Zero-One Integer Linear Programming for Program Synthesis

by

Arati Ashok Kaushik, B.Tech.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2016

Zero-One Integer Linear Programming for Program Synthesis

APPROVED BY

SUPERVISING COMMITTEE:

Thomas Dillig, Supervisor

Isil Dillig

Zero-One Integer Linear Programming for Program Synthesis

Arati Ashok Kaushik, M.S.
The University of Texas at Austin, 2016

Supervisor: Thomas Dillig

Program synthesis techniques generate code automatically for a given specification, while code reuse techniques adapt existing code to suit the user's requirements. These methods can be used to help developers implement hard-to-write functions which they find difficult to code by themselves. At the same time, they can also be used to automatically synthesize uninteresting glue code, thereby enabling programmers to concentrate on their own key goals. In this thesis, we describe how 0-1 Integer Linear Programming (ILP) can be utilized for program synthesis and code reuse, by discussing its employment in two recent applications.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Context

In today's programming culture of open-source software, source code is often freely available online. A large proportion of code available on websites like GitHub, Sourceforge, and Bitbucket come with open-source/free software licenses. In fact, programmers are frequently evaluated by peers and prospective employers based on their publicly available projects, thereby providing further incentive to share well-written and properly compilable open-source code. This has also consequently allowed and even encouraged programmers to modify and/or reuse existing code available online in their own applications. At the same time, the API (application program interface) economy has also seen a boom in recent years, making useful functions available for use by developers in several domains. More and more API libraries are being created and released for general use, most libraries providing hundreds or even thousands of functions.

In the general case, developers utilize publicly-available methods in their own code for performing relatively small, specific tasks. However, while both application code and API libraries are now ubiquitous, given the massive

availability of code online, just finding a method relevant to their purpose requires manual inspection of classes, function signatures and their descriptions. This task is both tedious and a huge time-sink, especially if the developer is unfamiliar with the codebase. Moreover, it takes valuable time away from the developer's own code, which is of more interest and importance to them. Programmers do not like to perform mundane tasks over and over again if it can be helped. Automatically detecting the functions of interest from a set of available methods, and using them to perform a specific computing task, can lead to a huge improvement in the productivity of programmers, allowing them to spend their time more efficiently on the larger problem at hand. Program synthesis is a promising technique which can help programmers accomplish this objective.

Programming languages have come a long way since the days of assembly-level coding. Today's programming languages are much more abstract and human-readable, making programming easier and easier, even for the naive user. Even present-day programming languages, however, work by providing the user a fixed number of "building blocks" in the form of explicit instructions, which detail the steps that need to be performed by the computer to fulfil the required task. In general, people find it more intuitive to describe their *intent*, rather than detailing the steps required to accomplish it. In other words, users find it more intuitive to give the *what*, and not the *how*. The current programming model, however, forces users to describe how to perform actions, and is not very conducive to allowing specifications of what it is that

users want to accomplish. Therefore, there is a growing interest in modern programming language research in program synthesis with the aim of making programming more intuitive to users.

Program synthesis refers to the automatic synthesis of program code. The main components of a system that performs program synthesis can be listed as follows:

- Specification
- Synthesizer
- Verifier

The user provides the system with a *specification* of the program that they want to synthesize, which can be in one of several forms including logical specifications, input-output examples, partial programs, or even natural language comments. The *synthesizer* accepts the specification and generates a *candidate program*. The *verifier* verifies the candidate program against the user specifications. If all specifications are satisfied, the program is output to the user as the final synthesized result. A particular form of program synthesis, which is labeled "Counterexample-guided Inductive Synthesis" (CEGIS), requires the synthesizer and verifier to work in tandem in a loop, with the verifier providing feedback in the form of *counterexamples* to the synthesizer if the provided candidate program fails to satisfy the required specifications. This is the primary form of program synthesis that will be discussed hereon.

1.2 Thesis Statement

In this thesis, we describe how to use an existing linear optimization technique, 0-1 Integer Linear Programming (0-1 ILP), to model constraints in program synthesis effectively for even very large problems. 0-1 ILP was utilized and implemented in two program synthesis techniques, one that synthesizes loop-free programs by composing methods from large Java API libraries, and another that reuses existing code from large code bases and adapts it to implement the desired method. Both systems, SyPet and Hunter were group projects developed with Yu Feng, Yuepeng Wang, and Dr. Ruben Martins, and supervised by Dr. Isil Dillig.

1.3 Outline

The rest of this thesis is organized in the following manner. Chapter 2 discusses some relevant background on boolean satisfiability solvers and integer linear programming. Chapter 3 discusses how 0-1 ILP has been incorporated into two Java applications for program synthesis and code reuse. Chapter 4 discusses related work, and Chapter 5 concludes the thesis.

Chapter 2

Technical Background

2.1 SAT solvers

The boolean satisfiability problem, more commonly known as the SAT problem, is the problem of finding whether a boolean formula is satisfiable or not. In other words, it is the problem of determining if there exists an assignment of values true or false to all variables in a boolean formula such that the entire formula becomes equivalent to true. If such a assignment exists, then the formula is said to be *satisfiable*, and the satisfying assignment is called a *model* of the boolean formula.

The SAT problem is one of the first computational complexity problems which was proven to be NP-complete. This means that all problems in the NP class, including several graph reachability algorithms, can be reduced to SAT. SAT solvers, or Boolean Satisfiability solvers, are systems which accept a boolean formula and determine whether it is satisfiable or not. Moreover, if the formula is found to be satisfiable, the solver returns a satisfying assignment, or model, for the formula.

SAT solvers are frequently used in synthesis techniques, since most program constraints can be encoded into boolean formulae with relative ease. One

such synthesis technique is SKETCH [26], which takes a *sketch*, or a program with **holes** with type restrictions, encodes these constraints into boolean formulae, and uses a SAT solver to solve these constraints and fill the sketch. We use a SAT solver in a similar fashion to generate candidate programs from a petri net.

2.2 Integer Linear Programming

An Integer Linear Programming (ILP) problem is a mathematical optimization problem, with an objective function to be maximized/minimized, and a set of constraints to be followed. Both the objective function and constraints are *linear*. Additionally, all variables are restricted to be integers. A general representation of an ILP problem is as follows:

maximize/minimize: $z = c^T x$

subject to: $Ax \leq b,\ x \geq 0$, and $x \in \mathbb{Z}^n$,

where c and b are vectors, x is an n-dimensional vector representing the variables, and A is an integer matrix. $Ax \leq b$ forms the set of constraints that need to be satisfied on the variables, and z represents the objective function which needs to maximized/minimized.

ILP is an *NP-hard* problem. However, the kind of ILP problem which we are interested in, which is 0-1 ILP, is NP-complete. In 0-1 ILP, the variables are constrained to take values of either 0 or 1. SAT problems can be expressed in 0-1 ILP, since the variables are binary. Other interesting problems which

can be formulated as 0-1 ILP problems are the *Traveling Salesman* and *Vertex Cover*.

Chapter 3

Formulation Details for Different Applications

In this section, we will describe three ways in which the 0-1 ILP formulation has been adapted for use in two program synthesis systems - SyPet [3] and Hunter [32].

3.1 SyPet

Component-based synthesis is a form of program synthesis that automatically generates programs (typically loop-free) by assembling them from their constituent base *components*. SyPet is a system for type-directed component-based synthesis from API libraries. It uses the *types* of components as specifications instead of requiring a logical specification describing the functionality of the constituent methods. SyPet takes a method signature, API library, and one or more test cases from the user, and generates the desired method by composing a sequence of method invocations from the provided API library.

Since this technique involves composing method calls to synthesize a required function, the crux of the problem lies in deciding which methods to compose, and how to combine them to generate the specific functionality, while

respecting the types of the language. A natural way to view this problem is to represent the components in the form of a graph. There are several graph representations which could be suitable for this function. The representation employed in SyPet is called a *Petri net* $\mathcal{N}$.

SyPet's architecture consists of the following three modules:

1. Petri net construction
2. Candidate sketch generation
3. Sketch completion

A Petri net $\mathcal{N}$ is a directed graph with two node types, *places* and *transitions* (represented by circles and solid bars respectively in the graph). Each place can contain *tokens* in it, which are considered resources representing availability of elements of that place. Each transition *consumes* a given number of tokens from one or more places, and *produces* tokens of another place. Consumptions are represented as edges to a transition, and productions by edges from a transition, while the number of tokens of a place P consumed or produced by a transition T is denoted by the *edge weight* of the edge between P and T (incoming to and outgoing from T, respectively). SyPet uses Petri nets by modeling places as *types* and transitions as *methods* of a given API library. A sample Petri net of this model is shown in Figure 3.1.

On a high level, the Petri net is first constructed from the target method signature and the components of the provided API. Once constructed, we perform a lazy graph-reachability problem by translating it to a 0-1 ILP problem

for the purpose of finding a feasible path. This path describes a possible sequence of method calls satisfying the signature requirements of the target function, which is then translated to a partial program or *sketch*, with placeholders for function arguments. The generated sketch is then instantiated by performing another 0-1 ILP run to find an assignment of variables to the placeholders, also called *holes*. The completed program is then tested against the user-provided tests, and if any tests fail, the process is repeated after taking action to block the same assignment from being produced again.

For the purposes of this thesis, we will focus on sketch generation and completion, since these are the two steps that employ 0-1 ILP for their implementation. Interested readers are referred to the paper [3] for more details on SyPet.

3.1.1 Candidate Sketch Generation

As briefly described in Section 3.1, the constructed Petri net $\mathcal{N}$ has places representing variable types, and transitions representing methods. An incoming edge from type τ of weight k to a transition T means that function T has k arguments of type τ. An outgoing edge from transition T to type τ means that function T returns the type τ. For example, in Figure 3.1, the method `average` takes two arguments of type `double` and one argument of type `int`, and returns a `double`. Since Petri nets consider places as *resources* to be consumed or produced by transitions, we can think of the `average` method as *consuming* two `double` types and an `int` type to *produce* another `double`

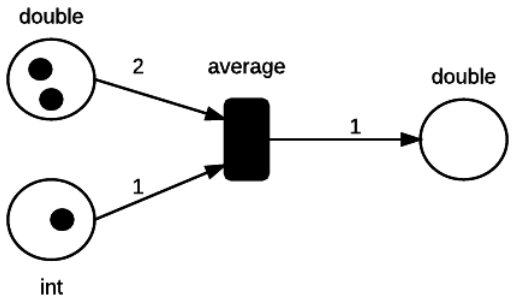
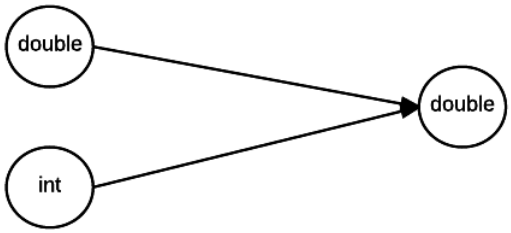


Figure 3.1: Petri net for the `average` method

If the user provides a signature of a function $f(\tau_1, \tau_2, \ldots \tau_n) \rightarrow \tau$, possible solutions to f can be determined by performing reachability analysis from $\tau_1, \tau_2, \ldots \tau_n$ to τ. Since multiple such feasible paths can exist, SyPet performs a *lazy* reachability analysis using 0-1 ILP by assigning *costs* to transitions. Therefore, each run of the reachability analysis algorithm returns a candidate path of the *lowest cost*. Each feasible path represents a sequence of method invocations, which can then easily be converted into a sketch.

To reduce the search space further, $\mathcal{N}$ is reduced to a normal directed graph called the *Induced graph* $\alpha(\mathcal{N})$, which has nodes for each type. Furthermore, $\alpha(\mathcal{N})$ introduces an edge between two types τ_1 and τ_2 only if there exits a transition T in $\mathcal{N}$ with an incoming edge from τ_1 and an outgoing edge to τ_2. For a function $f(\tau_1, \tau_2, \ldots \tau_n) \rightarrow \tau$, paths in $\mathcal{N}$ are considered only if in the corresponding path in $\alpha(\mathcal{N})$, $\tau_1, \tau_2, \ldots \tau_n are all backwards - reachable from \tau$. This is a sound assumption to make, since we want to utilize all input parameter types. The induced graph for the `average` method in Figure 3.1 is shown

below.



To represent the reachability analysis as a 0-1 ILP problem, we need to represent the objective function and constraints on $\mathcal{N}$ as described in Section 2.2. The constraints are encoded to a formula ϕ, representing the graph reachability problem from $\tau_1, \tau_2, \ldots \tau_n$ to τ, computed for increasing path lengths. The objective function is

$$z := min(\Sigma_i c_i x_i),$$

where c_i represents the cost of transition T_i. x_i is a binary variable that equals 1 if transition T_i is included in the path under consideration, and 0 if it is not. Therefore, the objective function for a feasible path is merely the sum of the costs of all components used in the path.

To obtain the cost values, distance metrics are first used to determine the similarity between the desired function and the component T_i. The function name is given particular importance, and the entire javadoc is used to find the similarity between the provided signature and the library function. This similarity measure (between 0 and 1) is then negatively scaled and rounded to the nearest integer to obtain c_i.

Once we obtain a satisfying assignment σ to the 0-1 ILP problem describing a feasible path p, we can easily translate it to a program sketch S. A sketch here is a sequence of method calls with unknown arguments. The following is an example of a sketch for a non-trivial function `rotate` with signature `Area rotate(Area A, Point p, double angle)` generated from the `java.awt.geom` library (taken from the motivating example of the paper):

```
x = #1.getX();
y = #2.getY();
t = new AffineTransform();
#3.setToRotation(#4, #5, #6);
a = #7.createTransformedArea(#8);
return #9;
```

To generate a program sketch S from a candidate solution, we create a new statement for each transition, namely its corresponding API method call. We then pass unknown arguments (denoted as `#i`) to each component. These placeholders for unknown arguments are called *holes*. This sketch construction guarantees type-checking, and ensures that the program is well-formed. However, there may be multiple ways to instantiate the holes in S. For instance, we must assign `#1` and `#2` to `pt`, but we can assign `#4` to either `angle`, `x`, or `y`, since the only requirement is that `#4` is of type `double`.

Once we have S, we use the technique described in Section 3.1.2 to complete the candidate sketch, and validate or invalidate it based on user-provided test cases. The synthesis segment synthesizes the code from the sketch and runs it on the provided test cases. If the tests pass, the program is returned to the user as the solution. If they fail, we need to prevent the system

from returning the same run in future iterations. The simplest way to achieve this is to add the *negation* of the final assignment to the constraint equation ϕ. However, we can block more unsatisfactory runs by performing a partial-order reduction on the current path p. This generates a stronger "blocking clause", which can then by conjoined with ϕ for use in all further iterations.

3.1.2 Sketch Completion

As described earlier, we can obtain a sketch corresponding to a satisfying path p in $\mathcal{N}$ by creating new statements for transitions in the path. Depending on the type of method that is being called, the corresponding statement in the sketch can vary slightly.

- General case: `T_o out = #1.foo(#2, #3, #4, ..., #n)`
- Virtual methods: `T_o out = #1.foo(#2, #3, #4, ..., #n+1)`
- Static method/Constructor: `T_o out = foo(#1, #2, #3, #4, ..., #n)`

Once we have the full sketch, our next task is to instantiate the holes with program variables. We also need the program to have certain properties to ensure that is is *well-formed*, namely:

1. It is well-typed
2. No variable is used before it is defined
3. All variables are used at least once
4. All holes are filled with exactly one variable

We encode these requirements as a propositional formula ψ, by introducing boolean variables of the form $h_v^{\#i}$. A value of 1 for $h_v^{\#i}$ denotes that

	v_1	v_2	...	v_n
h_1	0	1		0
h_2	1	0		0
...				
h_m	0	0		1

Figure 3.2: Matrix representation of constraint variables in Section 3.1.2

hole `#i` in the sketch is instantiated with program variable v. We only introduce $h_v^{\#i}$ if variable v matches the type of hole `#i` (Property 1), and if v is a function parameter or is defined in the code before hole `#i` (Property 2).

Additionally, the following formulae also have to hold to guarantee the remaining two properties:

- $\forall_v \forall_{\#i \in H^v} \sum h_v^{\#i} \geq 1$ (Property 3), where H^v denotes holes with the same type as v
- $\forall_{\#i} \forall_{v \in V^i} \sum h_v^{\#i} = 1$ (Property 4), where V^i denotes variables with the same type as hole `#i`

All constraints can be thought of as the cells of the matrix in Figure 3.2. The cell corresponding to row h_i and column v_j represents variable $h_{v_j}^{\#i}$. The

sum of each row in this matrix must be *exactly* 1, and the sum of each column must be *at least* 1.

Once we have the filled sketch, we can run the user-provided tests on it. If any of the tests fail, we get a different completion for the sketch by getting a model for $\psi \wedge \neg\sigma$ in the next iteration.

3.2 HUNTER

HUNTER is a type-directed code reuse tool implemented as an Eclipse plugin for Java programs. It takes a target method signature, a natural language description of the intended use of the method, and JUnit test cases from the user, and synthesizes the implementation for the function. On a high level, this functionality is similar to SYPET. The key difference is that instead of composing methods from an API library to synthesize the required function, HUNTER searches a large code base for methods which are the *most similar* to the target method, and synthesizes a *wrapper* for this candidate method. In fact, HUNTER uses SYPET internally to synthesize the adaptor code.

HUNTER has three main components in its workflow:

1. Code search
2. Interface alignment
3. Synthesis

The *Code search* component searches a large code base and retrieves a ranked list of methods which are the most *similar* to the target method.

Similarity is measured by calculating similarity metrics between the provided text for the target method (signature and description), and the methods available on the code base (Javadoc and function). The project containing the highest-ranked method is then downloaded and added to the user's current Eclipse project workspace. Next, the interface alignment proceeds to find the best *alignment* between the parameters of the candidate and target functions. This is achieved by formulation to a 0-1 ILP problem and getting a model of the alignment with the lowest *conversion cost* or *adaptation cost*. Once a candidate alignment is retrieved by the system, the *synthesis* part translates it to actual code, which is then verified by running the provided JUnit tests on it. Upon success, the synthesized wrapper code is made available to the user.

3.2.1 Interface Alignment

The interface alignment section of HUNTER is provided the method signature of the target method T, and also that of the highest-ranked candidate method C from the code search section. Let T have the signature

$$(p_1 : \tau_1, p_2 : \tau_2, \ldots p_n : \tau_n) \rightarrow p_0 : \tau_0,$$

and C have the signature

$$(q_1 : \eta_1, q_2 : \eta_2, \ldots q_n : \eta_n) \rightarrow q_0 : \eta_0,$$

where $p_i : \tau_i$ means that the i^{th} parameter of T is of type τ_i, $q_j : \eta_j$ means that the j^{th} parameter of C is of type η_j, and the terms with subscript 0 for both

functions refer to the return values of the functions.

Let M be a *mapping* from the set of parameters P of T to the set of parameters Q of C. We then require M to satisfy some conditions. First, all parameters of the adaptor method T are to be used, as it is presumably the the user's intention to utilize all parameters provided in the target signature. Furthermore, not all parameters of Q need to be used. In particular, one of the benchmark problems HUNTER was tested on, to synthesize a method implementing the Bresenham line-drawing algorithm, had a candidate method requiring `x` and `y` coordinates for both the initial and final points of the line, whereas the target method required only the final point, the initial point being the origin. The same can be true of functions which implement the desired functionality of the target, but require spurious boolean status variables. We allow mapping to such methods by assigning un-mapped parameters in Q one of a predefined set of default values (0/1 for `int`, true/false for `boolean`, etc).

Finally, owing to the presence of custom types and objects in Java, we allow M to be one-many (ex: $p_1 : Point \rightarrow (q_2 : double, q_3 : double)$) or many-one (ex: $(p_2 : double, p_3 : double) \rightarrow q_1 : Point$). However, a many-many mapping does not make sense in the context of interface alignment, and is consequently not allowed. Intuitively, these conditions make sense in terms of program behaviour. For example, let $P = p_1, p_2$, $Q = q_1, q_2$. Then, all the possible mappings are shown below:

- One-one: $x_{1\rightarrow1},\ x_{1\rightarrow2},\ x_{2\rightarrow1},\ x_{2\rightarrow2}$
- Many-one: $x_{(1,2)\rightarrow1},\ x_{(1,2)\rightarrow2}$

- One-many: $x_{1\rightarrow(1,2)},\ x_{2\rightarrow(1,2)}$

Of these, only the ones whose types are compatible with each other are considered. Also, if a parameter is not of a reference/custom type, it cannot have a mapping to/from multiple parameters.

Let x^{p_i} represent the set of all mapping variables which contain mappings from p_i, and x^{q_j} represent the set of all mapping variables which contain a mapping to q_j. Then the constraints on the variables are as below:

- $\forall_{p_i \in P} \Sigma_{x \in x^{p_i}} x = 1$
- $\forall_{q_j \in Q} \Sigma_{x \in x^{q_j}} x \leq 1$
- If the types are incompatible (`int` to `int[]`, etc), then the value of the variable is 0.

The cost of a mapping depends on the type-compatibility of the parameters involved. First, distance between pairs of parameters is calculated, after which they are combined in different manners depending on whether the mapping is one-many or many-one. Finally, the objective function is obtained by multiplying the cost for a mapping with the corresponding variable representing the mapping, and performing a summation over all variables. This was implemented using Sat4J for Java.

Chapter 4

Related Work

4.1 0-1 ILP

Constraint solving in programming is frequently employed in synthesis techniques in the context of program verification. Essentially, constraints on program variables are converted to logical pre- and post-conditions, and once a candidate solution satisfying these constraints is constructed, solvers are used as verification tools to either validate or refute them. Usually, constraints are encoded as boolean propositional formulas or quantified boolean formulas (QBF), with *universal* and *existantial* quantifiers ($\forall$ and $\exists$). To the best of our knowledge, however, 0-1 ILP specifically has not been applied to such techniques aside from SyPet [3] and Hunter [32], discussed in Section 3. SyPet's evaluation marked it as a scalable application, since it can handle up to two orders of magnitude more components than most current state-of-the-art synthesis tools, which can typically oonly handle up to 5-10 components. This allows users to utilize SyPet for synthesis from large APIs. Also, unlike most synthesis tools, SyPet does not require a logical specification from the user describing the target function, or from the API library describing each component. Since Hunter uses SyPet internally, these benefits are translated to Hunter as well. These findings demonstrate that the

ILP encoding for both systems when combined with the other modules, has a positive influence on the systems.

Since this thesis is motivated by the applications of program synthesis and code reuse, and is based on the internal implementation of the SyPet and Hunter systems, we also discuss prior work relevant to both systems. These fall under three broad categories - program synthesis, code reuse, and code completion, each of which is discussed below.

4.2 Program synthesis

As mentioned in Section 3.1, component-based synthesis refers to the synthesis of programs by assembling its constituent base components, like procedures from a given library. In the domain of component-based synthesis, the CodeHint tool [5], which also synthesizes Java code from APIs and takes test cases from the user, is the most similar to SyPet. Several other applications also utilize the component-based synthesis model across a wide range of domains like string and data structure transformations [4, 21], program deobfuscation [15], geometry constructions [8], and bit-vector algorithm construction [7]. Program sketching techniques like Sketch [14, 26–29] require a partial program with holes from the user, and construct the completed code. Programming-by-example approaches (PBE) [1, 4, 6, 11, 25], which generally target novice users, require input-output examples from the developer as partial specifications.

4.3 Code reuse

HUNTER is related to a long line of code search and reuse tools. Of these, S^6 [23], which also utilizes signatures, natural language descriptions and input-output test cases from the user to find relevant code and adapt it, is the tool most similar to HUNTER. The PROSPECTOR tool [17] is also closely related to both SYPET and HUNTER. PROSPECTOR synthesizes snippets of code called *jungloids* by composing functions with single arguments, and non-void return types. Other related tools can be broadly classified into type-directed approaches like PARSEWEB [30], graph-based approaches like SOURCERER [2] and PORTFOLIO [19], test-driven approaches like CODEGENIE [16] and CODECONJURER [13], and textual approaches like CODEEXCHANGE [18].

4.4 Code completion

Both HUNTER and SYPET are related to code completion techniques. While similar to component-based synthesis, code completion tools require a partial program from the user and provide a ranked list of completions, typically single-line. Several automated code completion tools [9, 10, 12, 17, 20, 22, 24, 31, 33] have arisen in recent years. Notable among these are PROSPECTOR [17], INSYNTH [10], and SLANG [22]. INSYNTH also uses theorem proving, while SLANG is based on machine learning techniques.

Chapter 5

Conclusion

Through this thesis, we have accomplished two main objectives. We first motivated the benefits of program synthesis and code reuse. They can help users both implement hard-to-write procedures, and also increase their productivity when utilized for the synthesis of mundane glue code, enabling users to focus on their own code. Next, we explained how the existing technique of 0-1 Integer Linear Programming can be applied to both techniques, by describing its utilization for two applications - SyPet and Hunter in detail.

Bibliography

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, pages 934–950. Springer-Verlag, 2013.

[2] Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, pages 241–259, 2014.

[3] Yu Feng, Yuepeng Wang, Ruben Martins, Arati Ashok Kaushik, and Isil Dillig. Type-directed Component-based Synthesis using Petri Nets. Technical Report TR-16-01, Department of Computer Science, UT-Austin, 2016.

[4] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[5] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663. ACM, 2014.

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

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[8] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61. ACM, 2011.

[9] Tihomir Gvero and Viktor Kuncak. Synthesizing Java expressions from free-form queries. In *OOPSLA*, pages 416–432, 2015.

[10] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[11] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[12] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125. ACM, 2005.

[13] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, pages 45–52, 2008.

[14] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Jsketch: Sketching for Java. In *ESEC/FSE*, pages 934–937. ACM, 2015.

[15] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. IEEE, 2010.

[16] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Videira Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 476–482, 2009.

[17] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61. ACM, 2005.

[18] Lee Martie, Thomas D. LaToza, and André van der Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 24–35, 2015.

[19] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 111–120, 2011.

[20] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286. ACM, 2012.

[21] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, page 43. ACM, 2014.

[22] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, page 44. ACM, 2014.

[23] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253. ACM, 2009.

[24] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM, 2006.

[25] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651. ACM, 2012.

[26] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[27] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178. ACM, 2007.

[28] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[30] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213. ACM, 2007.

[31] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213. ACM, 2007.

[32] Yuepeng Wang, Yu Feng, Ruben Martins, Isil Dillig, and Steven P. Reiss. Type-directed Code Reuse using Integer Linear Programming.

[33] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82. ACM, 2011.

Vita

Arati Ashok Kaushik was born in Chennai, Tamil Nadu, India. She received the degree of Bachelor of Technology in Electronics and Communication Engineering from Amrita Vishwavidyapeetham University, Coimbatore in 2012. She worked for 16 months at Cisco Systems Inc. Bangalore as a Software Engineer, and joined the University of Texas at Austin in August, 2014.

Permanent address: The University of Texas at Austin
Austin, Texas 78712

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.