RICE UNIVERSITY

Inductive Program Synthesis from Input-Output Examples

by

John K. Feser

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

[illegible]

Swarat Chaudhuri, Chair
Associate Professor of Computer Science

[illegible]

Christopher Jermaine
Associate Professor of Computer Science

[illegible]

Moshe Y. Vardi
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering

Houston, Texas

April, 2016

ABSTRACT

Inductive Program Synthesis from Input-Output Examples

by

John K. Feser

We present a method for example-guided synthesis of higher-order functional programs. Given a set of input-output examples, our method synthesizes the simplest program to fit the examples.

Our approach combines three technical ideas: enumerative search, higher-order functions for succinct programs, and deduction. First, we generalize the input-output examples into hypotheses about the structure of the target program. For each hypothesis, we use deduction to infer examples for the missing subexpressions, leading 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 best-first enumeration 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. The experiments demonstrate the scalability and broad scope of $\boldsymbol{\lambda^2}$.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Background and motivation

The last few years have seen a flurry of research on *automated program synthesis* [1, 2, 3, 4, 5]. This research area aims to radically simplify programming by allowing users to express their intent as non-deterministic, 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 [6, 7, 8]. 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 [6]. Example-guided synthesis has also been applied in many other application domains, ranging from bit-level algorithms to geometry constructions to text editing [9, 10, 7].

In this thesis, we present a new, general approach to inductive synthesis from examples, with a particular focus on programs which manipulate recursive data structures like lists and trees. Prior approaches to example-guided synthesis have not considered recursive data structures or can only consider a restricted class of programs [6, 14, 24]. Our approach is a step towards a program synthesis technique which is both general and scalable.

1.2 Prior work

Our work is inspired by two broad classes of prior work in program synthesis. First, a family of synthesis tools based on version space algebras [13, 6, 14]. Second, work on synthesizing recursive programs in the inductive programming community.

The first class of prior work includes synthesis tools like *FlashFill* [6] and *FlashRelate* [14] which synthesize programs from examples for performing common tasks like transforming strings or reformatting spreadsheets. These techniques use a combination of search and deduction to synthesize programs in restricted domain specific languages (DSLs). Each tool uses a synthesis algorithm which is customized for the DSL that it uses, so to cover new classes of programs new synthesizers must be created.

The second class of prior work has its roots in the artificial intelligence community. Example-guided synthesis has a long history in artificial intelligence [7, 8, 18]. Specifically, we build on the tradition of *inductive programming* [19, 20, 21], 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 [20], and those that *search* for implementations that fit the examples [22, 23]. Recent work [24] marries these two strands of research by restricting search to a space of candidate programs obtained through generalization. [24] uses deduction to generate candidates for search — i.e., each hypothesis must be deduced from some parent hypothesis. This prunes the search space to the class of programs which can be inferred deductively from the input-output examples. Then, [24] searches the space for a program which satisfies the examples.

Other prior work includes [15], which synthesizes recursive programs over data structures using enumerative search. Similarly, [16] is a generic method, based on enumerative search, for constructing program synthesizers for an arbitrary user-defined domain-specific language.

Prior work in the field of program synthesis does not adequately solve the problem of synthesizing recursive data structure manipulation programs from example specifications. Existing synthesis systems are specialized to a particular DSL, search program spaces that are limited to programs which can be deduced from the input specification, or lack scalability. We advance the state of the art with an algorithm that synthesizes programs in a general purpose language while cleanly integrating domain knowledge to retain scalability.

1.3 Contributions

In this thesis, we present a method for example-guided synthesis of recursive, higher-order functional programs, with a particular focus on synthesis of data structure manipulating programs. Our synthesis algorithm targets a functional programming language that permits higher-order functions 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 returns a program that fits the examples or times out.

Although our goal is to build a general purpose synthesizer, we cannot retain both perfect generality and adequate scalability. Our approach combines a *best-first enu-*

merative search with domain knowledge in a principled way such that the general purpose search is cleanly separated from the domain specific parts. Our synthesizer uses domain knowledge in two ways: (1) we extend our language with *higher-order functions* that encapsulate common patterns in data structure manipulating programs, (2) we use domain-specific *deduction rules* to guide the solution of subproblems.

1.3.1 Best-first enumerative search

The core of our synthesizer is a general purpose search procedure which enumerates candidate programs. 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. At each point in the search, it picks the *least-cost* item from this frontier, so candidate programs are enumerated in *best-first* order. In Section 4.4, we show that this search strategy allows us synthesize the simplest program that fits the examples.

Searching programs best-first means that the program that we synthesize is guaranteed to be the *least-cost* program in our language to fit the examples, according 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 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.

1.3.2 Higher-order functions

The set of primitive functions from which we build programs includes higher-order functions which encapsulate common recursion schemes over recursive data structures. This set of primitives is a form of *domain knowledge* that makes our synthesizer more effective at synthesizing data structure transformation programs. In our synthesizer, these primitives are functions like `map`, `fold`, and `filter`. Higher-order functions are a natural choice to encapsulate recursion over a data structure. For example, the `map` function takes a list l and a function f as arguments, and recursively applies f to each element of l, abstracting away the details of the recursion that is occurring internally. Providing recursion schemes as primitives means that we can synthesize succinct recursive programs, because the synthesizer need not generate the boilerplate code for the recursion.

1.3.3 Deduction

Our synthesizer also incorporates domain knowledge in the form of deduction rules. Our algorithm generates hypotheses in the form of program skeletons, or programs which contains holes. For instance, our algorithm might generate the hypothesis $\lambda x.$ `map` $f^* \ x$ if all input-output examples are of type $\text{list}(\tau) \rightarrow \text{list}(\tau)$. In this example, f^* is a hole which must be filled in to complete the hypothesis. In general, given a skeleton h, we must solve one or more sub-problems in order to synthesize the unknown functions that appear in h. Simply searching the space of programs which could complete h would take too long, so we use *automated deduction* to guide the search for solution to these 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 $h \equiv \lambda x.\ \texttt{map}\ f^*\ x$. 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 $\lambda x.\ \texttt{map}\ f^*\ x$ 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.

The use of deduction guides the search by eliminating hypotheses which have holes that cannot be filled and by narrowing the space of candidates for each subproblem.

1.3.4 Differences from prior work

Our approach improves on the state-of-the-art in three ways.

First, our approach to synthesis is not specialized to the language that we target. Although we focus on synthesizing programs over recursive data structures, our search procedure is not specialized to this problem. We extract out the domain-specific parts of the synthesizer into our deduction rules and our choice of higher-order functions, rather than building them into the search procedure.

Second, our algorithm is more scalable than algorithms which rely solely on enumerative search. We combine type-based pruning with deduction rules to significantly reduce the number of programs which must be examined. As we show in Ch. 5, these ideas are critical to the performance of our approach.

Finally, the space of programs that we search is not limited to those than can be deduced from the examples. We use deduction of examples solely 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 language.

1.4 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 synthesis tool[*].

In our benchmarks, we focus on programs that transform recursive data structures such as lists and trees. Such programs arise in many end-user programming scenarios and are naturally expressed as functional programs.

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 [11].

1.5 Organization

The thesis is organized as follows. In Ch. 2, we present three motivating examples for our approach. After formalizing the problem in Ch. 3, we present our synthesis

[*]The name $\boldsymbol{\lambda^2}$ stands for "Learning Lambdas"

algorithm in Ch. 4. An evaluation is presented in Ch. 5. Finally, we conclude with some discussion in Ch. 6.

Chapter 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. For simplicity, we assume that the scores for each student are distinct. 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:

```
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 [*].

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

In this example, note the complex interplay between scoping and higher-order functions where the `map`, `filter`, and `fold` operations are nested in a highly nontrivial way. 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:

$$[\,] \mapsto [\,]$$

$$[[1]] \mapsto [[\,]]$$

$$[[1,3,5],[5,3,2]] \mapsto [[3,5],[5,3]]$$

$$[[8,4,7,2],[4,6,2,9],[3,4,1,0]] \mapsto [[8,4,7][4,6,9],[3,4,1]]$$

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

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

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

Here, the operator `foldt` performs a fold over an unordered tree, and `concat` takes in a list of lists l_i and returns the concatenation of the l_i-s. 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:

$$\langle\,\rangle \mapsto [\,]$$
$$\langle(a, 1760)\ \langle(b, 1803)\rangle\ \langle(c, 1795)\rangle\rangle \mapsto [(b, 1803)]$$
$$\langle(a, 1771)\ \langle(b, 1815)\rangle\ \langle(c, 1818)\rangle\rangle \mapsto [(b, 1815), (c, 1818)]$$
$$\langle(a, 1812)\ \langle(b, 1846)\rangle\ \langle(c, 1852)\rangle\rangle \mapsto [(a, 1812)]$$

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. $\boldsymbol{\lambda^2}$ 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 [33]. Danvy and Spivey call this program "arrestingly beautiful" and believe it to be "the world's first functional pearl" [11].

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:

$$[[1,2,3],[4],[5,6]] \mapsto [[1,4,5],[1,4,6],[2,4,5],[2,4,6],[3,4,5],[3,4,6]]$$

Barron and Strachey implement this function as follows[33]:

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

Here, `foldr` is the standard right-fold operation. For an article-length explanation of how this function works, see [11].

We used $\boldsymbol{\lambda}^{\mathbf{2}}$ to synthesize the Cartesian product function from the following examples:

$$[\,] \mapsto [[\,]]$$

$$[[\,]] \mapsto [\,]$$

$$[[\,],[\,]] \mapsto [\,]$$

$$[[1,2,3][5,6]] \mapsto [[1,5],[1,6],[2,5],[2,6],[3,5],[3,6]]$$

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

Chapter 3

Problem formulation

In this section, we formally state our synthesis problem.

3.1 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:

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

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 construct "**match** e **with** ..." performs ML-style pattern-matching.

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

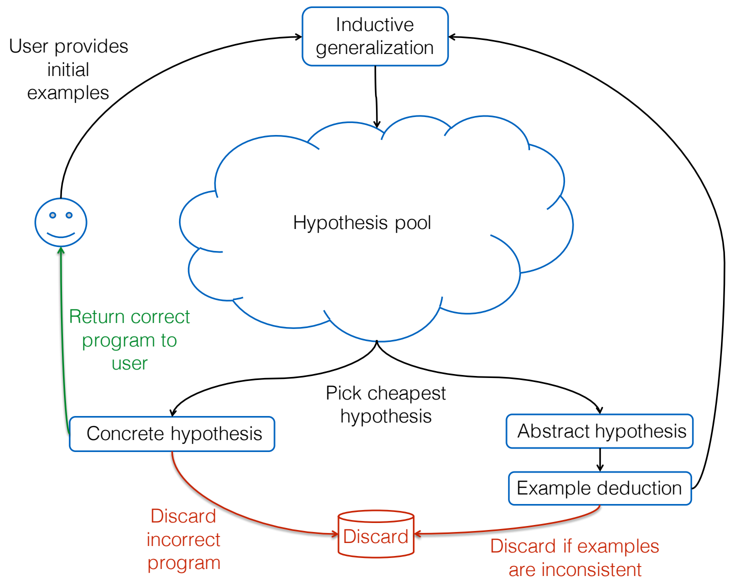

Figure 3.1 : High-level overview of our synthesis algorithm

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.

3.2 Components

Our synthesis algorithm is parameterized by a set of *components*, where a component is a function, constant, or variable. Components have a name, an arity, and a type.

Our implementation of the language comes prepackaged with certain components and type definitions. Predefined types include (polymorphic) lists and trees, encoded as variants. Predefined components include the standard arithmetic operators, if-then-else, and higher-order combinators like `map`, `foldl`, `foldr`, and `foldt` (see

Figure 3.2).

While synthesizing the inside of a lambda or a let binding, the component set is augmented with the variables in scope. We also will augment the built in component set with with external components and types for solving particular problems. For instance, in the synthesis of the `selectnodes` function in Sec. 2.2, `pr` is an external component.

3.3 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.

3.4 Specifications

Let an *input-output example* be a term $a_i \mapsto b_i$, where a_i and b_i are concrete programs. Each example has an implicit context Γ_i which contains the bindings of variables in the surrounding scope. Intuitively, this context should be treated as another input. For clarity of notation, we often omit Γ_i, but it is important to note that when

Hypothesis	Definition of Function
$\lambda x.$ `map` $f\ x$	`map` $::\ (a \to b) \to$ `list`$[a] \to$ `list`$[b]$ `map` $f\ [\,] = [\,]$ `map` $f\ (x : y) = (f\ x) :$ (`map` $y\ f$)
$\lambda x.$ `mapt` $f\ x$	`mapt` $::\ (a \to b) \to$ `tree`$[a] \to$ `tree`$[b]$ `mapt` f `tree`$()$ $=$ `tree`$()$ `mapt` f `tree`(x, y) $=$ `tree`$(f\ x,$ `mapt` $f\ y)$
$\lambda x.$ `filter` $f\ x$	`filter` $::\ (a \to$ `bool`$) \to$ `list`$[a] \to$ `list`$[a]$ `filter` $f\ [\,] = [\,]$ `filter` $(x : y)\ f =$ if $f\ x$ then $x :$ (`filter` $f\ y$) else `filter` $f\ y$
$\lambda x.$ `foldl` $f\ e\ x$	`foldl` $::\ (b \to a \to b) \to b \to$ `list`$[a] \to b$ `foldl` $f\ e\ [\,] = e$ `foldl` $f\ e\ (x : y) =$ `foldl` $f\ (f\ e\ x)\ y$
$\lambda x.$ `foldr` $f\ e\ x$	`foldr` $::\ (b \to a \to b) \to b \to$ `list`$[a] \to b$ `foldr` $f\ e\ [\,] = e$ `foldr` $f\ e\ (x : y) = f$ (`foldr` $f\ e\ y$) x
$\lambda x.$ `foldt` $f\ e\ x$	`foldt` $::$ (`list`$[b] \to a \to b) \to b \to$ `tree`$[a] \to b$ `foldt` $f\ e$ `tree`$()$ $= e$ `foldt` $f\ e$ `tree`$(x, y) = f$ (`map` ($\lambda z.$ `foldt` $f\ e\ z$) y) x

Figure 3.2 : Family of higher-order functions which are pre-packaged with $\boldsymbol{\lambda^2}$.

comparing the left-hand-sides of two examples, the contexts must also be compared. The input to our synthesis problem is a set $\mathcal{E}_{in}$ of such examples.

3.5 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 *abstract*. For instance, x and $\lambda x.f^{*}x$ are abstract hypotheses. In contrast, hypotheses that do not contain free variables are said to be *concrete*. For example, $\lambda x.\ \texttt{map}\ (\lambda y.\ y+1)\ x$ is a concrete 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.

Chapter 4

Synthesis algorithm

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

4.1 Algorithm architecture

Given a set of input-output examples, our goal is to compute a *minimal-cost* concrete 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.

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 *task pool* Q. Figure 3.1 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 concrete, 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 abstract, 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.

If hypothesis h is abstract 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 concrete and which does not conflict with the top-level examples $\mathcal{E}_{in}$.

Figure 1 gives pseudocode for the overall synthesis algorithm. Here, Q is 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 task pool Q. Now, if e is a concrete 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 the task pool Q.

Now, if e is an abstract 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 InductiveGen to perform type-aware inductive generalization. Suppose H is a list 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 concrete, there are no new unknowns to synthesize; hence, we add a single subtask $(e', \perp, \emptyset)$ to the task pool Q.

On the other hand, if e' is an abstract hypothesis, we generate new subtasks for synthesizing each hole in e'. Here, the procedure Deduce 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.

4.2 Inductive generalization

This section explains the hypothesis generation step of the synthesis algorithm in more detail. To generalize a hypothesis e, we must replace each hole in $h \in \text{Holes}(h)$ with a new hypothesis e'. We will now discuss the rules that we use to generate the hypotheses e'.

Algorithm 1 Synthesis procedure.

```
Require: $\mathcal{E}_{in}$ is a set of input-output examples.
Ensure: Returns a program which satisfies the examples in $\mathcal{E}_{in}$.
 1: function SYNTHESIZE($\mathcal{E}_{in}$)
 2:     $Q \leftarrow \{(f, f, \mathcal{E})\}$                    ▷ $f$ is a fresh variable name
 3:     while $Q \neq \emptyset$ do
 4:         Pick $(e, f, \mathcal{E})$ from $Q$ such that $e$ has minimal cost.
 5:         if $e$ is concrete and CONSISTENT($e, \mathcal{E}_{in}$) then
 6:             return $e$
 7:         else
 8:             continue
 9:         $\tau \leftarrow$ TYPEINFER($\mathcal{E}$)
10:         $H \leftarrow$ INDUCTIVEGEN($\tau$)
11:         for $h \in H$ do
12:             $e' \leftarrow e[h/f]$
13:             if $e'$ is concrete then
14:                 $Q \leftarrow Q \cup \{(e', \perp, \emptyset)\}$
15:             else
16:                 for $f^* \in$ HOLES($e'$) do
17:                     $\mathcal{E}^* \leftarrow$ DEDUCE($e', f^*, \mathcal{E}$)
18:                     if $\mathcal{E}^* = \perp$ then
19:                         break
20:                     $Q \leftarrow Q \cup \{(e', f^*, \mathcal{E}^*)\}$
```

If the hole type is of the form $\tau_1 \rightarrow \tau_2$, then we insert a lambda abstraction directly, rather than look for a higher-order component. In conjunction with the deduction rules for higher-order components, this simplification allows us avoid reasoning about higher-order candidate programs.

Otherwise, for holes of type τ, we select the set of components which are *type-compatible* with τ. We say that a component is type-compatible with a hole if (1) it is of type τ' and τ' unifies with τ or (2) it is of type $\tau_1 \rightarrow \tau_2$ and τ_2 unifies with τ. For components of type $\tau_1 \rightarrow \tau_2$, we infer types for the arguments to the component as a side effect of the type-compatibility check. After selecting the type-compatible components, we generate new holes corresponding to the arguments to the component.

The use of types to guide generalization is an important feature of our procedure and greatly helps to keep the search space manageable (see Ch. 5). In many practical cases, we are able to prune the set of generalizations using types. 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 3.2 from being possible generalizations.

4.2.1 Eliminating equivalent hypotheses

Our hypothesis generator 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 concrete hypothesis.

The rewrite system comes with a set of rules for the built-in components. As we provide most of the common arithmetic operators by default, and these operators have a rich set of axioms that relate them, there are many opportunities to simplify hypotheses. Hypotheses rewriting is a simple technique that can significantly reduce the size of the search space.

4.3 Inference of new examples using deduction

We now explain the DEDUCE procedure used in the synthesis algorithm from Figure 1. Recall that DEDUCE is used for inferring new input-output examples and detecting conflicts. 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* $\text{DEDUCE}(h, f, \mathcal{E}) = \mathcal{E}'$ *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.*

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 abstract hypothesis. However, since our deduction procedure is not complete, the existence of a solution to all synthesis subtasks does not imply the existence of a solution to the original problem. More technically, we define completeness as follows:

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)* $\textsc{Deduce}(h, f, \mathcal{E}) = \mathcal{E}'$ *and (ii) the synthesis problem defined by $\mathcal{E}', f$ does not have a solution.*

A consequence of incompleteness is that our synthesis procedure must check that a concrete hypothesis is consistent with the user-provided example set $\mathcal{E}_{in}$. This is done in line 5 of the SYNTHESIZE procedure in Figure 1.

We now describe the deduction engine underlying our procedure as 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 $\perp$, 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 $\mathcal{E}, h \vdash f : \perp$, then h cannot be a correct inductive generalization for examples $\mathcal{E}$.

The rules in Figure 4.1 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 $\perp$ 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 4.1 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.

$$\frac{\mathcal{E} = \bigcup_{1 \le i \le n} \{\Gamma_i, [a_{i1}, \ldots, a_{ig(i)}] \mapsto [b_{i1}, \ldots, b_{ih(i)}]\} \qquad \exists i.(1 \le i \le n \land g(i) \ne h(i))}{\mathcal{E}, \lambda x.\ \texttt{map}\ f\ x \vdash f : \bot}$$

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

$$\frac{\begin{array}{c} \mathcal{E} = \bigcup_{1 \le i \le n} \{\Gamma_i, [a_{i1}, \ldots, a_{ig(i)}] \mapsto [b_{i1}, \ldots, b_{ig(i)}]\} \\ \mathcal{E}' = \bigcup_{1 \le i \le n} \{\Gamma_i, a_{i1} \mapsto b_{i1}\ \ \ldots\ \ \Gamma_i, a_{ig(i)} \mapsto b_{ig(i)}\} \\ ((\Gamma, a \mapsto b) \in \mathcal{E}' \land (\Gamma', a' \mapsto b') \in \mathcal{E}' \land a = a' \land \Gamma = \Gamma') \implies b = b' \end{array}}{\mathcal{E}, \lambda x.\ \texttt{map}\ f\ x \vdash f : \mathcal{E}'}$$

Figure 4.1 : Deductive reasoning for `map`

$$\frac{\mathcal{E} = \bigcup_{1\leq i\leq n}\{\Gamma_i, s_i \mapsto t_i\} \qquad \exists i.(1 \leq i \leq n \wedge s_i \not\simeq t_i)}{\mathcal{E}, \lambda x.\ \texttt{mapt}\ f\ x \vdash f : \bot}$$

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

$$\frac{\begin{array}{c}\mathcal{E} = \bigcup_{1\leq i\leq n}\{\Gamma_i, s_i \mapsto t_i\} \qquad k = \text{size}(s_i) = \text{size}(t_i) \\ \mathcal{E}' = \bigcup_{1\leq i\leq n}\{\Gamma_i, s_{i1} \mapsto t_{i1}\ \ldots\ \Gamma_i, s_{ik} \mapsto t_{ik}\} \\ ((\Gamma, a \mapsto b) \in \mathcal{E}' \wedge (\Gamma', a' \mapsto b') \in \mathcal{E}' \wedge a = a' \wedge \Gamma = \Gamma') \implies b = b' \end{array}}{\mathcal{E}, \lambda x.\ \texttt{mapt}\ f\ x \vdash f : \mathcal{E}'}$$

Figure 4.2 : Deductive reasoning for `mapt`

$$\frac{\mathcal{E} = \bigcup_{1\leq i\leq n}\{\Gamma_i, [a_{i1},\ldots,a_{ig(i)}] \mapsto [b_{i1},\ldots,b_{ih(i)}]\} \qquad \exists i.(1 \leq i \leq n \ \wedge \ h(i) > g(i))}{\mathcal{E}, \lambda x.\ \texttt{filter}\ f\ x \vdash f : \bot}$$

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

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

Figure 4.3 : Deductive reasoning for `filter`

Example 1. *Consider the input-output examples* $[1, 2] \mapsto [2, 3]$ *and* $[2, 4] \mapsto [3, 5]$ *in a context* Γ *and the hypothesis* $\lambda x.$ `map` $f\ x$. *Using the third rule from Figure 4.1, we deduce the following new examples for* f*:* $\{1 \mapsto 2, 2 \mapsto 3, 4 \mapsto 5\}$.

Example 2. *Consider the input-output examples* $[1] \mapsto [1]$ *and* $[2, 1, 4] \mapsto [2, 3, 7]$ *in a context* Γ *and the hypothesis* $\lambda x.$ `map` $f\ x$. *We can derive a conflict using the second rule of Figure 4.1 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 rules in Figure 4.3 describe the deduction process for the `filter` combinator. Since $\lambda x.$`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 $\perp$ 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 4.3 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 3. *Consider the input-output example* $[1, 2] \mapsto [1, 2, 1, 2]$ *and the hypothesis* $\lambda x.$ `filter` $f\ x$. *Our procedure deduces a conflict using the first rule for* `filter`.

Example 4. *Consider the input-output example* $[1, 2, 3] \mapsto [1, 3, 2]$ *and the hypothesis* $\lambda x.$ `filter` $f\ x$. *This time, our procedure deduces* $\perp$ *using the second rule for* `filter`.

Example 5. *Consider the examples* $[1, 2, 3] \mapsto [2]$ *and* $[2, 8] \mapsto [2, 8]$ *and the hypothesis* $\lambda x.$ `filter` $f\ x$. *Using the last rule of Figure 4.3, we derive the following examples* *for f:* $\{1 \mapsto \text{false}, 2 \mapsto \text{true}, 3 \mapsto \text{false}, 8 \mapsto \text{true}\}$.

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 $\lambda x.$ `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. $\lambda x.$ `fold` $f\ c\ x$ where c is a constant
2. $\lambda x.$ `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 4.4 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 4.4. 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 $\perp$. Otherwise, if there is an example $[\,] \mapsto b$, we infer the initial seed value for `fold` to be b.

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:

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

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:

$$E'_1 : \quad [a_1, \dots, a_n] \mapsto b \qquad E'_2 : [a_1, \dots, a_{n-1}] \mapsto b'$$

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

$$\begin{aligned}
&\texttt{foldl}\ f\ y\ \ [a_1, \dots, a_n] \\
&\quad \equiv f\ a_n\ (\texttt{foldl}\ f\ y\ \ [a_1, \dots, a_{n-1}]) && \text{(property of } \texttt{foldl}\text{)} \\
&\quad \equiv f\ a_n\ b' && \text{(from } E'_2\text{)} \\
&\quad \equiv b && \text{(from } E'_1\text{)}
\end{aligned}$$

Hence, we can infer the input-output example $(a_n, b') \mapsto b$ for function f. However, as illustrated by the following example, these inference rules are not complete.

Example 6. *Consider the hypothesis $\lambda x.$ `foldl` $f\ y\ x$, and the following input-output examples provided by the user:*

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma, \mathrm{tree}(a, [b_1, \ldots, b_n]) \mapsto c \in \mathcal{E} \\ \forall i.(1 \leq i \leq n \Rightarrow (\Gamma, b_i \mapsto c'_i \in \mathcal{E}))\end{array}}{\mathcal{E}, \lambda x.\ \texttt{foldt}\ f\ c\ x \vdash f : \Gamma, ([c'_1, \ldots, c'_n], a) \mapsto c \in \mathcal{E}'}$$

Figure 4.4 : Deduction for `fold`.

Using the inference rules from Figure 4.4, 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.*

The next three rules for `foldt` are very similar to `foldl` and `foldr`; hence, we do not discuss them in detail.

4.4 Optimality of synthesis

Now we show that procedure Synthesize from Figure 1 correctly solves the synthesis problem defined in Ch. 3.

Theorem 1. *If* Synthesize *returns program e on examples $\mathcal{E}$, then e is a minimal-cost concrete program that satisfies $\mathcal{E}$.*

Proof. The program e is guaranteed to be concrete and satisfy the input-output examples by lines 5–6 in the procedure.

We prove optimality of e by contradiction. Assume there is a concrete 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 4, Synthesize would have picked $(e', f', \mathcal{E}')$ and returned e'.

However, Q must, at this point, contain an abstract 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 (Ch. 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. $\square$

Chapter 5

Evaluation

We have implemented the proposed synthesis algorithm in a tool called $\boldsymbol{\lambda^2}$, which consists of ∼13,000 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 shown in Tables A.5, A.6, and A.7, most of our benchmarks are textbook examples for functional programming and some are inspired by end-user synthesis tasks, such as those mentioned in Ch. 1 and Ch. 2.

Our main goal in the experimental evaluation is to assess (1) whether $\boldsymbol{\lambda^2}$ is able to synthesize programs from the specifications we collected, (2) how long each synthesis task takes, and (3) how many examples need to be provided for $\boldsymbol{\lambda^2}$ to generate the intended program. To answer these questions, we conducted an experimental evaluation by running $\boldsymbol{\lambda}^{\mathbf{2}}$ over our benchmark examples.[*]

The column labeled "Runtime" in Tables A.1, A.2, and A.3 shows the running time of $\boldsymbol{\lambda}^{\mathbf{2}}$ on each benchmark. The symbol $\perp$ indicates that $\boldsymbol{\lambda}^{\mathbf{2}}$ is unable to complete the synthesis task within a time limit of 10 minutes. As shown, $\boldsymbol{\lambda}^{\mathbf{2}}$ is able to successfully synthesize every benchmark program within its resource limits. Furthermore, we see that $\boldsymbol{\lambda}^{\mathbf{2}}$ is quite efficient: its median runtime is 0.43 seconds, as shown in Table A.4, 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. As shown, 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.

Additionally, we are interested in investigating the following questions:

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

In the following sections, we address these questions in more detail.

5.1 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 algo-

[*]Benchmarks were run on an Intel® Xeon® E5-2430 CPU (2.20 GHz) with 8GB of RAM.

rithm to ignore types when generating hypotheses. The column labeled "Runtime (no types)" 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.

5.2 Impact of deduction.

We also conducted an experiment to evaluate the effectiveness of deduction in the overall synthesis algorithm. Recall from Ch. 4 that our algorithm uses deduction for (1) inferring new input-output examples, and (2) 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 Sec. 4.3. The running times of this modified algorithm are presented under the column labeled "Runtime (no deduction)." While the impact of deduction is not as dramatic as the impact of type-aware hypothesis generation, it nonetheless has a significant impact. In particular, the original algorithm with deduction is 6 times faster on average than the modified algorithm with no deduction.

5.3 Impact of cost metric

We examined the impact of our cost metric on performance by testing the synthesizer with a very simple cost metric. The simple cost metric assigns each function, constant, and variable in our language a cost of one. Therefore, the cost of a program is simply the number of nodes in its syntax tree. The column labeled "Runtime (flat costs)" shows the running time of synthesis with the simple cost metric. This experiment shows that the cost metric is critical to performance; while synthesis performs well on simple examples regardless of the cost metric, complex examples suffer severe slowdowns or fail to terminate. More than 40% of the benchmarks fail to terminate within the time limit when the simple cost metric is used.

5.4 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 run the random example generator to generate k independent input-output examples. Given an example set of size k, we check whether $\boldsymbol{\lambda^2}$ is able to synthesize the correct function in 90% of the trials. If so, we conclude 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 increase the value of k and repeat this process.

The columns labeled "Random examples" and "Runtime (random)" respectively show the minimum number of examples and runtime for each benchmark when the examples are generated randomly. In Table A.4, 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`), we observe that $\boldsymbol{\lambda^2}$ is 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 turns 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.

5.5 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.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have presented a method for example-guided synthesis of functional programs over recursive data structures. Our method combines a general purpose search over hypotheses about program structure with two forms of domain knowledge: higher-order functions which encapsulate recursion schemes and deduction rules to guide the search. Our experimental results indicate that the proposed approach is promising.

6.2 Related work

The research in this thesis was originally published at PLDI 2015. Since then, inductive program synthesis has continued to be a vibrant research area, and a number of other works have been published.

Recent work, which was presented concurrently with $\boldsymbol{\lambda^2}$ at PLDI 2015, combines type-guided search and deduction of input-output examples by treating the examples as part of a refinement type [28]. During synthesis, the Myth system "pushes" examples towards the leaves of the hypothesis so that they can be used in pruning the search space. When synthesizing recursive functions, examples are used to determine the results of a recursive call. However, this requires the user to provide examples which allow the results of all recursive calls to be deduced. This differs from our approach: in $\boldsymbol{\lambda^2}$, examples are used purely for pruning, so no requirements are placed on the completeness of the example set.

Later work pushes the type-directed synthesis approach in Myth further by adding parametric polymorphism and refinement types [29]. Programs to be synthesized are specified by providing a *liquid type*. During the search, the Synquid system uses a specialized type inference algorithm to push specifications downwards to prune the search space. This approach performs well for programs which can be specified in a supported refinement logic, but requires that the refinement logic be extended to support wider classes of programs.

The authors of *FlashFill* and *FlashRelate* have since published *FlashMeta*, which is their framework for constructing program synthesizers [17]. DSL designers write "witness functions," which allow example based specifications to be pushed through the operators of the DSL, in much the same way that our deduction rules deduce examples for the holes in a hypothesis. When designing a new DSL, the programmer provides a grammar for the language and witness functions, and the *FlashMeta* framework handles the search procedure. This separation of concerns allows programmers to generates optimized synthesizers more quickly, but it still requires significant effort from a human programmer to implement a DSL and each implemented synthesizer remains specialized for a particular language.

6.3 Future work

There are many open directions for future work. Primarily, we are interested in extending $\boldsymbol{\lambda^2}$ to take advantage of existing code during synthesis. We also plan to study ways of exploiting parallelism in our synthesis algorithm.

6.3.1 Deduction

The synthesis algorithm that we have developed is general, so we believe that it is possible to extend the language used by the synthesizer without changing the fundamental synthesis algorithm. We plan to extend the target language with functions drawn from a corpus of existing code. The synthesizer will draw functions from the code corpus to build programs for a variety of applications. The primary research challenge is that the size of the search space increases dramatically as the number of functions in the target language increases. In $\boldsymbol{\lambda^2}$, we use deduction rules to prune the search space, but it will be infeasible to write custom deduction rules for every function in the corpus. The goal is to build a synthesizer that uses a corpus of existing code to allow users to synthesize programs in a variety of domains without manually adding domain-specific rules. Therefore, our focus for future work is to improve the deduction strategies in $\boldsymbol{\lambda^2}$ and to derive new deduction methods automatically from the functions in our corpus.

6.3.2 Parallelism

Parallelism is an under-explored possibility for program synthesis tools. Program synthesizers must be able to search large spaces of programs efficiently, but there are no state-of-the-art synthesizers which can scale to multiple machines. We believe that it will be possible to scale our synthesis algorithm effectively over many computers. This kind of horizontal scaling could significantly improve our ability to synthesize large programs.

Appendix A

Benchmarks

This section contains the benchmark results discussed in Chapter 5. In all tables, $\perp$ indicates that the program could not be synthesized in under 10 minutes.

Name	Runtime	Runtime (no deduction)	Runtime (no types)	Runtime (flat costs)	Expert examples	Random examples	Runtime (random)	Extra primitives
add	**0.04**	0.05	3.87	1.44	5	4	**0.04**	
append	**0.23**	0.49	⊥	⊥	3	16	0.93	
concat	**0.13**	0.22	68.95	5.65	5	23	0.20	
dedup	**231.05**	⊥	⊥	⊥	7	-	-	member
droplast	**316.39**	⊥	⊥	⊥	6	-	-	
dropmax	**0.12**	0.19	77.05	⊥	3	7	0.16	max
dupli	**0.11**	0.86	378.35	229.97	3	5	0.20	
evens	**7.39**	45.52	⊥	⊥	5	8	30.08	
last	0.02	0.06	1.80	**0.00**	4	4	0.03	
length	**0.01**	0.14	41.36	0.01	4	5	0.04	
max	**0.46**	9.53	⊥	⊥	7	8	8.19	
member	**0.35**	2.87	⊥	1.50	8	88	1.15	
multfirst	**0.01**	**0.01**	**0.01**	1.82	4	5	0.03	
multlast	0.08	0.51	⊥	**0.01**	4	7	0.27	
reverse	**0.01**	0.06	39.03	**0.01**	4	5	0.03	
shiftl	**0.89**	6.23	⊥	⊥	5	7	2.19	reverse
shiftr	**0.65**	3.79	⊥	⊥	6	13	6.58	reverse
sum	**0.01**	0.31	44.24	0.02	4	4	0.04	

Table A.1 : Benchmark results for list processing programs.

Name	Runtime	Runtime (no deduction)	Runtime (no types)	Runtime (flat costs)	Expert examples	Random examples	Runtime (random)	Extra primitives
count_leaves	0.44	2.69	$\perp$	**0.13**	8	10	0.67	sum
count_nodes	**0.62**	6.13	$\perp$	5.36	4	9	1.04	
flatten	**0.08**	0.09	102.24	1.82	3	6	0.14	join
height	**0.10**	0.27	83.12	8.55	6	7	0.20	max
incrt	0.02	**0.01**	1.90	0.04	3	4	0.03	
leaves	**0.52**	1.83	$\perp$	118.05	5	8	0.83	join
maxt	**10.59**	375.07	$\perp$	$\perp$	6	43	46.80	
membert	**4.66**	56.80	$\perp$	213.85	12	75	18.07	
selectnodes	**15.97**	94.91	$\perp$	$\perp$	4	9	66.81	join, `pr`
sumt	**0.59**	5.74	$\perp$	9.00	3	9	1.06	
tconcat	**551.84**	$\perp$	$\perp$	$\perp$	3	-	-	

Table A.2 : Benchmark results for tree processing programs.

Name	Runtime	Runtime (no deduction)	Runtime (no types)	Runtime (flat costs)	Expert examples	Random examples	Runtime (random)	Extra primitives
appendt	**1.03**	2.44	⊥	⊥	5	14	2.57	
cprod	**83.83**	⊥	⊥	⊥	4	-	-	
dropmins	**114.65**	452.07	⊥	⊥	4	-	-	min
flattenl	**0.08**	0.11	87.94	13.11	5	5	0.15	join
incrs	**0.12**	0.80	⊥	7.30	4	4	0.46	
join	**0.43**	2.13	⊥	3.26	4	6	1.01	
prependt	**0.01**	**0.01**	3.46	0.15	5	4	0.03	
replacet	**4.02**	10.22	⊥	⊥	4	8	10.94	
searchnodes	**4.28**	43.85	⊥	⊥	6	31	19.68	member
sumnodes	**0.16**	0.43	⊥	0.18	4	3	0.34	
sums	**0.12**	1.26	⊥	0.33	4	5	0.54	
sumtrees	**12.10**	77.21	⊥	⊥	3	5	49.55	

Table A.3 : Benchmark results for programs which process nested structures.

Runtime	Runtime (no deduction)	Runtime (no types)	Runtime (flat costs)	Expert examples	Random examples	Runtime (random)
0.43	2.13	⊥	1.50	4	8	0.93

Table A.4 : Median benchmark results.

Name	Description
add	Add a number to each element of a list.
append	Append an element to a list.
concat	Concatenate two lists together.
dedup	Remove duplicate elements from a list.
droplast	Drop the last element in a list.
dropmax	Drop the largest number(s) in a list.
dupli	Duplicate each element of a list.
evens	Remove the odd numbers from a list.
last	Return the last element in a list.
length	Return the length of a list.
max	Return the largest number in a list.
member	Check whether an item is a member of a list.
multfirst	Replace every item in a list with the first item.
multlast	Replace every item in a list with the last item.
reverse	Reverse a list.
shiftl	Shift all elements in a list to the left.
shiftr	Shift all elements in a list to the right.
sum	Return the sum of a list of integers.

Table A.5 : Descriptions of list processing benchmarks.

Name	Description
count_leaves	Count the number of leaves in a tree.
count_nodes	Count the number of nodes in a tree.
flatten	Flatten a tree into a list.
height	Return the height of a tree.
incrt	Increment each node in a tree by one.
leaves	Return a list of the leaves of a tree.
maxt	Return the largest number in a tree.
membert	Check whether an element is contained in a tree.
selectnodes	Sum the nodes of a tree of integers.
tconcat	Insert a tree under each leaf of another tree.

Table A.6 : Descriptions of tree processing benchmarks.

Name	Description
appendt	Append an element to each node in a tree of lists.
cprod	Return the cartesian product of a list if lists.
dropmins	Drop the smallest number in a list of lists.
flattenl	Flatten a tree of lists into a list.
incrs	Increment each number in a list of lists.
join	Concatenate a list of lists together.
prependt	Prepend an element to each list in a tree of lists.
replacet	Replace one element with another in a tree of lists.
searchnodes	Check if an element is contained in a tree of lists.
sumnodes	Replace each node with its sum in a tree of lists.
sums	For each list in a list of lists, sum the list.
sumtrees	Return the sum of each tree in a list of trees.

Table A.7 : Descriptions of nested structure benchmarks.

Bibliography

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

[2] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *PLDI*, pp. 316–329, 2010.

[3] S. Gulwani, "Dimensions in program synthesis," in *FMCAD*, 2010.

[4] 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.

[5] M. T. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," *STTT*, vol. 15, no. 5-6, pp. 413–431, 2013.

[6] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *POPL*, pp. 317–330, ACM, 2011.

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

[8] E. Kitzelmann, "Analytical inductive functional programming," in *Logic-Based Program Synthesis and Transformation*, pp. 87–102, Springer, 2009.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, pp. 62–73, 2011.

[10] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *PLDI*, pp. 50–61, ACM, 2011.

[11] O. Danvy and M. Spivey, "On barron and strachey's cartesian product function," in *ICFP*, pp. 41–46, 2007.

[12] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *STTT*, vol. 15, no. 5-6, pp. 497–518, 2013.

[13] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *Computer Aided Verification*, pp. 634–651, Springer, 2012.

[14] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *PLDI*, pp. 317–328, ACM, 2011.

[15] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 934–950, 2013.

[16] D. Perelman, S. Gulwani, D. Grossman, and P. Provost, "Test-driven synthesis," in *PLDI*, p. 43, 2014.

[17] O. Polozov and S. Gulwani, "FlashMeta: A Framework for Inductive Program Synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 107–126, ACM, 2015.

[18] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, "A machine learning framework for programming by example," in *ICML*, pp. 187–195, 2013.

[19] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *Approaches and Applications of Inductive Programming, Third International Workshop*, pp. 50–73, 2009.

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

[21] N. Lavrac and S. Dzeroski, "Inductive logic programming.," in *WLP*, pp. 146–160, Springer, 1994.

[22] 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.*, pp. 199–210, 2008.

[23] R. Olsson, "Inductive functional programming using incremental program transformation," *Artif. Intell.*, vol. 74, no. 1, pp. 55–8, 1995.

[24] E. Kitzelmann, "A combined analytical and search-based approach for the inductive synthesis of functional programs," *KI-Künstliche Intelligenz*, vol. 25, no. 2, pp. 179–182, 2011.

[25] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI*, vol. 47, pp. 275–286, ACM, 2012.

[26] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *PLDI*, vol. 48, pp. 27–38, ACM, 2013.

[27] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 48–61, 2005.

[28] P.-M. Osera and S. Zdancewic, "Type-and-example-directed Program Synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 619–630, ACM, 2015.

[29] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program Synthesis from Polymorphic Refinement Types," *arXiv:1510.08419 [cs]*, Oct. 2015. arXiv: 1510.08419.

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

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

[32] S. Chaudhuri, M. Clochard, and A. Solar-Lezama, "Bridging boolean and quantitative synthesis using smoothed proof search," in *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–220, ACM, 2014.

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

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

[35] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “ICE: A robust framework for learning invariants,” in *CAV*, pp. 69–87, 2014.