

YaleNUSCollege

**Towards Enhancing Deductive Synthesis
of Heap-Manipulating Programs
with Examples**

Tan Yao Hong Bryan

**Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences**

Supervised by: Dr. Ilya Sergey

AY 2020/2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

Unrestricted access

Make the Thesis immediately available for worldwide access.

Access restricted to Yale-NUS College for a limited period

Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):

After this period, the Thesis will be made available for worldwide access.

Other restrictions: (please specify if any part of your thesis should be restricted)

Tan Yao Hong Bryan / Saga College

Name & Residential College of Student

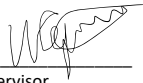


Signature of Student

03 April 2021

Date

Prof. Ilya Sergey



Name & Signature of Supervisor

26 March 2021

Date

Acknowledgements

This Capstone report would not have been possible, had it not been for the support of those who have shaped my experience at Yale-NUS.

First and foremost, I am tremendously grateful to my capstone advisor, Prof. Ilya Sergey, for his kind guidance. SUSLIK was conceived and implemented by himself and his colleagues; my work merely enhances it – a small feat, made infinitely easier for I could not only stand on the shoulders of giants, but also converse, learn, and receive mentorship from the giants themselves.

Secondly, many thanks to everyone who has played a part in my education at Yale-NUS. Plutarch describes in "On Listening" that "the mind is not a vessel that needs filling, but wood that needs igniting". I have benefitted greatly from the wild, unfettered sparks – some strong, others weak – in my liberal arts education. In particular, thank you Prof. Ilya, Prof. Danvy, Prof. Bodin, Prof. Spagnuolo, amongst others, who have bridged the liberal arts with a technical education.

I am eternally indebted to my parents who have spoiled me with love. A liberal arts education in Singapore was unheard of, but they had faith that I knew what I was doing when I took this leap (I didn't, and I still don't, but that's part of the process!). To my brother, I wish you were here.

Last but not least, to Chau, who has been a beacon of light through the tough times. You inspire me to do better, and I thank you for that.

The days are long, but the years are short. This Capstone report concludes my studies at Yale-NUS.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Towards Enhancing Deductive Synthesis of Heap-Manipulating Programs with Examples

by TAN YAO HONG BRYAN

SUSLIK is a state-of-the-art deductive program synthesizer that produces C-style imperative programs from concise user-provided specifications. However, relying on user-provided specifications is problematic: precise specifications are verbose; concise specifications are too loose. Since the synthesizer is dependent on these specifications, program synthesis may result in programs that – while correct – do not perform as intended by the user. This project provides an approach to alleviate this problem: one can provide input-output examples as "hints" to the synthesizer. This Capstone report thus describes the extension of SUSLIK towards example-driven synthesis by contributing an example-driven enhancement of SUSLIK's basic synthesis algorithm, and by also incorporating input-output examples into SUSLIK's SSL rules.

Keywords: Program Synthesis, Separation Logic, Formal Verification, Programming Languages

Contents

Acknowledgements	ii
Abstract	iii
1 Introduction	1
1.1 The Problem with Specifications	2
1.2 Input-Output Examples	3
1.3 Contribution of this Thesis	3
2 Background and Overview	4
2.1 Program Synthesis	4
2.1.1 About Hoare Logic	5
2.1.2 About Separation Logic	5
2.1.3 About Synthetic Separation Logic	6
2.1.4 About SUSLIK’s Synthesis	6
2.1.5 About Program Synthesis in General	7
2.2 Example-Driven Synthesis	8
2.2.1 Example-Driven Synthesis for Functional Programs	10
3 Synthesis with Examples, Algorithmically	12
3.1 SUSLIK’s Deductive Synthesis Algorithm	13
3.1.1 Program Derivation	13

3.1.2	Basic Rules of SSL	15
3.2	Using Examples in a Search for a Program	16
3.2.1	Input-Output Examples, Defined	17
3.2.2	Basic Synthesis Algorithm	19
3.2.3	Example-Driven Synthesis Algorithm	20
4	Synthesis with Examples, Logically	24
4.1	UNIFYHEAPS	24
4.1.1	The cost of Non-determinism	24
4.1.2	Removing unnecessary Non-determinism	25
4.1.3	The cost of removing Non-determinism	25
4.1.4	Re-introducing Non-determinism to UNIFYHEAPS	27
4.2	READ	28
4.2.1	A program that does nothing	28
4.2.2	READEG	29
4.2.3	A program that does something	30
5	Implementation and Case Studies	31
5.1	Running the Project	31
5.2	Case Study 1: fstElement	32
5.2.1	Inductive Heap Predicates	32
5.2.2	Synthesizing fstElement	32
5.3	Case Study 2: treeRightChild	36
5.3.1	Inductive Tree Predicates	36
5.3.2	Synthesizing treeRightChild	36
6	Discussion	38
6.1	Future Work	38

6.1.1	List notation/Storyboard programming	38
6.1.2	Interactive Mode & GUI	38
6.1.3	Optimizing the Search Procedure	39
6.1.4	Rules	39
7	Conclusion	40
	Bibliography	41
A	Logical Form of READEG	45

List of Figures

2.2	SL specification, and graphical description, for <code>listcopy</code>	9
2.3	The synthesized <code>listcopy</code> program	10
2.4	The examples and synthesized program for <code>stutter</code>	11
3.1	Derivation of <code>pick(x,y)</code> as c_1	13
3.2	The synthesized <code>pick</code> program	15
3.3	Basic rules of SSL.	16
4.1	Heap unification rule.	27
5.1	The synthesized <code>fstElement</code> program without examples	34
5.2	Derivation of <code>fstElement(ret)</code> as c_1	35
5.3	The synthesized <code>fstElement</code> program with examples	35
5.4	Derivation of <code>treeRightChild(ret)</code> as c_1	37
5.5	The synthesized <code>treeRightChild</code> program with examples	37
A.1	Logical form of READEG (1).	45
A.2	Logical form of READEG (2).	45

Chapter 1

Introduction

The notion of automatic programming has been considered as a "holy grail" [15] by computer scientists, and for good reason – why work when you can delegate work? One such example of automatic programming is program synthesis: given a user-provided specification, the program synthesizer *constructs* a procedure that satisfies the specification. That is, the role of the user is to simply inform the machine about what they desire; the machine then, through some – to the user – highly magical process, churns out a program that does exactly what the user wants. Successes of the field include FlashFill, a feature of Excel 2013 that can generate spreadsheet table transformations from examples [16], code completion in IDEs [22], and the recent observation that the synthesis of *correct-by-construction* imperative C-like programs with pointers can be implemented in Separation Logic¹ as proof search [21]. This observation, and its tool SUSLIK, forms the basis of this project.

¹A program logic used variously in Facebook's static analysis tool Infer [7], and the Verifiable Software Toolchain (VST) project [1]. Will be elaborated on in [chapter 2](#).

Furthermore, constructing programs in this fashion comes with numerous other benefits beyond appealing to the laziness of the programmer. Software bugs, often extremely costly,² are an inevitable product of programming, especially when we are concerned with stateful programs. Writing good specifications that describe what a program should do is very helpful in this regard – we can quickly verify that our program, for a certain input, satisfies this specification. However, it is significantly more difficult to ensure that this program satisfies the specification for *all* possible program inputs. Consequently, the automatic generation of programs that satisfy – in a fashion that can be proven mathematically – the provided specifications is incredibly appealing.

1.1 The Problem with Specifications

As with most things in life, however, one should not trust in claims that are too good to be true. Program synthesis comes with its own set of issues. This project seeks to remedy one of the most glaring – the trade-off between *strong* and *concise* specifications [9]. Like a mischievous genie, synthesizers tend to provide users with what they *asked* for, rather than what they *meant*: if a specification is verbose, its own formulation may be considerably more difficult than writing the program itself; on the other hand, if it is too concise, the intent of the user might not be captured by the synthesized program. While a deductive program synthesiser like SUSLIK generates *correct-by-construction* programs from their declarative

²One need only look at the Ariane 5 Disaster [10], which resulted because of software trying to cast a 64-bit variable to 16-bit; numerous bugs in Ethereum resulting in losses amounting to around 450 million [2], etc.

specifications, ultimately this "correctness" is reliant on the specification being able to encapsulate completely what the user means [4, 11].

1.2 Input-Output Examples

This project explores one possible solution to this. We extend SUSLIK by allowing it to take concrete, user-provided input-output examples, in addition to an incomplete specification. Consequently, the synthesizer can use these input-output examples to restrict the search space of SUSLIK, and so eliminate the possibility of programs that were semantically unintended by the user. We draw inspiration from the idea of *Storyboard Programming* [24] that allows for the transformation of a visually intuitive graphical representation of an example into a concise textual format.

1.3 Contribution of this Thesis

This thesis enhances SUSLIK with example-based synthesis by extending on SUSLIK's synthesis algorithm to allow for the evaluation of candidate programs with examples, and by incorporating examples into SUSLIK's rules. Notably, these examples should not only be easy to provide, but should also be able to precisely capture the intent of the user in a number of concrete scenarios. The thesis proceeds as follows. Firstly, we present an overview of program synthesis. Then, we discuss our representation of input-output examples, before showing how we extended SUSLIK's basic synthesis algorithm towards example-driven synthesis. Furthermore, we demonstrate how we incorporate examples into SUSLIK's rules. Finally, we conclude with case studies, and discuss future work.

Chapter 2

Background and Overview

The field of program synthesis aims to simplify programming by reducing the task of writing programs to writing specifications. Users express their intention through specifications. Then, a *program synthesizer* attempts to construct a program that provably satisfies the specification.

There are numerous types of program synthesis, ranging from deductive, inductive, type-directed, and example-driven. Here, we build on SUSLIK, an existing deductive synthesizer for programs with pointers. Recent work has enhanced SUSLIK with a post-hoc certification procedure, providing a fully automated interface for certified synthesis [26].

2.1 Program Synthesis

SUSLIK is a *deductive* synthesizer. This means that it begins from a user-provided logical specification. The specification is *declarative* – that is, it describes what the heap, or machine state, of the program should look like before and after a program is run, without saying how to get from one to the other. The program logic used by SUSLIK for its program specifications is Separation Logic, an extension of Hoare Logic [5, 14].

2.1.1 About Hoare Logic

Hoare logic is a formal system that allows one to reason rigorously about the correctness of computer programs [5]. Hoare logic centers around the notion of *Hoare triples*, which are assertions about a machine's state before and after some program is run on it. A Hoare triple takes the form of

$$\{\mathcal{P}\} c \{Q\}$$

where \mathcal{P} is referred to as the precondition, and Q is referred to as the postcondition. The Hoare triple $\{\mathcal{P}\} c \{Q\}$ expresses that whenever we are given a machine state that satisfies \mathcal{P} , the procedure c transforms this machine state such that Q is satisfied, provided that c terminates.

2.1.2 About Separation Logic

However, Hoare logic does not allow for reasoning about stateful programs involving pointer manipulations. Separation Logic (SL) bridges this gap and allows for similarly rigorous reasoning about heap-manipulating programs with pointers [23]. It is based upon the notion of a *separating conjunction*, denoted by $*$. Then, $\mathcal{P} * Q$ asserts that \mathcal{P} and Q hold for separate, disjoint portions of memory.

The separating conjunction has a property that makes it extremely useful – it supports *in-place reasoning*, whereby only a given portion of a given pre/postcondition is updated in-place, mirroring the operational locality of heap update [19]. This makes it amenable to exploitation by the Curry-Howard isomorphism: one can adapt SL to *derive* a program that transforms one state into the other instead of expecting a witness

program c [25]. This observation captures SUSLIK’s approach to program synthesis, through the framework of Synthetic Separation Logic [21].

2.1.3 About Synthetic Separation Logic

Synthetic Separation Logic (SSL) is a system of deductive synthesis rules which prescribes how to decompose SL specifications, while synthesizing the computations by composing the emitted code.

SUSLIK’s deductive synthesis functions via a proof search in a generalized proof system that combines entailment with Hoare-style reasoning for *as of yet unknown programs* [21]. SSL provides a *transforming entailment* judgement $\mathcal{P} \rightsquigarrow \mathcal{Q} | c$. The statement $\mathcal{P} \rightsquigarrow \mathcal{Q} | c$ denotes the existence of a witness program c such that the Hoare triple $\{\mathcal{P}\} c \{\mathcal{Q}\}$ holds, *i.e.*, that the assertion \mathcal{P} transforms into \mathcal{Q} via c . This unifies SL entailment $\mathcal{P} \vdash \mathcal{Q}$ and verification using Hoare logic as $\{\mathcal{P}\} c \{\mathcal{Q}\}$.

2.1.4 About SUSLIK’s Synthesis

Now, we can describe SUSLIK’s formalization of the synthesis problem. Program synthesis from a SL specification \mathcal{P}, \mathcal{Q} amounts to finding a program c , and a derivation of the SSL assertion $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} | c$, where Γ denotes the machine’s environment (set of immutable program variables).

SUSLIK performs a backtracking search in the space of SSL derivations, reducing the specification at each step until we get a *trivial* specification. Finally, the witness program emitted at each step of the SSL rules is composed, and when read in reverse is the synthesized program!

As a taste for this synthesis procedure, consider the specification described in [Equation 2.1](#) for a program that swaps the values stored at two

$\frac{\text{EMP}}{\text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) = \emptyset \quad \vdash \phi \Rightarrow \psi}{\Gamma; \{\phi; \text{emp}\} \rightsquigarrow \{\psi; \text{emp}\} \mid \text{skip}}$ $\frac{\text{READ} \quad a \in \text{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad y \notin \text{Vars}(\Gamma, \mathcal{P}, \mathcal{Q})}{\Gamma \cup \{y\}; [y/a]\{\phi; \langle x, \iota \rangle \mapsto a * P\} \rightsquigarrow [y/a]\{\mathcal{Q}\} \mid c}{\Gamma; \{\phi; \langle x, \iota \rangle \mapsto a * P\} \rightsquigarrow \{\mathcal{Q}\} \mid \text{let } y = *(x + \iota); c}$ $\frac{\text{WRITE} \quad \text{Vars}(e) \subseteq \Gamma \quad e \neq e'}{\Gamma; \{\phi; \langle x, \iota \rangle \mapsto e * P\} \rightsquigarrow \{\psi; \langle x, \iota \rangle \mapsto e * Q\} \mid c}{\Gamma; \{\phi; \langle x, \iota \rangle \mapsto e' * P\} \rightsquigarrow \{\psi; \langle x, \iota \rangle \mapsto e * Q\} \mid * (x + \iota) = e; c}$ $\frac{\text{FRAME} \quad \text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) \cap \text{Vars}(R) = \emptyset}{\Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c}{\Gamma; \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R\} \mid c}$	$\frac{\text{EMP with } c_7 = \text{skip}}{\{x, y, a_2, b_2\}; \{\text{emp}\} \rightsquigarrow \{\text{emp}\} \mid c_6 = c_7}$ $\frac{\text{FRAME}}{\{x, y, a_2, b_2\}; \{y \mapsto a_2\} \rightsquigarrow \{y \mapsto a_2\} \mid c_6}$ $\frac{\text{WRITE}}{\{x, y, a_2, b_2\}; \{y \mapsto b_2\} \rightsquigarrow \{y \mapsto a_2\} \mid c_5}$ $\frac{\text{FRAME}}{\{x, y, a_2, b_2\}; \{x \mapsto b_2 * y \mapsto b_2\} \rightsquigarrow \{x \mapsto b_2 * y \mapsto a_2\} \mid c_4}$ $\frac{\text{WRITE}}{\{x, y, a_2, b_2\}; \{x \mapsto a_2 * y \mapsto b_2\} \rightsquigarrow \{x \mapsto b_2 * y \mapsto a_2\} \mid c_3}$ $\frac{\text{READ}}{\{x, y, a_2\}; \{x \mapsto a_2 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a_2\} \mid c_2}$ $\frac{\text{READ}}{\{x, y\}; \{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid c_1}$
(A) Basic rules of SSL.	(B) Derivation of swap(x, y)

pointers. SUSLIK derives the witness program `swap` that satisfies [Equation 2.1](#) as follows. In [Figure 2.1a](#), we have the basic SSL inference rules.

$$\{x \mapsto a * y \mapsto b\} \text{ void swap}(\text{loc } x, \text{loc } y) \{x \mapsto b * y \mapsto a\} \quad (2.1)$$

[Figure 2.1b](#) shows the derivation of the program using these rules from the bottom-up. At each rule application, we have some emitted code c_i . Composing the code from the bottom-up, we have the witness program c , *i.e.*, the `swap` procedure, that satisfies the specification in [Equation 2.1](#). SUSLIK’s synthesis procedure will be examined in greater detail in [chapter 3](#), but for now, [Figure 2.1b](#) provides a flavour of how the process is orchestrated.

2.1.5 About Program Synthesis in General

Up to now, we have only described *deductive* synthesis, a prominent approach to program synthesis.

This process is *deductive* because we repeatedly apply deductive inference rules to formally prove that a witness program that satisfies the provided declarative specification exists [18]. Via the Curry-Howard isomorphism between programs and constructive proofs, a deductive synthesizer like SUSLIK then extracts, for a given specification, the witness program from the proof derivation. Owing to the soundness of this process, we know that the witness program must therefore be *correct-by-construction* according to the specification.

On the other hand, *inductive* synthesis takes as its specification a set of example program executions [13]. These example program executions specify behavior for only a subset of all possible inputs. The synthesizer then inductively generalizes them into a program that can handle all such possible inputs. One can immediately see the problem with such an approach: we cannot represent all possible inputs, and so specifications are *necessarily incomplete*. Thus, the synthesizer is prone to generating erroneous (in both senses of being wrong *wrt.* the provided specification, and of not matching the user's intent) programs. Nevertheless, for domain-specific use-cases like automated programming of bit-vectors, or string transformations in spreadsheets [16, 17], such an approach is useful since defining a complete specification is intractable, and yet, up to a certain threshold, inaccurate synthesis might still be acceptable.

2.2 Example-Driven Synthesis

Example-driven synthesis is a form of inductive synthesis; its goal is to generate a program that matches a given set of input-output examples.

$$\{r \mapsto x * ls(x, S)\} \text{listcopy}(r) \{r \mapsto y * ls(x, S) * ls(y, S)\}$$

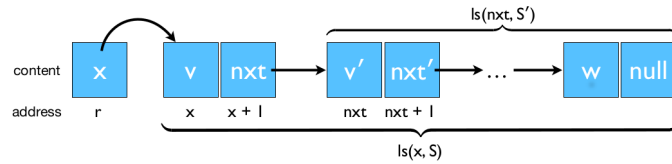


FIGURE 2.2: SL specification, and graphical description, for `listcopy`

To see why this enhancement to SUSLIK might be useful, consider a procedure `listcopy` that duplicates a list [9]. [Figure 2.2](#) describes its SL specification, and a graphical description. The predicate $ls(x, S)$ in the pre- and postcondition denotes that we have a symbolic heap corresponding to a linked list, with the head at pointer x , containing elements from the set S . Then, the postcondition simply describes the expected behavior of our `listcopy` function: the final heap, in addition to containing the original linked list, will also contain another linked list starting from y with elements from S .

Observe that the specification is deliberately concise: we do not specify anything about the structure of the linked lists, nor do we place any constraints on the elements in the list (except that they belong to S). It is our hope that the synthesizer is intelligent enough to avoid these pitfalls.

SUSLIK synthesizes the program described in [Figure 2.3](#). It is easy to check that the program satisfies the ascribed specification in [Figure 2.2](#). Furthermore, it correctly duplicates the original linked list, preserving the ordering of the contents. However, there is something strange about the structure of the resulting linked lists – the tails of each consecutive node in the two linked lists are swapped! While the synthesized program satisfies the specification, and is even correct in certain settings, in

```
1 void listcopy (loc r) {
2   let x = *r;
3   if (x = 0) {
4   } else {
5     let v = *x;
6     let nxt = *(x + 1);
7     *r = nxt;
8     listcopy(r);
9     let y1 = *r;
10    let y = malloc(2);
11    *(x + 1) = y1;
12    *r = y;
13    *(y + 1) = nxt;
14    *y = v;
15  } }
```

FIGURE 2.3: The synthesized listcopy program

a concurrent setting this program is actually incorrect: $ls(x, S)$ cannot be depended upon to remain unchanged.

Recent work has introduced a number of SL extensions that allow for read-only annotations on the symbolic heap [3, 6, 8], the most relevant of which directly extends SUSLIK [9]. While providing annotations alleviates the problem of program specification, one must still perform the identical task of *specifying exactly* which segments are to be made read-only. A more intuitive solution might be to allow users to provide examples of what they *mean*. This is exactly the task of this thesis.

2.2.1 Example-Driven Synthesis for Functional Programs

Program synthesis – deductive, example-driven, or otherwise – has typically operated in the domain of functional programs involving structured data, recursion, and higher-order functions in typed programming languages. Purely functional programs treat all functions as pure, and so do not have any side-effects [12, 20]. In contrast, impure procedures can have side-effects like modifying the program state and mutable data.

Consequently, functional programs tend to be easier to reason about and formally verify. A corollary of this is that program synthesis for purely functional programs tends to be simpler. There exist numerous examples of example-driven program synthesis [12, 20], but most of them operate in the domain of functional programs. *Storyboard programming* is an exception that works for data-structures with pointers [24]. However, it does not guarantee that its synthesized programs are *correct-by-construction*, while SUSLIK does. Existing works have demonstrated that using examples can prune the search space of the synthesis procedure, enabling efficient synthesis of non-trivial functional programs [20]. Consider the following toy example expressed in OCaml, a functional language.

```
1      (* Goal type refined by input/output ‘‘example worlds’’ *)
2      let stutter : list -> list |>
3      { [] => []
4        | [0] => [0;0]
5        | [1;0] => [1;1;0;0]
6      } = ?
7      (* Output: synthesized implementation of stutter *)
8      let stutter : list -> list =
9      let rec f1 (l1:list) : list =
10     match l1 with
11     | Nil -> l1
12     | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
13     in f1
```

FIGURE 2.4: The examples and synthesized program for stutter

By treating examples as “example worlds” (*i.e.* a hypothetical evaluation of `stutter`), candidate terms are evaluated early in the search process, pruning the search space dramatically [20]. Furthermore, this allows one to extract type and example information to synthesize recursive functions efficiently. This idea forms the basis for how we seek to prune our search space and reject candidate programs based on examples.

Chapter 3

Synthesis with Examples, Algorithmically

In this chapter, we discuss the extension of SUSLIK towards example-driven synthesis.

First, we deconstruct synthesis by discussing the derivation steps involved in synthesizing `pick`, our motivating example for this chapter. `pick` is a simple procedure that, given two pointers, assigns an arbitrary constant as their referends. `pick` was chosen because some of the rules involved in synthesizing it are *non-deterministic*. By taming non-determinism, examples can guide the synthesizer towards a particular program.

Having seen synthesis in action, we then elaborate on SUSLIK's synthesis algorithm. Finally, we present our extension, which uses examples to prune the search space of SUSLIK.

3.1 SUSLIK'S DEDUCTIVE SYNTHESIS ALGORITHM

Consider the following specification for `pick`:

$$\{x \mapsto a * y \mapsto b\} \text{ void pick}(\text{loc } x, \text{loc } y) \{x \mapsto z * y \mapsto z\} \quad (3.1)$$

3.1.1 Program Derivation

We have the resulting program derivation for `pick`:

$$\begin{array}{c}
 \frac{}{\{x, y, a2, b2\}; \{ \text{emp} \} \rightsquigarrow \{ \text{emp} \}} \text{EMP with } c_8 = \text{skip} \\
 c_7 = c_8 \\
 \frac{}{\{x, y, a2, b2\}; \{ x \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 \}} \text{FRAME} \\
 c_6 = *x = b2; c_7 \\
 \frac{}{\{x, y, a2, b2\}; \{ x \mapsto a2 \} \rightsquigarrow \{ x \mapsto b2 \}} \text{WRITE} \\
 c_5 = c_6 \\
 \frac{}{\{x, y, a2, b2\}; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto b2 \}} \text{FRAME} \\
 c_4 = c_5 \\
 \frac{}{\{x, y, a2, b2\}; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ b2 = z; x \mapsto z * y \mapsto b2 \}} \text{SUBSTRIGHT} \\
 c_3 = c_4 \\
 \frac{}{\{x, y, a2, b2\}; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \}} \text{UNIFYHEAPS} \\
 c_2 = \text{let } b2 = *y; c_3 \\
 \frac{}{\{x, y, a2\}; \{ x \mapsto a2 * y \mapsto b \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \}} \text{READ} \\
 c_1 = \text{let } a2 = *x; c_2 \\
 \frac{}{\{x, y\}; \{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \}} \text{READ} \\
 c_1
 \end{array}$$

FIGURE 3.1: Derivation of `pick(x, y)` as c_1 .

Before we go into detail about the rules used at each step, let us first traverse the program derivation of the specification described in [Equation 3.1](#). [Figure 3.1](#) shows the derivation of `pick` using SSL rules, namely FRAME, EMP, READ, WRITE, UNIFYHEAPS, and SUBSTRIGHT. The proof tree should be read bottom-up, and each subgoal's witness program is named c_i , where c_1 corresponds to `pick`'s body. Each intermediate subgoal has part of its specification highlighted in `gray boxes`, which are the

parts of the specification used in the particular SSL rule. The goal of the synthesis process is to "empty" the spatial components of the pre/post condition, so that we can conclude the derivation via the EMP rule.

Beginning from the SL specification (Equation 3.1), the referends of the program variable x are first READ in, turning into a new program variable $a2$ that is fresh in the original goal. This produces the witness program `let a2 = *x`. The same is done for the referends of the program variable y , producing the witness program `let b2 = *y`. Next, observe that in the spatial component of the postcondition, we have an existential variable z (i.e., a variable in the postcondition that doesn't appear in the precondition). Thus, the postcondition allows x and y to point to *any* value, so long as they point to the *same* value. Because z can be any arbitrary value, a unifying substitution σ must be found for some heaplets of the pre- and postcondition. This is achieved with help of the UNIFYHEAPS rule, which *non-deterministically* selects a heaplet and a unifying substitution. Taming this non-determinism will be significant in extending SUSLIK with example-driven synthesis, as will be explained later.

We make a simplifying assumption that UNIFYHEAPS selects the pointer y and its referend, and unifies it with $b2$. Note that, as it is non-deterministic, UNIFYHEAPS could have very well selected the heaplet $x \mapsto z$ in the postcondition, or chose to unify with $x \mapsto a2$ in the precondition instead. Now, instead of pointing to z , since UNIFYHEAPS has unified z with $b2$, $y \mapsto z$ is replaced with $y \mapsto b2$, and $b2 = z$ is introduced into the pure component of the postcondition. The unifying substitution $\sigma := [z = b2]$ is then applied with the SUBSTRIGHT rule to the postcondition, thereby also transforming $x \mapsto z$ to $z \mapsto b2$.

Finally, the matching heaplets $y \mapsto b2$ in the pre- and postcondition can be eliminated with `FRAME`, leaving us with $x \mapsto a2$ in the precondition and $y \mapsto b2$ in the postcondition. Applying `WRITE` and `FRAME` gives us empty heaplets in the pre- and postcondition together with the witness program for `WRITE`, $*x = b2$, and a simple application of `EMP` concludes the program derivation. This gives us the synthesized program c_1 , described in [Figure 3.2](#).

```

1 void pick (loc x, loc y) {
2   let a2 = *x;
3   let b2 = *y;
4   *x = b2;
5 }
```

FIGURE 3.2: The synthesized pick program

3.1.2 Basic Rules of SSL

The basic rules that were used in the program derivation are represented formally in [Figure 3.3](#) [21].

The `EMP` rule is applicable when we have empty heaplets in the pre- and postcondition, requiring that no existentials remain in the goal and that the pure precondition implies the pure postcondition. This pure formulae reasoning (the premise $\vdash \phi \Rightarrow \psi$) is resolved via an invocation of an oracle. `EMP` is a terminal rule, and thus its witness program is `skip`.

For brevity, we focus on the `UNIFYHEAPS` rule. The full Zoo of SSL rules, along with its formal treatment, can be found at [21]. `UNIFYHEAPS` deals with existentials in the heap by attempting to find a unifying substitution σ for *some* sub-heaplets of the pre- and postcondition. Because it can freely choose a sub-heap and a unifying substitution (with the restriction

$$\begin{array}{c}
\text{EMP} \\
\frac{\text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) = \emptyset \quad \vdash \phi \Rightarrow \psi}{\Gamma; \{\phi; \text{emp}\} \rightsquigarrow \{\psi; \text{emp}\} | \text{skip}} \\
\\
\text{READ} \\
\frac{a \in \text{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad y \notin \text{Vars}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad \Gamma \cup \{y\}; [y/a]\{\phi; \langle x, \iota \rangle \mapsto a * \mathcal{P}\} \rightsquigarrow [y/a]\{\mathcal{Q}\} | c}{\Gamma; \{\phi; \langle x, \iota \rangle \mapsto a * \mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} | \text{let } y = *(x + \iota); c} \\
\\
\text{WRITE} \\
\frac{\text{Vars}(e) \subseteq \Gamma \quad e \neq e' \quad \Gamma; \{\phi; \langle x, \iota \rangle \mapsto e * \mathcal{P}\} \rightsquigarrow \{\psi; \langle x, \iota \rangle \mapsto e * \mathcal{Q}\} | c}{\Gamma; \{\phi; \langle x, \iota \rangle \mapsto e' * \mathcal{P}\} \rightsquigarrow \left. \begin{array}{l} \{\psi; \langle x, \iota \rangle \mapsto e * \mathcal{Q}\} \\ \end{array} \right| *(x + \iota) = e; c} \\
\\
\text{FRAME} \\
\frac{\text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) \cap \text{Vars}(R) = \emptyset \quad \Gamma; \{\phi; \mathcal{P}\} \rightsquigarrow \{\psi; \mathcal{Q}\} | c}{\Gamma; \{\phi; \mathcal{P} * R\} \rightsquigarrow \{\psi; \mathcal{Q} * R\} | c} \\
\\
\text{UNIFYHEAPS} \\
\frac{[\sigma]R' = R \quad \text{frameable}(R') \quad \emptyset \neq \text{dom}(\sigma) \subseteq \text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad \Gamma; \{\mathcal{P} * R\} \rightsquigarrow [\sigma]\{\psi; \mathcal{Q} * R'\} | c}{\Gamma; \{\phi; \mathcal{P} * R\} \rightsquigarrow \{\psi; \mathcal{Q} * R'\} | c} \\
\\
\text{SUBSTRIGHT} \\
\frac{x \in \text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad \Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow [\psi'/x]\{\psi \wedge x = \psi', \mathcal{Q}\} | c}{\Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\psi \wedge x = \psi'; \mathcal{Q}\} | c}
\end{array}$$

FIGURE 3.3: Basic rules of SSL.

that the domain of σ contains only existentials), it introduces significant non-determinism into the synthesis procedure. Concretely, this means that the synthesis algorithm has to incorporate backtracking in order to handle the non-determinism.

Thus, one key insight for extending SUSLIK towards example-driven synthesis is that this non-determinism can be tamed by evaluating candidate programs and only accepting the program that satisfies our examples.

3.2 Using Examples in a Search for a Program

The non-determinism in the synthesis process comes from numerous sources: the order in which rules are applied, rules that have to make choices, and

so on.

Whenever the synthesizer runs into an unsolvable goal, it *backtracks* and tries to make another choice. However, some strand of nondeterminism (e.g., arising from UNIFYHEAPS) might lead to a goal that is solvable, since the nondeterminism might be coincidental. We make use of this to incorporate examples into the synthesis process – by lazily enumerating all possible programs on-demand, we can evaluate each candidate program with respect to our examples, only accepting a program that satisfies our examples. We check that a candidate program satisfies our input-output examples by writing a concrete interpreter. Given a candidate program, this interpreter has its machine state – a concrete heap – instantiated with the input example. Then, once the program is evaluated, the resulting machine state should be equivalent to the output example provided.

3.2.1 Input-Output Examples, Defined

We define the form of our input-output examples. These input-output examples reflect the idea of "example worlds" from [subsection 2.2.1](#).

An input-output example has three components. Firstly, it has an execution environment σ that maps variables to natural numbers. Since a heap in the heaplet model is represented as a finite function from natural numbers to integers [19], σ keeps track of the distinct memory location in the heap that is associated with each variable that is a pointer. Secondly, it has an *input* heap. The input heap maps from a memory address (an integer) to a value. The input heap represents the heap prior to running a

candidate program. Finally, it has an *output* heap, defined the same way. The output heap represents the heap *after* running a candidate program.

With these three components, for every pointer variable, we have an underlying map that stores a concrete memory address associated with the pointer, and the value at the memory address of the pointer. Thus, we are able to keep track of the machine state as we iterate through the program. The memory address associated with the pointer is only important insofar as it allows us to keep track of machine state. Hence, the specific memory address value is irrelevant, and so we can mechanically generate the execution environment σ from the input and output heap.

A user would present their input-output examples in a form resembling SL specifications, except we have square parentheses instead of curly parentheses to reflect that these are examples, and no function declaration is necessary. For instance, a user would input

$$[x \mapsto c * \dots * y \mapsto d] [x \mapsto e * \dots * y \mapsto f] \quad (3.2)$$

where c, d, e, f are values, as an input-output example. This example will get parsed into the three components mentioned above, with pointers x and y allocated arbitrary memory addresses. Then, the program state of a candidate program would be instantiated as a heap with the memory address associated with x mapping to c , and with y mapping to d . After running a candidate program, the program is considered to be "accepted" if the final machine state, or heap, of the interpreter resembles the output component of the example.

Algorithm 3.2.1: `synthesize` ($\mathcal{G} : \text{Goal}, \text{rules} : \text{Rule}^*$)

<p><i>Input:</i> Goal $\mathcal{G} = \langle f, \Sigma, \Gamma, \{\mathcal{P}\}, \{\mathcal{Q}\} \rangle$</p> <p><i>Input:</i> List <i>rules</i> of available rules to try</p> <p><i>Result:</i> Terminating program c, such that $\Sigma; \Gamma; \{\mathcal{P}\}c\{\mathcal{Q}\}$ is valid</p> <pre> 1 function synthesize ($\mathcal{G}, \text{rules}$) = 2 withRules($\text{rules}, \mathcal{G}$) 3 function withRules (rs, \mathcal{G}) = 4 match rs 5 case $[] \Rightarrow \text{Fail}$ 6 case $\mathcal{R} :: rs' \Rightarrow$ 7 match $\mathcal{R}(\mathcal{G})$ 8 case $\perp \Rightarrow \text{withRules}(rs')$ 9 case $\text{subderivs} \Rightarrow$ 10 tryAlts($\text{subderivs}, \mathcal{R}, rs', \mathcal{G}$) </pre>	<pre> 11 function tryAlts ($\text{derivs}, \mathcal{R}, rs, \mathcal{G}$) = 12 match derivs 13 case $[] \Rightarrow \text{withRules}(rs, \mathcal{G})$ 14 case $\langle \text{goals}, \mathcal{K} \rangle :: \text{derivs}' \Rightarrow$ 15 match solveSubgoals($\text{goals}, \mathcal{K}$) 16 case Fail $\Rightarrow \text{tryAlts}(\text{derivs}', \mathcal{R}, rs, \mathcal{G})$ 17 case $c \Rightarrow c$ 18 function solveSubgoals ($\text{goals}, \mathcal{K}$) = 19 $cs := []$ 20 $\text{pickRules} = \text{AllRules}$ 21 for $\mathcal{G} \leftarrow \text{goals}; c \neq \text{Fail}$ 22 $c = \text{synthesize}(\mathcal{G}, \text{pickRules}(\mathcal{G})); \text{do}$ 23 $cs := cs ++ [c]$ 24 if $cs < \text{goals}$ then Fail else $\mathcal{K}(cs)$ </pre> <hr/>
--	---

3.2.2 Basic Synthesis Algorithm

The SUSLIK synthesizer performs a goal-directed backtracking search over the space of all valid SSL derivations. We omit the discussion of optimizations in favour of the basic synthesis algorithm.

Algorithm 3.2.1 [21] describes the pseudocode of SUSLIK's synthesis procedure. The algorithm is comprised of four mutually-recursive functions:

- `synthesize` ($\mathcal{G}, \text{rules}$) is invoked on a goal, together with all the SSL rules. It passes control to the first auxiliary function, `withRules`.
- `withRules` (rs, \mathcal{G}) iterates through the list rs of remaining rules, trying to apply each one to the goal \mathcal{G} .

A successful application of rule \mathcal{R} results in one or more *sub-derivations* subderivs . A sub-derivation is a pair: the first component contains zero or more sub-goals which must all be solved, and the second component is a continuation \mathcal{K} that combines the results of solving subgoals into a final program. This is then passed to `tryAlts`.

On the other hand, if it fails, we try the next rule from the list rs' . If there are no rules left, then synthesis for the current goal has failed.

- `tryAlts` ($derivs, \mathcal{R}, rs, \mathcal{G}$) recursively processes the possible sub-derivations $derivs$ generated by \mathcal{R} .

On each sub-derivation, `solveSubgoals` is invoked to solve all the sub-goals $goals$ and apply the continuation \mathcal{K} that constructs the candidate program. If `solveSubgoals` is successful (line 17), the program c is returned.

If there are no sub-derivations left to try (line 13), we employ mutual recursion, calling `withRules` on the remaining rules rs .

- Finally, `solveSubgoals` ($goals, \mathcal{K}$) attempts to solve the provided set of subgoals.

It does this by calling `synthesize` recursively. This propagates the search problem a level deeper. If no goals fail, their results are combined via \mathcal{K} .

Observe that a successful application of a rule results in one or more sub-derivations. This reflects the non-determinism present in rules like `UNIFYHEAPS`.

3.2.3 Example-Driven Synthesis Algorithm

The extensions to the synthesis algorithm to incorporate examples are described in Algorithm 3.2.2. The gray boxes highlight parts of the pseudocode that were explicitly changed; note, however, that the algorithm now also takes as input the list of input-output examples described in

Algorithm 3.2.2: `synthesizeWithEg` ($\mathcal{G} : \text{Goal}, \text{rules} : \text{Rule}^*, \text{egs} : \text{examples}$)

<p><i>Input:</i> Goal $\mathcal{G} = \langle f, \Sigma, \Gamma, \{\mathcal{P}\}, \{\mathcal{Q}\} \rangle$ <i>Input:</i> List <i>rules</i> of available rules to try</p> <pre> 1 function synthesize ($\mathcal{G}, \text{rules}, \text{egs}$) = 2 match evaluateEgs(withRules(<i>rules</i>, \mathcal{G}, <i>egs</i>), <i>egs</i>) 3 case [] \Rightarrow Fail 4 case $c :: cs' \Rightarrow c$ 5 function withRules (<i>rs</i>, \mathcal{G}, <i>egs</i>) = 6 match <i>rs</i> 7 case [] \Rightarrow Fail 8 case $\mathcal{R} :: rs' \Rightarrow$ 9 match $\mathcal{R}(\mathcal{G})$ 10 case $\perp \Rightarrow$ withRules(<i>rs'</i>, \mathcal{G}) 11 case <i>subderivs</i> \Rightarrow 12 tryAlts(<i>subderivs</i>, \mathcal{R}, <i>rs'</i>, \mathcal{G}, <i>egs</i>) 13 function tryAlts (<i>derivs</i>, \mathcal{R}, <i>rs</i>, \mathcal{G}, <i>egs</i>) = 14 match <i>derivs</i> 15 case [] \Rightarrow withRules(<i>rs</i>, \mathcal{G}, <i>egs</i>) 16 case $\langle \text{goals}, \mathcal{K} \rangle :: \text{derivs}' \Rightarrow$ 17 match solveSubgoals(<i>goals</i>, \mathcal{K}, <i>egs</i>) 18 case Fail \Rightarrow 19 tryAlts(<i>derivs'</i>, \mathcal{R}, <i>rs</i>, \mathcal{G}, <i>egs</i>) 20 case $c \Rightarrow$ 21 $c :: \text{tryAlts}(\text{derivs}', \mathcal{R}, \text{rs}, \mathcal{G}, \text{egs})$ </pre>	<p><i>Input:</i> List <i>egs</i> of examples, Interpreter \mathcal{I} <i>Result:</i> Terminating program <i>c</i>, such that $\Sigma; \Gamma; \{\mathcal{P}\}c\{\mathcal{Q}\}$ is valid</p> <pre> 22 function solveSubgoals (<i>goals</i>, \mathcal{K}, <i>egs</i>) = 23 <i>cs</i> := [] 24 <i>pickRules</i> = AllRules 25 for $\mathcal{G} \leftarrow \text{goals}; c \neq \text{Fail}$ 26 $c = \text{synthesize}(\mathcal{G}, \text{pickRules}(\mathcal{G}), \text{egs});$ do 27 <i>cs</i> := <i>cs</i> ++ [<i>c</i>] 28 if $cs < \text{goals}$ then Fail else $\mathcal{K}(cs)$ 29 function evaluateEgs (<i>cs</i>, <i>egs</i>) = 30 <i>ls</i> := [] 31 for $c \leftarrow cs$ do 32 <i>p</i> := True 33 for $e \leftarrow \text{egs}$ do 34 <i>p</i> := <i>p</i> \wedge (evaluate (<i>c</i>, <i>e</i>)) 35 if <i>p</i> then <i>ls</i> := <i>c</i> ++ <i>ls</i> 36 <i>ls</i> 37 function evaluate (<i>c</i>, <i>e</i>) = 38 match <i>e</i> 39 case (<i>store</i>, <i>in_heap</i>, <i>out_heap</i>) \Rightarrow 40 <i>c_heap</i> := $\mathcal{I}(c, \text{in_heap}, \text{store})$ 41 <i>out_heap</i> $\subseteq c_heap$ 42 </pre>
---	--

subsection 3.2.1, *egs*, along with an interpreter that can be invoked to interpret the C-style program generated by SUSLIK's synthesis procedure.

- `tryAlts` now returns an enumeration of all possible candidate programs obtainable from a set of subderivations and goals, represented as a list of programs. This is because we want to prune the set of all possible candidate programs using our examples. While this may appear inefficient, in the actual implementation, optimizations like lazy enumeration and memoization feature heavily to ensure that we evaluate – on demand – each candidate program as they are constructed, instead of all at once.
- The interpreter \mathcal{I} takes a candidate program, an input concrete heap,

and an execution environment. The candidate program is written in a simple imperative C-style language, generated from SUSLIK. The syntax and semantics of this language can be found in Section 3, [21]. \mathcal{I} evaluates the candidate program according to the semantics of this language, but does so concretely instead of performing symbolic execution by instantiating the program with a concrete input heap. Then, since the language deliberately has no `return` statement, the interpreter returns the program state (heap) after the program has been interpreted as output.

- `evaluate` takes a candidate program and an example, and returns a Boolean value depending on whether or not the program satisfies the example. Recall that, from [subsection 3.2.1](#), an input-output example comprises of an execution environment σ , an input heap, and an output heap. `evaluate` invokes the interpreter for the program, \mathcal{I} , on the candidate program, instantiating it with the input heap. The interpreter then returns a final heap corresponding to the final state of the program. If the example's output heap is a subset of the final heap, the program *satisfies* the example, and so we return `True`. Else, we return `False`.
- `evaluateEgs` takes in two parameters: a list of candidate programs, and a list of input-output examples. It iterates through every candidate program. If the candidate program satisfies *all* the provided examples, then we add the program to our list of successful programs. Having iterated through all candidate programs, it returns the list of successful programs.

- Finally, synthesize. As in [Algorithm 3.2.1](#), synthesize gets the result from withRules. However, because of our changes to tryAlts, withRules now returns a *list* of programs reflecting all candidate programs instead of just *one* program. This list of programs is passed to our evaluation pipeline, starting from evaluateEgs, together with our list of examples. If an empty list is returned by evaluateEgs signifying that no candidate programs satisfy the examples, then we return Fail. Otherwise, if evaluateEgs returns a list of programs that satisfy the examples, then we take the head of the list and return that program as the synthesized program.

With these changes to SUSLIK's synthesis procedure, SUSLIK can now incorporate examples by pruning the space of candidate programs in its backtracking program search. This allows users to provide examples to guide the synthesis procedure. Notably, the examples can be under-specified, since the parser and interpreter can work together to automatically generate a concrete heap for each program as described in [subsection 3.2.1](#). Furthermore, as evaluate checks that the example's output heap is a *subset* of the final heap, the output heap component of each example has the flexibility of being loosely or tightly defined, allowing for examples to be *under-specified*.

This extension to SUSLIK is naive, functioning by pruning an expanded search space; yet, it forms the foundation for an example-driven synthesis of heap-manipulating programs. However, the algorithm does not describe how we can use the examples to *inform the SSL rules*. [Chapter 4](#) discusses how we can do this, providing another level of sophistication in the integration of examples into the synthesis procedure.

Chapter 4

Synthesis with Examples, Logically

In this chapter, we discuss how input-output examples can be incorporated into SSL rules. With "example-aware" SSL rules, we can have "example-aware" program derivations.

4.1 UNIFYHEAPS

Example-driven synthesis relies on the presence of non-determinism, since it allows for more than one possible program to be generated by the synthesis procedure before being pruned based on whether the candidate programs satisfy the examples.

4.1.1 The cost of Non-determinism

Accounting for non-determinism is computationally taxing: because the synthesis algorithm explores the space of all valid SSL derivations that

are rooted at the initial synthesis goal, non-determinism causes an exponential blowup. Thus, non-determinism that is coincidental to generation of correct programs can be removed to improve the runtime of SUSLIK.

4.1.2 Removing unnecessary Non-determinism

As we see in [subsection 3.1.1](#) and [subsection 3.1.2](#), UNIFYHEAPS introduces significant non-determinism as it is free to choose a sub-heap and a unifying substitution. Typically, non-determinism – as used in the sense of a non-deterministic finite automaton – is harnessed via a complete enumeration of all possible outcomes, since it is incredibly difficult, if not impossible, to introduce *truly random* choices to a model of computation.

Thus, UNIFYHEAPS can spawn an exponential amount of subgoals *wrt.* the number of existential variables in the SL specification and the number of sub-heaps. Because each choice of unifying substitution is central to the generation of a correct program, but the specific choice of sub-heap for the unifying substitution is not, UNIFYHEAPS has an optimization whereby it *eagerly* chooses the first possible sub-heap for the unifying substitution, discarding the other choices.

4.1.3 The cost of removing Non-determinism

But, consider the derivation of `pick(x, y)` again, whose SL specification is attached again below.

$$\{x \mapsto a * y \mapsto b\} \text{void pick}(\text{loc } x, \text{loc } y) \{x \mapsto z * y \mapsto z\} \quad (4.1)$$

At the step in our derivation of `pick(x, y)` wherein UNIFYHEAPS is invoked, as we see in [subsection 3.1.1](#),

$$\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\} \rightsquigarrow \{x \mapsto z * y \mapsto z\}$$

the sub-heap $y \mapsto z$ in the postcondition has been selected by UNIFYHEAPS with $y \mapsto b2$ in the precondition to form a unifying substitution, resulting in the subgoal

$$\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\} \rightsquigarrow \{b2 = z; x \mapsto z * y \mapsto b2\}$$

and the grayed portion triggers the SUBSTRIGHT rule, giving us

$$\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\} \rightsquigarrow \{x \mapsto b2 * y \mapsto b2\}$$

The program that will eventually be derived is identical to the program previously described in [Figure 3.2](#):

```

1 void pick (loc x, loc y) {
2   let a2 = *x;
3   let b2 = *y;
4   *x = b2;
5 }
```

This program is *provably* correct *wrt.* the SL specification described in [Equation 4.1](#). However, is the resultant program in agreement with what the user intended? All is well and good if the user intended for z to be equal to the value that y is pointing to, i.e., $b2$. Then, choosing the first possible sub-heap, in this case $y \mapsto b2$, for the unifying substitution would be aligned with the expectations of the user.

What if the user intended for z to be equal to the value that x is pointing to, which is $a2$, instead of $b2$?

Then, because of this optimization, the synthesized program does not reflect the user’s intentions, since by design it only chooses the first possible sub-heap for the unifying substitution in UNIFYHEAPS. As a consequence of this eagerness, the user will never be able to synthesize a program wherein both x and y point to the value, a_2 , initially pointed to by x .

4.1.4 Re-introducing Non-determinism to UNIFYHEAPS

As part of optimizations to SUSLIK’s runtime, non-determinism present in rules like UNIFYHEAPS which do not affect the correctness of generated programs was removed. However, the synthesized programs, while provably correct *wrt.* the provided SL specifications, do not accurately represent the user’s intent. We resolve this by re-introducing non-determinism to UNIFYHEAPS, thereby allowing user-provided input-output examples to prune the set of candidate programs instead of forcing the synthesizer to eagerly choose the first possible program.

Since the semantics of UNIFYHEAPS (cf. Figure 4.1) invoke non-determinism anyway, the optimization performed on the actual implementation of UNIFYHEAPS in SUSLIK was to eagerly return a singleton list containing the first result of the rule.

$$\begin{array}{c}
 \text{UNIFYHEAPS} \\
 \frac{\begin{array}{l} [\sigma]R' = R \\ \emptyset \neq \text{dom}(\sigma) \subseteq \text{Existentials}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Gamma; \{P * R\} \rightsquigarrow [\sigma]\{\psi; Q * R'\} \mid c \end{array}}{\Gamma; \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R'\} \mid c}
 \end{array}$$

FIGURE 4.1: Heap unification rule.

To re-introduce non-determinism, we simply adapt the implementation of UNIFYHEAPS to return a list containing all the possible results of the rule.

4.2 READ

We also incorporate into SUSLIK the ability to instantiate variables in a SL specification with a variable in the input-output examples.

4.2.1 A program that does nothing

Consider the following modified version of `pick`, `pick_mod`.

$$\{x \mapsto a * y \mapsto a\} \text{ void pick_mod}(\text{loc } x, \text{loc } y) \{x \mapsto z * y \mapsto z\} \quad (4.2)$$

The changes from `pick` (cf. Equation 3.1) are highlighted in gray: instead of the heaplet $y \mapsto b$, we have $y \mapsto a$. The synthesized program, according to the above SL specification described in Equation 4.2, is the empty program: a program that only performs a skip.

Looking at the program derivation, this is unsurprising. Firstly, the synthesizer applies the READ rule on the ghost variable a , *read*-ing in a as $a2$.

$$\{x, y, a2\}; \{x \mapsto a2 * y \mapsto a2\} \rightsquigarrow \{x \mapsto z * y \mapsto z\} \quad (4.3)$$

Next, it unifies z with $a2$ using the UNIFYHEAPS rule.

$$\{x, y, a2\}; \{x \mapsto a2 * y \mapsto a2\} \rightsquigarrow \{a2 = z; x \mapsto a2 * y \mapsto z\} \quad (4.4)$$

Unlike subsection 4.1.3, all that remains is to invoke SUBSTRIGHT, giving us

$$\{x, y, a2\}; \{x \mapsto a2 * y \mapsto a2\} \rightsquigarrow \{x \mapsto a2 * y \mapsto a2\} \quad (4.5)$$

which can then be FRAME-ed out. The goal is then solved using the EMP rule.

What is notable is that these rules, with the exception of READ, are non-operational, and the witness programs that serves as a proof term for them are all `skips`. For READ, it produces a witness program `let a2 = y,` but it is discarded since `a2` is never used. Consequently, the final program reflects this: it is a sequence of `skips`, *i.e.*, the empty program.

4.2.2 READEG

As before, the synthesized program is valid *wrt.* the provided specifications. However, a user might want for `z` to denote a value other than `a`. Thus, the user might provide an input-output example that specifies this as such:

$$[x \mapsto a * y \mapsto a] [x \mapsto v * y \mapsto v] \quad (4.6)$$

where `v` refers to the desired constant.

We want to unify the existential variable `z` (*cf.* Equation 4.2) with the heaplet `x ↦ v` or `y ↦ v` in the output component of Equation 4.6. `v` is read into the scope of the program if it is a ghost variable, bound in the pre- and postcondition, but in this case it is not.

This is handled by a non-deterministic example-*aware* rule called READEG that performs the operational READ and the non-operational unification UNIFYHEAPS together. The logical form of both cases of READEG is presented in Appendix A. READEG compares the SL specification with the example. If the example contains ghost variables that do not appear in the SL specification, then the ghost variables are "materialized" by reading them into local variables whose names are fresh in the scope. If there

are existential variables in the output component of the example, then they are unified with the corresponding heaplet in the postcondition. For both cases, READEG introduces a unifying substitution akin to UNIFY-HEAPS. However, to reflect the reification of the ghost variables in the first case, the READEG rule also has as its witness program the witness program of READ-ing in the ghost variables.

4.2.3 A program that does something

Suppose we used the input-output example described in Equation 4.6. Then, instead of unifying z with $a2$, as was done in Equation 4.3 to Equation 4.4, the synthesizer might invoke READEG. It unifies the heaplets in the postcondition, $x \mapsto z$ and $y \mapsto z$, with the heaplets in the output component of the examples, remembering the substitution $z = v$. This gives us

$$\{x, y, a2\}; \{x \mapsto a2 * y \mapsto a2\} \rightsquigarrow \{z = v; x \mapsto v * y \mapsto z\}$$

READEG, being example-aware, has introduced v into our subgoal. Then, after applying SUBSTRIGHT so that we also have $y \mapsto v$ in the postcondition, we can apply WRITE twice to write the value of v into x and y , finally deriving the following program:

```

1 void pick_mod (loc x, loc y) {
2   let a2 = *y;
3   *x = v;
4   *y = v;
5 }
```

This is a simpler example – since v is not a ghost variable, it does not need to be materialized. In general, we need to introduce ghost variables in the examples into the scope of the program, as we shall see in chapter 5.

Chapter 5

Implementation and Case Studies

We demonstrate our findings from [chapter 4](#) by synthesizing two programs. The two programs are specified using *incomplete* specifications, and are synthesized with the help of our input-output examples. The first program, `fstElement`, retrieves the first element of a linked list. The second program, `treeRightChild`, retrieves the right child of a tree.

5.1 Running the Project

We implemented our enhancements of SUSLIK with examples in an extended version of the SUSLIK tool, which we refer to as SUSLIX (Synthesis with Separation Logic and eXamples), available at: <https://github.com/tanyhb1/suslik/tree/suslix>. One can find details on running SUSLIX in the provided Github repository.

The changes to the original SUSLIK infrastructure amounted to around 1,000 lines of code, though the material changes to the standard SUSLIK algorithm and SSL rules affected around 200 lines of code. Other than that, SUSLIX augments SUSLIK by adding a concrete interpreter for

SUSLIK's target language, the definition of examples and a parser for examples, case studies, and glue code.

5.2 Case Study 1: `fstElement`

5.2.1 Inductive Heap Predicates

Firstly, let us look at how linked lists are defined in Separation Logic. SL can compositionally reason about linked data structures like linked lists that are defined recursively via inductive heap predicates. For instance, the definition of a linked list segment is given as below:

$$\begin{aligned}
 lseg(x, y, S) \triangleq & x = y \wedge \{S = \emptyset; emp\} \\
 & | x \neq y \wedge \{S = \{v\} \cup S_1; [x, 2] * \\
 & \quad x \mapsto v * \langle x, 1 \rangle \mapsto nxt * lseg(nxt, y, S_1)\}
 \end{aligned} \tag{5.1}$$

Thus, the predicate $lseg(x, y, S)$ describes a linked list: it begins at x , ends at location y , and contains a set of elements S .

5.2.2 Synthesizing `fstElement`

Synthesizing `fstElement` is not difficult if a complete specification is provided to the synthesizer. However, because linked data structures like linked lists are defined recursively, to provide a complete specification to SUSLIK, one would have to *manually* unfold the linked list predicate $lseg$, and specify explicitly what each heaplet points to. This could have been handled by the INDUCTION rule automatically. On the other hand, if the specification were not explicit enough, SUSLIK would not have enough

information to synthesize exactly the program that retrieves the first element.

We can provide input-output examples to handle this shortcoming. Consider the following *under-specified* SL specification for `fstElement`:

$$\{ret \mapsto x * lseg(x, 0, S1)\} \text{ void } fstElement(\text{loc } ret) \{ret \mapsto v * lseg(x, 0, S)\} \quad (5.2)$$

The specification is deliberately *under-specified*. It states that, prior to running `fstElement`, `ret` points to the head of the linked list, but after running `fstElement`, it points to some value v . Note that the set of elements in the linked list changes from $S1$ to S , reflecting the retrieval of a value by `fstElement`. In contrast, a complete specification for `fstElement` (*i.e.*, one that the synthesizer would synthesize correctly) would instead have

$$\{ret \mapsto x * [x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, 0, S1)\}$$

as the precondition of the specification in [Equation 5.2](#). However, this is clumsy, and requires one to unfold the `lseg` predicate themselves.

Using SUSLIK's basic synthesis algorithm without any input-output examples on our under-specified SL specification described in [Equation 5.2](#), the program described in [Figure 5.1](#) is derived since the critical step in the synthesis procedure is to simply invoke UNIFYHEAPS to unify the heaplets $ret \mapsto x$ in the precondition and $ret \mapsto v$ in the postcondition of the specification (rules like `READ`, *etc.*, are in actuality performed, but are here omitted for brevity).

```

1   void fstElement (loc ret) {
2   }

```

FIGURE 5.1: The synthesized `fstElement` program without examples

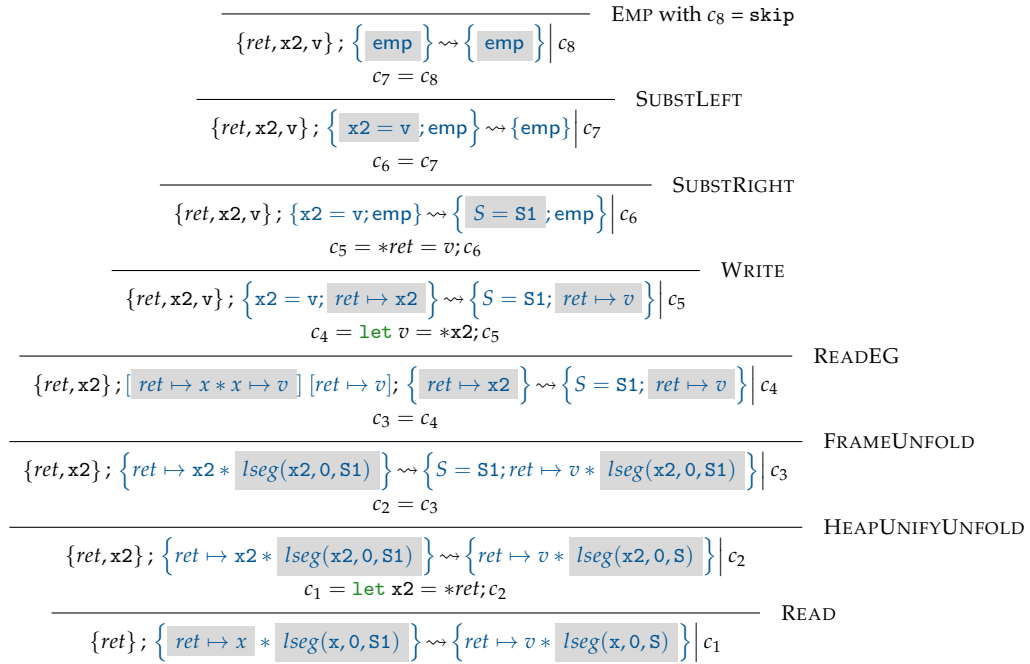
To inform the synthesizer of their intentions, the user can provide the following input-output example:

$$[ret \mapsto x * x \mapsto v] [ret \mapsto v] \quad (5.3)$$

The input-output example described in [Equation 5.3](#) states two things:

- **Input component.** The input component states that, before running a candidate program, the pointer `ret` should point to `x`, and that `x` in turn is a pointer to `v`, a logical variable whose scope captures the pre- and postcondition.
- **Output component.** The output component states that, after running a candidate program, `ret` should point to `v`, which was the referend of `x`.

The (trimmed) program derivation is described in [Figure 5.2](#). We omit failed paths that were backtracked and some rules for clarity. The rule `HEAPUNIFYUNFOLD` basically tries to unfold an inductive predicate in order to unify the underlying set. Furthermore, since `READEG` is non-deterministic as a consequence of being able to make multiple choices, we only include the path that led to a successful candidate program. In [Figure 5.2](#), when `READEG` is triggered, the provided example [Equation 5.3](#) is included in the proof tree to reflect it being used by the example-aware rule. Different candidate programs were also derived, but were pruned

FIGURE 5.2: Derivation of `fstElement(ret)` as c_1 .

by the updated synthesis algorithm (cf. [Algorithm 3.2.2](#)) since they did not satisfy the example.

The resulting program [Figure 5.3](#) does exactly what one expects of it: to retrieve the first element, it traverses through the linked list pointers starting with the head pointer. Thus, it reads in `x2`, and then dereferences it to retrieve the value that it points to. Finally, it returns this value by storing it to `ret`, according to the semantics of the SUSLIK's target language.

```

1 void fstElement (loc ret) {
2     let x2 = *ret;
3     let v = *x2;
4     *ret = v;
5 }

```

FIGURE 5.3: The synthesized `fstElement` program with examples

5.3 Case Study 2: treeRightChild

5.3.1 Inductive Tree Predicates

As before in [subsection 5.2.1](#), we present the definition of a tree:

$$\begin{aligned}
 \text{tree}(x, S) &\triangleq x = 0 \wedge \{S = \emptyset; \text{emp}\} \\
 &| x \neq 0 \wedge \{S = \{v\} \cup S_1 \cup S_2; [x, 3] * x \mapsto v * \langle x, 1 \rangle \\
 &\quad \mapsto l * \langle x, 2 \rangle \mapsto r * \text{tree}(l, S_1) * \text{tree}(r, S_2)\}
 \end{aligned} \tag{5.4}$$

That is, the predicate $\text{tree}(x, S)$ describes a tree containing a set of elements S : if $x = 0$, whereby 0 denotes the null pointer, then it is empty, otherwise, it contains a left and right tree.

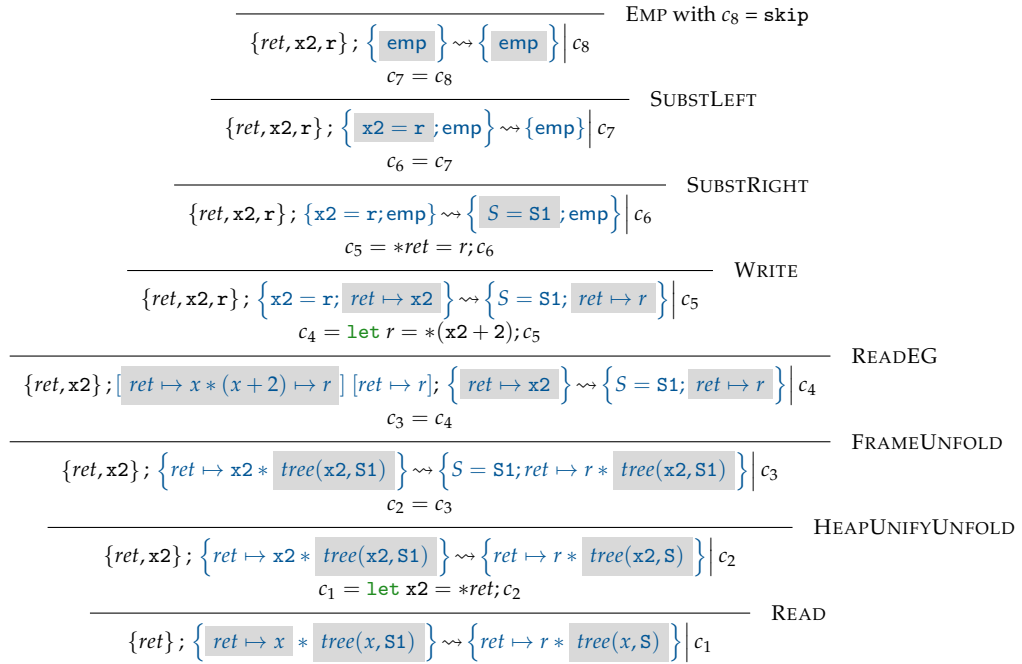
5.3.2 Synthesizing treeRightChild

An incomplete SL specification for `treeRightChild` is described in [Equation 5.5](#).

$$\{\text{ret} \mapsto x * \text{tree}(x, S_1)\} \text{void treeRightChild}(\text{loc ret}) \{\text{ret} \mapsto r * \text{tree}(x, S)\} \tag{5.5}$$

The incomplete specification states that before running `treeRightChild`, `ret` points to x which is a pointer to the payload of the tree. After running `treeRightChild`, it points to some value r . A complete specification, as explained before, would require that we unfold the inductive predicate `tree` ourselves. Instead, the user can provide the following input-output example:

$$[\text{ret} \mapsto x * (x + 2) \mapsto r] [\text{ret} \mapsto r] \tag{5.6}$$

FIGURE 5.4: Derivation of `treeRightChild(ret)` as c_1 .

whereby $(x + 2)$ is deliberately chosen since, according to the definition of `tree`, adding two to the memory address of the root will give the right child. The (trimmed) program derivation is described in Figure 5.4. The machinery used in the derivation is identical to that described in subsection 5.2.2, and so we omit its discussion.

The resulting program Figure 5.5 performs as expected: it retrieves the right child of the given tree by accessing two memory addresses ahead of the provided memory address.

```

1 void treeRightChild (loc ret) {
2     let x2 = *ret;
3     let r = *(x2 + 2);
4     *ret = r;
5 }

```

FIGURE 5.5: The synthesized `treeRightChild` program with examples

Chapter 6

Discussion

6.1 Future Work

6.1.1 List notation/Storyboard programming

A more sophisticated way to handle inductive predicates in example-driven synthesis is demonstrated in storyboard programming [24]. Here, ellipses are used to abstract away unnecessary details of the data-structure that are not manipulated or are manipulated inductively. In this case, the middle of the linked list is abstracted away as the node `mid`. This would allow us to abstract away unnecessary details in our inductive predicates.

6.1.2 Interactive Mode & GUI

SUSLIK also supports an interactive mode for users to interact with the synthesis procedure. SUSLIX can extend upon this to include the evaluation of candidate programs during the example-driven synthesis procedure: whenever SUSLIX generates a candidate program, it can display the candidate program together with the input-output examples to the user and the result of evaluating the program with the examples. The

user then has the liberty of deciding whether or not the candidate program is acceptable. If so, SUSLIX returns that program. Otherwise, the synthesis procedure continues.

Providing a clean GUI in the style of (<http://comcom.csail.mit.edu/comcom/#SuSLik>) would also be more user-friendly.

6.1.3 Optimizing the Search Procedure

Short-circuit evaluation based on examples can be implemented to optimize the search procedure as a form of example-based refutation. For instance, when unfolding an inductive predicate, it is often the case that numerous subgoals corresponding to the different branches (*e.g.*, every time a linked list predicate is unfolded, SUSLIK considers two cases: whether the list has been fully traversed, or not) are spawned. Therefore, some branches in the proof tree can be ruled out without *actually traversing them*. Information about this can be gleaned from examples (*e.g.*, whether a value is 0 – the null pointer – or not) and used to *short-circuit* branches that do not lead to a candidate program.

6.1.4 Rules

The incorporation of user-provided input-output examples into SUSLIK's rules described in this Capstone report are rudimentary, and there is a large space of possibilities for making rules example-aware. One way to do so while maintaining code cleanliness and modularity would be to have a non-example-aware version and an example-aware version of the rules, allowing SUSLIK to access the example-aware version of the rules only if users provide examples in the first place.

Chapter 7

Conclusion

Example-driven synthesis is not novel. It has been thoroughly researched in the domain of functional programming, and synthesis systems exist for low-level data-structure manipulations [12, 13, 20, 24]. Building on this, SUSLIK not only synthesizes heap-manipulating programs, but also guarantees that the programs are correct by construction [21]. However, being correct by construction – satisfying ascribed pre/postconditions – is not a guarantee that a user’s intent has been accurately captured, only a guarantee that the SSL derivations are sound.

The contribution of this thesis is thus to extend SUSLIK’s synthesis algorithm towards example-driven synthesis. Users can provide input-output examples that resemble SL specifications, except that they can be under-specified for ease of use. These examples provide "hints" about the user’s intent. Then, the extended synthesis algorithm prunes the search space of all candidate programs based on these input-output examples. Finally, we incorporate these examples into some of the SSL rules to allow for an "example-aware" program derivation. These enhancements are implemented in an extended version of the SUSLIK tool, which is named SUSLIX, for Synthesis with Separation Logic and eXamples.

Bibliography

- [1] Andrew W. Appel. “Verified Software Toolchain”. In: *ESOP*. Vol. 6602. LNCS. Springer, 2011, pp. 1–17 (cit. on p. 1).
- [2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A survey of attacks on ethereum smart contracts (sok)”. In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186 (cit. on p. 2).
- [3] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. “The design and formalization of Mezzo, a permission-based programming language”. In: *ACM (TOPLAS) 38.4* (2016), pp. 1–94 (cit. on p. 10).
- [4] Jason Gross Benjamin Delaware Clément Pit-Claudiel and Adam Chlipala. “Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant.” In: *POPL*. ACM. 2015, pp. 689–700 (cit. on p. 3).
- [5] Richard Bornat. “Proving pointer programs in Hoare logic”. In: *International Conference on Mathematics of Program Construction*. Springer. 2000, pp. 102–126 (cit. on pp. 4, 5).
- [6] John Boyland. “Checking interference with fractional permissions”. In: *ISAS*. Springer. 2003, pp. 55–72 (cit. on p. 10).

-
- [7] Cristiano Calcagno and Dino Distefano. “Infer: An Automatic Program Verifier for Memory Safety of C Programs”. In: *NASA Formal Methods*. Vol. 6617. LNCS. Springer, 2011, pp. 459–465 (cit. on p. 1).
- [8] Arthur Charguéraud and François Pottier. “Temporary read-only permissions for separation logic”. In: *ESOP*. Springer. 2017, pp. 260–286 (cit. on p. 10).
- [9] Andreea Costea et al. “Concise Read-Only Specifications for Better Synthesis of Programs with Pointers”. In: *ESOP*. Springer. 2020, pp. 141–168 (cit. on pp. 2, 9, 10).
- [10] Mark Dowson. “The Ariane 5 software failure”. In: *ACM SIGSOFT Software Engineering Notes* 22.2 (1997), p. 84 (cit. on p. 2).
- [11] Viktor Kuncak Etienne Kneuss Ivan Kuraj and Philippe Suter. “Synthesis modulo recursive functions”. In: *OOPSLA*. ACM. 2013, pp. 407–426 (cit. on p. 3).
- [12] John K Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 229–239 (cit. on pp. 10, 11, 40).
- [13] Pierre Flener and Serap Yilmaz. “Inductive synthesis of recursive logic programs: Achievements and prospects”. In: *The Journal of Logic Programming* 41.2-3 (1999), pp. 141–195 (cit. on pp. 8, 40).
- [14] Robert W Floyd. “Assigning meanings to programs”. In: *Program Verification*. Springer, 1993, pp. 65–81 (cit. on p. 4).
- [15] Eugene C. Freuder. “In Pursuit of the Holy Grail”. In: *Constraints* (1997) (cit. on p. 1).

-
- [16] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *ACM Sigplan Notices* 46.1 (2011), pp. 317–330 (cit. on pp. 1, 8).
- [17] Sumit Gulwani et al. “Synthesis of loop-free programs”. In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 62–73 (cit. on p. 8).
- [18] Zohar Manna and Richard Waldinger. “A deductive approach to program synthesis”. In: *ACM (TOPLAS)* 2.1 (1980), pp. 90–121 (cit. on p. 8).
- [19] Peter W O’Hearn. “A Primer on Separation Logic (and Automatic Program Verification and Analysis).” In: *Software Safety and Security* 33 (2012), pp. 286–318 (cit. on pp. 5, 17).
- [20] Peter-Michael Osera and Steve Zdancewic. “Type-and-example-directed program synthesis”. In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 619–630 (cit. on pp. 10, 11, 40).
- [21] Nadia Polikarpova and Ilya Sergey. “Structuring the synthesis of heap-manipulating programs”. In: *PACMPL* 3.POPL (2019), pp. 1–30 (cit. on pp. 1, 6, 15, 19, 22, 40).
- [22] Veselin Raychev, Martin Vechev, and Eran Yahav. “Code completion with statistical language models”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 419–428 (cit. on p. 1).
- [23] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74 (cit. on p. 5).

-
- [24] Rishabh Singh and Armando Solar-Lezama. “Synthesizing Data Structure Manipulations from Storyboards”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: Association for Computing Machinery, 2011, 289–299 (cit. on pp. 3, 11, 38, 40).
- [25] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006 (cit. on p. 6).
- [26] Yasunari Watanabe. “Building a Certified Program Synthesizer”. In: *Yale-NUS Capstone Thesis* (2020) (cit. on p. 4).

Appendix A

Logical Form of READEG

$$\begin{array}{c}
 \text{READEG (1)} \\
 \frac{
 \begin{array}{c}
 a \in \text{GV}(\Gamma, \mathcal{P}_\epsilon, \mathcal{Q}_\epsilon) \quad [\sigma]\mathcal{R}' = \mathcal{R} \quad \emptyset \neq \text{dom}(\sigma) \subseteq \text{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\
 y \notin \text{Vars}(\Gamma, \mathcal{P}, \mathcal{Q}) \quad \Gamma \cup \{y\}; \mathcal{E}; [\sigma][y/a]\{\phi; \mathcal{P} * \mathcal{R}\} \rightsquigarrow [y/a]\{\psi; \mathcal{Q} * \mathcal{R}'\} \Big|_c
 \end{array}
 }{
 \Gamma; [\phi; \langle x, l \rangle \mapsto a * \mathcal{P}_\epsilon] [\mathcal{Q}_\epsilon]; \{\phi; \mathcal{P} * \mathcal{R}\} \rightsquigarrow \{\psi; \mathcal{Q} * \mathcal{R}'\} \Big|_{\text{let } y = *(x + l); c}
 }
 \end{array}$$

FIGURE A.1: Logical form of READEG (1).

$$\begin{array}{c}
 \text{READEG (2)} \\
 \frac{
 \begin{array}{c}
 [\sigma]\mathcal{R}'_\epsilon = \mathcal{R}_\epsilon \quad \emptyset \neq \text{dom}(\sigma) \subseteq \text{Existentials}(\Gamma, \mathcal{P}_\epsilon, \mathcal{Q}_\epsilon) \\
 \Gamma; \mathcal{E}; \{\mathcal{P} * \mathcal{R}\} \rightsquigarrow [\sigma]\{\psi; \mathcal{Q} * \mathcal{R}'\} \Big|_c
 \end{array}
 }{
 \Gamma; [\mathcal{P}_\epsilon * \mathcal{R}_\epsilon] [\mathcal{Q}_\epsilon * \mathcal{R}'_\epsilon]; \{\phi; \mathcal{P} * \mathcal{R}\} \rightsquigarrow \{\psi; \mathcal{Q} * \mathcal{R}'\} \Big|_c
 }
 \end{array}$$

FIGURE A.2: Logical form of READEG (2).

Figure A.1 describes the logical form of the *first* case of the READEG rule in subsection 4.2.2, while Figure A.2 describes the *second* case. The *grayed*-out portions represent the changes from the standard READ rule for READEG (1) and from the standard UNIFYHEAPS rule for READEG (2). \mathcal{E} , positioned beside the environment Γ , refers to the provided input-output examples. \mathcal{P}_ϵ and \mathcal{Q}_ϵ correspond to the input/output components of the examples, \mathcal{R}_ϵ is a heaplet in the examples.