Synthesizing Data Structure Transformations from Input-Output Examples [*]

John K. Feser
Rice University, USA
feser@rice.edu

Swarat Chaudhuri
Rice University, USA
swarat@rice.edu

Isil Dillig
UT Austin, USA
isil@cs.utexas.edu

Abstract

We present a method for example-guided synthesis of functional programs over recursive data structures. Given a set of input-output examples, our method synthesizes a program in a functional language with higher-order combinators like `map` and `fold`. The synthesized program is guaranteed to be the simplest program in the language to fit the examples.

Our approach combines three technical ideas: inductive generalization, deduction, and enumerative search. First, we generalize the input-output examples into *hypotheses* about the structure of the target program. For each hypothesis, we use deduction to infer new input/output examples for the missing subexpressions. This leads to a new subproblem where the goal is to synthesize expressions within each hypothesis. Since not every hypothesis can be realized into a program that fits the examples, we use a combination of best-first enumeration and deduction to search for a hypothesis that meets our needs.

We have implemented our method in a tool called $\boldsymbol{\lambda^2}$, and we evaluate this tool on a large set of synthesis problems involving lists, trees, and nested data structures. The experiments demonstrate the scalability and broad scope of $\boldsymbol{\lambda^2}$. A highlight is the synthesis of a program believed to be the world's earliest functional pearl.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Program synthesis; programming by example; data transformations; search-based synthesis; automated deduction

1. Introduction

The last few years have seen a flurry of research on *automated program synthesis* [2, 10, 19, 30, 32]. This research area aims to radically simplify programming by allowing users to express their intent as nondeterministic, possibly-incomplete specifications. An algorithmic *program synthesizer* is then used to discover executable implementations of these specifications.

Inductive synthesis from examples is a particularly important form of program synthesis [1, 11, 16, 22, 27]. Here, a specification consists of a set of examples of the form $a \mapsto b$, where a is a sample input and b is the output of the desired program on input a. The synthesizer's task is to "learn" a program from these examples. This form of synthesis is especially appealing to end-users who need to perform programming tasks but lack the expertise or time to write traditional code. A prominent synthesizer of this sort is FlashFill, a feature of Excel 2013 that can generate spreadsheet table transformations from examples [11]. Example-guided synthesis has also been applied in many other domains, ranging from computer-aided design to game programming to text editing [22].

In this paper, we present a method for example-guided synthesis of programs that transform recursive data structures such as lists and trees. Such programs arise in many end-user programming scenarios. For instance, most professionals who work with numbers would sometimes want to programmatically manipulate lists or tables of numbers. Trees show up in end-user applications in many guises — concrete examples include family trees, HTML/XML documents, and directory trees in file systems. In fact, some end-user applications may demand data structures that are more general than lists or trees. For instance, a user interested in family trees may sometimes want to analyze trees for an unbounded list of families. In a specific family tree, a node for an individual may be equipped with a list of attributes for that person, and the user may want to transform these lists.

Transformations of recursive data structures are naturally expressed as functional programs. Therefore, our synthesis algorithm targets a functional programming language that permits higher-order functions, combinators like `map`, `fold` and `filter`, pattern-matching, recursion, and a flexible set of primitive operators and constants. The input to our algorithm is a set of input-output examples that define the behavior of the target program on certain small-sized instances. On such an input, the synthesis algorithm either times out or returns a program that fits the examples. In the latter case, the synthesized program is guaranteed to be the *least-cost* program in our language to fit the examples, according to a cost metric that assigns lower cost to simpler programs (for example, programs that are free of conditional branches).

A key advantage of the above optimality guarantee is that the synthesized program is not over-fitted to the examples. Specifically, given input-output examples $a_1 \mapsto b_1, \ldots, a_n \mapsto b_n$, our algorithm is unlikely to return a program that does an n-way case split

[*] This research was partially supported by NSF Awards #1162076 and #1156059, and DARPA MUSE award #FA8750-14-2-0270. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

PLDI'15 , June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

on its input and returns b_i whenever the input is equal to a_i. Instead, the algorithm tries to "generalize" the examples into a program that makes minimal use of conditional branches.

Although the synthesis algorithm's job is fundamentally difficult due to the combinatorial search space of possible programs, our algorithm addresses this challenge using a combination of three technical ideas: (1) *type-aware inductive generalization*, (2) the use of *deduction* to guide the solution of subproblems; and (3) *best-first enumerative search*.

Inductive generalization Rather than blindly searching for a target program, our method generalizes the user-provided examples into a set of *hypotheses* about this program. A hypothesis is either a concrete program or a "skeleton" that contains placeholders ("holes") for unknown programs. For instance, a hypothesis h for a program e might be of the form $\lambda x.\ \mathtt{map}\ f^*\ x$ where f^* stands for an unknown program. To synthesize a program from a hypothesis, we must substitute holes such as f^* by concrete programs.

Our algorithm generates hypotheses in a *type-aware* manner: We infer a type from the input-output examples and only generate hypotheses that can be concretized to programs of this type. For instance, our algorithm generates the hypothesis $\lambda x.\ \mathtt{map}\ f^*\ x$ only if all input-output examples are of type $\mathtt{list}[\tau] \to \mathtt{list}[\tau]$. This strategy often leads to significant pruning of the search space.

Deduction Once our algorithm generates a hypothesis h in the form of a program skeleton, we must solve one or more subproblems in order to synthesize the unknown functions that appear in h. For this purpose, our algorithm uses *automated deduction* to efficiently find a solution to the subproblems. In particular, we use deductive reasoning in two ways:

- ***Refutation.*** First, deduction is used to quickly *refute* certain hypotheses. For instance, consider an example of the form $[1, 1] \mapsto [2, 3]$ and the hypothesis $h \equiv \lambda x.\ \mathtt{map}\ f^*\ x$. Our deduction engine infers that this hypothesis h cannot be appropriate in this case, as no function maps the number 1 in the input list to two distinct numbers 2 and 3 in the output list.
- ***Example inference.*** Second, deduction is used to generate new examples that guide the search for missing functions. Consider again the hypothesis $\lambda x.\ \mathtt{map}\ f^*\ x$ and the example $[1, 2] \mapsto [3, 4]$. In this case, the deduction engine uses properties of the `map` combinator to infer two examples for f^*: $1 \mapsto 3$ and $2 \mapsto 4$. To find f^*, we invoke the synthesis algorithm on these examples.

Best-first enumerative search Generalization and deduction are complemented by *enumerative search*. For instance, suppose we have a hypothesis $\lambda x.\mathtt{map}\ f^*\ x$ where f^* is found to be of type $\mathtt{int} \to \mathtt{int}$. In this case, we use enumeration to find a substitution for f^* from within the space of unary arithmetic expressions. Also, in many cases, a set of examples can be generalized into multiple distinct hypotheses. Enumerative search is used to identify the hypotheses that can be realized into programs that fit the top-level goals.

Using the principle of *Occam's razor*, our search algorithm prioritizes simpler expressions and hypotheses. Specifically, the algorithm maintains a "frontier" of candidate expressions and hypotheses that need to be explored next and, at each point in the search, picks the *least-cost* item from this frontier. We show that this search strategy allows us synthesize the simplest program that fits the examples.

Results We have implemented our algorithm in a tool called $\boldsymbol{\lambda}^2$, and we empirically demonstrate that our technical insights can be combined into a scalable algorithm[1]. The benchmarks for our experiments include over 40 synthesis problems involving lists, trees, and nested data structures such as lists of lists and trees of lists. We show that $\boldsymbol{\lambda}^2$ can successfully solve these benchmarks, typically within a few seconds. The programs that $\boldsymbol{\lambda}^2$ synthesizes can be complex but also elegant. For example, $\boldsymbol{\lambda}^2$ is able to synthesize a program that is believed to be the world's earliest functional pearl [7].

Organization The paper is organized as follows. In Section 2, we present three motivating examples for our approach. After formalizing the problem in Section 3, we present our synthesis algorithm in Section 4. An evaluation is presented in Section 5, and related work is discussed in Section 6. We conclude with some discussion in Section 7.

2. Motivating Examples

In this section, we illustrate our method's capabilities using three examples.

2.1 Manipulating Lists of Lists

Consider a high-school teacher who wants to modify a collection of student scores. These scores are represented as a list $x = [l_1, \ldots, l_n]$ of lists, where each list l_i contains the i-th student's scores. The teacher's goal is to write a function `dropmins` that transforms x into a new list where each student's lowest score is dropped. For instance, we require that

```
dropmins [[1,3,5],[5, 3, 2]] = [3, 5], [5, 3].
```

Our $\boldsymbol{\lambda}^2$ system can synthesize the following implementation of this function in 114.65 seconds:

```
dropmins x = map f x
     where f y = filter g y
           where g z = foldl h False y
                where h t w = t || (w < z)
```

Here, `foldl`, `map`, and `filter` refer respectively to the standard left-fold, map, and filter operators [2].

Note the complex interplay between scoping and higher-order functions in this example. For example, the occurrence of `z` in line 4 is bound by the enclosing definition of `g`, and the occurrence of `y` in line 3 is bound by the enclosing definition of `f`.

The input-output examples used in the synthesis task are as follows.

```
[] ↦ []
[[1]] ↦ [[]]
[[1, 3, 5], [5, 3, 2]] ↦ [[3, 5], [5, 3]]
[[8, 4, 7, 2], [4, 6, 2, 9], [3, 4, 1, 0]] ↦
        [[8, 4, 7] [4, 6, 9], [3, 4, 1]]
```

2.2 Transforming Trees

Consider a user who wants to write a program to mine family trees. A node in such a tree represents a person; the node is annotated with a set of *attributes* including the year when the person was born. Given a family tree, the user's goal is to generate a list of persons in the family who were born between 1800 and 1820.

Suppose nodes of a family tree are labeled by pairs (v, by), where by is the birth year of a particular person and v represents the remaining attributes of that person. Given such a family tree, our synthesis task is to produce a program that generates a list of

[1] The name $\boldsymbol{\lambda}^2$ stands for "Lambda Learner".

[2] While $\boldsymbol{\lambda}^2$ generates its outputs in a λ-calculus, we use a Haskell-like notation for readability.

all labels (v, by) that appear in the tree and satisfy the predicate $\texttt{pr} \equiv \lambda by.\ 1800 \leq by \leq 1820$.

$\boldsymbol{\lambda^2}$ synthesizes the following program for this task in 15.97 seconds.

```
selectnodes x =  foldt f [] x
      where f z y =  foldl g (cons(y, concat z)) z
            where g t = filter pr t
```

Here, the operator `foldt` performs a fold over an unordered tree, `concat` takes in a list of lists l_i and returns the union of the l_i's, and `cons` is the standard list construction operator. Note that the predicate `pr` is *external* to the function. The user supplies the definition of this predicate along with the examples.

Let us represent trees using a bracket notation: <> represents the empty tree, and <*lab* S T> is a tree rooted at *lab* and containing child subtrees S and T. The examples needed to synthesize this program are as follows.

```
<> ↦ []
<(a,1760) <(b,1803)> <(c,1795)>> ↦ [(b,1803)]
<(a,1771) <(b,1815)> <(c,1818)>> ↦
                        [(b,1815), (c,1818)]
<(a,1812) <(b,1846)> <(c,1852)>> ↦ [(a,1812)]
```

Here, a, b, and c are *symbolic constants* that represent arbitrary values of v in labels (v, by). Note that there exist other definitions of `pr` — different from the one that we are using here — under which the synthesized program fits the examples. For instance, suppose we replaced our definition of `pr` by the predicate $\lambda by.\ by \bmod 3 = 0$ in the synthesized program. The resulting program would still satisfy the examples. The reason why $\boldsymbol{\lambda^2}$ does not output this alternative program is that it considers external predicates to be of especially low cost and prioritizes them during synthesis. This strategy formalizes the intuition that the user prefers the supplied predicate to appear in the synthesized program.

2.3 A Functional Pearl

We have used $\boldsymbol{\lambda^2}$ to synthesize a program originally invented by Barron and Strachey [3]. Danvy and Spivey call this program "arrestingly beautiful" and believe it to be "the world's first functional pearl" [7].

Consider a function `cprod` whose input is a list of lists, and whose output is the Cartesian product of these lists. Here is an input-output example for this function:

```
[[1,2,3], [4], [5,6]] ↦
 [[1,4,5],[1,4,6],[2,4,5],[2,4,6],[3,4,5],[3,4,6]].
```

Barron and Strachey implement this function as follows[3]:

```
cprod xss = foldr f [[]] xss
   where f xs yss = foldr g [] xs
      where g x zss = foldr h zss yss
         where h ys qss = cons(cons(x, ys), qss)
```

Here, `foldr` is the standard right-fold operation. For an article-length explanation of how this program works, see [7].

We used $\boldsymbol{\lambda^2}$ to synthesize an implementation of `cprod` from the following examples, in 83.83 seconds.

```
[] ↦ [[]]
[[]] ↦ []
[[], []] ↦ []
[[1, 2, 3] [5, 6]] ↦
[[1, 5], [1, 6], [2, 5], [2, 6], [3, 5], [3, 6]]
```

Remarkably, the program that $\boldsymbol{\lambda^2}$ synthesizes using these examples is precisely the one given by Barron and Strachey.

3. Problem Formulation

In this section, we formally state our synthesis problem.

Programming language Our method synthesizes programs in a λ-calculus with algebraic types and recursion. Let us consider *signatures* $\langle Op, Const, \mathcal{A} \rangle$, where Op is a set of *primitive operators*, $Const$ is a set of constants, and $\mathcal{A}$ is a set of equations that relate operators and constants. The syntax of programs e over such a signature is given by:

$$\begin{array}{rcl} e & ::= & x \mid c \mid \lambda x.e' \mid e_1\ e_2 \mid \mathbf{rec}\ f.(\lambda x.e') \mid (e_1, e_2) \mid \oplus\ e' \mid \\ & & \{l_1 : e_1, \ldots, l_k : e_k\} \mid e'.l \mid \langle l_i(e_i) \rangle \mid \\ & & \mathbf{match}\ e'\ \mathbf{with}\ \langle l_1(x_1) \Rightarrow e'_1, \ldots, l_k(x_k) \Rightarrow e'_k \rangle \end{array}$$

Here, x and f are variables, $\oplus \in Op$, and $c \in Const$. The syntax has standard meaning; in particular:

1. $\mathbf{rec}\ f.(\lambda x.e)$ is a recursive function f.
2. $e = \{l_1 : e_1, \ldots, l_k : e_k\}$ is a record whose field l_i has value e_i. We have $e.l_i = e_i$.
3. $e = \langle l_i(e_i) \rangle$ is a variant labeled l_i. The syntactic form "$\mathbf{match}\ e\ \mathbf{with}\ \ldots$" performs ML-style pattern-matching.

We assume the standard definition of free variables. A program is *closed* if it does not have any free variables.

As the operational semantics of the language is standard, we do not discuss it in detail. We simply assume that we have a relation $\rightsquigarrow$ such that $e_1 \rightsquigarrow e_2$ whenever e_1 evaluates to e_2 in one or more steps. Our programs are typed using an ML-style polymorphic type system. Since this system is standard, we skip a detailed description.

Our implementation of the language comes prepackaged with certain primitive operators, constants, and type definitions. Predefined types include (polymorphic) lists and trees, encoded as variants. Predefined operators include the standard arithmetic operators, operators for data structure construction and deconstruction, if-then-else, and a set of higher-order combinators like `map`, `foldl`, `foldr`, and `foldt` (see Figure 3 for a full list). In a particular synthesis task, we may augment this set with *external* operators, constants and types. For instance, in the synthesis of the `selectnodes` function in Section 2.2, `pr` is an external operator.

Cost model Each program e in the language has a *cost* $\mathcal{C}(e) \geq 0$. This cost is defined inductively. Specifically, we assume that each primitive operator $\oplus$ and constant c has a known, positive cost. Costs for more complex expressions satisfy constraints like the following (we skip some of the cases for brevity):

- $\mathcal{C}(\oplus\ e) > \mathcal{C}(\oplus) + \mathcal{C}(e)$
- $\mathcal{C}(\lambda x.e) > \mathcal{C}(e)$
- $\mathcal{C}(e_1\ e_2) > \mathcal{C}(e_1) + \mathcal{C}(e_2)$
- $\mathcal{C}(x) = 0$. Intuitively, we assign costs to the *definition*, rather than the *use*, of variables.

The synthesis problem Let an *input-output example* be a term $a_i \mapsto b_i$, where a_i and b_i are closed programs. The input to our synthesis problem is a set $\mathcal{E}_{in}$ of such examples. Our goal is to compute a *minimal-cost* closed program e that *satisfies* the examples — i.e., for each i, we have $(e\ a_i) \rightsquigarrow b_i$. In what follows, we refer to e as the *target program*.

Note that this problem formulation biases our synthesis procedure towards generating simpler programs. For example, since our implementation associates a higher cost with the `match` construct than the `fold` operators, our implementation favors fold-based implementations of list-transforming programs over those that use pattern-matching.

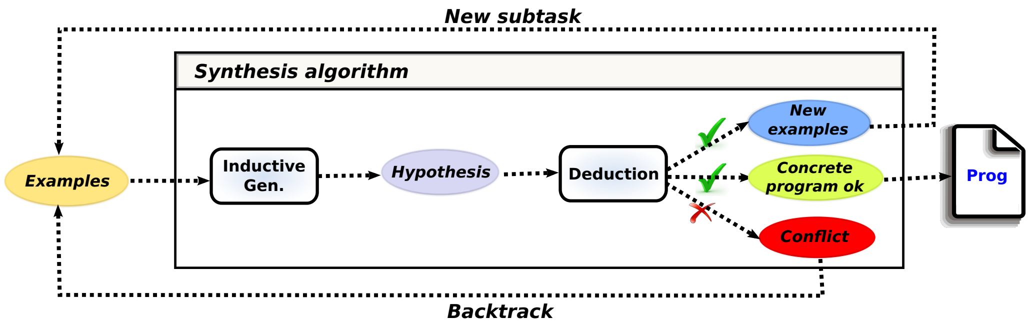

Figure 1. High-level overview of our synthesis algorithm

Hypotheses The concept of *hypotheses* about the structure of the target programs is key to our approach. Intuitively, a hypothesis is a program that may have placeholders for missing expressions. Formally, a *hypothesis* is a program that possibly has free variables. Free variables in a hypothesis are also known as *holes*, and a hypothesis with holes is said to be *open*. For instance, x and $\lambda x. f^* x$ are open hypotheses. In contrast, hypotheses that do not contain free variables are said to be *closed*. For example, $\lambda x. \texttt{map}\ (\lambda y.\ y + 1)\ x$ is a closed hypothesis.

A hypothesis h is typed under a *typing context* that assigns types to its holes. Given a type τ, we say that h is *consistent* with τ if there exists a typing context under which the type of h equals τ. This property can be decided using a standard type inference algorithm.

4. Synthesis Algorithm

This section describes our procedure for solving the synthesis problem from Section 3.

4.1 Algorithm Architecture

Our synthesis procedure performs an *enumerative search* that interleaves *inductive generalization* and *deductive reasoning*. Specifically, the procedure maintains a priority queue Q of *synthesis subtasks* of the form $(e, f, \mathcal{E})$, where e is a hypothesis, f is a hole in the hypothesis, and $\mathcal{E}$ is a set of examples. The interpretation of such a task is:

> Find a replacement e^* for the hole f such that e^* satisfies the examples $\mathcal{E}$, and the program $e[e^*/f]$ obtained by substituting f by e^* satisfies the top-level input-output examples $\mathcal{E}_{in}$.

The procedure iteratively processes subtasks in the queue Q (the *task pool*). Figure 1 gives an overview of our strategy for solving each subtask $(e, f, \mathcal{E})$. First, our algorithm performs inductive generalization over examples $\mathcal{E}$ to produce a lazy stream of hypotheses H about candidates e^* that can replace f. As mentioned earlier, these hypotheses are generated in a *type-aware* way, meaning that we rule out hypotheses that are inconsistent with the inferred type τ of examples $\mathcal{E}$.

Next, for each hypothesis h in H, our algorithm applies *deductive reasoning* to check for potential *conflicts*. If the hypothesis is closed, then a conflict arises if e does not satisfy the top-level (user-provided) input-output examples $\mathcal{E}_{in}$. In this case, the procedure simply picks a new synthesis subtask from the task pool Q. This corresponds to a form of *backtracking* in the overall algorithm.

If the hypothesis is open, a conflict indicates that the provided input-output examples violate a known axiom of a primitive operator used in the hypothesis (i.e., there is *no way* in which the hypothesis can be successfully completed). Upon conflict detection, our procedure again backtracks and considers a different inductive generalization.

```
SYNTHESIZE($\mathcal{E}_{in}$)
 1  $Q \leftarrow \{(f, f, \mathcal{E})\}$        // $f$ is a fresh variable name
 2  while $Q \neq \emptyset$
 3  do pick $(e, f, \mathcal{E})$ from $Q$ such that $e$ has minimal cost
 4      if $e$ is closed
 5        then if CONSISTENT($e, \mathcal{E}_{in}$)
 6                then return $e$
 7                else  continue
 8      $\tau \leftarrow$ TYPEINFER($\mathcal{E}$)
 9      $H \leftarrow$ INDUCTIVEGEN($\tau$)
10      for $h \in H$
11      do $e' \leftarrow e[h/f]$
12          if $e'$ is closed
13            then $Q \leftarrow Q \cup \{(e', \bot, \emptyset)\}$
14            else  for $f^* \in$ HOLES($e'$)
15                  do $\mathcal{E}^* \leftarrow$ DEDUCE($e', f^*, \mathcal{E}$)
16                      if $\mathcal{E}^* = \bot$  then break
17                      $Q \leftarrow Q \cup \{(e', f^*, \mathcal{E}^*)\}$
18  return  $\bot$
```

Figure 2. Synthesis procedure.

If hypothesis h is open and no conflicts are found, the procedure generates new subtasks for each hole f in h and uses deduction to learn new input-output examples. In more detail, each new subtask is of the form $(e', f^*, \mathcal{E}^*)$ where e' is a new hypothesis, f^* is a new hole to be synthesized, and $\mathcal{E}^*$ is the set of inferred input-output examples for f^*. This new subtask is now added to the task pool Q.

The procedure terminates once the search selects a subtask where the hypothesis is closed and which does not conflict with the top-level examples $\mathcal{E}_{in}$.

Figure 2 gives pseudocode for the overall synthesis algorithm. Here, Q is the task pool: a priority queue of tasks $(e, f, \mathcal{E})$ sorted according to the cost of hypothesis e. In each iteration of the outer loop, we pick a *minimum-cost* subtask $(e, f, \mathcal{E})$ from Q. Now, if e is a closed hypothesis, we use the routine CONSISTENT at line 5 to deductively check whether e satisfies examples $\mathcal{E}_{in}$. If this is the case, e must be a minimum-cost implementation consistent with $\mathcal{E}_{in}$; hence we return e as a solution to the synthesis problem. On the other hand, if e is not consistent with $\mathcal{E}_{in}$, we continue with a different candidate in Q.

Now, if e is an open hypothesis, we still need to synthesize the free variable f in e. For this purpose, we use the TYPEINFER procedure at line 8 to infer the type of f from examples $\mathcal{E}$ and then call the *hypothesis generator* INDUCTIVEGEN (see Section 4.2). This routine uses type-aware inductive generalization to compute a stream H of possible inductive generalizations for f. Now, we obtain a new hypothesis e' by replacing f with a hypothesis $h \in H$. If e' is closed, there are no new unknowns to synthesize; hence, we add a single subtask $(e', \bot, \emptyset)$ to the task pool Q.

Hypothesis	Definition of Combinator
$\lambda x.\, \texttt{map}\ f\ x$	$\texttt{map} :: (a \rightarrow b) \rightarrow \texttt{list}[a] \rightarrow \texttt{list}[b]$ $\texttt{map}\ f\ [\,] = [\,]$ $\texttt{map}\ f\ \texttt{cons}(x, y) = \texttt{cons}\ ((f\ x), (\texttt{map}\ y\ f))$
$\lambda x.\, \texttt{mapt}\ f\ x$	$\texttt{mapt} :: (a \rightarrow b) \rightarrow \texttt{tree}[a] \rightarrow \texttt{tree}[b]$ $\texttt{mapt}\ f\ \texttt{tree}() = \texttt{tree}()$ $\texttt{mapt}\ f\ \texttt{tree}(x, y) = \texttt{tree}(f\ x,\ \texttt{mapt}\ f\ y)$
$\lambda x.\, \texttt{filter}\ f\ x$	$\texttt{filter} :: (a \rightarrow \texttt{bool}) \rightarrow \texttt{list}[a] \rightarrow \texttt{list}[a]$ $\texttt{filter}\ f\ [\,] = [\,]$ $\texttt{filter}\ \texttt{cons}(x, y)\ f = \text{if } f\ x \text{ then } \texttt{cons}(x, (\texttt{filter}\ f\ y)) \text{ else } \texttt{filter}\ f\ y$
$\lambda x.\, \texttt{foldl}\ f\ e\ x$	$\texttt{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow \texttt{list}[a] \rightarrow b$ $\texttt{foldl}\ f\ e\ [\,] = e$ $\texttt{foldl}\ f\ e\ \texttt{cons}(x, y) = \texttt{foldl}\ f\ (f\ e\ x)\ y$
$\lambda x.\, \texttt{foldr}\ f\ e\ x$	$\texttt{foldr} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow \texttt{list}[a] \rightarrow b$ $\texttt{foldr}\ f\ e\ [\,] = e$ $\texttt{foldr}\ f\ e\ \texttt{cons}(x, y) = f\ (\texttt{foldr}\ f\ e\ y)\ x$
$\lambda x.\, \texttt{foldt}\ f\ e\ x$	$\texttt{foldt} :: (\texttt{list}[b] \rightarrow a \rightarrow b) \rightarrow b \rightarrow \texttt{tree}[a] \rightarrow b$ $\texttt{foldt}\ f\ e\ \texttt{tree}() = e$ $\texttt{foldt}\ f\ e\ \texttt{tree}(x, y) = f\ (\texttt{map}\ (\lambda z.\, \texttt{foldt}\ f\ e\ z)\ y)\ x$
$\lambda x.\, \texttt{recl}\ f\ e\ x$	$\texttt{recl} :: (a \rightarrow \texttt{list}[a] \rightarrow b) \rightarrow b \rightarrow \texttt{list}[a] \rightarrow b$ $\texttt{recl}\ f\ e\ [\,] =\ e$ $\texttt{recl}\ f\ e\ \texttt{cons}(x, y) =\ f\ x\ y$

Figure 3. Family of combinators from which we draw our open hypotheses.

On the other hand, if e' is an open hypothesis, we generate new subtasks for synthesizing each hole in e'. Here, the procedure DEDUCE (Section 4.3) infers new examples for a specified hole f^* in e'. If a conflict is detected (i.e., e' is not consistent with a known axiom), then DEDUCE returns $\perp$, which causes backtracking from the current hypothesis. If no conflicts are detected, we add new subtasks of the form $(e', f^*, \mathcal{E}^*)$ to Q.

Example 1 (List reverse)**.** Suppose we want to synthesize a program that reverses a list, starting from the following input-output examples.

```
[] ↦ []
[0] ↦ [0]
[0 1] ↦ [1 0]
[0 1 2] ↦ [2 1 0]
```

From this specification, our algorithm infers that the type of the target program is $\texttt{list}[\texttt{int}] \rightarrow \texttt{list}[\texttt{int}]$. Now it begins a search for this program, starting with the "empty" hypothesis $h = \lambda x.\ f$, where f is a hole of type $\texttt{list}[\texttt{int}]$. We begin filling in f with type-compatible expressions like $[\,]$, x, $\texttt{cons}(1, [\,])$, etc. Expressions are enumerated in order of increasing cost. Eventually, we start using higher-order combinators such as `map` and `foldl` to fill in f.

We use deductive reasoning in addition to the known type of f to determine whether to allow f to be filled by a particular combinator. For example, we rule out `filter` as a possibility because of the element reordering in the examples, but map, foldl and foldr remain possible.

When we fill f with the left-fold combinator `foldl`, we have a new hypothesis $\lambda x.(\texttt{foldl}\ h'\ [\,]\ x)$, where $h' = \lambda a.\lambda e.f'$ for a new hole f'. Higher-order combinators differ from operators like `cons` in that their arguments have certain restrictions. In this case, the last argument to `foldl` must be a name; it is not allowed to be an expression. The first argument must be a lambda-term as shown. Here, the second argument is inferred to be $[\,]$ because of the first example. In problems where the base case input-output example is not provided, the second argument to `foldl` will be a hole as well.

The new hole f' is of type $\texttt{list}[\texttt{int}]$. Now we use deduction to infer a set of input-output examples `foldl` (see Section 4.3 for the inference rule used for this). The inferred examples are:

```
([ ], 0) ↦ [0]
([0], 1) ↦ [1 0]
([1 0], 2) ↦ [2 1 0].
```

A solution is found for f' using the same process. We test the solution by filling in f' with the candidate and determining if the result fits the user-provided examples. Note that the inferred examples for f' are not used in this process; their purpose is only to *refute* the use of a combinator.

In the present scenario, we would not end up trying any candidates using higher-order combinators because f' can be filled by the low-cost expression $\texttt{cons}(e, a)$. The solution that our algorithm reports to the user is:

```
reverse x = foldl g [] x
     where g e a = cons (e, a).
```

4.2 Hypothesis Generation

This section describes the hypothesis generation procedure INDUCTIVEGEN used in our synthesis algorithm. The input of the procedure is a type τ inferred from a set of examples. The output is a stream of hypotheses that match this type.

The procedure treats the generation of open and closed hypotheses differently. An open hypothesis generated here has a particular form: it is the application of a higher-order combinator to a set of (known or unknown) arguments. Since such hypotheses are used to encapsulate common data structure transformation patterns, we only introduce them when the input examples correspond to recursive data structures. In other scenarios, for example when $\tau = \tau_1 \rightarrow \tau_2$ for primitive types τ_1 and τ_2, INDUCTIVEGEN only generates closed hypotheses. These hypotheses do not use higher-order combinators.

In general, the procedure separately generates a list of open hypotheses and a stream of closed hypotheses, each ordered according to cost. The stream returned to the top-level loop is obtained by merging this list and stream.

Generating open hypotheses The open hypotheses that we generate are drawn from Figure 3. The first column of Figure 3 shows the hypothesis which serves as our inductive generalization, and the second column shows the definition of the combinator used in the hypothesis. Observe that the free variables in the first column correspond to new synthesis subproblems.

Open hypothesis generation is guided by the inferred type of the input-output examples. For instance, suppose we have the examples $\{[\ ] \mapsto 0, [1] \mapsto 1, [1,2] \mapsto 3\}$. Here, since the inferred type of the transformation is $\texttt{list}[\texttt{int}] \rightarrow \texttt{int}$, our hypothesis generator immediately rules out the first three combinators from Figure 3 from being possible generalizations. The use of types to guide generalization is an important feature of our procedure and greatly helps to keep the search space manageable (see Section 5).

Generating closed hypotheses Generation of closed hypotheses is based on lazy, exhaustive enumeration. Specifically, we construct a lazy stream of candidate expressions that are free of higher-order combinators and are compatible with the type τ. The stream is ordered by increasing cost.

This construction of the stream is recursive. We start with a stream of type-compatible base expressions made from constants and bound variables). We then select the operators $\oplus$ that have return types that are compatible with τ. For each such $\oplus$, we generate an ordered stream for non-base expressions of form $\oplus(\dots)$. Such a stream is obtained by recursively generating a stream for each argument of $\oplus$ and applying $\oplus$ to these argument streams. The stream that we return is obtained by merging all these expression streams.

Our enumerative search also uses a rewrite system to avoid enumerating syntactically distinct but semantically equivalent hypotheses. Given a candidate hypothesis e, this rewrite system produces either the original hypothesis e or an equivalent, but lower-cost hypothesis e'. As our goal is to find a minimal-cost program, we can safely prune e from the search space in the latter case. For example, suppose our signature has addition as a primitive operator, 1 and 0 as constants, and the axiom $\forall x. x + 0 = 0 + x = 1$. In this case, our rewrite system will make sure that $(1 + 0)$ is not generated as a closed hypothesis.

4.3 Inference of New Examples Using Deduction

We now explain the DEDUCE procedure used in the synthesis algorithm from Figure 2. Recall that the purpose of this procedure is to infer new input-output examples and to detect conflicts.

The deduction engine underlying our procedure can be described by a set of inference rules of the form $\mathcal{E}, h \vdash f : \mathcal{E}'$. The meaning of this judgment is that, if $\mathcal{E}$ is a set of input-output examples with inductive generalization h, then $\mathcal{E}'$ is a new set of input-output examples for unknown f used in expression h. Here, $\mathcal{E}'$ may also be $\bot$, indicating that a conflict is detected and that the subproblem defined by f does not have a solution. Note that the soundness of deduction implies that, if $\mathcal{E}, h \vdash f : \bot$, then h cannot be a correct inductive generalization for examples $\mathcal{E}$.

Figure 4 shows the deductive reasoning performed for hypotheses involving the `map`, `mapt` and `filter` combinators. The first three rules in Figure 4 concern hypotheses involving `map`. Specifically, the first rule deduces a conflict if $\mathcal{E}$ contains an input-output example of the form $A \mapsto B$ such that A and B are lists but their lengths are not equal. Similarly, the second rule deduces $\bot$ if $\mathcal{E}$ contains a pair of input-output examples $A \mapsto B$ and $A' \mapsto B'$ such that $A_i = A'_j$ but $B_i \neq B'_j$. Since function f provided as an argument to the `map` combinator must produce the same output for a given input, the existence of such an input-output example indicates the hypothesis must be incorrect. Finally, the third rule in Figure 4 shows how to deduce new examples for f when no conflicts are detected. Specifically, for an input-output example of the form $A \mapsto B$ where A and B are lists, we can deduce examples $A_i \mapsto B_i$ for function f.

Example 2. Consider the input-output examples $[1,2] \mapsto [2,3]$ and $[2,4] \mapsto [3,5]$ and the hypothesis $\lambda x.\ \texttt{map}\ f\ x$. Using the third rule from Figure 4, we deduce the following new examples for f: $\{1 \mapsto 2, 2 \mapsto 3, 4 \mapsto 5\}$.

Example 3. Consider the input-output examples $[1] \mapsto [1]$ and $[2,1,4] \mapsto [2,3,7]$ and the hypothesis $\lambda x.\ \texttt{map}\ f\ x$. We can derive a conflict using the second rule of Figure 4 because f maps input 1 to two different values 1 and 3.

Since the rules for `mapt` are similar to those for `map`, we do not explain them in detail. We use the notation $s_i \not\simeq t_i$ to indicate that trees s_i and t_i have incompatible "shapes".

The last three rules of Figure 4 describe the deduction process for the `filter` combinator. Since $\lambda x.\texttt{filter}\ f\ x$ removes those elements of x for which $f(x)$ evaluates to false, the length of the output list cannot be greater than that of the input list. Hence, the first rule for `filter` derives a conflict if there exists an example $A \mapsto B$ such that the length of list B is greater than the length of list A. Similarly, the second rule for `filter` deduces $\bot$ if there is some element in list B that does not have a corresponding element in list A. If no conflicts are detected, the last rule of Figure 4 infers input-output examples for f such that an element x is mapped to true if and only if it is retained in output list B.

Example 4. Consider the input-output example $[1,2] \mapsto [1,2,1,2]$ and the hypothesis $\lambda x.\ \texttt{filter}\ f\ x$. Our procedure deduces a conflict using the first rule for `filter`.

Example 5. Consider the input-output example $[1,2,3] \mapsto [1,3,2]$ and the hypothesis $\lambda x.\ \texttt{filter}\ f\ x$. This time, our procedure deduces $\bot$ using the second rule for `filter`.

Example 6. Consider the examples $[1,2,3] \mapsto [2]$ and $[2,8] \mapsto [2,8]$ and the hypothesis $\lambda x.\ \texttt{filter}\ f\ x$. Using the last rule of Figure 4, we derive the following examples for f: $\{1 \mapsto \textit{false}, 2 \mapsto \textit{true}, 3 \mapsto \textit{false}, 8 \mapsto \textit{true}\}$.

We now describe the deductive reasoning performed for the `foldl` and `foldr` combinators (we collectively refer to these operators as `fold`). Consider a hypothesis of the form $\lambda x.\ \texttt{fold}\ f\ e\ x$ and suppose we want to learn new input-output examples characterizing f. We have found that deducing new examples for f in a sound and scalable way is only possible if the expression e is a constant, as oppposed to a function of x. Therefore, our procedure generates two separate hypotheses involving `fold`:

1. $\lambda x.\ \texttt{fold}\ f\ c\ x$ where c is a constant
2. $\lambda x.\ \texttt{fold}\ f\ e\ x$ where e is an arbitrary expression.

Our deduction engine can make inferences and generate useful examples only for case (1). The second case relies on enumerative search, which is possible even without examples. As new examples can prune the search space significantly, synthesis is more likely to be efficient in case (1), and we always try the first hypothesis before the second one. In what follows, we discuss the inference rules for `fold` under the assumption that it has a constant base case.

Figure 5 describes the deduction process for combinators involving `foldl`, `foldr` and `rec` using set inclusion constraints. Specifically, the set $\mathcal{E}'$ in these rules denotes the smallest set satisfying the generated constraints. Consider the first two rules of Figure 5. By the first rule, if there are two input-output examples of the form $[\ ] \mapsto b$ and $[\ ] \mapsto b'$ such that $b \neq b'$, this means that we cannot synthesize the program using a fold operator with a constant base case; hence we derive $\bot$. Otherwise, if there is an example $[\,] \mapsto b$, we infer the initial seed value for `fold` to be b.

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \\ \exists i.(1 \le i \le n \wedge g(i) \neq h(i))\end{array}}{\mathcal{E}, \lambda x.\, \texttt{map}\ f\ x \vdash f : \bot} \qquad \frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \\ \exists i,j,k,m. \left(\begin{array}{c} 1 \le i \le n \wedge 1 \le j \le n \wedge 1 \le k \le g(i) \wedge \\ 1 \le m \le h(j) \wedge a_{ik} = a_{jm} \wedge b_{ik} \neq b_{jm}) \end{array}\right)\end{array}}{\mathcal{E}, \lambda x.\, \texttt{map}\ f\ x \vdash f : \bot}$$

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ig(i)}]\} \\ \mathcal{E}' = \bigcup_{1\le i\le n}\{a_{i1} \mapsto b_{i1}, \ldots a_{ig(i)} \mapsto b_{ig(i)}\} \\ ((a,b) \in \mathcal{E}' \wedge (a',b') \in \mathcal{E}' \wedge a = a') \Rightarrow b = b'\end{array}}{\mathcal{E}, \lambda x.\, \texttt{map}\ f\ x \vdash f : \mathcal{E}'} \qquad \frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{s_i \mapsto t_i\} \\ \exists i.(1 \le i \le n \wedge s_i \not\simeq t_i)\end{array}}{\mathcal{E}, \lambda x.\, \texttt{mapt}\ f\ x \vdash f : \bot}$$

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{s_i \mapsto t_i\} \\ \exists i,j,k,m. \left(\begin{array}{c} 1 \le i \le n \wedge 1 \le j \le n \wedge 1 \le k \le \text{size}(s_i) \wedge \\ 1 \le m \le \text{size}(t_i) \wedge s_{ik} = s_{jm} \wedge t_{ik} \neq t_{jm}) \end{array}\right)\end{array}}{\mathcal{E}, \lambda x.\, \texttt{mapt}\ f\ x \vdash f : \bot} \qquad \frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{s_i \mapsto t_i\} \qquad k = \text{size}(s_i) = \text{size}(t_i) \\ \mathcal{E}' = \bigcup_{1\le i\le n}\{s_{i1} \mapsto t_{i1}, \ldots s_{ik} \mapsto t_{ik}\} \\ ((a,b) \in \mathcal{E}' \wedge (a',b') \in \mathcal{E}' \wedge a = a') \Rightarrow b = b'\end{array}}{\mathcal{E}, \lambda x.\, \texttt{mapt}\ f\ x \vdash f : \mathcal{E}'}$$

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \\ \exists i.(1 \le i \le n\ \wedge\ h(i) > g(i))\end{array}}{\mathcal{E}, \lambda x.\, \texttt{filter}\ f\ x \vdash f : \bot} \qquad \frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \\ \exists i.\exists j.1 \le i \le n \wedge 1 \le j \le g(i) \wedge \forall k.(k \ge j \Rightarrow b_{ij} \neq a_{ik})\end{array}}{\mathcal{E}, \lambda x.\, \texttt{filter}\ f\ x \vdash f : \bot}$$

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\le i\le n}\{[a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \qquad \forall i.1 \le i \le n \mapsto h(i) \le g(i) \\ \forall (A \mapsto B) \in \mathcal{E}.\forall b_i \in B.\exists a_j \in A.(j \ge i \wedge b_i = a_j) \\ \mathcal{E}_T = \{x \mapsto \text{true} \mid x \in A \wedge x \in B \wedge A \mapsto B \in \mathcal{E}\} \qquad \mathcal{E}_F = \{x \mapsto \text{false} \mid x \in A \wedge x \notin B \wedge A \mapsto B \in \mathcal{E}\}\end{array}}{\mathcal{E}, \lambda x.\, \texttt{filter}\ f\ x \vdash f : \mathcal{E}_T \cup \mathcal{E}_F}$$

Figure 4. Deductive reasoning for `map`, `mapt`, and `filter`

Let us now consider the third rule (i.e., `foldr`), and suppose we have an example $E_1 :\ [a_1,\ldots,a_n] \mapsto b$ and another example $E_2 :\ [a_2,\ldots,a_n] \mapsto b'$. Now, observe the following equivalences:

$$\begin{array}{lll} \texttt{foldr}\ f\ y\ [a_1,\ldots,a_n] & & \\ \quad \equiv f\ (\texttt{foldr}\ f\ y\ \ [a_2,\ldots,a_n])\ a_1 & & (\text{def. of } \texttt{foldr}) \\ \quad \equiv f\ b'\ a_1 & & (\text{from } E_2) \\ \quad \equiv b & & (\text{from } E_1) \end{array}$$

Since we have $f\ b'\ a_1 = b$, it is sound to deduce the input-output example $(b', a_1) \mapsto b$ for the unknown function f.

As shown in the fourth rule, we can apply similar reasoning to `foldl`. Suppose we have the following examples:

$$E_1' :\ [a_1,\ldots,a_n] \mapsto b \qquad E_2' : [a_1,\ldots,a_{n-1}] \mapsto b'$$

We can again expand the recursive definition of `foldl` to obtain the following equivalences:

$$\begin{array}{lll} \texttt{foldl}\ f\ y\ \ [a_1,\ldots,a_n] & & \\ \quad \equiv f\ a_n\ (\texttt{foldl}\ f\ y\ \ [a_1,\ldots,a_{n-1}]) & & (\text{property of } \texttt{foldl}) \\ \quad \equiv f\ a_n\ b' & & (\text{from } E_2') \\ \quad \equiv b & & (\text{from } E_1') \end{array}$$

Hence, we can infer the input-output example $(a_n, b') \mapsto b$ for function f.

The next three rules for `foldt` are very similar to `foldl` and `foldr`; hence, we do not discuss them in detail. The last rule in Figure 5 infers input-output examples for function f used in the general recursion combinator. Recall from Figure 3 that, given an input list of the form $x : xs$, the general recursion combinator applies function f to pair (x, xs). Hence, for any input example of the form $[a_1,\ldots,a_n] \mapsto b$, we can deduce the example $(a_1, [a_2,\ldots,a_n]) \mapsto b$ for unknown function f.

$$\frac{([] \mapsto b)\ \in \mathcal{E},\ \ ([] \mapsto b')\ \in \mathcal{E},\ \ b \neq b'}{\mathcal{E}, \lambda x.\, \texttt{fold}\ f\ c\ x \vdash c : \bot}$$

$$\frac{([\,] \mapsto b) \in \mathcal{E}}{\mathcal{E}, \lambda x.\, \texttt{fold}\ f\ c\ x \vdash c : \{b\}}$$

$$\frac{\begin{array}{c}([a_1,\ldots,a_n] \mapsto b) \in \mathcal{E} \\ ([a_2,\ldots,a_n] \mapsto b') \in \mathcal{E}\end{array}}{\mathcal{E}, \lambda x.\, \texttt{foldr}\ f\ c\ x \vdash f : ((b', a_1) \mapsto b) \in \mathcal{E}'}$$

$$\frac{\begin{array}{c}([a_1,\ldots,a_n] \mapsto b) \in \mathcal{E} \\ ([a_1,\ldots,a_{n-1}] \mapsto b') \in \mathcal{E}\end{array}}{\mathcal{E}, \lambda x.\, \texttt{foldl}\ f\ c\ x \vdash f : ((a_n, b') \mapsto b) \in \mathcal{E}'}$$

$$\frac{(\text{tree}() \mapsto b)\ \in \mathcal{E},\ \ (\text{tree}() \mapsto b')\ \in \mathcal{E},\ \ b \neq b'}{\mathcal{E}, \lambda x.\, \texttt{foldt}\ f\ c\ x \vdash c : \bot}$$

$$\frac{(\text{tree}() \mapsto b) \in \mathcal{E}}{\mathcal{E}, \lambda x.\, \texttt{foldt}\ f\ c\ x \vdash c : \{b\}}$$

$$\frac{\begin{array}{c}\text{tree}(a, [b_1,\ldots,b_n]) \mapsto c \in \mathcal{E} \\ \forall i.(1 \le i \le n \Rightarrow (b_i \mapsto c_i' \in \mathcal{E}))\end{array}}{\mathcal{E}, \lambda x.\, \texttt{foldt}\ f\ c\ x \vdash f : ([c_1',\ldots,c_n'], a) \mapsto c) \in \mathcal{E}'}$$

$$\frac{(x : xs \mapsto b) \in \mathcal{E}}{\mathcal{E}, \lambda x.\texttt{recl}\ f\ e\ x \vdash f : ((x, xs) \mapsto b) \in \mathcal{E}'}$$

Figure 5. Deduction for `fold` and general recursion.

Characterization and limitations The deduction procedure used in our synthesis algorithm is *sound* but *incomplete*. In this context, we define soundness and completeness as follows:

Definition 1. (Soundness of deduction) Let $\mathcal{E}$ be a set of input-output examples, and let h be a hypothesis that is an inductive generalization of $\mathcal{E}$. The DEDUCE procedure is sound, if for every unknown f in h, whenever DEDUCE$(h, f, \mathcal{E}) = \mathcal{E}'$ and the synthesis problem defined by $\mathcal{E}'$ and f does not have a solution, then the synthesis problem defined by $\mathcal{E}$ and h also does not have a solution.

Definition 2. (Completeness of deduction) Let $\mathcal{E}$ be a set of examples, and let h be a hypothesis that is an inductive generalization of $\mathcal{E}$. The completeness of DEDUCE means that, if the synthesis problem defined by $\mathcal{E}, h$ does not have a solution, then there exists some unknown f in h such that (i) DEDUCE$(h, f, \mathcal{E}) = \mathcal{E}'$ and (ii) the synthesis problem defined by $\mathcal{E}', f$ does not have a solution.

In other words, the deduction procedure is sound if the non-existence of a solution to any synthesis subtask implies the non-existence of a solution to the original problem. Hence, soundness implies that deduction will never cause our synthesis procedure to reject a valid open hypothesis. On the other hand, the procedure is complete if a solution to the original problem exists whenever all synthesis subtasks generated from it have solutions.

We can show that:

Theorem 1. *The procedure* DEDUCE *is sound and incomplete.*

To see how our inference rules are not complete, consider the following example.

Example 7. Consider the hypothesis $\lambda x.\ \texttt{foldl}\ f\ y\ x$, and the following input-output examples provided by the user:

$$\begin{array}{lll} [\,] \mapsto 0 & [1] \mapsto 1 & [1,2] \mapsto 3 \\ [2,3] \mapsto 5 & [1,2,3] \mapsto 6 & [2,3,5] \mapsto 10 \end{array}$$

Using the inference rules from Figure 5, we infer the following input-output examples for f:

$$(1,0) \mapsto 1, \quad (2,1) \mapsto 3, \quad (3,3) \mapsto 6, \quad (5,5) \mapsto 10$$

Observe that there is information provided by $[2,3] \mapsto 5$ that is not captured by the inferred examples for f.

A consequence of incompleteness is that our synthesis procedure must check that a closed hypothesis is consistent with the user-provided example set $\mathcal{E}_{in}$. This is done in line 5 of the SYNTHESIZE procedure in Figure 2.

Another limitation of our deduction procedure is that it critically depends on prior knowledge about the operators used for synthesis. This kind of knowledge is not in general available for external operators. However, we note that the SYNTHESIZE procedure applies even when DEDUCE does not return useful results. In these cases, it relies wholly on enumeration and generalization.

4.4 Optimality of Synthesis

Now we show that procedure SYNTHESIZE from Figure 2 correctly solves the synthesis problem defined in Section 3.

Theorem 2. *If* SYNTHESIZE *returns program e on examples $\mathcal{E}$, then e is a minimal-cost closed program that satisfies $\mathcal{E}$.*

Proof. The program e is guaranteed to be closed and satisfy the input-output examples by lines 4-6 in the procedure.

We prove optimality of e by contradiction. Assume there is a closed e' such that $e \neq e'$, $\mathcal{C}(e') < \mathcal{C}(e)$ and e' satisfies $\mathcal{E}$. Consider the point in time when SYNTHESIZE picks a task of the form $(e, f^*, \mathcal{E}^*)$ out of the task pool Q. A task of the form $(e', f', \mathcal{E}')$ cannot be in Q at this time or at any point of time until this point. Otherwise, because of line 3, SYNTHESIZE would have picked $(e', f', \mathcal{E}')$ and returned e'.

However, Q must, at this point, contain an open hypothesis h whose free variables can be instantiated to produce e'. This is because the pruning mechanisms in SYNTHESIZE are sound; as a result, the procedure does not rule out hypotheses from which satisfactory programs can be generated.

We know that $\mathcal{C}(h) \geq \mathcal{C}(e)$; otherwise SYNTHESIZE would have picked the task involving h in this loop iteration. However, note that in our cost model (Section 3), primitive operators and constants have positive costs and free variables have cost 0. Thus, any program of form $h[h'/f]$ must have strictly higher cost than h. This means that $C(e') > C(h)$, which implies $C(e') > C(e)$. This is a contradiction. □

5. Evaluation

We have implemented the proposed synthesis algorithm in a tool called $\boldsymbol{\lambda^2}$, which consists of ∼4,000 lines of OCaml code. In our current implementation, the cost function prioritizes open hypothesis generation over brute-force search for closed hypotheses. In particular, this is done by using a weighting function $W(c_a, c_b) = c_a + 1.5^{c_b}$ where c_a and c_b are the total costs of expressions obtained through open and closed hypothesis generation respectively. Intuitively, the weighting function attempts to balance the relative value of continued generalization and exhaustive search — the exponential term reflects that the exhaustive search space grows exponentially with maximum cost.

To evaluate our synthesis algorithm, we gathered a corpus of over 40 data structure manipulation tasks involving lists, trees, and nested data structures such as lists of lists or trees of lists. As described in the last column of Figure 6, most of our benchmarks are textbook examples for functional programming and some are inspired by end-user synthesis tasks, such as those mentioned in Section 1 and Section 2.

Our main goal in the experimental evaluation was to assess (i) whether $\boldsymbol{\lambda^2}$ is able to synthesize the benchmarks we collected, (ii) how long each synthesis task takes, and (iii) how many examples need to be provided by the user for $\boldsymbol{\lambda^2}$ to generate the intended program. To answer these questions, we conducted an experimental evaluation by running $\boldsymbol{\lambda^2}$ over our benchmark examples on an Intel(R) Xeon(R) E5-2430 CPU (2.20 GHz) with 8GB of RAM.

The column labeled “Runtime” in Figure 6 shows the running time of $\boldsymbol{\lambda^2}$ on each benchmark. The symbol $\perp$ indicates that $\boldsymbol{\lambda^2}$ is unable to complete the synthesis task within a time limit of 10 minutes. As shown in Figure 6, $\boldsymbol{\lambda^2}$ is able to successfully synthesize every benchmark program within its resource limits. Furthermore, we see that $\boldsymbol{\lambda^2}$ is quite efficient: its median runtime is 0.43 seconds, and it can synthesize 88% of the benchmarks in under a minute.

The column labeled “Expert examples” shows the number of examples we provided for each synthesis task (the examples were produced by a co-author of this paper). As shown in Figure 6, more than 75% of the benchmarks require 5 or fewer input-output examples, and there is only a single benchmark that requires more than 10 examples.

We were also interested in the following questions:

- What is the impact of using types in our algorithm?
- How effective is deduction?
- How does $\boldsymbol{\lambda^2}$ behave if its examples are not provided by an expert who is familiar with $\boldsymbol{\lambda^2}$?

In what follows, we address these questions in more detail.

Impact of types. Recall that our synthesis algorithm uses type-aware inductive generalization to prune the search space. To understand the impact of using types, we modified our algorithm to ignore types when generating hypotheses. The column labeled “Runtime (no types)” in Figure 6 shows the running time of the algorithm when we do *not* perform inductive generalization in a type-aware way. As shown by the data, types have a huge impact on the running time of the synthesis algorithm. In fact, without type-aware inductive generalization, more than 60% of the benchmarks do not terminate within the provided 10 minute resource limit.

Impact of deduction. We also conducted an experiment to evaluate the effectiveness of deduction in the overall synthesis algorithm. Recall from Section 4 that our algorithm uses deduction for (i) inferring new input-output examples, and (ii) refuting incorrect hypotheses. To evaluate the impact of deduction, we modified our algorithm so that it does not perform any of the reasoning described in Section 4.3. The running times of this modified algorithm are presented in Figure 6 under the column labeled “Runtime (no deduction)”. While the impact of deduction is not as dramatic as the

	Name	Runtime	Runtime (no deduction)	Runtime (no types)	Expert examples	Random examples	Runtime (random)	Extra primitives	Description
Lists	add	**0.04**	0.05	3.87	5	4	**0.04**		Add a number to each element of a list.
	append	**0.23**	0.49	⊥	3	16	0.93		Append an element to a list.
	concat	**0.13**	0.22	68.95	5	23	0.20		Concatenate two lists together.
	dedup	**231.05**	⊥	⊥	7	-	-	member	Remove duplicate elements from a list.
	droplast	**316.39**	⊥	⊥	6	-	-		Drop the last element in a list.
	dropmax	**0.12**	0.19	77.05	3	7	0.16	max	Drop the largest number(s) in a list.
	dupli	**0.11**	0.86	378.35	3	5	0.20		Duplicate each element of a list.
	evens	**7.39**	45.52	⊥	5	8	30.08		Remove the odd numbers from a list.
	last	**0.02**	0.06	1.80	4	4	0.03		Return the last element in a list.
	length	**0.01**	0.14	41.36	4	5	0.04		Return the length of a list.
	max	**0.46**	9.53	⊥	7	8	8.19		Return the largest number in a list.
	member	**0.35**	2.87	⊥	8	88	1.15		Check whether an item is a member of a list.
	multfirst	**0.01**	**0.01**	1.82	4	5	0.03		Replace every item in a list with the first item.
	multlast	**0.08**	0.51	⊥	4	7	0.27		Replace every item in a list with the last item.
	reverse	**0.01**	0.06	39.03	4	5	0.03		Reverse a list.
	shiftl	**0.89**	6.23	⊥	5	7	2.19	reverse	Shift all elements in a list to the left.
	shiftr	**0.65**	3.79	⊥	6	13	6.58	reverse	Shift all elements in a list to the right.
	sum	**0.01**	0.31	44.24	4	4	0.04		Return the sum of a list of integers.
Trees	count_leaves	**0.44**	2.69	⊥	8	10	0.67	sum	Count the number of leaves in a tree.
	count_nodes	**0.62**	6.13	⊥	4	9	1.04		Count the number of nodes in a tree.
	flatten	**0.08**	0.09	102.24	3	6	0.14	join	Flatten a tree into a list.
	height	**0.10**	0.27	83.12	6	7	0.20	max	Return the height of a tree.
	incrt	0.02	**0.01**	1.90	3	4	0.03		Increment each node in a tree by one.
	leaves	**0.52**	1.83	⊥	5	8	0.83	join	Return a list of the leaves of a tree.
	maxt	**10.59**	375.07	⊥	6	43	46.80		Return the largest number in a tree.
	membert	**4.66**	56.80	⊥	12	75	18.07		Check whether an element is contained in a tree.
	selectnodes	**15.97**	94.91	⊥	4	9	66.81	join, `pr`	Return a list of nodes in a tree that match a predicate `pr`.
	sumt	**0.59**	5.74	⊥	3	9	1.06		Sum the nodes of a tree of integers.
	tconcat	**551.84**	⊥	⊥	3	-	-		Insert a tree under each leaf of another tree.
Nested structures	appendt	**1.03**	2.44	⊥	5	14	2.57		Append an element to each node in a tree of lists.
	cprod	**83.83**	⊥	⊥	4	-	-		Return the Cartesian product of a list of lists.
	dropmins	**114.65**	452.07	⊥	4	-	-	min	Drop the smallest number in a list of lists.
	flattenl	**0.08**	0.11	87.94	5	5	0.15	join	Flatten a tree of lists into a list.
	incrs	**0.12**	0.80	⊥	4	4	0.46		Increment each number in a list of lists.
	join	**0.43**	2.13	⊥	4	6	1.01		Concatenate a list of lists together.
	prependt	**0.01**	**0.01**	3.46	5	4	0.03		Prepend an element to each list in a tree of lists.
	replacet	**4.02**	10.22	⊥	4	8	10.94		Replace one element with another in a tree of lists.
	searchnodes	**4.28**	43.85	⊥	6	31	19.68	member	Check if an element is contained in a tree of lists.
	sumnodes	**0.16**	0.43	⊥	4	3	0.34		Replace each node with its sum in a tree of lists.
	sums	**0.12**	1.26	⊥	4	5	0.54		For each list in a list of lists, sum the list.
	sumtrees	**12.10**	77.21	⊥	3	5	49.55		Return the sum of each tree in a list of trees.
	Median	0.43	2.13	⊥	4	8	0.93		

Figure 6. λ^2 performance. Times are in seconds. ⊥ indicates a timeout (>10 minutes) or an out of memory condition (>8GB).

impact of type-aware hypothesis generation, it nonetheless has a significant impact. In particular, the original algorithm with deduction is, on average, 6 times faster than the modified algorithm with no deduction.

Random example generation. Next, we examined the extent to which the effectiveness of our algorithm depends on the quality of user-provided examples. In particular, we aimed to estimate the behavior of $\boldsymbol{\lambda^2}$ on examples provided by a user who has no prior exposure to program synthesis tools. To this end, we built a *random example generator* that serves as a "lower bound" on a human user of $\boldsymbol{\lambda^2}$. The random example generator is, by design, quite naive. For instance, given a program with input type list[int], our example generator chooses a small list length and then populates the list with integers generated uniformly from a small interval. The corresponding output is generated by running a known implementation of the benchmark on this input.

Now, to determine the minimum number of examples that $\boldsymbol{\lambda^2}$ needs to synthesize the benchmark program, we ran the random example generator to generate k independent input-output examples. Given an example set of size k, we then checked whether $\boldsymbol{\lambda^2}$ is able to synthesize the correct function in 90% of the trials. If so, we concluded that the lay user should be able to successfully synthesize the target program with an example set of size k, and set the *runtime* of the benchmark to be the median runtime on these trials. Otherwise, we increased the value of k and repeat this process.

The columns labeled "Random examples" and "Runtime (random)" in Figure 6 respectively show the minimum number of examples and runtime for each benchmark when the examples are generated randomly. We see that the median values of these quantities are fairly low (8 and 0.93 seconds, respectively).

For a few benchmarks (e.g., dedup), $\boldsymbol{\lambda^2}$ was unable to synthesize the program for all trials with $k \leq 100$ examples while it requires a large number of examples for a few other benchmarks (e.g., member). However, upon further inspection, this turned out to be due to the naiveté of our random example generator rather than a shortcoming of $\boldsymbol{\lambda^2}$. For instance, consider the member benchmark which returns true iff the input list l contains a given element e. Clearly, any reasonable training set should contain a mix of positive and negative examples. However, when both the list l and the element e are randomly generated, it is very unlikely that l contains e. Hence, many randomly generated example sets contain only negative examples and fail to illustrate the desired concept. However, we believe it is entirely reasonable to expect that a human can provide both positive and negative examples in the training set.

Summary. Overall, our experimental evaluation validates the claim that $\boldsymbol{\lambda^2}$ can effectively synthesize representative, non-trivial examples of data structure transformations. Our experiments also show that type-aware inductive generalization and deductive reasoning have a significant impact on the running time of the algorithm. Finally, our experiments with randomly generated input-output examples suggest that using $\boldsymbol{\lambda^2}$ requires no special skill on the part of the user.

6. Related Work

Program synthesis has received much attention from the programming languages community lately. Many of these efforts [4, 30, 31] assume a programmer-provided "template" of the target program. Some others [19] do not need a template, but require a full logical specification of the target program. In contrast, our method performs synthesis from input-output examples.

Existing synthesis techniques that require only input-output examples are often restricted to programs over basic data types like numbers [29], strings [11], and tables [13], as opposed to recursive data structures. An exception is work by Albarghouthi et al. [1], which, like our method, synthesizes recursive programs over data structures. However, this approach is algorithmically quite different from ours: it is based on goal-directed enumerative search, and does not use types, inductive generalization, or deduction of conflicts and new examples. As we have shown in Section 5, these ideas are critical to the performance of our approach in most of our benchmarks.

In recent work, Perelman et al. [27] give a generic method called Test-Driven Synthesis (TDS) for constructing program synthesizers for user-defined domain-specific languages (DSLs). Also, Le and Gulwani [21] present a synthesis framework for DSLs called FlashExtract where programs are made from combinators such as map and filter. The problem domains targeted by these approaches and ours are similar; in fact, our synthesis problem could in principle be expressed in the DSL framework of TDS. However, the synthesis algorithm in TDS primarily relies on an optimized enumerative search, and the algorithm in FlashExtract is mostly based on deduction. In contrast, our approach uses a combination of deduction, generalization, and enumeration.

Example-guided synthesis has a long history in the artificial intelligence community [16, 22, 24]. Specifically, we build on the tradition of *inductive programming* [15, 18, 20], where the goal is to synthesize functional or logic programs from examples. Work here falls in two categories: those that inductively *generalize* examples into functions or relations [18], and those that *search* for implementations that fit the examples [14, 25]. Recent work by Kitzelmann [17] marries these two strands of research by restricting search to a space of candidate programs obtained through generalization. The main difference between this approach and ours lies in the roles of search and deduction. Specifically, Kitzelmann [17] uses deduction to generate candidates for search — i.e., each hypothesis must be deduced from some parent hypothesis. In contrast, we use deduction of examples as a *guide* to enumerative search. All hypotheses are allowed unless proven otherwise, and if deduction fails, we fall back on exhaustive search. This is a critical advantage because deductive inference is not necessarily applicable for every operator in a programming languge.

Since our technique uses types to guide synthesis, another line of related work is *type-directed program synthesis*. In particular, several recent papers use type-guided search to auto-complete program expressions involving complex API calls [12, 23, 26]. For instance, the InSynth tool formulates the code completion problem in terms of *type inhabitation* and generates a rank-ordered list of type inhabitants [12]. While our type-aware inductive generalization can be viewed as a form of type inhabitation problem, we simply use it for pruning the search space. Furthermore, rather than just finding a type inhabitant, our goal is to synthesize a program that is consistent with the input-output examples.

The technique we have proposed in this paper synthesizes a program that is not only consistent with the provided examples but is also a *minimum-cost* one according to some cost metric. Hence, our approach bears similarity to other efforts for *optimal program synthesis* [5, 6, 8]. In addition to addressing a different synthesis domain, we propose a different definition of *optimality* in this paper.

7. Conclusion

We have presented a method for example-guided synthesis of functional programs over recursive data structures. Our method combines three key ideas: type-aware inductive generalization, search over hypotheses about program structure, and use of deduction to guide the search. Our experimental results indicate that the proposed approach is promising.

There are many open directions for future work. We plan to study ways of exploiting parallelism in our synthesis algorithm.

We are also interested in using our method for synthesizing proofs. Several recent papers [9, 28] have studied the synthesis of program proofs from tests (which can be viewed as examples). We believe that $\boldsymbol{\lambda}^2$ can be used effectively for this purpose.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[3] D. W. Barron and C. Strachey. Programming. *Advances in Programming and Non-Numerical Computation*, pages 49–82, 1966.

[4] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.

[5] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification*, pages 140–156, 2009.

[6] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL*, pages 207–220, 2014.

[7] O. Danvy and M. Spivey. On Barron and Strachey's cartesian product function. In *ICFP*, pages 41–46, 2007.

[8] T. Dillig, I. Dillig, and S. Chaudhuri. Optimal guard synthesis for memory safety. 2014.

[9] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.

[10] S. Gulwani. Dimensions in program synthesis. In *FMCAD*, 2010.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[12] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, volume 48, pages 27–38. ACM, 2013.

[13] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[14] S. Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *Trends in Artificial Intelligence, 10th Pacific Rim International Conference on Artificial Intelligence.*, pages 199–210, 2008.

[15] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming, Third International Workshop*, pages 50–73, 2009.

[16] E. Kitzelmann. Analytical inductive functional programming. In *Logic-Based Program Synthesis and Transformation*, pages 87–102. Springer, 2009.

[17] E. Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *KI-Künstliche Intelligenz*, 25(2):179–182, 2011.

[18] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[19] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[20] N. Lavrac and S. Dzeroski. Inductive logic programming. In *WLP*, pages 146–160. Springer, 1994.

[21] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, pages 542–553, 2014.

[22] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[23] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[24] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[25] R. Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1):55–8, 1995.

[26] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, volume 47, pages 275–286. ACM, 2012.

[27] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, page 43, 2014.

[28] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[29] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[30] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[31] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[32] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. *STTT*, 15(5-6):413–431, 2013.