TÉCNICO LISBOA

Specification-Driven Synthesis of Summaries for Symbolic Execution

Rafael Henriques dos Santos Gonçalves

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Valentina Nisi
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. David Naumann

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I would like to thank my supervisors, Prof. Pedro Adão and Prof. José Santos, for their unwavering support in writing this thesis. Their advice and insight has proven invaluable in getting this project to where it is today, and has also greatly contributed to my personal development as a researcher.

Secondly, I want to extend my gratitude to all my colleagues at IST who, in one way or another, helped me through these last few months of hard work. In particular, I want to give a heartfelt thank you to Frederico Ramos for his invaluable help with the evaluation of our work, and in general for his time, patience and friendship.

Last but not least, I would like to thank my family and friends for the emotional support they showed me this past year. There are too many names that I would like to highlight, but know that I could not have done this without you.

Abstract

Symbolic execution is a program analysis technique that allows for the exploration of the execution paths of a given program up to a bound. Despite its popularity, two main challenges still hinder its use with real-world code: interactions with the runtime environment and path explosion. Symbolic summaries, which model the symbolic execution of concrete functions by directly interacting with the symbolic state, are one possible solution to address these problems. Summaries, however, are both error-prone and difficult to validate, making the task of writing them a tedious one. We present SUMSYNTH, a new tool for automatically synthesising symbolic summaries from separation-logic-style declarative specifications. SUMSYNTH supports the generation of two types of summaries: under-approximating summaries, which model a subset of the paths generated by the symbolic execution of the concrete function, and over-approximating summaries, which model a superset of the paths generated by the symbolic execution of the concrete function. Furthermore, SUMSYNTH can generate summaries for two separate back ends: C summaries that can be run on any tool that implements our symbolic reflection API, and Python summaries specifically generated for the *angr* symbolic execution tool. To evaluate SUMSYNTH, we generate 29 under-approximating summaries and 34 over-approximating summaries modelling 34 LIBC functions and use the generated summaries to symbolically test two highly used real-world C libraries obtained from GitHub. Results show that SUMSYNTH summaries are much easier to write than handcrafted summaries, and surpass them both in terms of correctness and performance.

Keywords

Symbolic Execution; Symbolic Summaries; Synthesis; Separation Logic; Specifications.

Resumo

Execução simbólica é uma técnica de análise de programas que permite a exploração dos caminhos de execução de um dado programa até um certo limite. Apesar da sua popularidade, dois desafios principais ainda dificultam o seu uso em programas reais: interações com o ambiente e explosão de caminhos. Sumários simbólicos, que modelam a execução simbólica de funções concretas através da interação direta com o estado simbólico, são uma possível solução para lidar com estes problemas. Sumários, no entanto, são propensos a erros e difíceis de validar, tornando a tarefa de escrevê-los entediante. Apresentamos o SUMSYNTH, uma nova ferramenta para sintetizar automaticamente sumários simbólicos a partir de especificações baseadas em lógica de separação. A ferramenta SUMSYNTH permite gerar dois tipos de sumários: de subaproximação, que modelam um subconjunto dos caminhos gerados pela execução simbólica da função concreta, e de sobreaproximação, que modelam um superconjunto dos caminhos gerados pela execução simbólica da função concreta. A ferramenta gera sumários para dois back ends distintos: sumários em C que podem ser corridos em qualquer ferramenta que implemente a nossa API de reflexão simbólica, e sumários em Python gerados especificamente para a ferramenta *angr*. Para avaliar o SUMSYNTH, geramos 29 sumários de subaproximação e 34 sumários de sobreaproximação modelando 34 funções da LIBC e usamos os sumários gerados para testar simbolicamente duas bibliotecas de C populares obtidas do GitHub. Os resultados mostram que os sumários da ferramenta SUMSYNTH são mais fáceis de escrever do que sumários manuais, e ultrapassam-los em termos de correção e desempenho.

Palavras Chave

Execução Simbólica; Sumários Simbólicos; Síntese; Lógica de Separação; Especificações.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

AST	Abstract Syntax Tree
IL	Intermediate Language
I/O	Input/Output
IR	Intermediate Representation
LoC	Lines of Code
OX	Over-Approximating
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories
SL	Separation Logic
SSL	Synthetic Separation Logic
UX	Under-Approximating

1

Introduction

Symbolic execution [1,2], first proposed in the 1970s by a number of software verification researchers, was envisioned in King's 1976 seminal paper [3] as a program analysis technique that would bridge the gap between the fields of program testing and program verification by allowing the execution of programs with symbolic values instead of concrete ones. The core idea is to have a *symbolic execution engine* explore the execution paths of a given program, maintaining for each: **(i)** a store mapping variables into symbolic values or expressions, called a *symbolic memory store*, and **(ii)** a first order formula, called *path condition*, describing the constraints on symbolic values that led execution towards that path. Eventually, a *solver* verifies the feasibility of the path and checks for inputs that might lead to undesired outcomes. In the past, symbolic execution has often been seen as costly and its usefulness limited to academic applications. Recent advances in Satisfiability Modulo Theories (SMT) [4] and compositional analysis approaches [5] have however revived the field, with an active research community now trying to answer its many open questions.

Despite the possibilities of symbolic execution, two main challenges still hinder its use with real-world code: interactions with the runtime environment and path explosion [6]. One of the most popular solutions deployed by modern engines to address these limitations is to use *symbolic summaries*, which allow for the modelling of both external functions and internal functions with a high degree of branching. Symbolic summaries are reusable code snippets that model the behaviour of concrete functions without actually having to symbolically execute them. They allow developers to directly interact with the symbolic state, which makes them a useful mechanism to contain the number of explored paths [6, 7].

Despite its popularity, the use of symbolic summaries to address the limitations of symbolic execution also presents us with some challenges. In particular, modern summaries suffer from the fact that they are written manually, usually in a tool-specific manner and without being verified [7]. This is far from ideal: the task of manually writing symbolic summaries is still a tedious one, being both error-prone and time-consuming. For this reason, even the most popular symbolic execution engines have limited support for summaries; *angr* [8], for instance, modeled 128 LIBC functions as of 2023[1].

Another issue with state-of-the-art symbolic summaries is the prevalence of bugs that compromise the correctness and/or coverage guarantees of symbolic execution engines. Notably, a study conducted by Ramos et al. [7] looked at 37 summaries from popular symbolic execution tools (specifically *angr* [8], *Binsec* [9] and *Manticore* [10]) and found bugs in more than half. The problem is made considerably worse by the relative lack of attention that the research community has shown towards symbolic summaries, resulting in a dire state of affairs where tool support is both scarce and prone to bugs.

Although some attention has recently been given to the problem of summary validation [7], the main challenges of manual summary development, namely its proneness to errors, have remained largely unaddressed. Nonetheless, it is clear that the current approach of writing summaries manually is neither

[1] For reference, the GNU LIBC implementation (*glibc*) offers 1720 functions.

practical, nor does it scale to a large number of functions, such as those offered by LIBC. In particular, the lack of correctness guarantees is a challenge that is largely a consequence of the manual development of summaries. Thus, there is an obvious need for automated support for symbolic summary construction, which would allow for the streamlined development of correct-by-construction summaries.

The main goal of this thesis is to automate the creation of non-mutating symbolic summaries by synthesising them from declarative specifications. More concretely, we design a system that receives a separation-logic-style specification of the target function (in the form of pre- and post-conditions) and synthesises the corresponding summary from this input. Separation Logic (SL) [11, 12] is an extension of Hoare logic developed by Reynolds et al. at the turn of the century that allows for reasoning about programs that access and mutate pointer data structures. Although its use has historically been associated with program verification, separation logic has also proven its usefulness in a variety of other areas. Notably, it underpins the inner workings of *Infer* [13], a static analyser used by Facebook to debug millions of lines of code every day, while Polikarpova and Sergey [14] showed that it can be used to synthesise concrete heap-manipulating functions. Unlike existing literature [14], however, we intend to synthesise symbolic summaries, not concrete functions.

In this thesis, we detail a methodology to implement these ideas, and introduce SUMSYNTH, a tool that applies them in practice. SUMSYNTH works as follows. First, we feed it a specification for the target function written in a custom SL-style assertion language. Then, SUMSYNTH produces the corresponding summary in an intermediate language (IL) and outputs it as a JSON file, which can later be used to drive the generation of actual executable summaries through a special-purpose transpiler. While our ultimate goal is to eventually create a tool for synthesising symbolic summaries for unrestricted functions, in this thesis we choose to focus only on non-mutating functions due to time constraints. We expect the same methodology to apply to other functions, with the additional challenge of having to come up with a way to model memory side effects at the summary level.

Importantly, our approach allows for generated summaries to be either under- or over-approximating. A summary is said to be under-approximating if the set of paths it models is contained in the set of paths generated by the symbolic execution of the concrete function, and over-approximating if the set of paths it models contains the set of paths generated by the symbolic execution of the concrete function. There is no one correct choice for which type of summary is better, as it often depends on the specific needs of the application. Hence, with SUMSYNTH, the developer can choose which of the two approaches better suits their needs.

An interesting consequence of our design is that one can change the output language of the summaries by changing only the transpiler, since there is an intermediate step where summaries are produced in a custom IL. This means that the actual overhead of generating summaries for different tools is relatively small. Thus, we choose to produce summaries in both C and Python. The first can be directly

executed by any tool that implements the API developed by Ramos et al. [7], while the latter can be executed by *angr* [8], which consistently ranks among the most popular symbolic execution engines.

In order to evaluate the viability of our approach, we generate a set of 29 under-approximating summaries and 34 over-approximating summaries (modelling 34 LIBC functions) compatible both with Ramos et al.'s tool-independent API [7] and *angr* [8]. We then compare the performance of automatically synthesised summaries with their handcrafted equivalents on two distinct data sets that make heavy use of LIBC functions. Our analysis finds that synthesised summaries do prove to be a viable alternative to manually written ones, with considerable gains both in terms of complexity and performance.

Additionally, we evaluate the correctness properties of the generated summaries with the help of the summary validation tool SUMBOUNDVERIFY [7], which verifies correctness up to a bound. We find synthesised summaries to be sufficiently expressive to ensure the desired correctness properties both in the case of under-approximating summaries and over-approximating summaries. Although guaranteeing that synthesised summaries are correct-by-construction would require a formal proof, this result strongly suggests that such is the case.

Contributions. The main focus of this thesis is the development of a methodology for automating the generation of symbolic summaries. In summary, we make the following contributions:

- A novel formally defined methodology for synthesising symbolic summaries and executable code from SL-style specifications.
- A tool for automatically synthesising summaries for a variety of symbolic execution engines.
- A library of 29 under-approximating summaries and 34 over-approximating summaries modelling 34 LIBC functions, compatible both with Ramos et al.'s tool-independent API [7] and *angr* [8].

Outline. The rest of this thesis is structured as follows. Chapter 2 presents the background theory of our project, focusing on symbolic execution with and without summaries, and on the theory and pragmatics of separation logic. Chapter 3 provides an overview of related work in the area. Chapter 4 describes how we can use specifications to drive function synthesis, while Chapter 5 does the same for symbolic summaries. Chapter 6 describes the architecture and implementation of SUMSYNTH. Chapter 7 covers the evaluation of the synthesised summaries. Finally, Chapter 8 concludes with some closing thoughts on our contributions and pointers for possible future work.

2

Background

Contents

We take a trip through our project's core concepts, from the basics of symbolic execution and summaries in §2.1 to the theory and pragmatics of separation logic in §2.2.

2.1 Symbolic Execution

We start by going over the basics of symbolic execution, exploring both its benefits and limitations when applied to real-world code, and then proceed to explore how symbolic summaries can prove to be a powerful tool for dealing with some of the main challenges affecting the field.

```
1  void foo(char *str1, char *str2) {
2      int len1 = strlen(str1);
3      int len2 = strlen(str2);
4
5      assert(len1 == len2);
6  }
```

Listing 2.1: Our running example: `foo`, a client function of `strlen`

A running example. Consider the function `foo` defined in Listing 2.1. This simple function receives two strings, `str1` and `str2`, computes their lengths by making a call to the `LIBC` function `strlen`, and asserts that the obtained lengths are equal. Throughout this section, we use it as our running example to first illustrate how symbolic execution is performed without summaries (in §2.1.1), and then with summaries (in §2.1.2).

2.1.1 Pure Symbolic Execution

Symbolic execution [1–3] is a program analysis technique that allows for the exploration of all possible execution paths in a program (up to a bound) by replacing unknown concrete values with symbolic ones. Symbolic execution is handled by a *symbolic execution engine* maintaining for each path: **(i)** a store mapping variables into symbolic values or expressions, called a *symbolic memory store*, and **(ii)** a first order formula, called *path condition*, describing constraints on symbolic values. These are updated by different types of instructions, as we will see further ahead when we explore them in more detail. When the engine hits a branching instruction, a *solver* verifies the feasibility of the path and checks for inputs that might lead to undesired outcomes (e.g., failed assertions).

Let us then look at each component more closely. A *symbolic memory store*, typically denoted by σ, is a function that maps program variables into symbolic expressions. An example would be $\sigma = \{i \mapsto \hat{n}, j \mapsto \hat{n} + 1\}$[1], meaning that a variable i holds the symbolic expression $\hat{n}$ and j holds $\hat{n} + 1$. The symbolic store is updated when an assignment is executed. Thus, if for instance we

[1] We use the caret (^) to denote symbolic variables.

encountered the instruction `i += j` in our previous example, the resulting memory store would be $\sigma = \{i \mapsto 2 * \hat{n} + 1, j \mapsto \hat{n} + 1\}$.

The *path condition*, denoted by π, is a first order formula consisting of constraints on symbolic values that describe the conditions that led execution along a certain path. Initially, the path condition is simply the Boolean value $true$, which is then updated accordingly each time the execution branches (i.e., due to conditionals or loops). A conditional branch update is simple enough: assuming $\sigma = \{i \mapsto \hat{n}\}$ and $\pi : true$, encountering the instruction `if (i > 0)` generates two branches with path conditions $\pi : \hat{n} > 0$ and $\pi : \hat{n} \leq 0$ respectively. Dealing with loops is more difficult, and constitutes an active field of study in symbolic execution. Proposed solutions include *loop unfolding* [15] and *bounded execution* of loops iterating over symbolic values [16]. For the purpose of simplicity, our pure symbolic executor uses a basic flavor of *loop unrolling* where loops are turned into a series of if-else statements, as we will see further ahead.

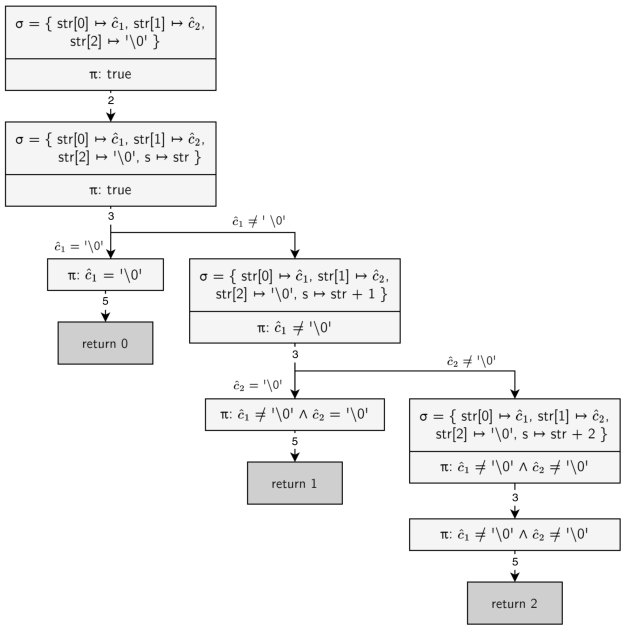
The final component of the symbolic execution engine is the *solver*, also referred to as the *model checker* in some of the literature [6]. The solver is responsible for checking that a path is *feasible*, i.e., that there is an assignment of concrete values to symbolic variables that satisfies the constraints, and whether any such assignment violates the desired properties of the program. Modern model checkers are typically based on Satisfiability Modulo Theories (SMT) solvers [4] such as *Z3* [17] and *CVC5* [18], which allow for reasoning about more complex formulas (operations involving arrays, for instance) than traditional Boolean Satisfiability (SAT) solvers.

Still, there are a number of problems plaguing pure symbolic execution. For instance, what happens when we explore unreachable paths (i.e., with unsatisfiable path conditions)? How do solvers deal with more complicated constraints, such as those involving non-linear operations? Such questions are outside of the scope of this project, but Baldoni et al. [6] provide a thorough overview of these and other challenges for the interested reader.

Symbolically executing `foo`. Let us now visit our running example, the client function `foo` given in Figure 2.1. Consider an execution of `foo` with `str1 = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']` and `str2 = [`$\hat{c}_3$`, `$\hat{c}_4$`, '\0']`, where $\hat{c}_1$, $\hat{c}_2$, $\hat{c}_3$ and $\hat{c}_4$ are symbolic variables (`chars`, in our case) and `\0` is the null terminator. We would expect the `assert` on line 5 to do nothing, since the strings appear to both have a length of 2; symbolically executing `foo`, however, proves otherwise.

In Figure 2.1(a) we can see a (simplified) concrete implementation of `LIBC`'s `strlen`. Given a string `str`, we iterate over it until a `\0` is found, and then return its length by subtracting the memory address of the first character from the address of the null terminator. In Figure 2.1(b) we present a symbolic execution tree for the execution of `strlen` with `str = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']`. According to the previously introduced notation, σ denotes the symbolic memory store, while π denotes the path condition; we omit σ in states where it does not change for brevity's sake. The execution starts with `str[0..2]` in the

```
1  size_t strlen(const char *str) {
2      const char *s = str;
3      while (*s) ++s;
4
5      return s - str;
6  }
```



(a) Simplified `strlen` implementation **(b)** Symbolic execution tree

Figure 2.1: Symbolic execution of LIBC's `strlen`

symbolic store; at line 2, a new variable is added, `s`, pointing to the address of `str`, and finally the loop is executed, yielding a return value of 0 if $\hat{c}_1 =$ '\0', 1 if $\hat{c}_1 \neq$ '\0' $\wedge\ \hat{c}_2 =$ '\0' and 2 if $\hat{c}_1 \neq$ '\0' $\wedge\ \hat{c}_2 \neq$ '\0'. Observe that we depict three return nodes in Figure 2.1(b), each corresponding to one of the three possible return values.

With this knowledge in hand, we can now derive the symbolic execution tree for `foo(str1, str2)` in Figure 2.2. As before, σ denotes the symbolic memory store, while π denotes the path condition. Observe that execution starts with a memory store containing `str1` and `str2` (written in a simplified form where $str \mapsto [\hat{c}_1, \hat{c}_2,$ '\0'] is taken to mean $str[0] \mapsto \hat{c}_1$, $str[1] \mapsto \hat{c}_2$, $str[2] \mapsto$ '\0') and a path condition set to $true$. After making the first call to `strlen`, however, we now have three paths, one for each possible return value. Then, with the second call to `strlen`, each of these paths again branches into three others. We can now see that our prediction was wrong: if $\hat{c}_1 =$ '\0', only $\hat{c}_3 =$ '\0' passes the `assert`; both $\hat{c}_3 \neq$ '\0' $\wedge\ \hat{c}_4 =$ '\0' and $\hat{c}_3 \neq$ '\0' $\wedge\ \hat{c}_4 \neq$ '\0' fail. The subtrees for $\hat{c}_1 \neq$ '\0' $\wedge\ \hat{c}_2 =$ '\0' and $\hat{c}_1 \neq$ '\0' $\wedge\ \hat{c}_2 \neq$ '\0' are omitted for brevity, but it is trivial to deduce that the `assert` similarly fails if the lengths yielded by the calls to `strlen` are different. In the end, out of nine paths, only three of them proved successful.

Our symbolic execution of `foo` highlighted another issue. From analysing the code and the obtained tree, one can see that the number of paths grows exponentially in relation to the number of calls to `strlen`. In our case this is not a problem, since nine paths is a manageable amount. Imagine, however,

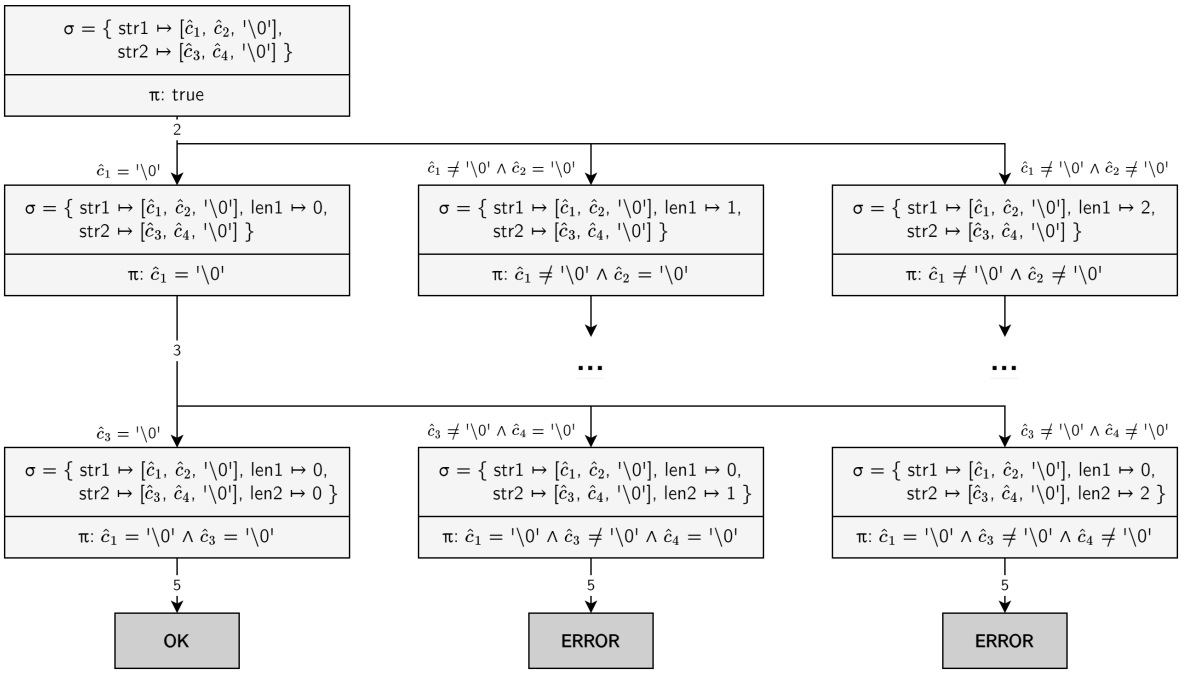


Figure 2.2: Symbolic execution of `foo` without summaries

what would happen if we increased the number of calls with similarly-sized strings. At five calls, for instance, the derived tree would have $3^5 = 243$ paths, while at ten that number would have already grown to $3^{10} = 59049$ paths!

2.1.2 Symbolic Execution with Summaries

Symbolic summaries are reusable code snippets that model the symbolic execution of concrete functions (both internal and external) by directly interacting with the symbolic state [6, 7]. The main idea is to allow the constraining of symbolic variables affected by the modeled function without actually having to symbolically execute it. This provides a higher degree of control over the symbolic state, which allows one to avoid problems often faced in pure symbolic execution, such as *interactions with the runtime environment* and the issue of *path explosion* [6], but comes at the cost of requiring the developer to implement summaries manually.

Symbolic summaries are, however, often difficult to produce, leading to a variety of tool-specific solutions. *angr* [8], for instance, implements *SimProcedures*, a collection of commonly used function summaries written in Python, while also allowing users to write their own summaries. *Manticore* [10] models a number of Linux system calls, but only a few are actually true summaries, with most being stubs that concretize the symbolic arguments. *AVD* [19] supports a number of external calls by implementing 36 LIBC functions, including commonly used system calls.

A more recent idea, proposed by Ramos et al. [7], is to move towards the creation of tool-independent summaries. The author specifically proposes a symbolic reflection API for implementing tool-independent symbolic summaries for the C programming language. The explicit manipulation of C symbolic states is achieved through the use of *symbolic reflection primitives* [20]. These include primitives for creating symbolic variables and constraints, checking for satisfiability, and extending the path condition, which is a set large enough to allow for the implementation of most summaries. The API supports a variety of symbolic execution tools targeting C code and allows for the formal definition of correctness properties for summaries.

There are various ways to group symbolic summaries in respect to one or more correctness properties. In this thesis, we classify summaries as being over-approximating [21], under-approximating [22] or exact [23]. A summary is:

- *Over-approximating* if the set of paths modeled by the symbolic summary contains the set of paths generated by the symbolic execution of the concrete function, i.e., if the constraints generated by the first are implied by the constraints generated by the latter.
- *Under-approximating* if the set of paths modeled by the symbolic summary is contained in the set of paths generated by the symbolic execution of the concrete function, i.e., if the constraints generated by the first imply the constraints generated by the latter.
- *Exact* if it is both over- and under-approximating, i.e., if the formulas generated by the symbolic execution of the concrete implementation and the summary are equivalent.

Exact summaries may seem ideal, but in practice they prove to be the hardest to design. Over-approximating summaries are best suited for applications in which one has to guarantee the absence of security flaws. Under-approximating summaries work well with debugging applications, in order to only show bugs that truly exist.

In line with our running example, we explore different flavors of `strlen` summaries over the next few pages. We specifically present over-approximating, under-approximating and exact summaries developed for the API proposed by Ramos et al. [7].

An example of an over-approximating summary. Listing 2.2 shows the implementation of an over-approximating summary for `strlen`. Given a string `s`, `strlen_over` iterates over it until it finds a concrete null terminator. If, in the meantime, it finds a symbolic character, it builds a *not-equal* constraint taking `s[i]` and `'\0'` as arguments (i.e., SAT only if $s[i] \neq$ '\0' is satisfiable). If the constraint is unsatisfiable, meaning that $s[i] =$ '\0', the summary returns the current length; otherwise, it returns a new symbolic variable `val` and assumes that `val` is greater than or equal to the current index `i` (i.e., SAT only if $val \geq i$, with val and i being signed values, is satisfiable), adding $r \equiv val \geq i$ to the path condition. In

```
size_t strlen_over(char* s) {
    int i = 0;
    char zero = '\0';

    while (1) {
        if (is_symbolic(&s[i],CHAR_SIZE)) {
            if (!is_sat(_solver_NEQ(&s[i], &zero, CHAR_SIZE)))
                break;
            else {
                symbolic val = new_sym_var(INT_SIZE);
                cnstr_t r = _solver_SGE(&val, &i, INT_SIZE);
                assume(r);

                return val;
            }
        }
        else if(s[i] == '\0') break;
        i++;
    }

    return i;
}
```

Listing 2.2: An over-approximating summary for `strlen`

practice, executing it with our example string `str = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']` yields `val` as its return value and adds $val \geq 0$ to the path condition. This highlights why the summary is over-approximating: all the paths generated by the symbolic execution of the concrete function (i.e., lengths of 0, 1 or 2) are contained in the paths modeled by `strlen_over` (i.e., any lengths greater than or equal to 0). The summary is not, however, exact, since there are paths modeled by the summary that are not concrete paths of `strlen(str)` (e.g., lengths greater than 2).

An example of an under-approximating summary. Listing 2.3 shows the implementation of an under-approximating summary for `strlen`. Similarly to its over-approximating counterpart, given a string `s`, `strlen_under` iterates over it until it finds a concrete null terminator. If, in the meantime, it finds a symbolic character, it builds a *not-equal* constraint taking `s[i]` and `'\0'` as arguments (i.e., SAT only if $s[i] \neq$ '\0' is satisfiable). If the constraint is unsatisfiable, meaning that $s[i] =$ '\0', the summary returns the current length; otherwise, it adds $cnstr \equiv s[i] \neq$ '\0' to the path condition and continues iterating. Executing it with our example string `str = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']` yields 2 as its return value and adds $\hat{c}_1 \neq$ '\0' $\wedge\, \hat{c}_2 \neq$ '\0' to the path condition. This highlights why the summary is under-approximating: the path modeled by `strlen_under` (i.e., a length of 2) is contained in the set of the paths generated by the symbolic execution of the concrete function (i.e., lengths of 0, 1 or 2). The summary is not, however, exact, since there are concrete paths of `strlen(str)` that are not modeled by the summary (e.g., lengths smaller than 2).

```
1  size_t strlen_under(char* s) {
2      int i = 0;
3      char char_zero = '\0';
4
5      while (1) {
6          if (is_symbolic(&s[i], CHAR_SIZE)) {
7              cnstr_t cnstr = _solver_NEQ
8                      (&s[i], &charZero, CHAR_SIZE);
9
10             if (!is_sat(cnstr)) break;
11             else assume(cnstr);
12         }
13         else if (s[i] == charZero) break;
14         i++;
15     }
16
17     return i;
18 }
```

Listing 2.3: An under-approximating summary for `strlen`

An example of an exact summary. Finally, Listing 2.4 shows the implementation of an exact summary for `strlen`. Similarly to the previous examples, `strlen_exact` receives a string `s` as argument; this time, however, it first searches for the index of the first occurrence of a concrete null terminator (which is a trivial upper bound on the length of `s`). Then, it creates a symbolic variable `ret` for the return value and iterates over the symbolic characters *backwards*, recursively creating an *if-then-else* constraint where $if \equiv s[i] =$ '\0', $then \equiv ret = i$ and $else \equiv ite\_prev$ (with *ite_prev* being the constraint built in the previous iteration). In the provided code snippet, this refers to line 19's `ret_cnstr`, where $ret\_cnstr \equiv ite(c\_eq\_zero, ret\_eq\_i, ret\_cnstr)$, with $c\_eq\_zero \equiv if$, $ret\_eq\_i \equiv then$ and $ret\_cnstr \equiv else$. After the loop finishes, the summary returns the symbolic variable `ret` and adds the constraint $ret\_cnstr \equiv ite(s[0] =$ '\0', $ret = 0$, $ite(s[1] =$ '\0', ...)) to the path condition. Observe that now the execution of `strlen_exact` over our example string `str = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']` yields ret as its return value, adding $ite(\hat{c}_1 =$ '\0', $ret = 0$, $ite(\hat{c}_2 =$ '\0', $ret = 1$, $ret = 2))$ to the path condition. This demonstrates the exactness of the summary: all the paths it models (i.e., lengths of 0, 1 or 2) are paths generated by the symbolic execution of the concrete function (i.e., lengths of 0, 1 or 2), and vice-versa; the summary is thus simultaneously under- and over-approximating and, therefore, exact.

Revisiting our running example. We are now in a position to revisit the symbolic execution of `foo`. This time, however, we consider an execution where the calls to the concrete implementation of `strlen` in lines 2 and 3 are replaced by calls to the symbolic summary `strlen_exact`. As before, we invoke `foo` with `str1 = [`$\hat{c}_1$`, `$\hat{c}_2$`, '\0']` and `str2 = [`$\hat{c}_3$`, `$\hat{c}_4$`, '\0']` as arguments and take σ, π and $str \mapsto [\hat{c}_1, \hat{c}_2, \text{'\0'}]$ to assume meanings according to the usual notation. With that in mind, we can now derive the symbolic execution tree presented in Figure 2.3. Observe that, similarly to the previous execution, we start with a

```
1  size_t strlen_exact(char* s) {
2      int i = 0;
3      char char_zero = '\0';
4
5      while (is_symbolic(&s[i],CHAR_SIZE) || s[i] != '\0') {
6          i++;
7      }
8
9      int len = i;
10     symbolic ret = new_sym_var(INT_SIZE);
11     cnstr_t ret_cnstr = _solver_EQ(&ret, &len, INT_SIZE);
12
13     for (i = len - 1; i >= 0; i--) {
14         if (is_symbolic(&s[i],CHAR_SIZE)) {
15             cnstr_t c_eq_zero = _solver_EQ
16                     (&s[i], &char_zero, CHAR_SIZE);
17             cnstr_t ret_eq_i = _solver_EQ
18                     (&ret, &i, INT_SIZE);
19
20             ret_cnstr = _solver_IF
21                     (c_eq_zero, ret_eq_i, ret_cnstr);
22         }
23     }
24
25     assume(ret_cnstr);
26     return ret;
27 }
```

Listing 2.4: An exact summary for `strlen`

symbolic memory store containing `str1` and `str2` and a path condition set to $true$. After the first call to `strlen`, though, the path no longer branches; instead, the call to the summary produces a new symbolic variable $\hat{r}_1$, after which we add $len1 \mapsto \hat{r}_1$ to the memory store and $\hat{r}_1 = ite(\hat{c}_1 = \text{'\0'}, 0, ite(\hat{c}_2 = \text{'\0'}, 1, 2))$ to the path condition. This process is then repeated for `str2`, resulting in a similar outcome for `len2`. In the end, it is only in line 5 that a degree of branching is introduced by the constraint solver, when it determines that the `assert` is only successful if $\hat{r}_1 = \hat{r}_2 \Leftrightarrow (\hat{c}_1 = \text{'\0'} \wedge \hat{c}_3 = \text{'\0'}) \vee (\hat{c}_1 \neq \text{'\0'} \wedge \hat{c}_2 = \text{'\0'} \wedge \hat{c}_3 \neq \text{'\0'} \wedge \hat{c}_4 = \text{'\0'}) \vee (\hat{c}_1 \neq \text{'\0'} \wedge \hat{c}_2 \neq \text{'\0'} \wedge \hat{c}_3 \neq \text{'\0'} \wedge \hat{c}_4 \neq \text{'\0'})$.

One important thing to note is how the use of summaries addressed the path explosion problem. Recall that our first symbolic execution of `foo` branched into nine possible paths, and that these grew exponentially, with more calls to `strlen` easily resulting in a number of paths reaching into the thousands. By simply switching the calls to the concrete `strlen` with calls to an equivalent symbolic summary, however, we have managed to streamline the symbolic execution of `foo` to one single path, regardless of the number of times `strlen` is called. This reduces the strain of LIBC calls on the symbolic execution engine, instead shifting it to the constraint solver, and thus allows for more efficient executions of code with repeated calls to such functions.

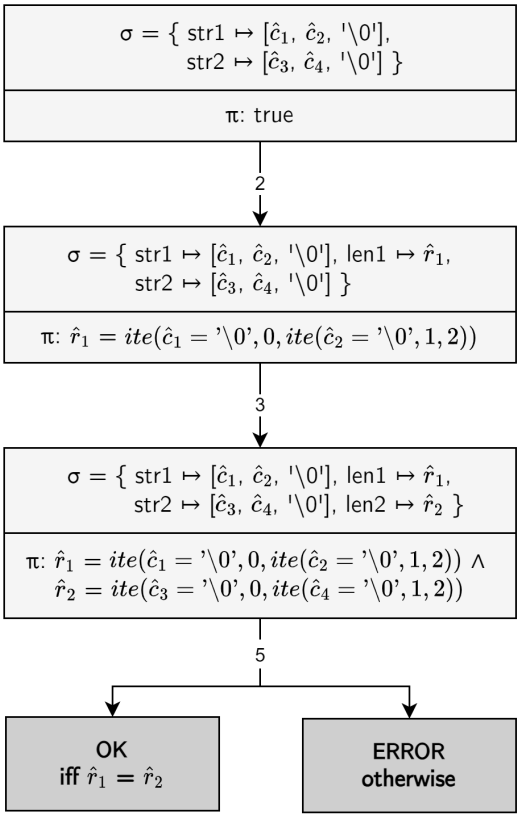


Figure 2.3: Symbolic execution of `foo` with summaries

2.2 Separation Logic

We explore the theory and pragmatics of separation logic (SL) [11, 12], the other key component of our project. We start by going over its Hoare logic foundations, address the major breakthrough that was the separating conjunction *P * Q*, and finish with an exploration of the possibilities that SL offers as a way to derive function specifications.

2.2.1 Foundations: Hoare Logic

The foundations of separation logic lie in Hoare logic [24], a formal system for reasoning mathematically about the correctness of programs. The central feature of Hoare logic is the *Hoare triple*:

$$\{P\}\, C\, \{Q\} \tag{2.1}$$

This notation is also called a *partial correctness specification*, and it means that for a command *C*, *P* and *Q* are respectively its pre- and post-conditions. From this it follows that the Hoare triple necessarily

holds if and only if when C is executed in a state satisfying the pre-condition *P* and C terminates, the resulting state satisfies the post-condition *Q*. An example of an Hoare triple that evaluates to *true* would be:

$$\{X = x\}\,\texttt{Y := X}\,\{X = x \wedge Y = x\} \tag{2.2}$$

There are a number of additional intricacies in Hoare logic, but they are not of particular interest to us and are thus omitted. For now, we note only that while it works extremely well for programs manipulating primitive data types (e.g., integers, strings), proofs involving pointer data structures are much harder to derive when working with standard Hoare logic. Due to these difficulties, a number of methods for reasoning about pointers were proposed over the years, but it was only at the turn of the century that one such idea gained major traction.

2.2.2 The Separating Conjunction

That idea was separation logic [11, 12], developed by Reynolds and a few others around the 2000s, and their main contribution was the *separating conjunction P * Q*. Separation logic extends traditional Hoare logic by modelling not only a store (i.e., a mapping of variables into values), but also a heap, which supports pointers by finitely mapping locations (or addresses) into values. Values are often simply assumed to be integers, although in practice they can also represent addresses or atomic values. Mathematically, we can define them as:

$$\begin{gathered} Store \triangleq Vars \to Vals \qquad Heap \triangleq Addrs \rightharpoonup_{fin} Vals \\ Vals = Ints \supseteq Addrs \cup Atoms \quad Addrs \cap Atoms = \varnothing \quad \mathbf{nil} \in Atoms \end{gathered} \tag{2.3}$$

What distinguishes separation logic from other proposed Hoare logic extensions, however, is the *separating conjunction*. This conjunction, denoted in the literature as *P * Q*, states that assertions *P* and *Q* hold for disjoint sections of memory [12]. This seemingly trivial operator allows us to effectively model much of the behaviour of functions that access and mutate pointer data structures, and is the foundation upon which all of separation logic is built.

The following example comes from the work of O'Hearn [25], and is often considered to be the *hello world* of separation logic:

$$x \mapsto y * y \mapsto x \tag{2.4}$$

Which is to say that *x* points to *y* and *separately* (i.e., in another memory location) *y* points to *x*. Since the separating conjunction states that the memory sections must be disjoint, this means that *x* and *y* must be different values, i.e., *x* (respectively *y*) cannot point to itself. We can visualize this as:

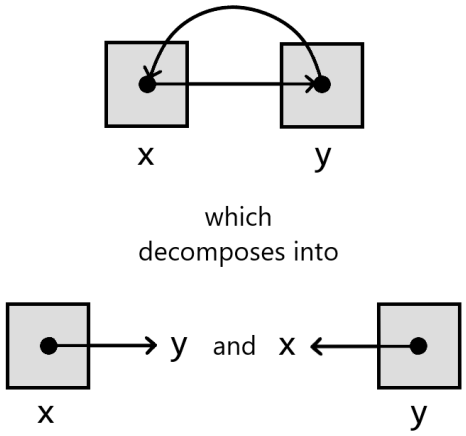


Figure 2.4: Visualizing $x \mapsto y * y \mapsto x$

What does this mean in practice? Let us suppose that $x = 10$ and $y = 20$. From the perspective of the program, both *x* and *y* are aware of these equalities. This means that memory cell $x = 10$ holds the value 20, represented as $x \mapsto 20$, while memory cell $y = 20$ holds the value 10, represented as $y \mapsto 10$. Thus, from the perspective of the first cell, $x = 10 \wedge y = 20 \wedge x \mapsto 20$ holds, but no information regarding the contents of $y = 20$ is known, and vice-versa for the latter.

The field of separation logic is vast and there is a large amount of research in the area that falls outside of the scope of this project. Further directions can be found in the work of O'Hearn [25] for the interested reader.

2.2.3 Specifications

And so we arrive at function specifications. A *specification* is a Hoare triple $\{P\}\, C\, \{Q\}$ where instead of a simple command, C is a function signature, while P and Q are its respective pre- and post-conditions. Specifications prove themselves useful as a way to mathematically characterize the behaviour of functions. If, for instance, we had a procedure for swapping the values of two pointers, `x` and `y`, we could write the following specification:

$$\{x \mapsto a * y \mapsto b\}\, \texttt{void swap(int *x, int *y)}\, \{x \mapsto b * y \mapsto a\} \tag{2.5}$$

A question, then, naturally arises: could we scale this to other (perhaps external) functions, such as those provided by LIBC? The answer is yes, as we will show in the following paragraphs.

Finding a specification for `strlen`. In line with our example from the previous section, let us now find a specification for `strlen`. Recall that `strlen` receives a string `s` as its argument and returns its length.

How can we find this specification? The pre-condition is simple enough: we know that `s` is a string with a certain length (let us say ν). We can represent this as the predicate $\mathrm{str}(s, \nu)$. We do not know as of yet how $\mathrm{str}(s, \nu)$ is defined; for now we consider only that it is an opaque predicate which correctly characterizes a string. The post-condition is similarly simple: `strlen` does not change the string itself, so the assertion $\mathrm{str}(s, \nu)$ remains. What does change is the return value, which we now know to be ν. Let us call the return value ret and derive the assertion $ret = \nu$. With this we have our post-condition: $\mathrm{str}(s, \nu) * ret = \nu$.

Taking our pre- and post-conditions, we thus obtain the following specification for `strlen`:

$$\{\mathrm{str}(s, \nu)\}\ \texttt{size\_t strlen(char *s)}\ \{\mathrm{str}(s, \nu) * ret = \nu\} \tag{2.6}$$

And how do we define the predicate $\mathrm{str}(s, \nu)$? Let us recall what we know about strings: they are null-terminated sequences of characters whose length is equal to the number of characters up until the null terminator. This yields an obvious result: if the first character is the null terminator, the string's length is 0. We can represent this as $s \mapsto \varnothing * \nu = 0$.

The more common case is when the first character is *not* the null terminator ($s \mapsto c * c \neq \varnothing$). Here, we do not know the length of `s`, since the next characters are opaque. We do know, however, that the length must be equal to 1 plus the length of `s + 1`, whatever the latter is. Since the predicate does not concern itself with the actual length, only that such a length does exist, we can write this as $\exists \kappa.\, s \mapsto c * c \neq \varnothing * \nu = \kappa + 1 * \mathrm{str}(s + 1, \kappa)$. This yields the full definition of $\mathrm{str}(s, \nu)$:

$$\begin{aligned} \mathrm{str}(s, \nu) \triangleq\ & s \mapsto \varnothing * \nu = 0 \\ & \vee \exists \kappa.\, s \mapsto c * c \neq \varnothing * \nu = \kappa + 1 * \mathrm{str}(s + 1, \kappa) \end{aligned} \tag{2.7}$$

3

Related Work

Contents

We survey the existing literature relating to our research. In particular, we give an overview of different types of summaries for symbolic execution in §3.1 and discuss some of the existing techniques for separation-logic-based synthesis in §3.2.

3.1 Summaries in Symbolic Execution

There is a vast body of work on symbolic summaries, particularly in the fields of *first-order summaries* [5, 26, 27] and *structured summaries* [28, 29]. Interestingly, and despite their widespread use in practice, *operational summaries* [7] have been much more neglected by researchers. Below, we give an overview of all three types regarding both academic research and use in popular symbolic execution tools.

3.1.1 Operational Summaries

We say operational summaries to be those of the style described in §2.1, i.e., reusable code snippets that model the symbolic execution of concrete functions by directly interacting with the symbolic state [6, 7]. In other words, operational summaries are a loose grouping of symbolic summaries developed for the express purpose of usage in symbolic execution tools. Support is varied: popular examples include *angr* [8], *Binsec* [9] and *Manticore* [10] (which we previously mentioned), but also *KLEE* [30], *Otter* [31] and *S2E* [32], all of which implement some variation of symbolic summaries in a tool-specific manner. On that note, we also highlight Ramos et al. [7], who propose a tool-independent API for developing operational summaries.

In Table 3.1, we present the abridged results of a survey of these tools conducted by Ramos et al. [7]. We specifically highlight the language in which the summaries were written, the number of implemented summaries and the median/maximum number of lines of code (LoC) across the tool's summaries. Interesting findings include the clear advantage of *angr* in terms of implemented summaries, with *Otter* and *Manticore* closely following. Also of note is the fact that even though the length of the most complex summaries (in terms of LoC) varies substantially across tools, the median length usually stays in the 10-20 LoC range (with the notable exception of *Binsec*), suggesting that most summaries tend to be relatively short irrespective of the tool in question.

3.1.2 First-Order Summaries

A first-order summary is a simple first-order formula with either limited support for reasoning about heap memory or no support at all. Early work in the area is attributed to Godefroid [5], with the author introducing *SMART*, an algorithm for dynamic test generation that extends previous solutions by doing it compositionally. *SMART*'s main idea is to test functions in isolation and produce first-order summaries

Table 3.1: Support for operational summaries in popular symbolic execution tools

Tool	Language	$N_{Summaries}$	Med. LoC	Max. LoC
angr	Python	128	9.5	238
Manticore	Python	86	14	59
KLEE	C	55	18	144
Otter	C	109	12	151
S2E	C++	26	26	69
Binsec	OCaml	20	50	102

that can later be reused to analyse higher-level functions. Later work builds upon these insights by leveraging lazy exploration [33] and under-/over-approximating summaries [34], in addition to applying first-order summaries in the context of loop summarization [35]. Still, no support is provided for reasoning about the heap.

Gopan and Reps [26] propose a method for automatically constructing first-order summaries for library functions through an analysis of their low-level implementation. The produced summaries consist of a set of error triggers and a set of summary transformers. Error triggers are assertions over a program state that, if satisfied, indicate the possible occurrence of a program failure, while summary transformers specify how the function call might affect the program state. Generation of summaries consists of three main phases: **(i)** intermediate representation (IR) recovery, **(ii)** numeric program generation, and **(iii)** numeric analysis and summary construction. In the first phase, a value-set analysis is performed to recover low-level information (e.g., accessed variables, parameter-passing details). In the second phase, a numeric program is generated from the obtained IR, capturing the behaviour of the library function. Finally, in the third phase, the generated numeric program is fed into an off-the-shelf numeric analyser, which then generates the set of error triggers and the set of summary transformers. As is the case with other first-order summaries, produced formulas can only express numerical properties, with operations such as those involving heap manipulation not being accurately modeled.

3.1.3 Structured Summaries

Structured summaries are similar to first-order summaries in many respects, but distinguish themselves by introducing some sort of structure to the elements of symbolic execution, which allows them to reason about the heap. Qiu et al. [29], for instance, introduce *memoization trees*, a tree-like data structure that captures the various paths in a function and their respective path conditions (including constraints on the heap). Fragoso Santos et al. [28], on the other hand, propose *JaVerT 2.0*, a compositional symbolic execution tool for JavaScript that allows for the generation of separation-logic-based specifications, which can then be used as function summaries by its symbolic execution engine at a later stage.

3.2 SL-Based Synthesis

There is a long line of work on separation-logic-based synthesis, with many different applications having already been proposed. We choose to focus on *test synthesis* [16, 36, 37], *program synthesis* [14] and *wrapper synthesis* [38], detailing how we can make use of some of the ideas put forth by each of them. We finish with some closing remarks about the *status quo* of SL-based synthesis.

3.2.1 Test Synthesis

Early work on test synthesis can be attributed to Claessen and Hughes [36], who introduced *Quickcheck*, a Haskell-based tool for generating comprehensive test-suites from type declarations. Years later, Seidel et al. [37] proposed *Target*, which added support for precise refinement types. *Cosette*, from Fragoso Santos et al. [16], builds upon these ideas by allowing for the generation of symbolic tests from separation logic specifications. Crucially, it uses the concept of *unification* to find bindings for variables in assertions and then translate these bindings to actual symbolic tests. *Cosette* unification was the basis of *unification plans* [28], which aim at addressing the problem of dependencies between simple assertions and are of particular interest to us.

3.2.2 Program Synthesis

Polikarpova and Sergey [14] propose an extension to separation logic in the form of synthetic separation logic (SSL). SSL is based on the insight that since type theories are proof systems[1], proof search is also program synthesis. Through the introduction of a *transforming entailment* judgement $P \rightsquigarrow Q \mid c$ (meaning that a heap satisfying an assertion P transforms into a heap satisfying an assertion Q via a program c), the authors were able to derive a set of rules that allow for the synthesis of functions from SL specifications. These ideas were then implemented in *SuSLik*, which was proven to be able to handle functions manipulating structures such as lists and trees. More recently, the authors further extended their research to the fields of *cyclic program synthesis* [40] and *synthesis certification* [41]. Itzhaky et al. [42] present a thorough overview of the current state-of-the-art in program synthesis.

3.2.3 Wrapper Synthesis

Nguyen et al. [38] propose *SLICK*, a separation-logic-based runtime checker for Java programs. Crucially, *SLICK* receives a program annotated with pre- and post-conditions and generates function wrappers that *require* the pre-condition to be met at invocation time and *ensure* that the post-condition is met when execution finishes. It achieves this by translating assertions to executable code which ensures

[1] From the Curry-Howard correspondence [39].

that the current heap is compatible with the assertion (i.e., the program state is a model of the formula). Interestingly, *SLICK* makes use of a partial ordering by means of a topological sort not unlike *unification plans* [28], which allows the system to find bindings for variables that are not known *a priori*. These ideas form the basis of *matching plans*, as discussed in §4.4.

3.2.4 Closing Remarks

The key takeaway of this section is that, despite the existence of an extensive body of work on separation-logic-based synthesis, no solutions exist as of yet for symbolic summary synthesis (to the best of our knowledge). We detail our proposal for such a solution in subsequent chapters.

4

Specification-Driven Function Synthesis

Contents

We introduce a methodology to synthesise *functions* from SL-style specifications. We start by giving an overview of the process in §4.1, and then proceed to discuss the specifics in sections §4.2 through §4.5. Finally, we detail the actual code generation procedure in §4.6.

4.1 Overview

In the following pages, we delve into the synthesis of functions from specifications. In particular, we aim to have a procedure for translating specifications into functions by the end of this chapter. The main steps of the translation process are: **(i)** pre-condition resource consumption, and **(ii)** post-condition resource/constraints production.

Before moving on to the specifics, let us start by looking at a simple example. Consider the specification for a function f shown below:

$$\{\underbrace{x \mapsto \#y * x+1 \mapsto \#z * \#z \mapsto \#w}_{Pre}\}\, f(x)\, \{\, Pre * ret = \#y + \#w \,\} \tag{4.1}$$

Figure 4.1 offers a visualization of the program state at the start of f (i.e., a visualization of Pre). Intuitively, f takes the first two elements of a linked list (at addresses x and $\#z$) and adds them, returning the obtained value as described in the assertion $ret = \#y + \#w$. Furthermore, one can also see that the compound assertion Pre appears in both the pre- and post-conditions, which means that this particular function does not change the program state.

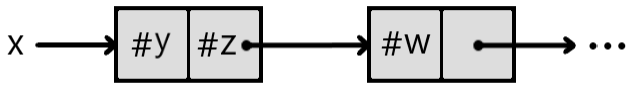


Figure 4.1: Visualizing the program state

Given such a specification, our goal is to produce the code that computes ret. To do this, we first need to synthesise the code that determines the values of $\#y$ and $\#w$ from the value of x. In this example, this is trivial: we find $\#y$ by reading the value stored at address x and $\#w$ by reading the value at address $\#z$ (which itself is stored at address $x+1$). Translating this to code is straightforward:

$$Pre \quad \rightsquigarrow \quad \begin{array}{l} \texttt{int y = *x;} \\ \texttt{int w = *(*(x + 1));} \end{array} \tag{4.2}$$

Having computed $\#y$ and $\#w$, computing ret is trivial: we need only return the result of evaluating $\#y + \#w$. Putting it all together, this yields the full synthesised function f presented in Figure 4.2.

```
1  int f(int *x) {
2      int y = *x;
3      int w = *(*(x + 1));
4
5      return y + w;
6  }
```

Figure 4.2: Full synthesised function f

In general, for a specification such as f's, the two main tasks to perform are: **(i)** synthesising the code that finds the bindings of the logical variables, and **(ii)** producing the return value with the help of those bindings. These can roughly be equated with the pre-condition consumption and post-condition production steps we previously hinted at, and will be explored in greater detail in subsequent sections.

Before closing this section, a word about the different groupings of specifications is warranted. As we mentioned previously, f does not affect the program state, and is therefore classified as having no side effects. Such a statement, however, implies that there must also be a grouping for specifications with side effects. These would be specifications for functions which change the program state by writing to the memory (e.g., a function that copies a list). Such specifications would require an additional step to update the memory in the manner specified by the post-condition. In this thesis, we focus only on functions without side effects, leaving the handling of functions with side effects for future work.

4.2 Syntax

The syntax of our assertion language is given below. Values, $v \in \mathcal{V}$, include numbers (integers and floats) and characters. Types, $\tau \in \mathcal{T}$, are the standard C types: integers and floats of varying sizes (e.g., 32-bit, 64-bit, etc.), both signed and unsigned, pointers, and the generic type $\top$ (equivalent to C's $\mathsf{void}*$). Expressions, $e \in \mathcal{E}$, include values, program variables $x \in \mathcal{X}$, and various unary and binary algebraic operators; Boolean assertions, $\pi \in \Pi$, include the truth values true and false, the standard relational operators (i.e., $=$, $\neq$, $>$, $\geq$, $<$ and $\leq$), as well as the logical operators $\wedge$, $\vee$ and $\neg$.

The syntax of our assertion language

$$
\begin{array}{rcl}
v \in \mathcal{V} & \triangleq & n \in \mathcal{N} \mid c \in \mathcal{C} \\
\tau \in \mathcal{T} & \triangleq & \mathsf{int8} \mid \mathsf{int16} \mid \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{uint8} \mid \mathsf{uint16} \mid \mathsf{uint32} \mid \mathsf{uint64} \mid \mathsf{float32} \mid \mathsf{float64} \mid * \tau \mid \top \\
e \in \mathcal{E} & \triangleq & v \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 \\
\pi \in \Pi & \triangleq & \mathsf{true} \mid \mathsf{false} \mid e_1 \oplus_r e_2 \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi \\
p, q \in \mathcal{SA} & \triangleq & \pi \mid e_1 := e_2 \mid e_1 \mapsto_\tau e_2 \mid \alpha(\bar{e}; e') \\
P, Q \in \mathcal{A} & \triangleq & \mathsf{emp} \mid p \mid P * Q
\end{array}
$$

$$
\mathsf{pred}\ \alpha(\bar{x}; y)\ \{\bar{P}\} \in \mathcal{P}red \qquad\qquad \{P\}\ \mathsf{fn}\ f(\bar{x})\ \{Q\} \in \mathcal{S}pec
$$

Simple assertions, $p \in \mathcal{SA}$, are either Boolean assertions, directed equalities (not to be confused with the regular equality; see §4.3), cell assertions, or predicate assertions. The cell assertion, $e_1 \mapsto_\tau e_2$, states that the memory cell pointed to by e_1 holds the value e_2, and that e_2 has type τ. Predicate assertions are of the form $\alpha(\bar{e}; e')$, where $\bar{e}$ is a list of arguments and e' is a special (optional) argument to be used for synthesis purposes (see §4.3). Assertions, $P \in \mathcal{A}$, are either the constant emp, a simple assertion, or two assertions connected by the separating conjunction. Predicate definitions are of the form pred $\alpha(\bar{x}; y)\,\{\bar{P}\}$, where α is the predicate name, $\bar{x}$ a list of the formal parameters, y a special parameter, and $\bar{P}$ a sequence of assertions, each corresponding to a predicate case (i.e., the full predicate definition is a disjunction of the elements of $\bar{P}$). Function specifications are of the form $\{P\}$ fn $f(\bar{x})\,\{Q\}$, where P and Q are the pre- and post-conditions respectively, f is the function identifier, and $\bar{x}$ is the set of formal parameters. Notice that specifications do not include the function body (i.e., they use function signatures); this is because our purpose is precisely to synthesise the body of a function from its pre- and post-conditions.

4.3 Input/Output Parameters

The overall goal of our synthesis process is to find the bindings of logical variables. Thus, a natural question arises: how do we find such bindings when matching an assertion against a given program state? Understanding the answer is crucial if we are to synthesise any sort of useful code from the specifications we are given. Consider, for instance, the assertion $x \mapsto \#y * \#y \mapsto \#z$ and the state $\sigma = \{x \mapsto l\}$ (where l is some memory position). It is rather intuitive that we need to know $\#y$ to learn $\#z$, and we can learn $\#y$ from x, but how do we formalise this? Is there a way to generalize it for other types of assertions?

Definition 4.1 gives such a formalisation. The idea is to define a set of input variables and a set of output variables for a simple assertion. An input variable will then be a variable that we must "know" before encountering the simple assertion, while an output variable will be the one that we "learn". For example, in the simple assertion $x \mapsto \#y$, x would be the input variable, while $\#y$ would be the output variable.

Definition 4.1 (Input and Output Variables)**.** *Given $p \in \mathcal{SA}$, the function* io $: \mathcal{SA} \to \wp(\mathcal{X}) \times \wp(\mathcal{X})$ *is defined as follows:*

$$
\begin{aligned}
\mathsf{io}(\pi) &\triangleq (\mathsf{vars}(\pi),\, \varnothing) \\
\mathsf{io}(e_1 := e_2) &\triangleq (\mathsf{vars}(e_2),\, \mathsf{outs}_{\mathcal{E}}(e_1)) \\
\mathsf{io}(e_1 \mapsto_\tau e_2) &\triangleq (\mathsf{vars}(e_1),\, \mathsf{outs}_{\mathcal{E}}(e_2)) \\
\mathsf{io}(\alpha(e_1, \ldots, e_n; e') &\triangleq (\textstyle\bigcup_{i=1}^{n} \mathsf{vars}(e_i),\, \mathsf{outs}_{\mathcal{E}}(e'))
\end{aligned}
$$

where $\mathsf{vars} : \mathcal{E} \cup \Pi \to \wp(\mathcal{X})$ *returns the set of variables in an expression* $e \in \mathcal{E}$ *or* $\pi \in \Pi$*, and* $\mathsf{outs}_{\mathcal{E}} : \mathcal{E} \to \wp(\mathcal{X})$ *is defined as:*

$$\mathsf{outs}_{\mathcal{E}}(x) \triangleq \{x\} \qquad \mathsf{outs}_{\mathcal{E}}(e) \triangleq \varnothing \ \ (\text{if } e \notin \mathcal{X})$$

Given the definition of io*, the sets of input and output variables of* p*, respectively* $\mathsf{ins} : \mathcal{SA} \to \wp(\mathcal{X})$ *and* $\mathsf{outs} : \mathcal{SA} \to \wp(\mathcal{X})$*, are defined as follows:*

$$\frac{\mathsf{io}(p) = (X, -)}{\mathsf{ins}(p) \triangleq X} \qquad \frac{\mathsf{io}(p) = (-, Y)}{\mathsf{outs}(p) \triangleq Y}$$

Here also lies the necessity for distinguishing regular and directed equalities. Suppose we had a single type of equality; how would we know if we needed to "learn" any bindings from it (as opposed to it simply being a Boolean assertion/constraint)? For instance, what would be the input variable(s) of $\#y = x$? Just x? Both $\#y$ and x? The solution is to introduce a directed equality, $\#y := x$, such that the input variable is x and the output variable is $\#y$. Conversely, the input variables of $\#y = x$ are both x and $\#y$, and there are no output variables.

What about *parameters*? In §4.2, we mentioned that a predicate definition has a special parameter, but why the need to distinguish it from it the rest? As it turns out, not all parameters are made equal. Input parameters serve a similar purpose to formal parameters in function specifications, in the sense that they must be known before invoking the predicate. Output parameters, on the other hand, are what we learn from the predicate. For instance, $\mathsf{pred}\ \alpha\ (x, y; z)\ \{\bar{P}\}$ has input parameters x and y and output parameter z. In general, predicate definitions may have an arbitrary number of output parameters; for the sake of simplicity, however, we admit that they have only one.

4.4 Matching Plans

So far, we have seen that we can define what we "know" and what we "learn" about simple assertions through the concept of input and output parameters. But this knowledge is relatively uninteresting for an isolated simple assertion; after all, it should be relatively trivial to learn the bindings of logical variables in a correctly formulated directed equality or cell assertion. Much more interesting is what happens in compound assertions. How can we learn bindings in those cases? Is there only one way to do it? Are there multiple ways? And what are the building blocks of this "learning" process?

We start by addressing the latter question. Recall from §4.2 that an assertion is either either the constant emp, a simple assertion, or two assertions connected by the separating conjunction. Thus, the building blocks of assertions are simple assertions, and for any assertion we can derive a sequence of

$$\textsc{Empty}\quad \mathsf{MPG}([],\, X) \rightsquigarrow []$$

$$\textsc{Cont}\quad \frac{\vec{p} = \vec{p}_1 +\!\!+ [p] +\!\!+ \vec{p}_2 \quad \mathsf{ins}(p) \subseteq X \qquad \mathsf{MPG}(\vec{p}_1 +\!\!+ \vec{p}_2,\, X \cup \mathsf{outs}(p)) \rightsquigarrow \vec{p}\,'}{\mathsf{MPG}(\vec{p},\, X) \rightsquigarrow p : \vec{p}\,'}$$

Figure 4.3: Matching Plan Generation

the simple assertions that compose it (in the emp case, we derive the empty sequence). We call this sequence a *matching plan* [16, 28].

An interesting fact to note is that matching plans are not unique: in fact, every permutation of a matching plan is still a matching plan of the original assertion. The question is, are all of them useful? Take, for instance, the assertion $x \mapsto \#y * \#z \mapsto \#w * x + 1 \mapsto \#z$. An obvious matching plan is $[x \mapsto \#y, \#z \mapsto \#w, x + 1 \mapsto \#z]$. Let us assume that x is the only variable that we know *a priori*. There is a point in the sequence (specifically, $\#z \mapsto \#w$) where we want to learn $\#w$ without knowing $\#z$. Notice that this would not happen for the matching plan $[x \mapsto \#y, x + 1 \mapsto \#z, \#z \mapsto \#w]$. Such a matching plan allows us to learn the bindings of all unknown variables, and is thus said to be *valid*. Valid matching plans are the backbone of our synthesis process, and are formalised in Definition 4.2.

Definition 4.2 (Matching Plan and Valid Matching Plan)**.** *A matching plan $\vec{p} \in \mathbb{MP}$ is a sequence of simple assertions. A matching plan $\vec{p} = [p_1, p_2, \ldots, p_n]$ is said to be valid w.r.t. to a set $X \subseteq \mathcal{X}$ of known variables, written* $\mathsf{valid}(\vec{p},\, X)$*, if and only if:*

$$\forall\, 1 \leq i \leq n.\ \mathsf{ins}(p_i) \subseteq X \cup \bigcup_{k=1}^{i-1} \mathsf{outs}(p_k)$$

Finding a valid matching plan for a given assertion is a problem of combinatorial nature which, in practice, requires the implementation of a search algorithm with backtracking to solve. For the sake of simplicity, we model the behaviour of this algorithm through a set of non-deterministic rules. We refer the reader to the implementation to see the transposition of these rules to a deterministic context.

Figure 4.3 defines a non-deterministic procedure $\mathsf{MPG} : \mathbb{MP} \times \wp(\mathcal{X}) \to \mathbb{MP}$ that, given a matching plan $\vec{p}$ and a set of known variables X, derives a *valid* rearrangement of $\vec{p}$ if any such rearrangement exists. In particular, we write $\mathsf{MPG}(\vec{p},\, X) \rightsquigarrow \vec{p}\,'$ to mean that $\vec{p}\,'$ is a valid rearrangement of $\vec{p}$. MPG has two main cases:

- **[Empty]**. The empty matching plan is simply mapped to itself.
- **[Cont]**. When given a non-empty matching plan $\vec{p}$, the procedure MPG finds a simple assertion p in $\vec{p}$ for which we already know all input variables; this will be the head of the generated valid matching plan. Then, the procedure calls itself recursively on the matching plan resulting from the extraction of p from $\vec{p}$ (i.e., $\vec{p}_1 +\!\!+ \vec{p}_2$), adding the output variables of p to the set X of known variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{MPG}([],\, \{x, \#y, \#z, \#w\}) \rightsquigarrow []}\,\textsc{Empty}}{\mathsf{ins}(\#z \mapsto \#w) = \{\#z\} \subseteq \{x, \#y, \#z\} \quad \mathsf{outs}(\#z \mapsto \#w) = \{\#w\}}\,\textsc{Cont}}{\mathsf{MPG}([\#z \mapsto \#w],\, \{x, \#y, \#z\}) \rightsquigarrow [\#z \mapsto \#w]}}{\mathsf{ins}(x+1 \mapsto \#z) = \{x\} \subseteq \{x, \#y\} \quad \mathsf{outs}(x+1 \mapsto \#z) = \{\#z\}}\,\textsc{Cont}}{\mathsf{MPG}([\#z \mapsto \#w,\, x+1 \mapsto \#z],\, \{x, \#y\}) \rightsquigarrow [x+1 \mapsto \#z,\, \#z \mapsto \#w]}$$

$$\dfrac{\mathsf{ins}(x \mapsto \#y) = \{x\} \subseteq \{x\} \quad \mathsf{outs}(x \mapsto \#y) = \{\#y\}}{\mathsf{MPG}([x \mapsto \#y,\, \#z \mapsto \#w,\, x+1 \mapsto \#z],\, \{x\}) \rightsquigarrow [x \mapsto \#y,\, x+1 \mapsto \#z,\, \#z \mapsto \#w]}\,\textsc{Cont}$$

Figure 4.4: Deriving a valid matching plan for $[x \mapsto \#y,\, \#z \mapsto \#w,\, x+1 \mapsto \#z]$

Figure 4.4 shows an execution of MPG for the matching plan $[x \mapsto \#y,\, \#z \mapsto \#w,\, x+1 \mapsto \#z]$ and the set of known variables $\{x\}$. The derivation process is straightforward: we apply the CONT rule until $\vec{p}$ is the empty matching plan and we have already learned all variables; then, we apply the EMPTY rule.

We can now formalise the correctness of matching plans generated by MPG. Theorem 4.1 states that a generated matching plan $\vec{p}'$ is valid and comprised of the same set of simple assertions as $\vec{p}$. Note that the derivation is stated in terms of an equality instead of the $\rightsquigarrow$ operator, denoting a deterministic application of the non-deterministic procedure.

Theorem 4.1 (Correctness of Matching Plan Generation)**.** *For any matching plan $\vec{p} \in \mathbb{MP}$ and set of variables $X \subseteq \mathcal{X}$, it holds that:*

$$\mathsf{MPG}(\vec{p},\, X) = \vec{p}' \implies \mathsf{asrts}(\vec{p}') = \mathsf{asrts}(\vec{p}) \wedge \mathsf{valid}(\vec{p}',\, X)$$

where $\mathsf{asrts}(\vec{p})$ *denotes the set of simple assertions in* $\vec{p}$*.*

Proof. The proof is by induction on the structure of the derivation of $\mathsf{MPG}(\vec{p},\, X) \rightsquigarrow \vec{p}'$. For convenience, we restate the hypothesis of the theorem: $\mathsf{MPG}(\vec{p},\, X) = \vec{p}'$ **(H1)**. The base case is **[EMPTY]**, whereas the inductive case is **[CONT]**.

[EMPTY] Let $\vec{p} = []$. In this case, it follows from **H1** that:

I1. $\vec{p}' = \mathsf{MPG}(\vec{p},\, X) = \mathsf{MPG}([],\, X) = []$

From **I1**, we conclude that:

I2. $\mathsf{asrts}(\vec{p}') = \varnothing = \mathsf{asrts}(\vec{p})$

I3. $\mathsf{valid}(\vec{p}',\, X) = \mathsf{valid}([],\, X) = True$

[CONT] Let $\vec{p} = \vec{p}_1 +\!\!+ [p] +\!\!+ \vec{p}_2$ **(H2)** and $\mathsf{ins}(p) \subseteq X$ **(H3)**. In this case, it follows from **H1** that there is a sequence of assertions $\vec{p}''$ such that:

I1. $\mathsf{MPG}(\vec{p}_1 +\!\!+ \vec{p}_2,\, X \cup \mathsf{outs}(p)) = \vec{p}''$

I2. $\vec{p}' = p : \vec{p}''$

Applying the induction hypothesis to **I1**, we conclude that:

I3. $\mathsf{asrts}(\vec{p}\,'') = \mathsf{asrts}(\vec{p}_1 +\!\!+ \vec{p}_2)$

I4. $\mathsf{valid}(\vec{p}\,''\!,\, X \cup \mathsf{outs}(p))$

From **H2**, it follows that:

I5. $\mathsf{asrts}(\vec{p}) = \mathsf{asrts}(\vec{p}_1) \cup \mathsf{asrts}(\vec{p}_2) \cup \{p\}$

From **I2**, it follows that:

I6. $\mathsf{asrts}(\vec{p}\,') = \{p\} \cup \mathsf{asrts}(\vec{p}\,'')$

From **I3**, **I5** and **I6**, we conclude that:

I7.
$$\begin{aligned}\mathsf{asrts}(\vec{p}\,') &= \{p\} \cup \mathsf{asrts}(\vec{p}\,'') \\ &= \{p\} \cup \mathsf{asrts}(\vec{p}_1 +\!\!+ \vec{p}_2) \\ &= \{p\} \cup \mathsf{asrts}(\vec{p}_1) \cup \mathsf{asrts}(\vec{p}_2) \\ &= \mathsf{asrts}(\vec{p})\end{aligned}$$

From **I4** and **H3**, it follows from Definition 4.2 that:

I8. $\mathsf{valid}(p : \vec{p}\,''\!,\, X) \equiv \mathsf{valid}(\vec{p}\,'\!,\, X)$

□

4.5 Matching Trees

Moving up in the syntactic ladder, we arrive at predicate definitions and function specifications. We set aside the latter for now, and focus on the former. Why do we need to address predicate definitions? Because we want to know how to compute the output parameter from the input parameters, which is of the utmost importance when matching predicate assertions. Thus, we revisit a previous question: how can we learn the bindings of logical variables in predicate definitions? Can we use matching plans, similarly to how we did in the case of assertions?

The answer, as it turns out, is no, although the reason why might not be immediately obvious. Consider the following predicate definition, which models the absolute value of a given number:

$$\begin{array}{l}\mathsf{pred}\ abs(x; y)\,\{ \\ \quad x \geq 0 \ast y := x; \\ \quad x < 0 \ast y := -x \\ \}\end{array} \tag{4.3}$$

We could derive the matching plans $[x \geq 0,\, y := x]$ and $[x < 0,\, y := -x]$ for the first and second case respectively, but these do not tell us much. Informally, we know that what the predicate is actually

saying is that $y := x$ if $x \geq 0$, and $y := -x$ otherwise, but we would not be able to learn this from the matching plans alone.

So why can we not use matching plans to express predicate definitions? Because predicates are (normally) a disjunction of assertions, each corresponding to a possible case, and we do not know *a priori* which of the cases applies. This means that we cannot express predicate definitions in terms of a single matching plan, and instead require a structure that can coalesce the matching plans corresponding to the various cases in a single format. We call this structure a *matching tree*, and formalise it in Definition 4.3. A matching tree is either the leaf node $\bullet$, a single node $\langle p, \psi \rangle$ (corresponding to a non-branching simple assertion) or a double node $\langle \pi, \psi_1, \psi_2 \rangle$ (corresponding to a branching simple assertion). We can now express the predicate abs as the matching tree $\psi = \langle x \geq 0, \langle y := x, \bullet \rangle, \langle y := -x, \bullet \rangle \rangle$.

Definition 4.3 (Matching Tree)**.** *A matching tree $\psi \in \Psi$ is defined as:*

$$\psi \in \Psi \triangleq \bullet \mid \langle p, \psi \rangle \mid \langle \pi, \psi_1, \psi_2 \rangle$$

Similarly to matching plans, we also want to be able to express that a matching tree is able to learn the bindings of all unknown variables, i.e., that it is *valid*. In the case of matching trees, this means that we want all paths that they coalesce to be valid matching plans. Thus, in order to formally define a valid matching tree, we first need to define an auxiliary function $\mathtt{tp}$ that, given a matching tree ψ, returns the set of all possible paths from root to leaves. We formalise this function in Definition 4.4.

Definition 4.4. *Given $\psi \in \Psi$, the function $\mathtt{tp} : \Psi \to \wp(\mathbb{MP})$ is defined as follows:*

$$\begin{aligned} \mathtt{tp}(\bullet) &\triangleq \{[]\} \\ \mathtt{tp}(\langle p, \psi \rangle) &\triangleq \{p : \vec{p} \mid \vec{p} \in \mathtt{tp}(\psi)\} \\ \mathtt{tp}(\langle \pi, \psi_1, \psi_2 \rangle) &\triangleq \{\pi : \vec{p}_1 \mid \vec{p}_1 \in \mathtt{tp}(\psi_1)\} \cup \{\neg\pi : \vec{p}_2 \mid \vec{p}_2 \in \mathtt{tp}(\psi_2)\} \end{aligned}$$

We are now in a position to formally define valid matching trees, which we do in Definition 4.5. Notice that if we apply this to the matching tree derived for abs, we conclude that it is indeed valid, since both $[x \geq 0, y := x]$ and $[x < 0, y := -x]$ are valid matching plans.

Definition 4.5 (Valid Matching Tree)**.** *A matching tree $\psi \in \Psi$ is said to be valid w.r.t. to a set $X \subseteq \mathcal{X}$ of known variables, written $\mathtt{valid}(\psi, X)$, if and only if every $\vec{p} \in \mathtt{tp}(\psi)$ is a valid matching plan w.r.t. to X.*

As is the case with matching plans, deriving a valid matching tree from a set of assertions is also a problem of combinatorial nature. In practice, we solve this by first deriving valid matching plans for each individual assertion and then searching for a matching tree that correctly coalesces them. For the sake of simplicity, we again model the behaviour of this algorithm through a set of non-deterministic rules, and refer the reader to the implementation to see their transposition to a deterministic context.

$$\textsc{Empty}\quad \frac{\forall \vec{p} \in \overrightarrow{pp}.\ \vec{p} = []}{\mathsf{MTG}(\overrightarrow{pp},\ X)\ \rightsquigarrow\ \bullet}$$

$$\textsc{Single Node}\quad \frac{\forall \vec{p} \in \overrightarrow{pp}.\ p \in \vec{p} \quad \mathsf{ins}(p) \subseteq X \qquad \mathsf{MTG}(\overrightarrow{pp} \setminus_{ss} p,\ X \cup \mathsf{outs}(p))\ \rightsquigarrow\ \psi}{\mathsf{MTG}(\overrightarrow{pp},\ X)\ \rightsquigarrow\ \langle p,\ \psi \rangle}$$

$$\textsc{Double Node}\quad \frac{\overrightarrow{pp} = \overrightarrow{pp}_t \uplus \overrightarrow{pp}_f \quad \forall \vec{p} \in \overrightarrow{pp}_t.\ \pi \in \vec{p} \quad \forall \vec{p} \in \overrightarrow{pp}_f.\ \neg\pi \in \vec{p} \quad \mathsf{ins}(\pi) \subseteq X \qquad \mathsf{MTG}(\overrightarrow{pp}_t \setminus_{ss} \pi,\ X)\ \rightsquigarrow\ \psi_t \quad \mathsf{MTG}(\overrightarrow{pp}_f \setminus_{ss} \neg\pi,\ X)\ \rightsquigarrow\ \psi_f}{\mathsf{MTG}(\overrightarrow{pp},\ X)\ \rightsquigarrow\ \langle \pi,\ \psi_t,\ \psi_f \rangle}$$

Figure 4.5: Matching Tree Generation

Before formally presenting these rules, we need to define two auxiliary operators, $\setminus_s : \mathbb{MP} \to \mathcal{SA} \to \mathbb{MP}$ and $\setminus_{ss} : \wp(\mathbb{MP}) \to \mathcal{SA} \to \wp(\mathbb{MP})$. Informally, $\vec{p} \setminus_s p$ represents the matching plan obtained by removing the simple assertion p from $\vec{p}$ (e.g., $[p_1, p_2, p_3] \setminus_s p_2 = [p_1, p_3]$), while $\overrightarrow{pp} \setminus_{ss} p$ is the result of mapping $\vec{p} \setminus_s p$ to a set $\overrightarrow{pp}$. Put formally:

$$\frac{\exists \vec{p}_1, \vec{p}_2.\ \vec{p} = \vec{p}_1 +\!\!+ [p] +\!\!+ \vec{p}_2}{\vec{p} \setminus_s p \triangleq \vec{p}_1 +\!\!+ \vec{p}_2} \qquad \overrightarrow{pp} \setminus_{ss} p \triangleq \{\, \vec{p} \setminus_s p \mid \vec{p} \in \overrightarrow{pp} \,\}$$

Figure 4.5 defines a non-deterministic procedure $\mathsf{MTG} : \wp(\mathbb{MP}) \times \wp(\mathcal{X}) \to \Psi$ that, given a set of (not necessarily valid) matching plans $\overrightarrow{pp}$ and a set of known variables X, derives a matching tree coalescing *valid* rearrangements of the matching plans in $\overrightarrow{pp}$ (henceforth referred to as a valid tree-rearrangement of $\overrightarrow{pp}$) if any such rearrangement exists. In particular, we write $\mathsf{MTG}(\overrightarrow{pp},\ X)\ \rightsquigarrow\ \psi$ to mean that ψ is a valid tree-rearrangement of $\overrightarrow{pp}$. MTG has three main cases:

- **[EMPTY]**. The set containing only empty matching plans is simply mapped to a leaf node.
- **[SINGLE NODE]**. If there is a single assertion p, for which we already know all input variables, such that p occurs in every $\vec{p} \in \overrightarrow{pp}$, MTG generates a single node and labels it with p. Then, the procedure calls itself recursively on the set of matching plans resulting from the extraction of p from every $\vec{p} \in \overrightarrow{pp}$ (i.e., $\overrightarrow{pp} \setminus_{ss} p$). Finally, the matching tree resulting from the recursive call is added as a child of the single node.
- **[DOUBLE NODE]**. If there is a Boolean assertion π, for which we already know all input variables, such that, for every $\vec{p} \in \overrightarrow{pp}$, either π or $\neg\pi$ occurs in $\vec{p}$, MTG generates a double node and labels it with π. Then, the procedure calls itself recursively on the set of matching plans resulting from the extraction of π from every $\vec{p} \in \overrightarrow{pp}$ where $\pi \in \vec{p}$ (i.e., $\overrightarrow{pp}_t \setminus_{ss} \pi$) and the set of matching plans resulting from the extraction of $\neg\pi$ from every $\vec{p} \in \overrightarrow{pp}$ where $\neg\pi \in \vec{p}$ (i.e., $\overrightarrow{pp}_f \setminus_{ss} \neg\pi$). Finally, the matching trees resulting from the recursive calls are added as children of the double node (the former as the left branch and the latter as the right branch).

$$\dfrac{\dfrac{\dfrac{}{\mathsf{MTG}([],\,\{x,y\}) \rightsquigarrow \bullet}\,\textsc{Empty}}{\mathsf{MTG}(\{[y := x]\},\,\{x\}) \rightsquigarrow \langle y := x,\, \bullet\rangle}\,\textsc{Single Node} \qquad \dfrac{\dfrac{}{\mathsf{MTG}([],\,\{x,y\}) \rightsquigarrow \bullet}\,\textsc{Empty}}{\mathsf{MTG}(\{[y := -x]\},\,\{x\}) \rightsquigarrow \langle y := -x,\, \bullet\rangle}\,\textsc{Single Node}}{\mathsf{MTG}(\{[x \geq 0,\, y := x], [x < 0,\, y := -x]\},\,\{x\}) \rightsquigarrow \langle x \geq 0,\, \langle y := x,\, \bullet\rangle,\, \langle y := -x,\, \bullet\rangle\rangle}\,\textsc{Double Node}$$

$$\mathsf{outs}(y := x) = \{y\} \quad \mathsf{ins}(y := x) = \{x\} \subseteq \{x\}$$

$$\mathsf{outs}(y := -x) = \{y\} \quad \mathsf{ins}(y := -x) = \{x\} \subseteq \{x\}$$

$$x \geq 0 \in [x \geq 0,\, y := x] \quad x < 0 \in [x < 0,\, y := -x] \quad \mathsf{ins}(x \geq 0) = \{x\} \subseteq \{x\}$$

$$\{[x \geq 0,\, y := x], [x < 0,\, y := -x]\} = \{[x \geq 0,\, y := x]\} \uplus \{[x < 0,\, y := -x]\}$$

Figure 4.6: Deriving a valid matching tree for $\{[x \geq 0,\, y := x], [x < 0,\, y := -x]\}$

Figure 4.6 shows an execution of MTG for the set of matching plans $\{[x \geq 0,\, y := x], [x < 0,\, y := -x]\}$ (corresponding to the abs predicate) and the set of known variables $\{x\}$. The derivation process works as follows:

- We start by applying the DOUBLE NODE rule, since the two matching plans share no simple assertions. Hence, we select the assertion $x \geq 0$, which occurs in the matching plan $[x \geq 0,\, y := x]$, and whose negation (i.e., $x < 0$) occurs in the matching plan $[x < 0,\, y := -x]$. Having selected $x \geq 0$ as our branching assertion, we proceed to check that its input parameters are in the set of known variables $\{x\}$, which is indeed the case. Finally, we simply have to derive the matching trees corresponding to the extraction of $x \geq 0$ from the first matching plan and the extraction of $x < 0$ from the second (respectively the derivations of $[y := x]$ and $[y := -x]$).
- As the derivations of $[y := x]$ and $[y := -x]$ are similar, we explain only the former. The derivation of a matching tree for $[y := x]$ is straightforward, as it contains a single matching plan consisting of a single directed equality. Thus, we simply have to check that the input parameters of the assertion are contained in the set of known variables $\{x\}$; since that is the case, we apply the SINGLE NODE rule. Finally, we need to derive a matching tree for the set containing only an empty matching plan, which, following the EMPTY rule, corresponds to the leaf node.

As was the case with matching plans, we can also formalise the correctness of matching trees generated by MTG. Theorem 4.2 states that a generated matching tree ψ is valid and comprised of the same set of simple assertions as $\overrightarrow{pp}$. Again, we state the derivation in terms of an equality instead of the $\rightsquigarrow$ operator, denoting a deterministic application of the non-deterministic procedure.

Theorem 4.2 (Correctness of Matching Tree Generation)**.** *For any set of matching plans $\overrightarrow{pp} \in \wp(\mathbb{MP})$ and set of variables $X \subseteq \mathcal{X}$, it holds that:*

$$\mathsf{MTG}(\overrightarrow{pp},\, X) = \psi \implies \mathsf{asrts}(\psi) = \mathsf{asrts}(\overrightarrow{pp}) \wedge \mathsf{valid}(\psi,\, X)$$

where $\mathsf{asrts}(\overrightarrow{pp})$ (likewise, $\mathsf{asrts}(\psi)$) denotes the set of simple assertions in $\overrightarrow{pp}$ (likewise, ψ).

Proof. The proof is by induction on the structure of the derivation of $\mathsf{MTG}(\overrightarrow{pp}, X) \rightsquigarrow \psi$. For convenience, we restate the hypothesis of the theorem: $\mathsf{MTG}(\overrightarrow{pp}, X) = \psi$ **(H1)**. The base case is **[EMPTY]**, whereas the inductive cases are **[SINGLE NODE]** and **[DOUBLE NODE]**.

[EMPTY] Let $\overrightarrow{pp} = \{[], \ldots, []\}$. In this case, it follows from **H1** that:

I1. $\psi = \mathsf{MTG}(\overrightarrow{pp}, X) = \mathsf{MTG}(\{[]\}, X) = \bullet$

From **I1**, we conclude that:

I2. $\mathsf{asrts}(\psi) = \varnothing = \mathsf{asrts}(\overrightarrow{pp})$

I3. $\mathsf{valid}(\psi, X) = \mathsf{valid}(\bullet, X) = True$

[DOUBLE NODE] It follows from the case that there are two sets of matching plans $\overrightarrow{pp}_t$ and $\overrightarrow{pp}_f$, an assertion π, and matching trees ψ_t and ψ_f such that:

H2. $\overrightarrow{pp} = \overrightarrow{pp}_t \uplus \overrightarrow{pp}_f$

H3. $\forall \vec{p} \in \overrightarrow{pp}_t.\ \pi \in \vec{p}$

H4. $\forall \vec{p} \in \overrightarrow{pp}_f.\ \neg\pi \in \vec{p}$

H5. $\mathsf{ins}(\pi) \subseteq X$

H6. $\mathsf{MTG}(\overrightarrow{pp}_t \setminus_{ss} \pi, X) = \psi_t$

H7. $\mathsf{MTG}(\overrightarrow{pp}_f \setminus_{ss} \neg\pi, X) = \psi_f$

H8. $\psi = \langle \pi, \psi_t, \psi_f \rangle$

Applying the induction hypothesis to **H6** and **H7**, we conclude that:

I1. $\mathsf{asrts}(\psi_t) = \mathsf{asrts}(\overrightarrow{pp}_t \setminus_{ss} \pi)$

I2. $\mathsf{valid}(\psi_t, X)$

I3. $\mathsf{asrts}(\psi_f) = \mathsf{asrts}(\overrightarrow{pp}_f \setminus_{ss} \neg\pi)$

I4. $\mathsf{valid}(\psi_f, X)$

From **H2**-**H4**, it follows that:

I5. $\mathsf{asrts}(\overrightarrow{pp}) = \mathsf{asrts}(\overrightarrow{pp}_t \setminus_{ss} \pi) \cup \mathsf{asrts}(\overrightarrow{pp}_f \setminus_{ss} \neg\pi) \cup \{\pi, \neg\pi\}$

From **H8**, it follows that:

I6. $\mathsf{asrts}(\psi) = \mathsf{asrts}(\psi_t) \cup \mathsf{asrts}(\psi_f) \cup \{\pi, \neg\pi\}$

From **I1**, **I3**, **I5** and **I6**, we conclude that:

I7. $\mathsf{asrts}(\psi) = \mathsf{asrts}(\overrightarrow{pp})$

From **I2**, **I4**, **H5** and **H8**, it follows from Definition 4.5 that:

I8. $\mathsf{valid}(\langle \pi, \psi_t, \psi_f \rangle, X) \equiv \mathsf{valid}(\psi, X)$

[SINGLE NODE] The argument is similar to **[DOUBLE NODE]**.

□

Matching Trees for Specifications. So far, we have seen that matching trees provide us with a mechanism to determine the bindings of existentially quantified variables present in disjunctions of assertions. In particular, when applied to predicate definitions, they offer us a mechanism to calculate the output parameter of a given predicate from its input parameters; for example, given the previously introduced abs predicate, the derived valid matching tree explicitly shows how to calculate the absolute value of a number x. But how can we leverage matching trees to reason about program specifications? Consider the specification shown below for a function f that calculates the absolute value of the first element stored in a linked list:

$$\begin{array}{l} \{\, x \mapsto \#y \ast \#y \geq 0 \,\} \ \mathsf{fn}\ f(x)\ \{\, ret := \#y \,\} \\ \{\, x \mapsto \#y \ast \#y < 0 \,\} \ \mathsf{fn}\ f(x)\ \{\, ret := -\#y \,\} \end{array} \tag{4.4}$$

This specification is composed of two Hoare triples, each modeling one of the cases: in the first case, x points to a positive value, so the function returns the value itself; in the second case, x points to a negative value, so the function returns the symmetric. Just as with the various cases of a predicate, we can construct a matching tree that models the various possible pre-conditions of a given function. In this case, a valid matching tree would be $\psi = \langle x \mapsto \#y, \langle \#y \geq 0, \bullet, \bullet \rangle \rangle$. This matching tree makes it easy to determine the value of the logical variable $\#y$ from the parameter x. Similarly, using the value of the variable $\#y$ to compute the function's return value as specified in the post-condition through the variable ret is a trivial task.

Notice that one could also model this behaviour through a predicate capturing the various cases of the specification. For instance, we could write an auxiliary predicate f_{aux} as follows:

$$\begin{array}{l} \mathsf{pred}\ f_{aux}(x; z)\,\{ \\ \quad x \mapsto \#y \ast \#y \geq 0 \ast z := \#y; \\ \quad x \mapsto \#y \ast \#y < 0 \ast z := -\#y \\ \} \end{array} \tag{4.5}$$

capturing the behaviour of the function f given above. Then, we could rewrite f as:

$$\{\, f_{aux}(x; z) \,\} \ \mathsf{fn}\ f(x)\ \{\, ret := z \,\} \tag{4.6}$$

Hence, we expect developers to model branching within specifications using a branching predicate appearing in the pre-condition. Note that this does not dispense with the need for matching trees in the

context of function specifications; it simply means that those matching trees will have a single branch (i.e., they will coalesce a single matching plan corresponding to the pre-condition).

In what follows, we use matching trees both in the context of predicate definitions to calculate the output parameter of a predicate from its input parameters, and in the context of function specifications to calculate the values of logical variables appearing in the pre-conditions from the function's parameters.

4.6 Code Generation

Having formalised the derivation of matching trees, we are now in a position to explain how we use them to drive the actual code synthesis process. In §4.6.1 we define a pseudo-language for the generated code based on *statements*; in §4.6.2 we explain the compilation process from matching trees to our statement language.

4.6.1 Syntax

The syntax of statements is given below. Statements, $s \in \mathcal{S}tmt$, include: **(i)** the skip primitive; **(ii)** typed operations: the standard variable assignment; symbolic variable generation, $x := (\tau)\ \mathsf{symvar}()$; the memory read operation, $x := (\tau)\ *e$; and function calls, $x := (\tau)\ f(\bar{e})$; **(iii)** sequenced statements; **(iv)** the if-then-else conditional; **(v)** the assume and assert commands; and **(vi)** the return command. Functions are of the form $\mathsf{fn}\ f(\bar{x})\ \{\ s\ \}$, where f is the function identifier, $\bar{x}$ the set of parameters, and s is the function body.

The syntax of statements

$$s \in \mathcal{S}tmt \triangleq \mathsf{skip} \mid x := (\tau)\ e \mid x := (\tau)\ \mathsf{symvar}() \mid x := (\tau)\ *e \mid x := (\tau)\ f(\bar{e}) \mid s_1; s_2 \mid \mathsf{if}\ (\pi)\ \{s_1\}\ \mathsf{else}\ \{s_2\} \mid \mathsf{assume}\ \pi \mid \mathsf{assert}\ \pi \mid \mathsf{return}\ e$$

4.6.2 Compilation

We can now explain how the compilation process works. In particular, our goal is to show how to compile a function specification to executable code that can be run symbolically. For instance, consider the specification shown below for a function f that models the absolute value of a given number through the abs predicate (see §4.5):

$$\{\ abs(x; y)\ \}\ \mathsf{fn}\ f(x)\ \{\ ret := y\ \} \tag{4.7}$$

$$
\begin{aligned}
\mathcal{C}_{\mathsf{asrt}}(\pi) &\triangleq \mathsf{assert}\ \pi \\
\mathcal{C}_{\mathsf{asrt}}(x := e) &\triangleq x := e \\
\mathcal{C}_{\mathsf{asrt}}(x \mapsto_\tau y) &\triangleq y := (\tau) * x \\
\mathcal{C}_{\mathsf{asrt}}(x \mapsto_\tau v) &\triangleq y := (\tau) * x; \mathsf{assert}\ y = v \\
\mathcal{C}_{\mathsf{asrt}}(\alpha(\bar{e}; x)) &\triangleq x := \mathit{fold}_\alpha(\bar{e}) \\
\mathcal{C}_{\mathsf{asrt}}(\alpha(\bar{e}; v)) &\triangleq x := \mathit{fold}_\alpha(\bar{e}); \mathsf{assert}\ x = v
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{\mathsf{tree}}(\bullet) &\triangleq \mathsf{skip} \\
\mathcal{C}_{\mathsf{tree}}(\langle p, \psi \rangle) &\triangleq \mathcal{C}_{\mathsf{asrt}}(p); \mathcal{C}_{\mathsf{tree}}(\psi) \\
\mathcal{C}_{\mathsf{tree}}(\langle \pi, \psi_1, \psi_2 \rangle) &\triangleq \mathsf{if}\ (\pi)\ \{\mathcal{C}_{\mathsf{tree}}(\psi_1)\} \\
&\qquad\qquad \mathsf{else}\ \{\mathcal{C}_{\mathsf{tree}}(\psi_2)\}
\end{aligned}
$$

Figure 4.7: Compilation Functions: $\mathcal{C}_{\mathsf{asrt}}$ (left) and $\mathcal{C}_{\mathsf{tree}}$ (right)

Here, our goal is to compile the following functions:

$$
\begin{array}{l}
\mathsf{fn}\ \mathit{fold}_{abs}(x)\,\{ \\
\quad \mathsf{if}\ (x \geq 0)\ \{\, y := x \,\} \\
\quad \mathsf{else}\ \{\, y := -x \,\}; \\
\quad \mathsf{return}\ y \\
\}
\end{array}
\tag{4.8}
$$

$$
\begin{array}{l}
\mathsf{fn}\ f(x)\,\{ \\
\quad y := \mathit{fold}_{abs}(x); \\
\quad \mathsf{return}\ y \\
\}
\end{array}
\tag{4.9}
$$

The compilation process for such a function is a complex procedure comprised of a series of simpler steps. We explain the process in the following (bottom-up) order:

- **Assertions and Matching Trees.** Assertions are the smallest unit in the compilation process and are directly compiled to statements that compute the assertion's output parameters from its input parameters. Likewise, matching trees are compiled to statements that, given the set of input parameters, compute the output parameters of the matching tree.
- **Predicate Definitions.** A predicate definition is compiled to a function that computes the output parameter of the predicate from its input parameters. We call these *fold functions*, written fold_α for a given predicate α.
- **Function Specifications.** The compilation of function specifications is the ultimate goal of the compilation process. Thus, a specification is compiled to a function that computes its return value from the set of formal parameters, along with any free logical variables appearing in the post-condition.

Assertions and Matching Trees. We start by discussing the compilation of assertions and matching trees. For instance, given the abs predicate, we want to transform the matching tree $\psi = \langle x \geq 0, \langle y := x, \bullet \rangle, \langle y := -x, \bullet \rangle \rangle$ into the following statement:

$$
\mathsf{if}\ (x \geq 0)\ \{\, y := x \,\}\ \mathsf{else}\ \{\, y := -x \,\} \tag{4.10}
$$

To this end, we introduce a compilation function $\mathcal{C}_{\mathsf{tree}} : \Psi \to \mathcal{S}tmt$, formalised in Figure 4.7, along with an auxiliary function $\mathcal{C}_{\mathsf{asrt}} : \mathcal{SA} \to \mathcal{S}tmt$ for compiling simple assertions. Informally, $\mathcal{C}_{\mathsf{tree}}$ receives

a valid matching tree $\psi \in \Psi$ and compiles the corresponding statement, while $\mathcal{C}_{\mathsf{asrt}}$ does the same for simple assertions. Broadly, $\mathcal{C}_{\mathsf{asrt}}$ has four main cases:

- **Boolean Assertions.** A Boolean assertion π is compiled to the statement $\mathsf{assert}\ \pi$ (i.e., the generated code checks whether π holds in the current program state), since a Boolean assertion has no output parameters.
- **Directed Equalities.** A directed equality, $x := e$, is compiled to the equivalent statement, which computes the output parameter x.
- **Cell Assertions.** We consider two possibilities: if the cell assertion is of the form $x \mapsto_\tau y$, with $y \in \mathcal{X}$, it is compiled to a statement that assigns to y the value pointed to by x (with type τ); if it is of the form $x \mapsto_\tau v$, with $v \in \mathcal{V}$, we assign the value pointed to by x to a temporary variable y and add an $\mathsf{assert}\ y = v$ statement.
- **Predicate Assertions.** We consider two possibilities: if the predicate assertion is of the form $\alpha(\bar{e}; x)$, with $x \in \mathcal{X}$, it is compiled to a statement that assigns to x the value returned by $fold_\alpha(\bar{e})$ (i.e., the output parameter of the predicate); if it is of the form $\alpha(\bar{e}; v)$, with $v \in \mathcal{V}$, we assign the value returned by $fold_\alpha(\bar{e})$ to a temporary variable x and add an $\mathsf{assert}\ x = v$ statement. Intuitively, we assume the existence of a predicate definition for α, as well as its corresponding fold function.

Conversely, $\mathcal{C}_{\mathsf{tree}}$ has three cases:

- **Leaf Nodes.** The leaf node is simply compiled to the skip instruction.
- **Single Nodes.** A single node $\langle p, \psi \rangle$ is compiled as follows: first, we compile the simple assertion p with $\mathcal{C}_{\mathsf{asrt}}$; then, we compile the child tree ψ with $\mathcal{C}_{\mathsf{tree}}$ and sequence both.
- **Double Nodes.** A double node $\langle \pi, \psi_1, \psi_2 \rangle$ is compiled to the if-then-else conditional, with π as the condition and the compiled trees ψ_1 and ψ_2 as each branch respectively.

Predicate Definitions. Having gone through the compilation of matching trees, compiling a predicate definition is then relatively straightforward. Informally, our goal is to compile a predicate definition to a function $fold_\alpha$ that, given its input parameters, computes the output parameter. For instance, we can compile the full abs predicate into the following function:

$$
\begin{array}{l}
\mathsf{fn}\ fold_{abs}(x)\,\{ \\
\quad \mathsf{if}\ (x \geq 0)\ \{\, y := x \,\} \\
\quad \mathsf{else}\ \{\, y := -x \,\}; \\
\quad \mathsf{return}\ y \\
\}
\end{array}
\tag{4.11}
$$

Algorithm 4.1 describes our procedure for compiling predicate definitions of the form pred $\alpha(\bar{x}; y)\,\{\bar{P}\}$. We start by deriving a valid matching tree from the sequence of assertions $\bar{P}$ (line 1), which we then compile to a statement s with $\mathcal{C}_{\mathsf{tree}}$ (line 2). Finally, we sequence s with a return y command for the output parameter, which serves as the body of the compiled function (line 3).

Algorithm 4.1: COMPILEPRED compiles a predicate definition

Input: A predicate definition of the form pred $\alpha(\bar{x}; y)\,\{\bar{P}\}$
Output: The corresponding function fn $f(\bar{x})\,\{\,s\,\}$

1 $\psi \leftarrow \mathsf{MTG}([\,\mathsf{asrts}(P_i) \mid P_i \in \bar{P}\,],\, \bar{x})$
2 $s \leftarrow \mathcal{C}_{\mathsf{tree}}(\psi)$
3 **return** fn $fold_{\alpha}(\bar{x})\,\{\,s;$ return $y\,\}$

Function Specifications. Recall that compiling a function specification should generate the code that computes the function's return value based on its parameters. The difficulty lies in finding the bindings of logical variables that occur in the pre-condition and are used to describe the function's output in the post-condition; for example, the post-condition of the function f uses the logical variable y that appears in the pre-condition. To determine the bindings of logical variables that occur in the pre-condition, we simply derive and compile the corresponding valid matching tree. Once the bindings of the logical variables in the pre-condition are determined, computing the function's return value is straightforward.

Compiling specifications, however, is not always so simple. A common challenge in the compilation process is related to free logical variables in the post-condition. Consider, for example, the specification shown below for a function g that returns a random positive integer:

$$\{\,\mathsf{emp}\,\}\,\mathsf{fn}\,g()\,\{\,ret := \hat{x} * \hat{x} > 0\,\} \tag{4.12}$$

There are two issues here: how do we compute the return value not knowing the value of $\hat{x}$, and, having done so, how do we *restrict* the value of $\hat{x}$? Our mechanism for compiling specifications creates a new symbolic variable for each free variable in the post-condition and restricts the values these variables can take using an assume statement. In the case of g, it would generate the function:

$$\begin{array}{l}\mathsf{fn}\,g()\,\{\\ \quad \hat{x} := \mathsf{symvar}();\\ \quad \mathsf{assume}\ \hat{x} > 0;\\ \quad \mathsf{return}\ \hat{x}\\ \}\end{array} \tag{4.13}$$

Algorithm 4.2 describes our procedure for compiling function specifications. For the sake of simplicity, it assumes specifications of the form $\{P\}\,\mathsf{fn}\,f(\bar{x})\,\{\,ret := e * \pi\,\}$, where ret is the return value and π

a series of restrictions on free logical variables; nonetheless, it should be noted that the theory stays largely the same when dealing with multiple Hoare triples (such as those we saw in §4.5). Similarly to the compilation of predicate definitions, we start by deriving a valid matching tree from the assertion P (line 1), which we compile to a statement s with $\mathcal{C}_{\mathsf{tree}}$ (line 2). Then, we determine the set xs containing the free logical variables of the post-condition (line 3) and generate a fresh symbolic variable for each of these (line 4). Finally, we add an assume command for the constraints π and the return e statement for ret, and return the compiled function with the sequenced statements as its body (line 5).

Algorithm 4.2: COMPILESPEC compiles a function specification

Input: A spec of the form $\{P\}\ \mathsf{fn}\ f(\bar{x})\ \{\ ret := e * \pi\ \}$

Output: The corresponding function $\mathsf{fn}\ f(\bar{x})\ \{\ s\ \}$

1 $\psi \leftarrow \mathsf{MTG}([\mathsf{asrts}(P)],\ \bar{x})$

2 $s \leftarrow \mathcal{C}_{\mathsf{tree}}(\psi)$

3 $xs \leftarrow \mathsf{vars}(e) \cup \mathsf{vars}(\pi) \setminus \mathsf{vars}(P)$

4 $\vec{s}_{xs} \leftarrow [\ \hat{s}_i\ :=\ \mathsf{symvar}()\ \mid\ \hat{s}_i \in xs\ ]$

5 **return** $\mathsf{fn}\ f(\bar{x})\ \{\ s;\ \vec{s}_{xs};\ \mathsf{assume}\ \pi;\ \mathsf{return}\ e\ \}$

5

Specification-Driven Summary Synthesis

Contents

We introduce modifications to the code generation process in order to synthesise *summaries* instead of *functions*. In particular, we discuss the limitations of function synthesis (§5.1) and detail the generation process for *under-approximating* (§5.2) and *over-approximating* (§5.3) summaries.

5.1 Limitations of Function Synthesis

In §4.6, we described a procedure to compile function specifications to executable code. In practice, however, this procedure has a number of limitations. Consider, for instance, the specification shown below for the LIBC function strlen, which models the length of a C-style string pointed to by s:

$$\{\ str(s;\nu)\ \}\ \mathsf{fn}\ \mathsf{strlen}(s)\ \{\ ret := \nu\ \} \tag{5.1}$$

The pre-condition states that s points to a string of length ν (through a predicate str), while the post-condition simply states that the return value is the length of the string. The str predicate is shown below:

$$\begin{array}{l}\mathsf{pred}\ str(s;\nu)\ \{\\ \quad s \mapsto c\ *\ c = \text{`}\backslash 0\text{'}\ *\ \nu := 0;\\ \quad s \mapsto c\ *\ c \neq \text{`}\backslash 0\text{'}\ *\ \nu := \kappa + 1\ *\ str(s+1,\ \kappa)\\ \}\end{array} \tag{5.2}$$

The predicate models two cases: **(i)** s points to the null character (i.e., s is the empty string), in which case the length is 0, or **(ii)** s points to a non-null character (i.e., s is a non-empty string), in which case the length of s is obtained by adding 1 to the length of the string starting at $s + 1$. Thus, compiling the specification, we obtain the following functions:

$$\begin{array}{l}\mathsf{fn}\ \mathsf{strlen}(s)\ \{\\ \quad \nu := fold_{str}(s);\\ \quad \mathsf{return}\ \nu\\ \}\end{array} \tag{5.3}$$

$$\begin{array}{l}\mathsf{fn}\ fold_{str}(s)\ \{\\ \quad c := *s;\\ \quad \mathsf{if}\ (c = \text{`}\backslash 0\text{'})\ \{\ \nu := 0\ \}\\ \quad \mathsf{else}\ \{\\ \quad\quad \kappa := fold_{str}(s+1);\\ \quad\quad \nu := \kappa + 1\\ \quad \};\\ \quad \mathsf{return}\ \nu\\ \}\end{array} \tag{5.4}$$

Intuitively, the synthesised strlen function receives as argument a string s and returns its respective length. This is in line with expectations, since we are computing the function's return value from the set of formal parameters, as we emphasised in §4.

Let us now look at what happens in practice when invoking the synthesised function. We consider two cases: the concrete case and the symbolic case. The first is rather simple; take, for example, the concrete string $[\text{`}f\text{'}, \text{`}o\text{'}, \text{`}o\text{'}, \text{`}\backslash 0\text{'}]$. Clearly, passing it to strlen would return 3, which is indeed the correct length. Consider, however, the case of symbolic strings; what happens, for instance, if we give it $[\hat{c}_1, \hat{c}_2, \text{`}\backslash 0\text{'}]$ as argument? Does strlen return a length of 0, 1 or 2? In fact, a symbolic execution engine will branch once it hits the statement if $(c = \text{`}\backslash 0\text{'})$ if c is a symbolic value, meaning that both $\hat{c}_1$ and $\hat{c}_2$ cause the engine to branch. Thus, strlen models all possible lengths (i.e., 0, 1 and 2), but does so by branching, which means that the issue of path explosion is still present in the synthesised functions.

Our goal is to synthesise *summaries*, which we expect to address the path explosion problem. To that end, we need to introduce a number of changes to the code generation process. In particular, these changes allow us to compile function specifications to *under-approximating* (§5.2) or *over-approximating* (§5.3) summaries.

5.2 Under-Approximating Compilation

An under-approximating (UX) summary, as we have already seen, models a subset of the paths generated by the symbolic execution of the concrete function. Thus, a possible approach to synthesise such summaries is to drop feasible paths each time the symbolic execution might branch. This is indeed the approach we take, as we allow developers to annotate their specifications with a default path that will never be dropped. The responsibility of choosing this default path lies with the developer, and the choice can be made according to their specific needs. Then, in the synthesised code, if statements are generated in such a way that they avoid branching; that is to say, when branching is possible, the symbolic execution engine follows the default branch and discards non-default ones.

To illustrate our methodology, let us again consider the str predicate. When analysing a symbolic character within a string, the most common case is for it to be a non-null character. In line with this reasoning, the developer may choose to specify the recursive case of the predicate as the default path. In the following, we identify the default case by underlining it, as shown below:

$$\begin{array}{l}\text{pred } str(s;\nu)\,\{\\ \quad s \mapsto c * c = \text{`}\backslash 0\text{'} * \nu := 0;\\ \quad \underline{s \mapsto c * c \neq \text{`}\backslash 0\text{'} * \nu := \kappa + 1 * str(s+1,\,\kappa)}\\ \}\end{array} \tag{5.5}$$

Taking the default case information into account, we synthesise the $fold_{str}$ function as follows:

$$
\begin{array}{l}
\mathsf{fn}\ fold_{str}(s)\,\{ \\
\quad c := *s; \\
\quad \mathsf{if}\ (\mathsf{isCertain}(c = \text{`}\backslash 0\text{'}))\ \{\, \nu := 0 \,\} \\
\quad \mathsf{else}\ \{\, \mathsf{assume}\ c \neq \text{`}\backslash 0\text{'};\ \kappa := fold_{str}(s+1);\ \nu := \kappa + 1 \,\}; \\
\quad \mathsf{return}\ \nu \\
\}
\end{array}
\tag{5.6}
$$

where $\mathsf{isCertain}(\pi)$ is a special predicate that checks if a Boolean assertion π is implied by the current path condition.

Notice that if we now feed the symbolic string $[\hat{c}_1, \hat{c}_2, \text{`}\backslash 0\text{'}]$ to strlen, only the path where the length is 2 is modelled. Why? Because the symbolic execution engine cannot reason with certainty about $\hat{c}_1$ and $\hat{c}_2$; thus, it assumes the default path (i.e., that they are non-null characters), and continues calling $fold_{str}$ recursively. Only in the case of the concrete null terminator does it follow the corresponding path, since $\mathsf{isCertain}(\text{`}\backslash 0\text{'} = \text{`}\backslash 0\text{'})$ is trivially true.

A natural question to ask at this point is whether the same applies to function specifications. If a specification leads to branching, can we use a similar approach for summary synthesis? The answer is yes, although in practice specifications do not allow branching, since we restrict ourselves to single Hoare triples (see §4.5).

Luckily, most of our infrastructure for code synthesis can be reused for the generation of under-approximating summaries. In particular, only the compilation function $\mathcal{C}_{\mathsf{tree}}$ needs to be changed to take default paths into account. To that end, we introduce *under-approximating matching trees* $\zeta \in Z$, which come annotated with their respective default path:

$$
\zeta \in Z \triangleq \circledcirc \mid \lhd p, \zeta \rhd \mid \lhd \pi, \zeta, \psi \rhd_l \mid \lhd \pi, \psi, \zeta \rhd_r \tag{5.7}
$$

where: **(i)** the leaf node $\circledcirc$ and the single node $\lhd p, \zeta \rhd$ are similar in form and purpose to those of regular matching trees; **(ii)** the left-default double node $\lhd \pi, \zeta, \psi \rhd_l$ denotes a double node where the default path is the left branch; and **(iii)** the right-default double node $\lhd \pi, \psi, \zeta \rhd_r$ denotes a double node where the default path is the right branch. Notice that the non-default paths are just regular matching trees. With this information in mind, we can derive an under-approximating matching tree for the str predicate as follows:

$$
\lhd\ s \mapsto c, \lhd c = \text{`}\backslash 0\text{'}, \langle \nu := 0, \bullet \rangle, \lhd str(s+1, \kappa), \lhd \nu := \kappa + 1, \circledcirc \rhd \rhd \rhd_r \rhd \tag{5.8}
$$

Note that the double node that captures the test $c = \text{`}\backslash 0\text{'}$ is a right-default double node, expressing the fact that if it cannot be proven that c is the null terminator, we assume that it is not.

$$
\begin{aligned}
\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\circledcirc) &\triangleq \mathsf{skip} \\
\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\lhd p,\, \zeta \rhd) &\triangleq \mathcal{C}_{\mathsf{asrt}}(p);\, \mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\zeta) \\
\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\lhd \pi,\, \zeta,\, \psi \rhd_l) &\triangleq \mathsf{if}\ (\mathsf{isCertain}(\neg\pi))\ \{\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\psi)\} \\
&\qquad \mathsf{else}\ \{\mathsf{assume}\ \pi;\, \mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\zeta)\} \\
\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\lhd \pi,\, \psi,\, \zeta \rhd_r) &\triangleq \mathsf{if}\ (\mathsf{isCertain}(\pi))\ \{\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\psi)\} \\
&\qquad \mathsf{else}\ \{\mathsf{assume}\ \neg\pi;\, \mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}(\zeta)\}
\end{aligned}
\qquad
\begin{aligned}
\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\bullet) &\triangleq \mathsf{skip} \\
\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\langle p,\, \psi\rangle) &\triangleq \mathcal{C}_{\mathsf{asrt}}(p);\, \mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\psi) \\
\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\langle \pi,\, \psi_1,\, \psi_2\rangle) &\triangleq \mathsf{if}\ (\mathsf{isCertain}(\pi))\ \{\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\psi_1)\} \\
&\qquad \mathsf{else}\ \{\mathsf{if}\ (\mathsf{isCertain}(\neg\pi))\ \{\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}(\psi_2)\} \\
&\qquad \mathsf{else}\ \{\mathsf{assume}\ \mathsf{false}\}\}
\end{aligned}
$$

Figure 5.1: Compilation Functions: $\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}$ (left) and $\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}$ (right)

Having established a new format for under-approximating trees, we must now define a new compiler that translates them into executable code. Figure 5.1 introduces the new compiler, modeled as a function $\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}} : Z \to \mathcal{S}tmt$ that maps under-approximating matching trees to the corresponding statements. The compiler $\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}$ makes use of an auxiliary compiler $\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}} : \Psi \to \mathcal{S}tmt$ that translates exact matching trees to under-approximating code. As is the case in regular compilation, the synthesised code should compute the output parameters of the matching tree from the set of input parameters, but it should also guide execution towards the default path when it cannot reason about the symbolic parameters with absolute certainty. Thus, the compiler $\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}$ has four cases:

- **Leaf Nodes.** The leaf node is simply compiled to the skip instruction.
- **Single Nodes.** A single node $\lhd p,\, \zeta \rhd$ is compiled by sequencing the compilation of p with the compilation of ζ.
- **Left-Default Double Nodes.** A left-default double node $\lhd \pi,\, \zeta,\, \psi \rhd_l$ means that the left branch is the default one. Consequently, we only follow the right branch if $\neg\pi$ is certain; if it is not, we follow the left and assume its constraints by beginning the branch with an $\mathsf{assume}\ \pi$ command.
- **Right-Default Double Nodes.** A right-default double node is compiled in an analogous way to the left-default one.

Note that when compiling non-default branches, we use the auxiliary compiler instead of the main one. The key distinction between the compilers lies in their treatment of conditionals. The auxiliary compiler $\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}$ only allows a conditional branch to be taken if its guard is implied by the current path condition. In contrast, the main compiler $\mathcal{C}^{\mathsf{ux}}_{\mathsf{tree}}$ allows for the default branch to be taken even when it cannot prove its guard. In essence, we exclusively employ the auxiliary compiler for non-default paths, for which one has to prove their exact conditions in order to follow them. The auxiliary compiler $\mathcal{A}^{\mathsf{ux}}_{\mathsf{tree}}$ has three cases:

- **Leaf Nodes.** The leaf node is simply compiled to the skip instruction.
- **Single Nodes.** A single node $\langle p,\, \psi\rangle$ is compiled by sequencing the compilation of p with the compilation of ψ.

- **Double Nodes.** A double node $\langle \pi, \psi_1, \psi_2 \rangle$ means that neither ψ_1 nor ψ_2 are the default path. Consequently, we should only follow them if either π or $\neg\pi$ is certain. If that is not the case, we discard the current path by adding an assume false command.

5.3 Over-Approximating Compilation

An over-approximating (OX) summary models a superset of the paths generated by the symbolic execution of the concrete function. In contrast to the under-approximating case, one cannot discard paths when synthesising over-approximating summaries, since they have to model all paths that the symbolic execution of the concrete function would generate. In the case of matching trees, this means that we cannot simply choose a default branch when compiling double nodes. Instead, we need to compute a set of simple assertions shared by both branches, and use these to generate a single over-approximating case. More concretely, a double node is compiled to an if-then-else conditional with three cases:

- The case where the guard is proven to hold.
- The case where the negation of the guard is proven to hold.
- The over-approximating case, where we create a fresh symbolic variable representing the output parameter of the predicate and constrain it with the the simple assertions shared by both branches.

An important caveat of our methodology is that it does not automatically find implied shared assertions. Thus, if the developer wants any such assertion to be taken into account by the synthesis procedure, they should add them explicitly to the various branches of a predicate.

To illustrate our methodology, let us again look at the str predicate. Although no shared set of simple assertions is explicit, it is relatively straightforward to see that one can augment both cases with a redundant simple assertion $\nu \geq 0$, capturing the fact that the length of a string is always non-negative. We can therefore rewrite the str predicate with this shared simple assertion highlighted, as shown below:

$$
\begin{array}{l}
\mathsf{pred}\ str(s; \nu)\,\{ \\
\quad s \mapsto c * c = \text{`}\backslash 0\text{'} * \nu := 0 * \boxed{\nu \geq 0}; \\
\quad s \mapsto c * c \neq \text{`}\backslash 0\text{'} * \nu := \kappa + 1 * str(s+1, \kappa) * \boxed{\nu \geq 0} \\
\}
\end{array}
\tag{5.9}
$$

$$
\begin{aligned}
\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\bullet, X) &\triangleq \mathsf{skip} \\
\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\langle p,\, \psi\rangle, X) &\triangleq \mathcal{C}_{\mathsf{asrt}}(p);\, \mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\psi, X \cup \mathsf{outs}(p)) \\
\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\langle \pi,\, \psi_1,\, \psi_2\rangle, X) &\triangleq \mathsf{if}\ (\mathsf{isCertain}(\pi))\ \{\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\psi_1, X)\} \\
&\quad\ \mathsf{else}\ \{\mathsf{if}\ (\mathsf{isCertain}(\neg\pi))\ \{\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}(\psi_2, X)\} \\
&\quad\ \mathsf{else}\ \{\mathsf{OX\text{-}CODE}(\psi_1, \psi_2, X)\}\}
\end{aligned}
$$

Figure 5.2: Compilation Function: $\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}$

Equipped with the knowledge of the shared assertion, we synthesise the $fold_{str}$ function as follows:

$$
\begin{array}{l}
\mathsf{fn}\, fold_{str}(s)\,\{ \\
\quad c := *s; \\
\quad \mathsf{if}\ (\mathsf{isCertain}(c = \text{`}\backslash 0\text{'}))\ \{\, \nu := 0 \,\} \\
\quad \mathsf{else}\ \{ \\
\quad\quad \mathsf{if}\ (\mathsf{isCertain}(c \neq \text{`}\backslash 0\text{'}))\ \{\, \kappa := fold_{str}(s+1);\, \nu := \kappa + 1 \,\} \\
\quad\quad \mathsf{else}\ \{\, \nu := \mathsf{symvar}();\, \mathsf{assume}\ \nu \geq 0 \,\} \\
\quad \} \\
\quad \mathsf{return}\ \nu \\
\}
\end{array}
\tag{5.10}
$$

Note that, similarly to the under-approximating case, the synthesised strlen function does not actually change, only the fold function.

If we now feed the symbolic string $[\hat{c}_1, \hat{c}_2, \text{`}\backslash 0\text{'}]$ to strlen, the summary will output a fresh symbolic variable, say $\hat{v}$, that is assumed to be greater than or equal to 0, effectively reflecting the fact that it models all lengths greater than or equal to 0. In contrast, if $\hat{c}_1$ was a concrete character (for instance, in the string $[\text{`}a\text{'}, \hat{c}_2, \text{`}\backslash 0\text{'}]$), we would only model lengths greater than or equal to 1.

In contrast to the under-approximating case, over-approximating compilation does not require changes to the internal structure of matching trees. Thus, we simply need to redefine the compiler so that it can translate regular matching trees to over-approximating code. Figure 5.2 introduces the new compiler, modeled as a function $\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}} : \Psi \times \wp(\mathcal{X}) \to \mathcal{S}tmt$ that, given a set of known variables X, maps regular matching trees to the corresponding statements. In addition to computing the output parameters of the matching tree from the set of input parameters, the synthesised code should also capture the constraints of the over-approximating case when it cannot reason about the symbolic parameters with absolute certainty. Thus, the compiler $\mathcal{C}^{\mathsf{ox}}_{\mathsf{tree}}$ has three cases:

- **Leaf Nodes.** The leaf node is simply compiled to the skip instruction.
- **Single Nodes.** A single node $\langle p, \psi\rangle$ is compiled by sequencing the compilation of p with the compilation of ψ.
- **Double Nodes.** A double node $\langle \pi, \psi_1, \psi_2\rangle$ means that the execution has three possible outcomes:

either π is certain, in which case the chosen branch is ψ_1, $\neg\pi$ is certain, in which case the chosen branch is ψ_2, or neither is certain, in which case we follow the over-approximating path, as explained above.

Notice that the compiler makes use of a procedure OX-CODE to synthesise the over-approximating code. This procedure should address two main questions: how do we find the set of shared simple assertions, and how do we compile them once found? Algorithm 5.1 describes our implementation of OX-CODE. The algorithm works as follows: first, we extract the matching plans coalesced by ψ_1 and ψ_2 with the help of the tp function from §4.5 (line 1). Then, we find the Boolean assertions that appear in all matching plans (i.e., the set of simple assertions shared by ψ_1 and ψ_2) and create a new formula π with their conjunction (line 2). We then determine the set xs containing the free logical variables that occur in π (line 3); these will typically include the output parameter of the predicate. Lastly, we generate a fresh symbolic variable for each of the free variables (line 4), appending a final assume command to add π to the current path condition and returning the result (line 5).

Algorithm 5.1: OX-CODE compiles the set of simple assertions shared by the two branches of a double node

Input: Two matching trees ψ_1 and ψ_2 and a set of known variables X

Output: The compilation of the set of simple assertions shared by ψ_1 and ψ_2

1 $\overrightarrow{pp}_1, \overrightarrow{pp}_2 \leftarrow \mathsf{tp}(\psi_1), \mathsf{tp}(\psi_2)$

2 $\pi \leftarrow \bigwedge \{\, \pi' \mid \forall \vec{p} \in \overrightarrow{pp}_1 \cup \overrightarrow{pp}_2.\ \pi' \in \vec{p} \,\}$

3 $xs \leftarrow \mathsf{vars}(\pi) \setminus X$

4 $\vec{s}_{xs} \leftarrow [\, \hat{s}_i := \mathsf{symvar}() \mid \hat{s}_i \in xs \,]$

5 **return** $\vec{s}_{xs}$; assume π

6

Architecture and Implementation

Based on our methodology, we introduce SUMSYNTH, a tool that synthesises symbolic summaries from declarative specifications. More concretely, SUMSYNTH receives as input an SL-style specification for some target function as input and a flag indicating the synthesis mode (under-approximating or over-approximating), and outputs the corresponding symbolic summary. Importantly, SUMSYNTH allows the developer to synthesise summaries for a variety of languages by simply changing the code generation module, as will later become clear. Even though in our implementation we only synthesise summaries in C and Python, one could easily extend SUMSYNTH to other languages without much difficulty.

Let us now look at the inner workings of SUMSYNTH in more detail. Figure 6.1 gives an overview of the high-level architecture of SUMSYNTH. There are 4 main modules in the pipeline: **(1)** the parser module, **(2)** the matching engine, **(3)** the summary generator, and **(4)** the code generator. Below, we describe each of these:

1. SL Parser Module The parser module simply parses a function specification received as input, generating the corresponding abstract syntax tree (AST). Specifications are written in our custom SL-style assertion language (which we formalised in §4.2).
2. Matching Engine The matching engine is responsible for deriving matching plans and trees for the provided specifications, following the derivation procedures described in §4.4 and §4.5.
3. Summary Generator The summary generator module produces symbolic summaries in the intermediate language (IL) described in §4.6.1 and outputs them as an AST encoded as a JSON file. The generation process either follows the methodology for generating under-approximating summaries (§5.2) or the one for over-approximating summaries (§5.3). Notably, this module is also capable of synthesising functions according to the methodology described in §4.6.
4. Code Generator The code generator module (not to be confused with the code generation procedure described in §4.6) is simply a special-purpose transpiler to convert summaries produced in our IL to actual executable summaries that can then be given to a symbolic execution engine. In our implementation, we opt to generate summaries in C and Python. The C summaries are compatible with the tool-independent API developed by Ramos et al. [7], whereas the Python summaries are compatible with *angr* [8]; however, the developer may choose to expand this module to generate summaries for other languages.

How do these modules work in practice? We demonstrate the architecture via an example execution. Consider the specification given in Listing 6.1 for the LIBC function `atoi`, which converts a string to an integer. Notice that the specification is over-approximating, meaning that at the end we will generate an over-approximating summary. Let us then examine the synthesis process:

1. SL Parser Module The specification for `atoi` is fed to SUMSYNTH and parsed by the SL parser module. Notice that the specification follows the formalised grammar rather closely. There are,

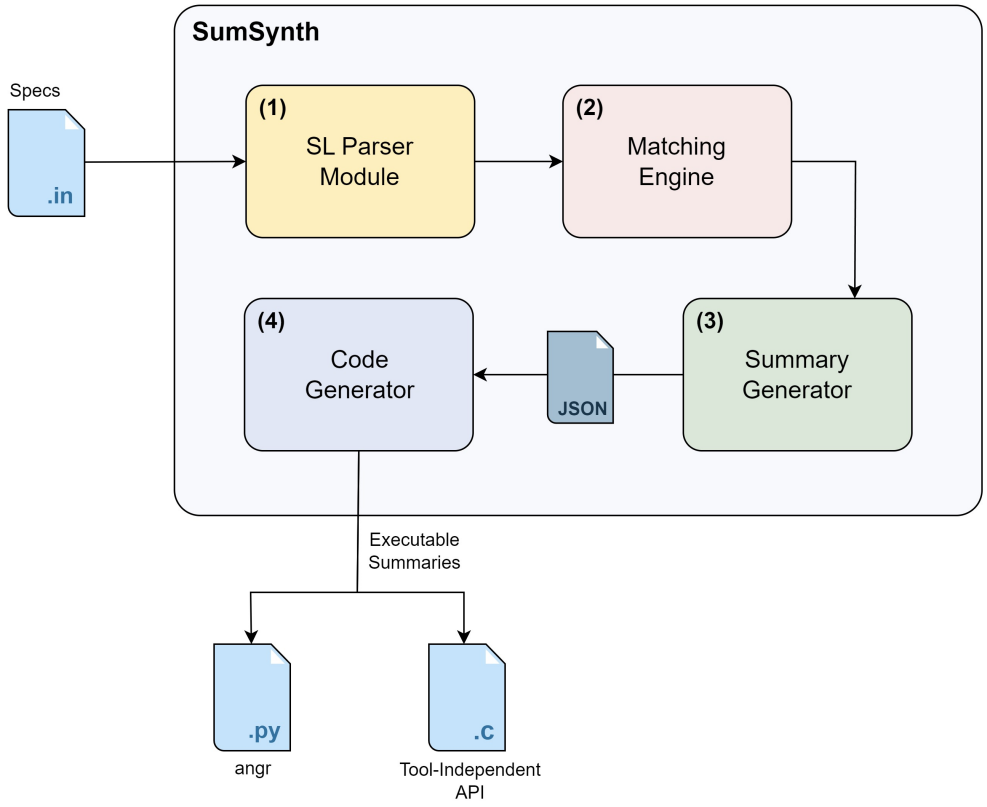


Figure 6.1: SUMSYNTH architecture

however, three main differences: **(i)** all types are explicit, whereas in the formal grammar only some operations are typed; **(ii)** assertions are not implicitly assumed to be existentially quantified; instead, they have to be explicitly annotated as such; **(iii)** there is no distinction between regular and directed equalities; there is a single equality operation (i.e., ==), and SUMSYNTH infers which of the two applies in each case based on the current set of known variables.

2. Matching Engine After the parsing step, the information is then passed to the matching engine, which is responsible for deriving valid matching trees for the specification and any predicate definitions that might exist. If we look at our running `atoi` example, for instance, we would derive the following matching tree for the predicate `str`[1]:

$$\langle s \mapsto c,\ \langle c = \text{`}\backslash 0\text{'},\ \langle \nu := 0,\ \ldots\rangle,\ \langle str(s+1,\ \kappa),\ \ldots\rangle\rangle\rangle \tag{6.1}$$

Note that the derivation process depends on the desired summary type, i.e., if we were generating an under-approximating summary, we would derive an under-approximating matching tree. Since the specification for `atoi` is over-approximating, we derive a regular matching tree (see §5.3).

[1] We omit parts of the matching tree for readability.

```
fn atoi(s: ^char) := {
    locals: (n: int32, i: int32, ret: int32)
    pre: str(s; n)
    post: ret == i <*> i <= 10 ** n <*> i >= -10 ** (n - 1)
}

pred str(s: ^char; n: int32) := {
    s -> '\0' <*> n == 0 <*> n >= 0;
    exists [c: char, k: int32] s -> c <*> c != '\0' <*>
                                  n == k + 1 <*> str(s + 1; k) <*> n >= 0
}
```

Listing 6.1: An over-approximating specification for `atoi`

3. Summary Generator The step that follows is the actual summary synthesis procedure. As we have previously mentioned, the purpose of the summary generator module is to synthesise summaries in our IL, which is syntactically similar to the statement language we formalised in §4.6.1. There are three different synthesis modes: function, under-approximating and over-approximating. The first follows the procedure detailed in §4.6, and is not of particular interest to us. The other two produce under- and over-approximating summaries according to the methodologies described in §5.2 and §5.3 respectively. Since the specification for `atoi` is over-approximating, SUMSYNTH can only synthesise an over-approximating summary (the converse is true for the under-approximating case). If the specification was exact, the developer would be able to specify a synthesis mode to generate either an under- or an over-approximating summary. In the end, the generated summary is encoded as an AST and passed on to the code generator module in a JSON file. Listing 6.2 shows an excerpt of the JSON encoding of the synthesised summary for `atoi`.

4. Code Generator In the final step, the JSON file produced by the summary generator is fed to a special-purpose transpiler to convert summaries produced in our IL to executable summaries in our target language. In our implementation, we offer the infrastructure to synthesise summaries both in C and Python, leaving to the developer the choice of possibly extending this to other languages. Listing 6.3 shows an excerpt of the synthesised Python summary for `atoi`, which is compatible with *angr* [8]. The corresponding C summary, omitted here for brevity, would follow a similar structure, using the symbolic reflection primitives proposed by Ramos et al. [7] instead of the *angr* primitives.

Implementation. SUMSYNTH was written in Haskell (using *stack*[2]) and Python. The Haskell portion includes Modules **(1)**-**(3)** (i.e., up until the summary generation step), while the Python portion implements Module **(4)**. The total number of lines of code (LoC) is ∼2.4k (source code only), roughly split between 1.5k LoC in Haskell and 850 LoC in Python. Additionally, there are multiple unit and end-to-end tests implemented across the various modules.

[2] https://docs.haskellstack.org/en/stable/

```
...
"name": "atoi",
"params": [ {
    "node": "decl",
    "name": "s",
    "type": {
        "node": "ptr",
        "type": {
            "node": "type",
            "name": "s",
            "type": {
                "node": "typeid",
                "type": "int8"
            }
        }
    }
} ]
...
```

Listing 6.2: JSON encoding of the synthesised summary for `atoi` (excerpt)

```
class atoi(Summary):
    def run(self, s):
        return self.atoi(s)

    def atoi(self, s):
        var1 = self.fold_str(s)
        aux1 = self.sym_var(
            self.arch.sizeof['int']
        )

        ret = aux1
        self.assume(self.And(
            self.Le(aux1, (10 ** var1)),
            self.Ge(aux1, -(10 ** (var1 - 1)))
        ))

        return ret
...
```

Listing 6.3: Synthesised Python summary for `atoi` (excerpt)

A major challenge during the development of SUMSYNTH was choosing the right solution to address the challenges arising from our intended architecture. Thus, we highlight some of the technologies we used and choices we made while developing SUMSYNTH:

- We use *Parsec*[3] [43] to handle the parsing of specifications. *Parsec* is a parser combinator library for Haskell that is both efficient and relatively easy to use, besides providing all the functionality we need for our tool.
- We use *MTL*[4] [44, 45] to tackle stateful computations, I/O actions and error handling. In particular, we make heavy use of the state monad [46] in the matching engine and summary generator modules, and of the I/O and error monads across the whole Haskell pipeline.
- We use *HSpec*[5], a popular testing framework, to test the Haskell modules.
- We use *pycparser*[6], a C parser written in Python, to parse ASTs from our IL to C ASTs. Besides allowing us to compile C summaries almost directly, the C ASTs generated by *pycparser* remove any ambiguities that might exist in the original AST. This means that, even when compiling Python code, it is useful to generate a C AST as an intermediate step before generating the Python AST.

[3] https://hackage.haskell.org/package/parsec
[4] https://hackage.haskell.org/package/mtl
[5] https://hspec.github.io/
[6] https://github.com/eliben/pycparser

7

Evaluation

Contents

This chapter answers the following evaluation questions:

EQ1: Is SUMSYNTH capable of synthesising under- and over-approximating summaries that are correct by construction?

EQ2: How complex are generated summaries in comparison to handcrafted summaries?

EQ3: How does the performance of generated summaries compare against that of handcrafted summaries (both tool-independent and tool-specific)?

7.1 EQ1: Synthesis Correctness

In order to evaluate the correctness of the synthesis process, we use SUMSYNTH to generate 63 symbolic summaries covering 34 LIBC functions from 4 header files (*string.h*, *stdlib.h*, *stdio.h* and *ctype.h*). As we mentioned in Chapter 6, we can choose between two synthesis modes: under-approximating and over-approximating. For most of the functions, we generate both types of summaries. In cases where that is not possible (i.e., when we can only write either an under- or over-approximating specification), we generate a single type. Thus, from the 63 synthesised summaries, 29 follow the under-approximating synthesis procedure and 34 follow the over-approximating one.

Table 7.1 shows, for each function, the type of the respective specification and of the generated summary(ies). For each summarized function, we place a checkmark in the Spec Under/Over/Exact column if the developed specification for that function matches the respective specification type. Analogously, we place a checkmark in the Summary Under/Over column if the generated summary is of the respective type. Rows *string.h*, *stdlib.h*, *stdio.h* and *ctype.h* show the aggregate number of specifications/summaries pertaining to each of the four libraries. The correctness properties of synthesised summaries were all double-checked with the help of SUMBOUNDVERIFY [7]. SUMBOUNDVERIFY is a summary validation tool that checks the correctness of summaries by comparing the paths modelled by the summary with the paths generated by the symbolic execution of the concrete function.

We checked the correctness of all summaries except for those modelling I/O functions, which we highlighted with an asterisk ($*$). These cannot be verified against their reference implementations because they interact with the runtime environment through *system calls*, meaning that their verification extends beyond the boundaries of the language and, consequently, of the symbolic execution engine. Nonetheless, we include them in our experiments, since we choose to ignore environment side effects.

It is notoriously challenging to write exact specifications for some functions (notably, in the case of number-parsing). We choose to over-approximate such functions, meaning that we can only synthesise the corresponding over-approximating summaries. The remaining functions are relatively straightforward to define through exact specifications. This implies that we can synthesise both under- and over-approximating summaries modelling these functions, as our results show.

Table 7.1: Correctness properties of the synthesised summaries

Function	Spec Under	Spec Over	Spec Exact	Summary Under	Summary Over
string.h	–	–	13	13	13
memchr	–	–	✓	✓	✓
memcmp	–	–	✓	✓	✓
strcasecmp	–	–	✓	✓	✓
strcasencmp	–	–	✓	✓	✓
strchr	–	–	✓	✓	✓
strcmp	–	–	✓	✓	✓
strcspn	–	–	✓	✓	✓
strlen	–	–	✓	✓	✓
strncmp	–	–	✓	✓	✓
strpbrk	–	–	✓	✓	✓
strrchr	–	–	✓	✓	✓
strspn	–	–	✓	✓	✓
strstr	–	–	✓	✓	✓
stdlib.h	–	4	1	1	5
abs	–	–	✓	✓	✓
atof	–	✓	–	–	✓
atoi	–	✓	–	–	✓
atol	–	✓	–	–	✓
rand	–	✓	–	–	✓
*stdio.h**	–	1	2	2	3
getchar	–	✓	–	–	✓
putchar	–	–	✓	✓	✓
puts	–	–	✓	✓	✓
ctype.h	–	–	13	13	13
isalnum	–	–	✓	✓	✓
isalpha	–	–	✓	✓	✓
iscntrl	–	–	✓	✓	✓
isdigit	–	–	✓	✓	✓
isgraph	–	–	✓	✓	✓
islower	–	–	✓	✓	✓
isprint	–	–	✓	✓	✓
ispunct	–	–	✓	✓	✓
isspace	–	–	✓	✓	✓
isupper	–	–	✓	✓	✓
isxdigit	–	–	✓	✓	✓
tolower	–	–	✓	✓	✓
toupper	–	–	✓	✓	✓

One of the key strengths of our methodology lies in how easy it is to write specifications and synthesise the corresponding summaries. Unlike their handcrafted counterparts, our summaries are correct by construction. As a result, we found them to be bug-free, which would be inconceivable if we wrote the summaries manually. Furthermore, our approach allowed us to swiftly develop summaries for 34 LIBC functions. Given the time, we would be able to synthesise an even greater number of summaries.

7.2 EQ2: Summary Complexity

We compare the complexity of summaries synthesised with SUMSYNTH against that of handcrafted summaries. In particular, we discuss whether there are any major differences between the two and what the developer gains from using our tool. In order to measure how complex a summary is, we focus on two metrics: the number of lines of code (LoC) and the number of calls to the API of the corresponding back end (N_{API}). To this end, we first discuss some of the challenges that arise when measuring complexity across different tools (§7.2.1), and then present our results (§7.2.2).

7.2.1 Challenges

The main challenge of measuring complexity across different back ends is the lack of uniformity between the various implementations. We subdivide this into three simpler challenges:

C1: How can we compare the complexity of summaries that satisfy different properties, and thus inherently possess distinct characteristics? We find that, on average, unsound summaries tend to be the shortest, exact summaries tend to be the longest, and under-approximating summaries tend to be shorter than over-approximating summaries. How can we make a fair comparison between different back ends using summaries satisfying different properties with varying degrees of correctness?

Solution. Our synthesis tool allows the developer to choose between generating an under- or over-approximating summary, so we measure their respective complexity separately. On the other hand, both the tool-independent API developed by Ramos et al. [7] and *angr* [8] lack uniformity: the tool-independent API often offers multiple summaries per function with varying degrees of correctness, while *angr* offers a single (usually exact) summary per function. To deal with this issue, we try to choose the summary that most closely resembles the synthesised summaries whenever possible.

C2: How are summaries linked to the symbolic execution engine? How does that affect their complexity? Consider, for instance, the `atoi` function. The corresponding summary tends to consistently have around 30-40 LoC, except in the case of the handcrafted *angr* summary, which has 224. This is because the latter is part of the `strtol` family of summaries, meaning that a lot of its code is shared with other summaries and not actually used in the particular `atoi` case.

Solution. We find that, in general, the variation in complexity due to linking differences is negligible. When it is not, as was the case with `atoi`, we explicitly highlight the function.

C3: How do we define the concept of an API call? In particular, how can we ensure that this definition is fair across both back ends, taking into account the fact that different tools use different symbolic reflection primitives?

Solution. We choose a definition that is consistent with our synthesis procedure, and at the same time fair to both back ends. Thus, an API call is defined as any call to the symbolic execution engine except those whose purpose is to build a constraint (e.g. *equal-to*, *greater-than*, *ITE*, etc.) or create a constant value (as is the case with *angr*'s *BVV*).

7.2.2 Results

In order to evaluate the complexity of synthesised summaries, we focus on the same set of functions as in §7.1. Table 7.2 shows, for each function, the complexity of the respective tool-independent summaries. Analogously, Table 7.3 shows the complexity of the *angr* summaries. For each function, the Spec column refers to the corresponding specification, the Manual column to the handcrafted summaries, and the Gen-Under and Gen-Over columns to the under- and over-approximating summaries synthesised with SUMSYNTH. The Type column under Manual refers to the correctness of the corresponding summary: UX for under-approximating, OX for over-approximating, X for exact and U for unsound. Rows *string.h*, *stdlib.h*, *stdio.h* and *ctype.h* show the median number of LoC/API calls for each of the four libraries. Note that some of these values might be misleading, since the number of implemented summaries varies considerably between handcrafted and synthesised summaries. Additionally, functions with an asterisk ($*$) in front indicate considerable variations in complexity resulting from differences in linking between both back ends (**C2**, §7.2.1).

Results show that synthesised summaries are, on average, less complex than their handcrafted counterparts. In particular, we see that under-approximating summaries tend to be the simplest, followed by over-approximating and then handcrafted ones. One might speculate that this is due to two factors. The first is that many of the handcrafted summaries are exact, which tend to be more complex than both under- and over-approximating summaries. The second, which is more interesting to our analysis, is that these summaries were written manually, without adhering to any specific structure and mostly through trial and error. Furthermore, we should also consider the fact that even the complexity of synthesised summaries is overstating the difficulty of developing them. In fact, one needs only to write a specification for the target function, as the actual summary generation process is handled by SUMSYNTH. As our results show, this is a relatively straightforward task; the number of LoC in a specification tends to be only a fraction of the length of the corresponding summaries.

Table 7.2: Complexity of the synthesised summaries (C)

Function	Spec	Manual			Gen-Under		Gen-Over	
	LoC	LoC	N_{API}	Type	LoC	N_{API}	LoC	N_{API}
string.h	14	38	12	–	47	6	71	12
memchr	10	38	12	UX	30	4	49	11
memcmp	10	34	6	U	31	4	45	6
strcasecmp	14	–	–	–	47	6	71	12
strcasencmp	15	–	–	–	56	8	87	15
strchr	10	73	16	X	30	4	49	11
strcmp	10	50	14	U	31	4	45	6
strcspn	15	–	–	–	56	8	89	17
strlen	9	23	4	X	22	2	32	6
strncmp	11	91	18	U	40	6	61	9
strpbrk	15	–	–	–	55	8	88	17
strrchr	17	38	6	X	56	8	92	20
strspn	15	–	–	–	56	8	89	17
strstr	16	–	–	–	65	10	105	20
stdlib.h	9	41	6	–	14	2	34	8
abs	9	–	–	–	14	2	30	6
atof	10	–	–	–	–	–	52	13
atoi	9	38	6	OX	–	–	34	8
atol	9	44	6	OX	–	–	34	8
rand	5	–	–	–	–	–	7	2
stdio.h	5	5	1	–	13.5	1	6	1
getchar	5	4	1	OX	–	–	6	1
putchar	5	5	1	X	5	0	5	0
puts	9	21	4	UX	22	2	32	6
ctype.h	9	25	3	–	20	2	27	3
isalnum	17	–	–	–	50	6	71	9
isalpha	9	–	–	–	20	2	27	3
iscntrl	9	–	–	–	20	2	27	3
isdigit	9	–	–	–	20	2	27	3
isgraph	13	–	–	–	35	4	49	6
islower	9	–	–	–	20	2	27	3
isprint	9	–	–	–	20	2	27	3
ispunct	29	–	–	–	95	12	137	18
isspace	9	20	3	X	20	2	27	3
isupper	9	–	–	–	20	2	27	3
isxdigit	13	–	–	–	35	4	49	6
tolower	9	25	3	X	20	2	30	6
toupper	9	25	3	X	20	2	30	6

Table 7.3: Complexity of the synthesised summaries (Python)

Function	Spec	Manual			Gen-Under		Gen-Over	
	LoC	LoC	N_{API}	Type	LoC	N_{API}	LoC	N_{API}
string.h	14	164	33	–	37	6	51	12
memchr	10	–	–	–	25	4	38	11
memcmp	10	54	12	X	26	4	34	6
strcasecmp	14	–	–	–	37	6	51	12
strcasencmp	15	–	–	–	42	8	61	15
strchr*	10	106	25	X	25	4	38	11
strcmp*	10	317	58	U	26	4	34	6
strcspn	15	–	–	–	42	8	63	17
strlen	9	77	15	X	21	2	27	6
strncmp*	11	222	41	U	31	6	44	9
strpbrk	15	–	–	–	41	8	62	17
strrchr	17	–	–	–	42	8	64	20
strspn	15	–	–	–	42	8	63	17
strstr*	16	368	68	U	47	10	73	20
stdlib.h	9	224	29	–	14	2	29	8
abs	9	–	–	–	14	2	25	6
atof	10	–	–	–	–	–	41	13
atoi*	9	224	29	U	–	–	29	8
atol*	9	224	29	U	–	–	29	8
rand	5	–	–	–	–	–	11	2
stdio.h	5	21	2	–	15	1	10	1
getchar	5	21	2	OX	–	–	10	1
putchar	5	8	1	X	9	0	9	0
puts*	9	89	16	X	21	2	27	6
ctype.h	9	4	0	–	19	2	22	3
isalnum	17	–	–	–	39	6	48	9
isalpha	9	–	–	–	19	2	22	3
iscntrl	9	–	–	–	19	2	22	3
isdigit	9	–	–	–	19	2	22	3
isgraph	13	–	–	–	29	4	35	6
islower	9	–	–	–	19	2	22	3
isprint	9	–	–	–	19	2	22	3
ispunct	29	–	–	–	69	12	87	18
isspace	9	–	–	–	19	2	22	3
isupper	9	–	–	–	19	2	22	3
isxdigit	13	–	–	–	29	4	35	6
tolower	9	4	0	X	19	2	25	6
toupper	9	4	0	X	19	2	25	6

Finally, we observe that there is a considerable number of functions for which there are neither tool-independent nor *angr* summaries. This happens even in popular functions such as `atof` and `strpbrk`, mostly due to a combination of implementation overhead and being deemed unimportant. With the help of SUMSYNTH, however, it is much easier to write summaries, meaning that developers need not neglect these functions any longer.

7.3 EQ3: Summary Performance

We compare the performance of summaries synthesised with SUMSYNTH against that of handcrafted summaries and measure the performance gains. In order to carry out this analysis, we use the symbolic test suite developed by Ramos et al. [7], which makes heavy use of LIBC functions, as opposed to other popular test suites [47, 48]. We give further details about the experimental setup in §7.3.1; then, we present our results in §7.3.2.

7.3.1 Experimental Setup

As the test bed for our experiments, we used *angr* [8] extended with support for synthesised summaries (both tool-independent and native). All tests were run on a Ubuntu machine (18.04.5 LTS) with an Intel Xeon E5–2620 CPU and 32GB of RAM. Each test was given 16GB of RAM, and had a maximum timeout of 30 minutes (1800 seconds).

Test Suites. To compare the performance of of summaries synthesised with SUMSYNTH against that of handcrafted summaries, we use a symbolic test suite developed by Ramos et al. [7]. The test suite is based on two open-source C libraries, both of which make heavy use of LIBC functions: **(i)** the *HashMap*[1] library, which provides an implementation of a standard hash table, and **(ii)** the *Dynamic Strings*[2] library, which augments the LIBC string handling functionality by adding support for heap-allocated strings. More concretely, there are actually two symbolic test suites, one for each library. The test suite for *HashMap* comprises 10 symbolic tests, while the test suite for *Dynamic Strings* comprises 12. Furthermore, the symbolic test suites were designed so as to cover all the functions exposed by the two libraries that interact with LIBC functions.

7.3.2 Results

We ran both test suites in *angr* using: **(i)** the C reference implementation of each function (Concrete); **(ii)** the handcrafted tool-independent summaries (C-Summaries); **(iii)** the handcrafted *angr* summaries

[1] https://gist.github.com/Richard-W/9568649
[2] https://github.com/antirez/sds

(Py-Summaries); **(iv)** the synthesised tool-independent summaries (C-Gen-Under and C-Gen-Over); and **(v)** the synthesised *angr* summaries (Py-Gen-Under and Py-Gen-Over). The reference implementations were obtained from *Verifiable C*[3], *glibc*[4] and *libiberty*[5], while the handcrafted C summaries were obtained from the tool-independent API developed by Ramos et al. [7] and the Python summaries from *angr* [8].

We present the summarised results in Table 7.4. We show for each test suite run: **(i)** the number of tests that failed because the engine ran out of memory (Mem. Out); **(ii)** the number of tests that failed because they exceeded the time limit (Timeout); **(iii)** the number of tests that executed successfully (Success); **(iv)** the average number of explored paths per test (Avg. N_{Paths}); **(v)** the average number of calls to LIBC functions per test (Avg. N_{LIBC}); **(vi)** the average number of API calls (Avg. N_{API}); and **(vii)** the average execution time per test (Avg. $Time$). Note that the average execution time for some of the runs might be misleading, since tests that exceed the time limit are recorded with the fixed maximum timeout of 1800 seconds. In contrast, the execution time for tests that fail due to lack of memory is considered to be the time elapsed until failure.

Table 7.4: Performance of the synthesised summaries

		Mem. Out ✗	Timeout ✗	Success ✓	Avg. N_{Paths}	Avg. N_{LIBC}	Avg. N_{API}	Avg. $Time$ (s)
Hash Map	Concrete	7	0	3	2.2k	6.4k	3.7k	1056.58
	C-Summaries	0	0	10	80	419	7.7k	201.66
	Py-Summaries	0	0	10	72	390	222	74.81
	C-Gen-Under	0	0	10	1	54	755	19.54
	C-Gen-Over	0	0	10	111	464	1.7k	86.07
	Py-Gen-Under	0	0	10	1	54	755	9.79
	Py-Gen-Over	0	0	10	109	454	1.7k	61.83
Dynamic Strings	Concrete	6	2	4	2.3k	3.8k	4.3k	744.25
	C-Summaries	1	0	11	424	483	4.0k	276.64
	Py-Summaries	1	0	11	454	361	97	132.184
	C-Gen-Under	1	0	11	340	288	1.5k	114.99
	C-Gen-Over	5	1	6	512	946	5.5k	636.33
	Py-Gen-Under	1	0	11	374	291	1.7k	101.71
	Py-Gen-Over	4	1	7	714	1.3k	25.5k	586.06

Unsurprisingly, results show that summaries, both handcrafted and automatically synthesised, easily surpass the performance of the reference implementations. Perhaps more interesting, however, is the comparison between the performance of handcrafted and synthesised summaries. In particular, we

[3] https://softwarefoundations.cis.upenn.edu/
[4] https://www.gnu.org/software/libc/
[5] https://gcc.gnu.org/onlinedocs/libiberty/

make two main observations. Firstly, we note that the automatically generated under-approximating summaries tend to be the most performant of the bunch. Notably, only a single test in the *Dynamic Strings* test suite, which also fails in all other cases, fails when using under-approximating summaries. In contrast, the execution time in the tests that do not fail is, on average, 3 to 5 times lower than that in the case of handcrafted summaries. This is to be expected, as such summaries drop some of the paths generated by the symbolic execution of the concrete function. It follows that the execution time should indeed be much lower than in other cases, since we model only a fraction of the paths we would otherwise.

On the other hand, the over-approximating case is not as straightforward. In particular, we observe that the use of automatically synthesised over-approximating summaries actually leads to an increase in performance in the *Hash Map* test suite. In contrast, in the *Dynamic Strings* test suite, there is a decrease in performance. In fact, the latter is to be expected, as over-approximating summaries have the opposite problem of their under-approximating counterparts: they model a superset of the execution paths generated by the symbolic execution of the concrete function, meaning that they might not drop spurious paths. The fact that over-approximating summaries are actually faster in the *Hash Map* test suite seems to indicate that the synthesis procedure is powerful enough for the performance of our over-approximating summaries to surpass even that of exact handcrafted summaries, although further research would have to be conducted to confirm this hypothesis.

Both these results highlight an important point: we cannot make a fair comparison between handcrafted and synthesised summaries if their correctness properties differ. Consider, for instance, the common case where the handcrafted summary is exact. If we compare it against an under-approximating summary, we compromise on the coverage guarantees. If we compare it against an over-approximating summary, we compromise on performance. Thus, we cannot conduct a comprehensive study on the performance of synthesised summaries unless we are able to automatically synthesise exact summaries. We plan to address this limitation in future work.

8

Conclusion

Contents

8.1 Conclusions

Symbolic summaries are a powerful way for modern symbolic execution engines to address the challenges of modelling interactions with the runtime environment and path explosion. Despite their potential, however, summaries run into a key problem: they require significant developer effort to produce, since interactions with the symbolic state must be implemented manually.

In this thesis, we proposed a new methodology to automate the creation of non-mutating symbolic summaries by synthesising them from declarative specifications. Our approach allows developers to automatically synthesise both under- and over-approximating summaries in a correct-by-construction manner through our SL-based synthesis tool. The tool is compatible with a variety of symbolic execution engines, and was successfully used to generate a set of 29 under-approximating summaries and 34 over-approximating summaries, modelling 34 LIBC functions. In order to evaluate the viability of our approach, we tested the performance and correctness of synthesised summaries against those of handcrafted summaries. Results clearly show that SUMSYNTH summaries surpass handcrafted summaries both in terms of correctness and, in some benchmarks, performance.

8.2 Future Work

So far, our work has focused on the synthesis of under- and over-approximating symbolic summaries. This leaves an obvious gap in the absence of a way to synthesise exact summaries. An exact summary, as we have mentioned previously, models the same set of paths generated by the symbolic execution of the concrete function. Thus, an approach to synthesise such summaries would be to use if-then-else constraints instead of conditionals when modeling branching.

Another notable research path is the synthesis of mutating summaries (i.e., summaries with side effects). As we have seen before, our tool only allows for the generation of non-mutating summaries, but this also means that there is a good number of LIBC functions that we cannot model, since they affect the heap. We intend to explore ways to solve this problem in the future.

Finally, we would like to formalise the soundness of the actual synthesis process. In particular, we intend to mathematically prove that the code generation procedure is sound, thus ensuring the correctness of the synthesised summaries.

Bibliography

[1] R. S. Boyer, B. Elspas, and K. N. Levitt, “SELECT—a formal system for testing and debugging programs by symbolic execution,” in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: Association for Computing Machinery, 1975, p. 234–245.

[2] J. C. King, “A new approach to program testing,” in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: Association for Computing Machinery, 1975, p. 228–233.

[3] ——, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[4] C. Barrett, D. Kroening, and T. Melham, *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*, ser. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, Jun. 2014.

[5] P. Godefroid, “Compositional dynamic test generation,” *SIGPLAN Not.*, vol. 42, no. 1, p. 47–54, Jan. 2007.

[6] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.

[7] F. Ramos, N. Sabino, P. Adão, D. A. Naumann, and J. Fragoso Santos, “Toward tool-independent summaries for symbolic execution,” in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. Ali and G. Salvaneschi, Eds., vol. 263. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 24:1–24:29.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SOK: (State of) The art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[9] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, “Binsec/SE: A dynamic symbolic execution toolkit for binary-level analysis,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 653–656.

[10] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[11] P. W. O’Hearn, J. C. Reynolds, and H. Yang, “Local reasoning about programs that alter data structures,” in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL ’01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 1–19.

[12] J. Reynolds, “Separation logic: a logic for shared mutable data structures,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[13] C. Calcagno and D. Distefano, “Infer: An automatic program verifier for memory safety of C programs,” in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[14] N. Polikarpova and I. Sergey, “Structuring the synthesis of heap-manipulating programs,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[15] L. Song and K. Kavi, “What can we gain by unfolding loops?” *SIGPLAN Not.*, vol. 39, no. 2, p. 26–33, Feb. 2004.

[16] J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, “Symbolic execution for JavaScript,” in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP ’18. New York, NY, USA: Association for Computing Machinery, 2018.

[17] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[18] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “CVC5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

[19] N. Sabino, “Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, 2019.

[20] E. Torlak and R. Bodik, “Growing solver-aided languages with Rosette,” in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–152.

[21] K. R. Apt, “Ten years of Hoare’s logic: A survey—part i,” *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 4, p. 431–483, Oct. 1981.

[22] P. W. O’Hearn, “Incorrectness logic,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019.

[23] P. Maksimović, C. Cronjäger, J. Sutherland, A. Lööw, S.-E. Ayoun, and P. Gardner, “Exact separation logic,” 2022.

[24] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969.

[25] P. O’Hearn, “Separation logic,” *Commun. ACM*, vol. 62, no. 2, p. 86–95, Jan. 2019.

[26] D. Gopan and T. Reps, “Low-level library analysis and summarization,” in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 68–81.

[27] Y. Lin, T. Miller, and H. Søndergaard, “Compositional symbolic execution using fine-grained summaries,” in *2015 24th Australasian Software Engineering Conference*, 2015, pp. 213–222.

[28] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, “JaVerT 2.0: Compositional symbolic execution for JavaScript,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[29] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid, “Compositional symbolic execution with memoized replay,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 632–642.

[30] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USA: USENIX Association, 2008, p. 209–224.

[31] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, “Using symbolic evaluation to understand behavior in configurable software systems,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 445–454.

[32] V. Chipounov, V. Kuznetsov, and G. Candea, “The S2E platform: Design, implementation, and applications,” *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012.

[33] S. Anand, P. Godefroid, and N. Tillmann, “Demand-driven compositional symbolic execution,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381.

[34] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, “Compositional may-must program analysis: Unleashing the power of alternation,” in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 43–56.

[35] P. Godefroid and D. Luchaup, “Automatic partial loop summarization in dynamic test generation,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 23–33.

[36] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of haskell programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279.

[37] E. L. Seidel, N. Vazou, and R. Jhala, “Type targeted testing,” in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 812–836.

[38] H. H. Nguyen, V. Kuncak, and W.-N. Chin, “Runtime checking for separation logic,” in *Verification, Model Checking, and Abstract Interpretation*, F. Logozzo, D. A. Peled, and L. D. Zuck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 203–217.

[39] P. Urzyczyn, M. H. S. Rensen, and M. H. Sorensen, *Lectures on the Curry-Howard Isomorphism*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier Science & Technology, Jul. 2006.

[40] S. Itzhaky, H. Peleg, N. Polikarpova, R. N. S. Rowe, and I. Sergey, “Cyclic program synthesis,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 944–959.

[41] Y. Watanabe, K. Gopinathan, G. Pîrlea, N. Polikarpova, and I. Sergey, “Certifying the synthesis of heap-manipulating programs,” *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, aug 2021.

[42] S. Itzhaky, H. Peleg, N. Polikarpova, R. N. S. Rowe, and I. Sergey, “Deductive synthesis of programs with pointers: Techniques, challenges, opportunities,” in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 110–134.

[43] D. Leijen and E. Meijer, “Parsec: A practical parser library,” *Electronic Notes in Theoretical Computer Science*, vol. 41, no. 1, pp. 1–20, 2001.

[44] P. Wadler, “Monads for functional programming,” in *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer, 1995, pp. 24–52.

[45] S. Liang, P. Hudak, and M. Jones, “Monad transformers and modular interpreters,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’95. New York, NY, USA: Association for Computing Machinery, 1995, p. 333–343.

[46] J. Launchbury and S. L. P. Jones, “State in Haskell,” *LISP and Symbolic Computation*, vol. 8, pp. 293–341, 1995.

[47] D. Beyer, “Competition on software verification and witness validation: SV-COMP 2023,” in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 495–522.

[48] ——, “Software testing: 5th comparative evaluation: Test-Comp 2023,” in *Fundamental Approaches to Software Engineering*, L. Lambers and S. Uchitel, Eds. Cham: Springer Nature Switzerland, 2023, pp. 309–323.