for compositionality, defining example propagation for basic language constructs, including constructors and pattern matches.

Presenting Scrybe. In this paper, we explore how the techniques of λ^2 and Smyth can be combined to create a general-purpose, compositional example-driven synthesizer, which we will call Scrybe. Figure 1 shows four different interactions with Scrybe, where the function dupli is synthesized with different sets

[1] In this example, as well as in the rest of this paper, we leave type constraints implicit.

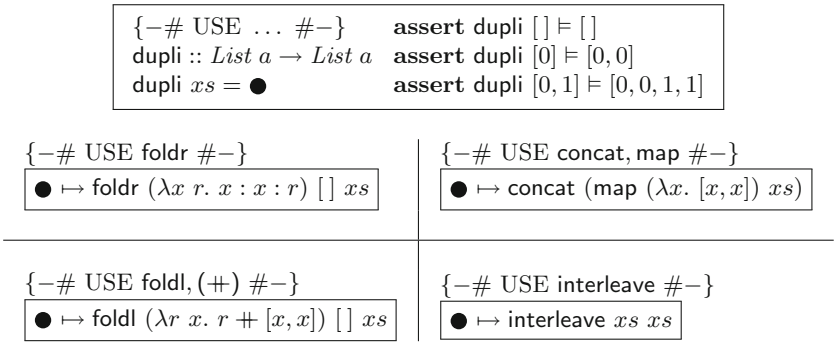

Fig. 1. (Top) A program sketch in Scrybe for synthesizing the function dupli, which duplicates each value in a list. (Bottom) Different synthesis results returned by Scrybe, for different sets of included functions.

of functions. Scrybe is able to propagate examples through all of the provided functions using live bidirectional evaluation as introduced by Lubin et al. [8] for their synthesizer Smyth, originally intended to support sketching [13,14]. By choosing the right set of functions (for example, the set of combinators used in λ^2), Scrybe is able to cover different synthesis domains. Additionally, allowing the programmer to choose this set of functions opens up a new way for the programmer to express their intent to the synthesizer, without going out of their way to provide an exact specification.

Main Contributions. The contributions of this paper are as follows:

- We give an overview of example propagation and how it can be used to perform program synthesis (Sect. 2).
- We show how live bidirectional evaluation as introduced by Lubin et al. [8] allows arbitrary sets of functions to be used as refinements during program synthesis (Sect. 3).
- We present Scrybe, an extension of Smyth [8], and evaluate it against existing benchmarks from different synthesis domains (Sect. 4).

2 Example Propagation

Example constraints give a specification of a function in terms of input-output pairs. For example, the following constraint represents the function mult that multiplies two numbers.

$$\left\{\begin{array}{ll} 0 & 1 \mapsto 0 \\ 1 & 1 \mapsto 1 \\ 2 & 3 \mapsto 6 \end{array}\right\}$$

The constraint consists of three input-output examples. Each arrow ($\mapsto$) maps the inputs on its left to the output on its right. A function can be checked against

an example constraint by evaluating it on the inputs and matching the results against the corresponding outputs. During synthesis, we want to check that generated expressions adhere to this constraint. For example, to synthesize mult, we may generate a range of expressions of type $Nat \to Nat \to Nat$ and then check each against the example constraint. The expression $\lambda x\ y.\ \mathsf{double}\ (\mathsf{plus}\ x\ y)$ will be discarded, as it maps the inputs to 2, 4, and 10, respectively. It would be more efficient, however, to recognize that any expression of the form $\lambda x\ y.\ \mathsf{double}\ e$, for some expression e, can be discarded, since there is no natural number whose double is 1.

To discard incorrect expressions as early as possible, we incrementally construct a sketch, where each hole (denoted by $\bullet$) is annotated with an example constraint. Each time a hole is filled, its example constraint is propagated to the new holes and checked for contradictions. Let us start from a single hole $\bullet_0$. We refine the sketch by eta-expansion, binding the inputs to the variables x and y.

$$\bullet_0 \vDash \left\{ \begin{array}{ll} 0 & 1 \mapsto 0 \\ 1 & 1 \mapsto 1 \\ 2 & 3 \mapsto 6 \end{array} \right\} \quad \xrightarrow{\textbf{eta-expand}} \quad \lambda x\ y.\ (\bullet_1 \vDash \left\{ \begin{array}{ll|l} x & y & \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 2 & 3 & 6 \end{array} \right\})$$

A new hole $\bullet_1$ is introduced, annotated with a constraint that captures the values of x and y. Example propagation through double should be able to recognize that the value 1 is not in the codomain of double, so that the hole filling $\boxed{\bullet_1 \mapsto \mathsf{double}\ \bullet_2}$ can be discarded.

2.1 Program Synthesis Using Example Propagation

Program synthesizers based on example propagation iteratively build a program by filling holes. At each iteration, the synthesizer may choose to fill a hole using either a *refinement* or a *guess*. A *refinement* is an expression for which example propagation is defined. For example, eta-expansion is a refinement, as shown in the previous example. To propagate an example constraint through a lambda abstraction, we simply bind the inputs to the newly introduced variables. A *guess* is an expression for which example propagation is *not* defined. The new holes introduced by a guess will not have example constraints. Once you start guessing, you have to keep guessing! Only when all holes introduced by guessing are filled can the expression be checked against the example constraint. In a sense, guessing comes down to brute-force enumerative search. Refinements are preferred over guesses, since they preserve constraint information. It is, however, not feasible to define example propagation for every possible expression.

2.2 Handcrafted Example Propagation

One way to implement example propagation is to use handcrafted propagation rules on a per function basis. Consider, for example, map, which maps a function over a list. Refinement using map replaces a constraint on a list with constraints

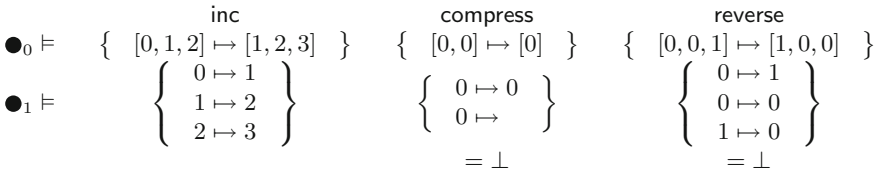

Fig. 2. Inputs and outputs for example propagation through the hole filling ●0 ↦ map ●1, for example constraints taken from the functions inc, compress, and reverse. The latter two cannot be implemented using map, which is reflected in the contradictory constraints: for compress there is a length mismatch, and for reverse the same input is mapped to different outputs.

on its elements, while checking that the input and output lists have equal length and that no value in the input list is mapped to different values in the output list. Figure 2 illustrates how specific input-output examples (taken from constraints on common list functions) are propagated through map. Note that synthesis works on many input-output examples at the same time. The function inc, which increments each number in a list by one, can be implemented using map. As such, example propagation succeeds, resulting in a constraint that represents incrementing a number by one. The function compress, which removes consecutive duplicates from a list, cannot be implemented using map, since the input and output lists can have different lengths. As such, example propagation fails, as seen in Fig. 2. The function reverse, which reverses a list, has input and output lists of the same length. It can, however, not be implemented using map, as map cannot take the positions of elements in a list into account. This is reflected in the example in Fig. 2, where the resulting constraint is inconsistent, mapping 0 to two different values.

For their tool λ^2, Feser et al. [3] provide such handcrafted propagation in terms of deduction rules for a set of combinators, including map, foldr, and filter. This allows λ^2 to efficiently synthesize complex functions in terms of these combinators. For example, λ^2 is able to synthesize a function computing the Cartesian product in terms of foldr, believed to be the first functional pearl [2]. Feser et al. [3] show that synthesis using example propagation is feasible, but λ^2 is not general purpose. Many synthesis problems require other recursion schemes or are defined over different types. Example propagation can be added for other functions in a similar fashion by adding new deduction rules, but this is very laborious work.

2.3 Example Propagation for Algebraic Datatypes

As shown by Osera and Zdancewic [10] in their synthesizer Myth, example constraints can be propagated through constructors and pattern matches, as long as they operate on algebraic datatypes. To illustrate this, we will use Peano-style natural numbers (we use the literals 0, 1, 2, etc. as syntactic sugar):

$$\textbf{data}\ \mathit{Nat} = \textsc{Zero} \mid \textsc{Succ}\ \mathit{Nat}$$

Constructors. To propagate a constraint through a constructor, we have to check that all possible outputs agree with this constructor. For example, the constraint $\{\textsc{Zero}\}$ can be refined by the constructor Zero. No constraints need to be propagated, since Zero has no arguments. In the next example, the constraint on $\bullet_0$ has multiple possible outputs, depending on the value of x.

$$\bullet_0 \vDash \left\{\begin{array}{c|l} x & \\ \ldots & \textsc{Succ}\ \textsc{Zero} \\ \ldots & \textsc{Succ}\ (\textsc{Succ}\ \textsc{Zero}) \end{array}\right\} \xrightarrow{\textsc{Succ}} \textsc{Succ}\ (\bullet_1 \vDash \left\{\begin{array}{c|l} x & \\ \ldots & \textsc{Zero} \\ \ldots & \textsc{Succ}\ \textsc{Zero} \end{array}\right\})$$

Since every possible output is a successor, the constraint can be propagated through Succ by removing one Succ constructor from each output, i.e. decreasing each output by one. The resulting constraint on $\bullet_1$ cannot be refined by a constructor, since the outputs do not all agree.

Pattern Matching. The elimination of constructors (i.e. pattern matching) is a bit more involved. For now, we will only consider non-nested pattern matches, where the scrutinee has no holes. Consider the following example, wherein the sketch $\mathsf{double} = \lambda n.\ \bullet_0$ is refined by propagating the constraint on $\bullet_0$ through a pattern match on the local variable n.

$$\bullet_0 \vDash \left\{\begin{array}{c|l} n & \\ 0 & 0 \\ 1 & 2 \\ 2 & 4 \end{array}\right\} \xrightarrow{\textbf{pattern match}} \begin{array}{l} \textbf{case}\ n\ \textbf{of} \\ \quad \textsc{Zero} \rightarrow \bullet_1 \vDash \{0\} \\ \quad \textsc{Succ}\ m \rightarrow \bullet_2 \vDash \left\{\begin{array}{c|l} m & \\ 0 & 2 \\ 1 & 4 \end{array}\right\} \end{array}$$

Pattern matching on n creates two branches, one for each constructor of *Nat*, with holes on the right-hand side. The constraint on $\bullet_0$ is propagated to each branch by splitting up the constraint based on the value of n. For brevity, we leave n out of the new constraints. The newly introduced variable m is exactly one less than n, i.e. one Succ constructor is stripped away.

Tying the Knot. By combining example propagation for algebraic datatypes with structural recursion, Myth is able to perform general-purpose, propagation-based synthesis. To illustrate this, we show how Myth synthesizes the function double, starting from the previous sketch. Hole $\bullet_1$ is easily refined with Zero. Hole $\bullet_2$ can be refined with Succ twice, since every output is at least 2:

$$\bullet_2 \vDash \left\{\begin{array}{c|l} m & \\ 0 & 2 \\ 1 & 4 \end{array}\right\} \xrightarrow{\textsc{Succ}} \ldots \xrightarrow{\textsc{Succ}} \textsc{Succ}\ (\textsc{Succ}\ (\bullet_3 \vDash \left\{\begin{array}{c|l} m & \\ 0 & 0 \\ 1 & 2 \end{array}\right\}))$$

At this point, to tie the knot, Myth should introduce the recursive call $\mathsf{double}\ m$. Note, however, that double is not yet implemented, so we cannot directly test the correctness of this guess. We can, however, use the original constraint (on $\bullet_0$) as a partial implementation of double. The example constraint on $\bullet_3$ is a subset of this original constraint, with m substituted for n. This implies that $\mathsf{double}\ m$ is a valid refinement. This property of example constraints, i.e. that the specification for recursive calls is a subset of the original constraint, is known as *trace completeness* [4], and is a prerequisite for synthesizing recursive functions in Myth.

2.4 Evaluation-Based Example Propagation

In their synthesizer Smyth, Lubin et al. [8] extend Myth with sketching, i.e. program synthesis starting from a sketch, a program containing holes. Lubin et al. define evaluation-based example propagation, in order to propagate global example constraints through the sketch, after which Myth-style synthesis takes over using the local example constraints.

Take for example the constraint $\{\, [0,1,2] \mapsto [0,2,4] \,\}$, which represents doubling each number in a list. The programmer may provide the sketch $\mathsf{map}\ \bullet$ as a starting point for the synthesis procedure. Unlike λ^2, Smyth does not provide a handcrafted rule for map. Instead, Smyth determines how examples are propagated through functions based on their implementation.

The crucial idea is that the sketch is first evaluated, essentially inlining all function calls[2] until only simple language constructs remain, each of which supports example propagation. Omar et al. [9] describe how to evaluate an expression containing holes using live evaluation. The sketch is applied to the provided input, after which map is inlined and evaluated as far as possible:

$$\mathsf{map}\ \bullet\ [0,1,2] \rightsquigarrow [\bullet\ 0,\ \bullet\ 1,\ \bullet\ 2]$$

At this point, the constraint can be propagated through the resulting expression. Lubin et al. [8] extend Myth-style example propagation to work for the primitives returned by live evaluation. The constraint is propagated through the result of live evaluation.

$$[\bullet\ 0,\ \bullet\ 1,\ \bullet\ 2] \vDash \{\, [0,2,4] \,\} \quad \longrightarrow^* \quad \begin{array}{l} [\ (\bullet \vDash \{\, 0 \mapsto 0 \,\})\ 0 \\ ,\ (\bullet \vDash \{\, 1 \mapsto 2 \,\})\ 1 \\ ,\ (\bullet \vDash \{\, 2 \mapsto 4 \,\})\ 2\ ] \end{array}$$

The constraints propagated to the different occurrences of $\bullet$ in the evaluated expression can then be collected and combined to compute a constraint for $\bullet$ in the input sketch.

$$\mathsf{map}\ (\bullet \vDash \left\{ \begin{array}{l} 0 \mapsto 0 \\ 1 \mapsto 2 \\ 2 \mapsto 4 \end{array} \right\})$$

[2] Note that function calls within the branches of a stuck pattern match are not inlined.

$$\frac{\varphi_0 \neq \bot}{\mathcal{P} \leftarrow \{\bullet_0\}}\ \textsc{init}$$

$$\frac{p \in \mathcal{P} \qquad \bullet_i \in \mathit{holes}(p) \qquad f \in \mathcal{D}_i \qquad \bullet_i \vDash \varphi_i \xrightarrow{f} f(\bullet_1 \vDash \varphi_1, \ldots, \bullet_n \vDash \varphi_n)}{\mathcal{P} \leftarrow \mathcal{P} \cup \{[\bullet_i \mapsto f(\bullet_1, \ldots, \bullet_n)]p\}}\ \textsc{expand}$$

$$\frac{p \in \mathcal{P} \qquad \mathit{holes}(p) = \emptyset}{p \text{ is a solution}}\ \textsc{final}$$

Fig. 3. Program synthesis using example propagation as a set of guarded rules that can be applied non-deterministically.

$$\begin{aligned} \bullet_0 &\mapsto \mathsf{map}\ (\lambda x.\ \bullet_1)\ \bullet_2 \\ \bullet_1 &\mapsto \textsc{Succ}\ \bullet_3 \\ \bullet_2 &\mapsto xs \\ \bullet_3 &\mapsto x \end{aligned}$$

Fig. 4. A set of hole fillings synthesizing an expression that increments each value in a list, starting from the sketch $\lambda xs.\ \bullet_0$.

This kind of example propagation based on evaluation is called live bidirectional evaluation. For a full description, see Lubin et al. [8]. Smyth uses live bidirectional evaluation to extend Myth with sketching. Note, however, that Smyth does not use live bidirectional evaluation to introduce refinements during synthesis.

3 Program Synthesis Using Example Propagation

Inspired by Smith and Albarghouthi [12], we give a high-level overview of program synthesis using example propagation as a set of guarded rules that can be applied non-deterministically, shown in Fig. 3. We keep track of a set of candidate programs $\mathcal{P}$, which is initialized by the rule init and then expanded by the rule expand until the rule final applies, returning a solution.

The rule init initializes $\mathcal{P}$ with a single hole $\bullet_0$, given that the initial constraint φ_0 is not contradictory. Each invocation of the rule expand non-deterministically picks a program p from $\mathcal{P}$ and fills one of its holes $\bullet_i$ using a refinement f from the domain $\mathcal{D}_i$, given that the constraint φ_i can be propagated through f. The new program is considered a valid candidate and added to $\mathcal{P}$. As an invariant, $\mathcal{P}$ only contains programs that do not conflict with the original constraint φ_0. A solution to the synthesis problem is therefore simply any program $p \in \mathcal{P}$ that contains no holes.

To implement a synthesizer according to these rules, we have to make the non-deterministic choices explicit: we have to decide in which order programs are expanded ($p \in \mathcal{P}$); which holes are selected for expansion ($\bullet_i \in \mathit{holes}(p)$); and how refinements are chosen ($f \in \mathcal{D}_i$). How each of these choices is made is described in Sects. 3.1 to 3.3 respectively.

3.1 Weighted Search

To decide which candidate expressions are selected for expansion, we define an order on expressions by assigning a weight to each of them. We implement $\mathcal{P}$

as a priority queue and select expressions for expansion in increasing order of their weight. The weight of an expression can be seen as a heuristic to guide the synthesis search. Intuitively, we want expressions that are closer to a possible solution to have a lower weight.

Firstly, we assign a weight to each *application* in an expression. This leads to a preference for smaller expression, which converge more quickly and are less prone to overfitting. Secondly, we assign a weight to each *lambda abstraction*, since each lambda abstraction introduces a variable, increasing the amount of possible holes fillings. Additionally, this disincentivizes an overuse of higher-order functions, as they are always eta-expanded. Lastly, we assign a weight to complex *hole constraints*, i.e. hole constraints that took very long to compute or that have a large number of disjunctions. We assume that smaller, simpler hole constraints are more easily resolved during synthesis and are thus closer to a solution.

3.2 Hole Order

Choosing in which order holes are filled during synthesis is a bit more involved. Consider, for example, the hole fillings in Fig. 4, synthesizing the expression $\lambda xs.\ \mathsf{map}\ (\lambda x.\ \textsc{Succ}\ x)\ xs$ starting from $\lambda xs.\ \bullet_0$. There are three different synthesis paths that lead to this result, depending on which holes are filled first. More specifically, $\bullet_2$ can be filled independently of $\bullet_1$ and $\bullet_3$, so it could be filled before, between, or after them. To avoid generating the same expression three times, we should fix the order in which holes are filled, so that there is a unique path to every possible expression.

Because our techniques rely heavily on evaluation, we let evaluation guide the hole order. After filling hole $\bullet_0$, we live evaluate.

$$\mathsf{map}\ (\lambda x.\ \bullet_1)\ \bullet_2 \rightsquigarrow \textbf{case}\ \bullet_2\ \textbf{of}\ \ldots$$

At this point, evaluation cannot continue, because we do not know which pattern $\bullet_2$ will be matched on. We say that $\bullet_2$ blocks the evaluation. By filling $\bullet_2$, the pattern match may resolve and generate new example constraints for $\bullet_1$. Conversely, filling $\bullet_1$ does not introduce any new constraints. Hence, we always fill blocking holes first. Blocking holes are easily computed by live evaluating the expression against the example constraints.

3.3 Generating Hole Fillings

Hole fillings depend on the local context and the type of a hole and may consist of constructors, pattern matches, variables, and function calls. To avoid synthesizing multiple equivalent expressions, we will only generate expressions in *β-normal*, *η-long* form. An expression is in *β-normal*, *η-long* form exactly if no η-expansions or β-reductions are possible. During synthesis, we guarantee *β-normal*, *η-long* form by greedily η-expanding newly introduced holes and always fully applying functions, variables, and constructors. Consider, for example, the

function map. To use map as a refinement, it is applied to two holes, the first of which is η-expanded:

$$\mathsf{map}\ (\lambda x.\ \bullet_0)\ \bullet_1$$

We add some syntactic restrictions to the generated expressions: pattern matches are non-nested and exhaustive; and we do not allow constructors in the recursive argument of recursion schemes such as foldr. These are similar to the restrictions on pattern matching and structural recursion in Myth [4,10] and Smyth [8]. Additionally, we disallow expressions that are not in normal form, somewhat similar to equivalence reduction as described by Smith and Albarghouthi [12]. Currently, our tool provides a handcrafted set of expressions that are not in normal form, which are prohibited during synthesis. Ideally, these sets of disallowed expressions would be taken from an existing data set (such as HLint[3]), or approximated using evaluation-based techniques such as QuickSpec [1].

3.4 Pruning the Program Space

For a program $p \in \mathcal{P}$ and a hole $\bullet_i \in \mathit{holes}(p)$, we generate a set of possible hole fillings based on the hole domain $\mathcal{D}_i$. For each of these hole fillings, we try to apply the Expand rule. To do so, we must ensure that the constraint φ_i on $\bullet_i$ can be propagated through the hole filling. We use evaluation-based example propagation in the style of Smyth to compute hole constraints for the newly introduced holes and check these for consistency. If example propagation fails, we do not extend $\mathcal{P}$, essentially pruning the program space.

Diverging Constraints. Unfortunately, example propagation is not feasible for all possible expressions. Consider, for instance, the function sum. If we try to propagate a constraint through sum $\bullet$, we first use live evaluation, resulting in the following partially evaluated result, with $\bullet$ in a scrutinized position:

$$\begin{array}{l}\textbf{case}\ \bullet\ \textbf{of}\\ \quad [\,] \quad\ \to 0\\ \quad x : xs \to \mathsf{plus}\ x\ (\mathsf{sum}\ xs)\end{array}$$

Unlike Myth, Smyth (and by extension Scrybe) allows examples to be propagated through pattern matches whose scrutinee may contain holes, considering each branch separately under the assumption that the scrutinee evaluates to the corresponding pattern. This introduces disjunctions in the example constraint. Propagating $\{\,\textsc{Zero}\,\}$ through the previous expression results in a constraint that cannot be finitely captured in our constraint language:

$$(\bullet \vDash \{\,[\,]\,\}) \vee (\bullet \vDash \{\,[\textsc{Zero}]\,\}) \vee (\bullet \vDash \{\,[\textsc{Zero}, \textsc{Zero}]\,\}) \vee \ldots$$

Without extending the constraint language it is impossible to compute such a constraint. Instead, we try to recognize that example propagation diverges, by

[3] https://github.com/ndmitchell/hlint.

setting a maximum to the amount of recursive calls allowed during example propagation. If the maximum recursion depth is reached, we cancel example propagation.

Since example propagation through $\mathsf{sum}\ \bullet$ always diverges, we could decide to disallow it as a hole filling. This is, however, too restrictive, as example propagation becomes feasible again when the length of the argument to sum is no longer unrestricted. Take, for example, the following constraint, representing counting the number of TRUEs in a list, and a possible series of hole fillings:

$$\bullet_0 \vDash \left\{ \begin{array}{c|c} xs & 0 \\ [\textsc{False}] & 1 \\ [\textsc{False}, \textsc{True}] & 2 \\ [\textsc{True}, \textsc{True}] & \end{array} \right\} \qquad \boxed{\begin{array}{l} \bullet_0 \mapsto \mathsf{sum}\ \bullet_1 \\ \bullet_1 \mapsto \mathsf{map}\ (\lambda x.\ \bullet_2)\ \bullet_3 \\ \bullet_3 \mapsto xs \end{array}}$$

Trying to propagate through $\boxed{\bullet_0 \mapsto \mathsf{sum}\ \bullet_1}$ diverges, since $\bullet_1$ could be a list of any length. At this point, we could decide to disregard this hole filling, but this would incorrectly prune away a valid solution. Instead, we allow synthesis to continue guessing hole fillings, until we get back on the right track: after guessing $\boxed{\bullet_1 \mapsto \mathsf{map}\ (\lambda x.\ \bullet_2)\ \bullet_3}$ and $\boxed{\bullet_3 \mapsto xs}$, the length of the argument to sum becomes restricted and example propagation no longer diverges:

$$\mathsf{sum}\ (\mathsf{map}\ (\lambda x.\ \bullet_2 \vDash \left\{ \begin{array}{c|c} x & \\ \textsc{False} & 0 \\ \textsc{True} & 1 \end{array} \right\})\ xs)$$

At this point, synthesis easily finishes by pattern matching on x. Note that, unlike λ^2, MYTH, and SMYTH, SCRYBE is able to interleave refinements and guesses.

Exponential Constraints. Even if example propagation does not diverge, it still might take too long to compute or generate a disproportionally large constraint, slowing down the synthesis procedure. Lubin et al. [8] compute the falsifiability of an example constraint by first transforming it to disjunctive normal form (DNF), which may lead to exponential growth of the constraint size. For example, consider the function or, defined as follows:

$$\begin{array}{l} \mathsf{or} = \lambda a\ b.\ \textbf{case}\ a\ \textbf{of} \\ \quad \textsc{False} \to b \\ \quad \textsc{True} \ \to \textsc{True} \end{array}$$

Propagating the example constraint $\{\, \textsc{True} \,\}$ through the expression $\mathsf{or}\ \bullet_0\ \bullet_1$ puts the hole $\bullet_0$ in a scrutinized position, resulting in the following constraint:

$$(\bullet_0 \vDash \{\, \textsc{False} \,\} \wedge \bullet_1 \vDash \{\, \textsc{True} \,\}) \vee \bullet_0 \vDash \{\, \textsc{True} \,\}$$

This constraint has size three (the number of hole occurrences). We can extend this example by mapping it over a list of length n as follows:

$$\mathsf{map}\ (\lambda x.\ \mathsf{or}\ \bullet_0\ \bullet_1)\ [0, 1, 2, \ldots] \vDash \{\, [\textsc{True}, \textsc{True}, \textsc{True}, \ldots] \,\}$$

Propagation generates a conjunction of n constraints that are all exactly the same apart from their local context, which differs in the value of x. This constraint, unsurprisingly, has size $3n$. Computing the disjunctive normal form of this constraint, however, results in a constraint of size of $2^n \times \frac{3}{2}n$, which is exponential.

In some cases, generating such a large constraint may cause example propagation to reach the maximum recursion depth. In other cases, example propagation succeeds, but returns such a large constraint that subsequent refinements will take too long to compute. In both cases, we treat it the same as diverging example propagation.

4 Evaluation

We have implemented our synthesis algorithm in the tool Scrybe[4]. To evaluate Scrybe, we combine the benchmarks of Myth [10] and λ^2 [3]. We ran the Scrybe benchmarks using the Criterion[5] library on an HP Elitebook 850 G6 with an Intel® Core™ i7-8565U CPU (1.80 GHz) and 32 GB of RAM.

This evaluation is not intended to compare our technique directly with previous techniques in terms of efficiency, but rather to get insight into the effectiveness of example propagation as a pruning technique, as well show the wide range of synthesis problems that Scrybe can handle.

For ease of readability, the benchmark suite is split up into a set of functions operating on lists (Table 1) and a set of functions operating on binary trees (Table 2). We have excluded functions operating on just booleans or natural numbers, as these are all trivial and synthesize in a few milliseconds. For consistency, and to avoid naming conflicts, the names of some of the benchmarks are changed to better reflect the corresponding functions in the Haskell prelude. To avoid confusion, each benchmark function comes with a short description.

Each row describes a single synthesis problem in terms of a function that needs to be synthesized. The first two columns give the name and a short description of this function. The third and fourth columns show, in milliseconds, the average time Scrybe takes to correctly synthesize the function with example propagation (*EP*) and without example propagation (*NoEP*), respectively. Some functions may fail to synthesize ($\perp$) within 10 s and some cannot straightforwardly be represented in our language (-). The last three columns show, for Myth, Smyth, and λ^2, respectively, whether the function synthesizes (✓), fails to synthesize (✗), or is not included in their benchmark (-).

Each benchmark uses the same context as the original benchmark (if the benchmark occurs in both Myth and λ^2, the context from the Myth benchmark is chosen). Additionally, benchmarks from Myth use a catamorphism in place of structural recursion, except for list_set_insert and tree_insert, which use a paramorphism instead.

[4] https://github.com/NiekM/scrybe.
[5] https://github.com/haskell/criterion.

Table 1. Benchmark for functions acting on lists. Each row describes a single benchmark task and the time it takes for each function to synthesize with example propagation (*EP*) and without (*NoEP*) respectively. Some tasks cannot be synthesized within 10 s (⊥) and others are omitted, since they cannot straightforwardly be translated to our language (-).

Name	Description	Runtime		Myth	Smyth	λ^2
		EP	*NoEP*			
list_add	Increment each value in a list by n	4.70	5.65	-	-	✓
list_append	Append two lists	13.35	41.36	✓	✓	✓
list_cartesian	The cartesian product	⊥	⊥	-	-	✓
list_compress	Remove consecutive duplicates from a list	⊥	⊥	✓	✗	-
list_copy_first	Replace each element in a list with the first	30.71	20.55	-	-	✓
list_copy_last	Replace each element in a list with the last	14.89	28.00	-	-	✓
list_delete_max	Remove the largest numbers from a list	16.05	31.95	-	-	✓
list_delete_maxs*	Remove the largest numbers from a list of lists	1875.43	⊥	-	-	✓
list_drop‡	All but the first n elements of a list	554.96	⊥	✓	✓	-
list_dupli	Duplicate each element in a list	6.43	7.60	✓	✓	✓
list_evens	Remove any odd numbers from a list	1.84	2.62	-	-	✓
list_even_parity	Whether a list has an odd number of Trues	42.72	201.73	✓	✗	-
list_filter	The elements in a list that satisfy p	80.25	162.70	✓	✓	-
list_flatten	Flatten a list of lists	7.54	8.12	✓	✓	✓
list_fold	A catamorphism over a list	9.48	6.43	✓	✓	-
list_head†	The first element of a list	1.40	2.20	✓	✓	-
list_inc	Increment each value in a list by one	5.38	24.36	✓	✓	-
list_incs	Increment each value in a list of lists by one	12.79	19.73	-	-	✓
list_index‡	Index a list starting at zero	79.01	2956.53	✓	✓	-
list_init†	All but the last element of a list	115.59	453.28	-	-	✓
list_last†	The last element of a list	9.63	15.14	✓	✓	✓
list_length	The number of elements in a list	1.33	2.16	✓	✓	✓
list_map	Map a function over a list	2.82	4.75	✓	✓	-
list_maximum	The largest number in a list	120.38	231.44	-	-	✓
list_member	Whether a number occurs in a list	212.45	1145.07	-	-	✓
list_nub	Remove duplicates from a list	450.56	9245.26	-	-	✓
list_reverse*	Reverse a list	131.04	574.82	✓	✓	-
list_set_insert	Insert an element in a set	⊥	⊥	✓	✓	-
list_shiftl	Shift all elements in a list to the left	723.97	525.29	-	-	✓
list_shiftr	Shift all elements in a list to the right	369.27	620.67	-	-	✓
list_snoc	Add an element to the end of a list	6.77	55.76	✓	✓	✓
list_sum	The sum of all numbers in a list	4.36	17.05	✓	✓	✓
list_sums	The sum of each nested list in a list of lists	69.71	677.12	-	-	✓
list_swap*	Swap the elements in a list pairwise	-	-	✓	✓	-
list_tail†	All but the first element of a list	1.65	5.43	✓	✓	-
list_take‡	The first n elements of a list	462.50	⊥	✓	✓	-
list_to_set	Sort a list, removing duplicates	37.06	41.18	✓	✓	-

The benchmarks list_head, list_tail, list_init, and list_last are all partial functions (marked †). We do not support partial functions, and therefore these functions are replaced by their total counterparts, by wrapping their return type in *Maybe*. For example, list_last is defined as follows, where the outlined hole filling is the result returned by Scrybe (input-output constraints are omitted for brevity):

Table 2. Benchmark for functions acting on binary trees. Each row describes a single benchmark task and the time it takes for each function to synthesize with example propagation (*EP*) and without (*NoEP*) respectively. Some tasks cannot be synthesized within 10 s (⊥) and others are omitted, since they cannot straightforwardly be translated to our language (-).

Name	Description	Runtime		Myth	Smyth	λ^2
		EP	*NoEP*			
tree_cons	Prepend an element to each list in a tree of lists	3.38	5.08	-	-	✓
tree_flatten	Flatten a tree of lists into a list	68.34	72.29	-	-	✓
tree_height	The height of a tree	7.01	34.89	-	-	✓
tree_inc	Increment each element in a tree by one	1.40	2.21	-	-	✓
tree_inorder	Inorder traversal of a tree	15.77	18.28	✓	✓	✓
tree_insert	Insert an element in a binary tree	⊥	⊥	✓	✗	-
tree_leaves	The number of leaves in a tree	14.77	45.67	✓	✓	✓
tree_level‡	The number of nodes at depth n	⊥	⊥	✓	✗	-
tree_map	Map a function over a tree	3.38	11.77	✓	✓	-
tree_maximum	The largest number in a tree	25.02	114.68	-	-	✓
tree_member	Whether a number occurs in a tree	907.81	⊥	-	-	✓
tree_postorder	Postorder traversal of a tree	19.50	44.79	✓	✗	-
tree_preorder	Preorder traversal of a tree	9.09	21.10	✓	✓	-
tree_search	Whether a number occurs in a tree of lists	1218.10	5253.07	-	-	✓
tree_select	All nodes in a tree that satisfy p	652.12	1268.76	-	-	✓
tree_size	The number of nodes in a tree	31.48	252.63	✓	✓	✓
tree_snoc	Append an element to each list in a tree of lists	57.13	2156.82	-	-	✓
tree_sum	The sum of all nodes in a tree	18.21	89.11	-	-	✓
tree_sum_lists	The sum of each list in a tree of lists	19.32	285.03	-	-	✓
tree_sum_trees	The sum of each tree in a list of trees	1000.26	⊥	-	-	✓

$$\{-\#\ \text{USE foldr}\ \#-\}$$
$$\text{list\_last} :: \mathit{List}\ a \to \mathit{Maybe}\ a$$
$$\text{list\_last}\ xs = \bullet$$
$$\boxed{\bullet \mapsto \text{foldr}\ (\lambda x\ r.\ \textbf{case}\ r\ \textbf{of}\ \begin{matrix}\textsc{Nothing} & \to \textsc{Just}\ x \\ \textsc{Just}\ y & \to r\end{matrix})\ \textsc{Nothing}\ xs}$$

The benchmarks list_drop, list_index, list_take, and tree_level (marked ‡) all recurse over two datatypes at the same time. As such, they cannot be implemented using foldr as it is used in Sect. 3.3. Instead, we provide a specialized version of foldr that takes an extra argument. For example, for list_take:

$$\{-\#\ \text{USE foldr} :: (a \to (c \to b) \to (c \to b)) \to (c \to b) \to \mathit{List}\ a \to c \to b\ \#-\}$$
$$\text{list\_take} :: \mathit{Nat} \to \mathit{List}\ a \to \mathit{List}\ a$$
$$\text{list\_take}\ n\ xs = \bullet$$
$$\boxed{\bullet \mapsto \text{foldr}\ (\lambda x\ r\ m.\ \textbf{case}\ m\ \textbf{of}\ \begin{matrix}\textsc{Zero} & \to [\,] \\ \textsc{Succ}\ o & \to (x : r\ o)\end{matrix})\ (\lambda\_.\ [\,])\ xs\ n}$$

A few functions (marked *) could not straightforwardly be translated to our approach:

- Function list_delete_maxs replaces the function list_delete_mins from λ^2. The original list_delete_mins requires a total function in scope that returns the minimum number in a list. This is not possible for natural numbers, as there is no obvious number to return for the empty list.
- Function list_swap uses nested pattern matching on the input list, which is not possible to mimic using a fold.
- Function list_reverse combines a set of benchmarks from Myth that synthesize reverse using different techniques, which are not easily translated to our language.

4.1 Results

Scrybe is able to synthesize most of the combined benchmarks of Myth and λ^2, with a median runtime of 19 ms. Furthermore, synthesis with example propagation is on average 5.1 times as fast as without example propagation, disregarding the benchmarks where synthesis without example propagation failed. λ^2 noticed a similar improvement (6 times as fast) for example propagation based on automated deduction, which indicates that example propagation using live-bidirectional evaluation is similar in effectiveness, while being more general.

Some functions benefit especially from example propagation, in particular those that can be decomposed into smaller problems. Take, for example, the function tree_snoc, which can be decomposed into list_snoc and tree_map. By preserving example constraints between these subproblems, Scrybe greatly reduces the search space.

$$
\begin{array}{l}
\{-\#\ \text{USE}\ \mathsf{mapTree}, \mathsf{foldr}, \ldots\ \#-\} \\
\mathsf{tree\_snoc} :: a \to \mathit{Tree}\ (\mathit{List}\ a) \to \mathit{Tree}\ (\mathit{List}\ a) \\
\mathsf{tree\_snoc}\ x\ t = \bullet \\
\boxed{\bullet \mapsto \mathsf{mapTree}\ (\lambda xs.\ \mathsf{foldr}\ (\lambda y\ r.\ y : r)\ [x]\ xs)\ t}
\end{array}
$$

On the other hand, for some functions, such as list_shiftl, synthesis is noticeably faster without example propagation, showing that the overhead of example propagation sometimes outweighs the benefits. This indicates that it might be helpful to use some heuristics to decide when example propagation is beneficial. A few functions that fail to synthesize, such as list_compress, do synthesize when a simple sketch is provided:

$$
\begin{array}{l}
\{-\#\ \text{USE}\ \mathsf{foldr}, (\equiv), \ldots\ \#-\} \\
\mathsf{compress} :: \mathit{List}\ \mathit{Nat} \to \mathit{List}\ \mathit{Nat} \\
\mathsf{compress}\ xs = \mathsf{foldr}\ (\lambda x\ r.\ \bullet_0)\ \bullet_1
\end{array}
\qquad
\boxed{
\begin{array}{l}
\bullet_0 \mapsto x : \mathbf{case}\ r\ \mathbf{of} \\
\quad [\,] \quad\ \to r \\
\quad y : ys \to \mathbf{if}\ x \equiv y\ \mathbf{then}\ ys\ \mathbf{else}\ r \\
\bullet_1 \mapsto [\,]
\end{array}
}
$$

Since our evaluation is not aimed at sketching, we still denote list_compress as failing ($\bot$) in the benchmark.

5 Conclusion

We presented an approach to program synthesis using example propagation that specializes in compositionality, by allowing arbitrary functions to be used as refinement steps. One of the key ideas is holding on to constraint information as long as possible, rather than resorting to brute-force, enumerative search. Our experiments show that we are able to synthesize a wide range of synthesis problems from different synthesis domains.

There are many avenues for future research. One direction we wish to explore is to replace the currently ad hoc constraint solver with a more general purpose SMT solver. Our hope is that this paves the way for the addition of primitive data types such as integers and floating point numbers.

Acknowledgements. We would like to thank Alex Gerdes, Koen Claessen, and the anonymous reviewers of HATRA 2022 and PADL 2023 for their supportive comments and constructive feedback.

References

1. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: guessing formal specifications using testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_3
2. Danvy, O., Spivey, M.: On Barron and Strachey's cartesian product function. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, pp. 41–46. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1291151.1291161
3. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Not. **50**(6), 229–239 (2015). https://doi.org/10.1145/2813885.2737977
4. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. SIGPLAN Not. **51**(1), 802–815 (2016). https://doi.org/10.1145/2914770.2837629
5. Guo, Z., et al.: Program synthesis by type-guided abstraction refinement. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371080
6. Katayama, S.: Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 199–210. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89197-0_21
7. Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N.: Searching entangled program spaces. Proc. ACM Program. Lang. **6**(ICFP) (2022). https://doi.org/10.1145/3547622
8. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proc. ACM Program. Lang. **4**(ICFP) (2020). https://doi.org/10.1145/3408991
9. Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290327

10. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. ACM SIGPLAN Not. **50**(6), 619–630 (2015). https://doi.org/10.1145/2813885.2738007
11. Peleg, H., Gabay, R., Itzhaky, S., Yahav, E.: Programming with a read-eval-synth loop. Proc. ACM Program. Lang. **4**(OOPSLA) (2020). https://doi.org/10.1145/3428227
12. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_2
13. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis, Berkeley (2008)
14. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_3