YaleNUSCollege

PROGRAM SYNTHESIS WITH ACCUMULATORS

NAY CHI WINT NAING

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Ilya Sergey

AY 2023/2024

Yale-NUS College Capstone Project

DECLARATION & CONSENT

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

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

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

   o Unrestricted access
   Make the Thesis immediately available for worldwide access.

   ● Access restricted to Yale-NUS College for a limited period
   Make the Thesis immediately available for Yale-NUS College access only from 04/2024 (mm/yyyy) to 04/2026 (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
   For the possibility of publishing this research in the future.

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

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

Nay Chi Wint Naing, Elm
Name & Residential College of Student

[illegible]
Signature of Student

07/ 04/2024
Date

Ilya Sergey [illegible]
Name & Signature of Supervisor

07 April 2024
Date

Acknowledgements

This thesis would not have been possible without the support and guidance of several key individuals whom I am immensely grateful for.

First and foremost, I would like to extend my deepest gratitude to my supervisor Professor Ilya Sergey, whose guidance and unwavering support throughout my thesis work. His passion, valuable advice, and insightful feedback have been instrumental in shaping this work.

I would also like to express my sincere thanks to Ziyi Yang, whose collaboration and assistance have been invaluable. Ziyi's dedication, patience, and willingness to share their profound knowledge and technical skills have significantly contributed to both the success of this project and my personal development.

I would also like to thank my parents for offering endless encourage, and unwavering support in times of stress and celebration in moments of achievement. Their belief in me has been a constant source of motivation and strength in completing this milestone.

Finally, I would like to thank my friends for their encouragement, understanding, and support. From short conversations at dinner table to endless nights, your support has been a crucial part of my Yale-NUS journey and I am really grateful to have such an incredible group of people by my side.

Thank you all for making this long journey enjoyable and rewarding.

YALE-NUS COLLEGE

Abstract

Mathematical, Computational and Statistical Sciences

B.Sc (Hons)

Program Synthesis with Accumulators

by Nay Chi NAING

Deductive synthesis is a powerful technique to automate the process of program implementation, by leveraging formal logic. SuSLik, employing deductive synthesis technique, is capable of synthesizing functional programs from a pair of pre- and postconditions. However, the synthesized code is not always efficient, and one of the sources of inefficiency stems from the tool's incapability to introduce the use of accumulators. Hence, through this work, we aim to provide a pipeline that could help introduce the use of accumulators through specifications. We have implemented an algorithm that can transform user-provided specifications into ones with an accumulator and is capable of verifying their correctness. We have also explored methods of guiding the tool to utilize the accumulator correctly to produce efficient programs.

Keywords: SuSLik, Deductive Synthesis, Accumulators, Separation Logic

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Program Synthesis

Program synthesis is an automated program construction method that satisfies a given set of desired behaviours. By automating the development process, it can significantly alleviate the burden of manual programming, reducing both errors and the time required, especially for complex programs. Various approaches to program synthesis are tailored to specific types of specifications and application domains. These approaches range from *constraint-based synthesis*, which relies on solving logical constraints, to *inductive synthesis*, which learns from examples. Among these various techniques, we will focus on *deductive synthesis*, a method that heavily relies on logical foundations and a theorem-proving approach (Manna and Waldinger, 1980).

1.2 Deductive Synthesis

A deductive synthesis is a specific approach that regards program synthesis as a theorem-proving task. It relies on a theorem-proving method that combines the features of transformation rules, unification, and mathematical induction within a single framework to generate programs.

At its core, deductive synthesis translates high-level specifications into executable code by systematically applying logical transformations. Each transformation is rigorously verified through theorem proving to ensure it adheres to the specified behaviour. This method leverages formal logic to bridge the gap between specification and implementation, ensuring that the resulting program performs the desired tasks and adheres to specified correctness properties. In this work, we will be working with the tool SUSLIK, which employs deductive synthesis framework.

1.3 Shortcomings of Deductive Synthesis

Despite being a powerful technique in synthesising recursive functions, deductive synthesis has some critical flaws due to its over-reliance on the precision and comprehensiveness of the specifications. Any ambiguity in these premises could lead to incorrect code. Additionally, even with the correct premises, the deductive synthesis might not always find the most efficient solutions as the tool only focuses on the correctness of the program, i.e. to satisfy the specification requirement. This shortcoming becomes evident in cases where the time and/or space complexity of the synthesised function could be hugely improved with introduction of new parameters to the specification such as accumulators.

1.4 Essence of an Accumulator

Functional programming, unlike imperative programming, emphasises the concept of immutability and statelessness. It often leverages recursion – a fundamental concept that allows the problem to be broken down into smaller instances of the same problem – to iterate over data structures and perform repetitive tasks. However, it is challenging to maintain state across these recursive calls without any mutable variables.

One solution to this problem is to introduce accumulators to the program as an argument. The accumulator is parsed to each recursive call and keeps track of the intermediate states by being updated with the current computation results, accumulating a final result without mutating any external state. Accumulators can further optimise the recursive functions into tail-recursive ones where the recursive call is the last operation performed in the function. This optimization allows for efficient memory usage and avoids stack overflow errors, making accumulator-based recursive functions particularly suited for handling large datasets or deeply nested recursive computations.

To illustrate the significance of accumulators in optimising the efficiency of recursive programs, let us consider the task of flattening a binary tree into a singly linked list. The pseudocode examples in Figures 1.1 and 1.2 respectively demonstrate this process both with and without the use of an accumulator.

The approach of `flattenTree` in Figure 1.1 involves recursively flattening its left and right sub-trees, and then temporarily storing these results in singly linked lists named `left_flattened` and `right_flattened`. We then concatenate these results with the root value of the tree `tree.value`.

```
function flattenTree(tree)
    if tree is empty
        return []
    else
        left_flattened = flattenTree(tree.left)
        right_flattened = flattenTree(tree.right)
        return concatenate([tree.value], left_flattened,
            right_flattened)
```

FIGURE 1.1: Flattening a Tree into a List without an Accumulator

Even though this approach delivers functionally correct results, it incurs a quadratic time complexity, $O(n^2)$, where n is the number of nodes in the tree. This is due to the necessity of traversing the tree twice: once during the flattening and once while concatenating.

However, if we employ an accumulator, we could accomplish this task of flattening a binary tree into a linked list with just a linear time complexity of $O(n)$ as illustrated in Figure 1.2.

```
function flattenTreeWithAccumulator(tree, accumulator)
    if tree is empty
        return accumulator
    else
        accumulator.append(tree.value)
        accumulator = flattenTreeWithAccumulator(tree.left,
            accumulator)
        accumulator = flattenTreeWithAccumulator(tree.right,
            accumulator)
        return accumulator
```

FIGURE 1.2: Flattening a Tree into a List with an Accumulator

The approach in Figure 1.2 first instantiates an empty list as an accumulator and stores the root value inside. Afterward, the left subtree is recursively flattened, and its results are appended to the accumulator. Similarly, the right subtree is recursively flattened, and the results are also appended to the accumulator instance. Upon termination of the recursive call to the right tree, the accumulator will have retained all the nodes from the tree and is then returned, satisfying the program requirement. As such, this implementation ensures a single traversal of the tree, and thereby, enhances the time complexity of the process to $O(n)$.

1.5 Problem Statement

As demonstrated above, we can see that the use of accumulators can result in more efficient programs. However, the current tool is not capable of introducing an accumulator to its synthesis as it only focuses on producing the correct program for the given requirement but not the most efficient result.

1.6 Contributions

Through this work, we will attempt to provide a pipeline that would provide the users with the option to synthesise programs using accumulators. Since deductive synthesis heavily relies on the correctness and completeness of the specifications, we implement a pipeline that could help transform a user specification into an accumulator-based specification and ensure that it is a correct one. Additionally, we explore how SuSLik could be guided to utilise accumulators in the "correct" manner to synthesise the most efficient version possible.

Chapter 2

Background

2.1 Separation Logic

Separation Logic, an extension of Hoare Logic, provides a formal framework to reason about programs that manipulate a dynamically allocated memory in a heap-like structure. According to this logic, memory is segmented into distinct, disjoint regions, where each segment is "owned" by different program elements. This principle of separation facilitates localised reasoning, allowing verification of program correctness in a given memory region without needing to consider the entire heap's state. Such an approach is instrumental in verifying the safety properties of programs with complex pointer manipulations and dynamic memory allocation (O'Hearn, 2012).

2.1.1 Specifications in Separation Logic

A specification or a synthesis goal formally outlines the desired behavior or properties the synthesised program should exhibit. Among various types of specifications, SUSLIK deals with declarative functional specifications which describe what the heap should look like before and after program execution, without saying how to get from one to the other (Polikarpova and Sergey, 2019).

A specification generally takes the form of a Hoare-triple $\{\mathcal{P}\}\ f(...)\ \{\mathcal{Q}\}$. The essence of such a specification is to formally assert that if the precondition $\mathcal{P}$ is true, then after the execution of the program f, the postcondition $\mathcal{Q}$ will be true. Both assertions $\mathcal{P}$ and $\mathcal{Q}$ are further divided into two segments: the pure and spatial parts. Specifically, $\mathcal{P}$ can be expressed as a pair $\{\phi; P\}$ of pure part ϕ and a spatial part P while $\mathcal{Q}$ is represented as a combination of $\{\psi; Q\}$, encompassing pure part ψ and a spatial part Q (Polikarpova and Sergey, 2019).

The pure part represents the logical constraints of the program about the relationship between the variables. The spatial part is represented by a collection of disjoint symbolic heap fragments called heaplets joined via the separation conjunction operation $*$ which is both commutative and associative Reynolds, 2002. The simplest kinds of heaplets are empty heap assertions (emp) and points-to assertions taking the form of $x \rightarrow e$, where x is a single memory address in the heap storing a payload of e (Polikarpova and Sergey, 2019). Let us consider the following assertion: $\{p \neq q; x \mapsto p * y \mapsto q\}$. Here, the spatial part denotes two separate, non-overlapping memory segments x and y storing symbolic values a and b respectively and the pure part states that p and q are distinct values.

2.1.2 Inductive Predicates

In order to express linked data structures such as linked lists, binary trees, and graphs, we can employ inductive heap predicates -– a fundamental concept of Separation Logic. These predicates enable the recursive definition of data structures, capturing both their shape and content within the heap. For example, a binary tree could be inductively defined as below: (O'Hearn, 2012)

$$\begin{aligned}\mathsf{tree}(x, s) \triangleq\ & x = 0 \Rightarrow \{s = \emptyset; \mathsf{emp}\} \\ & \mid\ x \neq 0 \Rightarrow \{s = \{v\} \cup s_l \cup s_r; \\ & [x, 3] * x \mapsto v * \langle x, 1\rangle \mapsto l * \langle x, 2\rangle \mapsto r * \\ & \mathsf{tree}(l, s_l) * \mathsf{tree}(r, s_r)\}\end{aligned} \tag{2.1}$$

The above tree predicate recursively defines a binary tree rooted at address `x` with a payload set `s`. It is defined in two parts: 1) the base case and 2) the inductive case. The first clause represents the base case, when the tree is empty and applies when the root pointer `x` is null. In this scenario, the payload set s would also be empty. The second clause, the inductive case describes a non-empty tree. In this scenario, a binary tree node is conceptualised as a record occupying three contagious memory slots, starting at address `x`. The first location at address `x` stores the node value, `v` while the other two stores a pointer to the left and right subtrees: `l` and `r`. The pure part of this second clause indicates that the payload of the whole tree, `s`, is comprised of `v` and the payloads of the left and right subtrees, `s1` and `s2` respectively.

2.2 Synthetic Separation Logic

Synthetic Separation Logic(SSL), extending Separation Logic(SL), is a system of deductive synthesis rules to provide a framework on decomposing specifications of complex programs into simpler ones. The tool SuSLik is a deductive synthesiser for heap-manipulating programs and leverages the principles of SSL. It takes in a specification of a function to be synthesised, expressed in Separation Logic (SL) along with a library of inductive predicates, a list of auxiliary function specifications (Polikarpova and Sergey, 2019).

Given a separation logic specification, the deductive synthesis will construct a proof derivation tree by employing the rules stated in Figure.2.1 (Itzhaky et al., 2021).

$$\textsc{Emp}\quad \frac{\vdash \phi \Rightarrow \psi}{\{\phi; \mathsf{emp}\} \leadsto \{\psi; \mathsf{emp}\} \mid \texttt{skip}} \qquad \textsc{Frame}\quad \frac{\{\phi; P\} \leadsto \{\psi; Q\} \mid c}{\{\phi; P * R\} \leadsto \{\psi; Q * R\} \mid c}$$

$$\textsc{Read}\quad \frac{\mathrm{y}\ \text{is fresh} \quad a \notin \mathsf{PV} \qquad [\mathrm{y}/a]\{\phi; \langle \mathrm{x}, \iota \rangle \mapsto a * P\} \leadsto [\mathrm{y}/a]\{\mathcal{Q}\} \mid c}{\{\phi; \langle \mathrm{x}, \iota \rangle \mapsto a * P\} \leadsto \{\mathcal{Q}\} \mid \mathbf{let}\ \mathrm{y} = *(\mathrm{x} + \iota); c}$$

$$\textsc{Write}\quad \frac{\mathsf{Vars}(e) \subseteq \mathsf{PV} \quad e \neq e' \qquad \{\phi; \langle \mathrm{x}, \iota \rangle \mapsto e * P\} \leadsto \{\psi; \langle \mathrm{x}, \iota \rangle \mapsto e * Q\} \mid c}{\{\phi; \langle \mathrm{x}, \iota \rangle \mapsto e' * P\} \leadsto \{\psi; \langle \mathrm{x}, \iota \rangle \mapsto e * Q\} \mid *(\mathrm{x} + \iota) = e; c}$$

FIGURE 2.1: Selected Separation Logic Rules

The rules in Figure 2.1 are basic inference rules that are employed by SuSLik to deduce simple functions that do not require a recursive call. In order to understand how these rules work, we will consider the implementation of `swap`, which is responsible for swapping two values stored at distinct memory locations. The derivation of `swap` is shown in Figure 2.2.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textsc{Emp}\;\dfrac{}{\{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \texttt{skip}}}{\textsc{Frame}\;\; \{\mathtt{x} \mapsto \mathtt{b1} * \mathtt{y} \mapsto \mathtt{a1}\} \rightsquigarrow \{\mathtt{x} \mapsto \mathtt{b1} * \mathtt{y} \mapsto \mathtt{a1}\} \mid \texttt{skip}}}{\textsc{Write}\;\; \{\mathtt{x} \mapsto \mathtt{b1} * \mathtt{y} \mapsto \mathtt{b1}\} \rightsquigarrow \{\mathtt{x} \mapsto \mathtt{b1} * \mathtt{y} \mapsto \mathtt{a1}\} \mid \texttt{*y = a1}}}{\textsc{Write}\;\; \{\mathtt{x} \mapsto \mathtt{a1} * \mathtt{y} \mapsto \mathtt{b1}\} \rightsquigarrow \{\mathtt{x} \mapsto \mathtt{b1} * \mathtt{y} \mapsto \mathtt{a1}\} \mid \begin{array}{l}\texttt{*x = b1;}\\ \texttt{*y = a1}\end{array}}}{\textsc{Read}\;\; \{\mathtt{x} \mapsto \mathtt{a1} * \mathtt{y} \mapsto b\} \rightsquigarrow \{\mathtt{x} \mapsto b * \mathtt{y} \mapsto \mathtt{a1}\} \mid \begin{array}{l}\texttt{let b1 = *y;}\\ \texttt{*x = b1;}\\ \texttt{*y = a1}\end{array}}}{\textsc{Read}\;\; \{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\} \rightsquigarrow \{\mathtt{x} \mapsto b * \mathtt{y} \mapsto a\} \mid \begin{array}{l}\texttt{let a1 = *x;}\\ \texttt{let b1 = *y;}\\ \texttt{*x = b1;}\\ \texttt{*y = a1}\end{array}}
$$

FIGURE 2.2: Derivation of `swap`.

The derivation in Figure 2.2 starts with a pair of pre-and post condition assertions. Each inference rule in Figure 2.1 is applied to simplify the synthesis goal until both the pre- and postheaps are empty. First, the tool applies the READ rule twice, reading logical variables a and b from locations x and y respectively, turning them into fresh program variables $a1$ and $b1$ correspondingly. Afterward, the WRITE rule is applied twice to write the value stored at $b1$ into x first, and then the value stored at $a1$ into y. At this stage, the pre- and postheap are equal. As such, the FRAME rule, responsible for removing the shared sub-heaps in the pre- and post-conditions to reduce the original synthesis goal into a smaller one, kicks in, leaving EMP on both sides of the goal. This will trigger the terminal rule EMP resulting in a trivial program `skip` and thereby concluding the derivation (Polikarpova and Sergey, 2019).

2.2.1 Inference Rules for Inductive Predicates

In order to deal with inductive predicates and deducing recursive functions, SSL features additional rules such as OPEN, CLOSE, CALL, and ABDUCECALL rules as shown in Figure 2.3(Itzhaky et al., 2021). The

OPEN and CLOSE rules are responsible for unfolding the inductive predicates as per their definition while the CALL and ABDUCECALL rules can synthesise recursive calls when the intermediate assertion can be unified with the top-level goal or any previous intermediate assertions.

OPEN

$$\frac{\Gamma \cup \forall\overline{\omega_{jk}}\,;\ \{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * P\} \rightsquigarrow \mathcal{Q} \,|\, c_j \quad \textit{for } \underline{\textit{all}}\ j{=}1..r\mathsf{p}^{\alpha}(\overline{v_i}) : \overline{e_j \Rightarrow \exists\overline{\omega_{jk}}.\{\chi_j; R_j\}}_{j=1..r} \in \Sigma \quad \omega_{jk} \notin \mathsf{Vars}(\Gamma), \mathsf{GV}(t_i) = \varnothing}{\Gamma;\ \{\phi; \mathsf{p}^{\alpha}(\overline{t_i}) * P\} \rightsquigarrow \mathcal{Q} \,|\, \textbf{if}\ \ (e_1)\ \{c_1\}\textbf{else}\ \ \textbf{if}\ \ (e_2)\ \{c_2\}\ \textbf{else}\ \cdots}$$

CLOSE

$$\frac{\Gamma \cup \exists\overline{\omega_{jk}}\,;\ \mathcal{P} \rightsquigarrow \{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * Q\} \,|\, c_j \quad \textit{for } \underline{\textit{some}}\ j{=}1..r\mathsf{p}^{\alpha}(\overline{v_i}) : \overline{e_j \Rightarrow \exists\overline{\omega_{jk}}.\{\chi_j; R_j\}}_{j=1..r} \in \Sigma \quad \omega_{jk} \notin \mathsf{Vars}(\Gamma)}{\Gamma;\ \mathcal{P} \rightsquigarrow \{\phi; \mathsf{p}^{\alpha}(\overline{t_i}) * Q\} \,|\, c}$$

CALL

$$\frac{\{\phi_f; P\} \rightsquigarrow \{\psi_f; S\} \,|\, f(\overline{x_i}) \qquad \vdash \phi \Rightarrow [\sigma]\phi_f \qquad \{\phi \wedge [\sigma]\psi_f; [\sigma]S * R\} \rightsquigarrow \{\mathcal{Q}\} \,|\, c}{\{\phi; [\sigma]P * R\} \rightsquigarrow \{\mathcal{Q}\} \,|\, f(\sigma(\overline{x_i})); c}$$

ABDUCECALL

$$\frac{\begin{array}{c}\mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f * F_f\}\{\psi_f; Q_f\} \in \Sigma \\ F_f \text{ has no predicate instances} \qquad [\sigma]P_f = P \quad F_f \neq \mathsf{emp} \quad F' \triangleq [\sigma]F_f \\ \Sigma;\Gamma;\ \{\phi; F\} \rightsquigarrow \{\phi; F'\} \,|\, c_1 \qquad \Sigma;\Gamma;\ \{\phi; P * F' * R\} \rightsquigarrow \{\mathcal{Q}\} \,|\, c_2\end{array}}{\Sigma;\Gamma;\ \{\phi; P * F * R\} \rightsquigarrow \{\mathcal{Q}\} \,|\, c_1; c_2}$$

FIGURE 2.3: More SSL Inference Rules for Recursive Functions and Inductive Predicates.

To better understand the rules in Figure 2.3, we will consider the implementation of `sll_free` which is responsible for deallocating a singly-linked list `sll`. The specification of `sll_free` along with the inductive predicate definition of `sll` are as follows:

$$\{\mathsf{sll}(x, s)\}\ \texttt{void free}(\texttt{loc}\ x)\ \{\mathsf{emp}\} \tag{2.2}$$

$$\begin{aligned}\mathsf{sll}(x, s) \triangleq\ & x = 0 \Rightarrow \{s = \varnothing; \mathsf{emp}\} \\ |\ & x \neq 0 \Rightarrow \{s = \{v\} \cup s1; [x, 2] * \\ & x \mapsto v * \langle x, 1\rangle \mapsto nxt * \mathsf{sll}(nxt, s1)\}\end{aligned} \tag{2.3}$$

The first synthesis step of implementing `sll_free` is by unfolding the predicate instance of the goal in the pre-condition, i.e. `sll` as per the inductive definition listed in Eq. 2.3. This step is complemented by the

OPEN rule, generating two sub-goals, one for each clause of the predicate:

$$\{\mathtt{x} = 0 \land s = \emptyset; \mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid c_1 \tag{2.4}$$

$$\{\mathtt{x} \neq 0 \land [\mathtt{x}, 2] * \mathtt{x} \mapsto v * \langle \mathtt{x}, 1\rangle \mapsto nxt * \mathsf{sll}(nxt, s1)\{\mathsf{emp}\} \mid c_2 \tag{2.5}$$

We can combine the two programs c_1 and c_2 as `if (x = 0)` { c_1 } `else` { c_2 } to obtain the final program which satisfies the given specification in Eq. 2.2. The first sub-goal (in Eq. 2.4) will be trivially resolved by the EMP rule, resulting in a program `skip`. For the second subgoal (in Eq. 2.5), the READ rule will kick in to transform the logical variable *nxt* into a program variable *n*1 as follows:

$$\{\mathtt{x} \neq 0 \land [\mathtt{x}, 2] * \mathtt{x} \mapsto v * \langle \mathtt{x}, 1\rangle \mapsto n1 * \mathsf{sll}(n1, s1)\} \rightsquigarrow \{\mathsf{emp}\} \tag{2.6}$$

Now, the new heaplet `sll(n1, s1)` can be unified with `sll(x, s)` from the top-level specification in Eq. 2.2), and therefore the CALL rule is triggered, synthesising the recursive call. Since the postcondition of `sll_free` is emp, the goal becomes:

$$\{\mathtt{x} \neq 0 \land [\mathtt{x}, 2] * \mathtt{x} \mapsto v * \langle \mathtt{x}, 1\rangle \mapsto n1 * \mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \tag{2.7}$$

Finally, the FREE rule is applied to dispose of the two consecutive memory blocks stored at x, resulting in both pre and postheaps being emp, and thus triggering the EMP rule for termination of synthesis. The synthesised code for `sll_free` would be as shown in Figure 2.4.

```
void sll_free (loc x) {
  if (x == 0) {
  } else {
    let n = *(x + 1);
    sll_free(n);
    free(x);
  }
}
```

FIGURE 2.4: Singly Linked List (sll) free program synthesised by SUSLIK

To further understand the capabilities of the CALL rule, let us revisit the example of flattening a tree into a singly-linked list (sll) as described by the specification in Figure 2.5.

$$\begin{array}{l}\{\mathsf{z} \mapsto p * \mathsf{tree}(p, s)\}\\ \texttt{void flatten}(\texttt{loc}\ z)\\ \{\mathsf{z} \mapsto q * \mathsf{sll}(q, s)\}\end{array}$$

FIGURE 2.5: Specification for Flattening a Tree to SLL in Separation Logic

The tool will begin with the READ rule to turn the program variable z into logical variable x. Then, similar to the previous example, the tool will continue with the OPEN rule to unfold the tree predicate as per the inductive definition, producing two sub-goals c_1 and c_2:

$$\{\mathrm{x} = 0 \land s = \emptyset; \mathsf{emp}\} \leadsto \{\mathsf{emp}\} \mid c_1 \tag{2.8}$$

$$\{\mathrm{x} \neq 0; [\mathrm{x}, 3] * \boxed{\mathrm{x} \mapsto v} * \boxed{\langle \mathrm{x}, 1\rangle \mapsto l} * \boxed{\langle \mathrm{x}, 2\rangle \mapsto r} * \mathsf{tree}(l, s_l) * \mathsf{tree}(r, s_r)\} \leadsto \{\mathsf{sll}(x, s)\} \mid c_2 \tag{2.9}$$

Similarly to the previous example, the first subgoal (Eq. 2.8) would be solved by the EMP rule, emitting a program `skip`. In the case of the second subgoal (Eq. 2.9), the tool will apply the READ rule three times to turn the logical variables x, $x+1$, and $x+2$ into program variables v, l, and r correspondingly. Afterward, we want the tool to unify both heaplets l and r with the top-level specification to trigger the CALL rule on them.

However, the precondition in specification requires `z` to point to the root of the sub-trees to make this recursive call. As the WRITE rule application process is guided by the desired postheap, it is not capable of making this decision on its own. This is where the ABDUCECALL rule comes in. It will prepare the symbolic heap for the recursive call, i.e. $\{\ldots;\mathtt{z} \mapsto \mathtt{l}\} \rightsquigarrow \{\mathtt{z} \mapsto \mathtt{l'}\}$, which will help trigger the WRITE rule to produce `*z = l'` (Polikarpova and Sergey, 2019). Now, the CALL rule can be triggered on the predicate instance of the left sub-tree which will store the result in program variable y. This process of applying the ABDUCE-CALL, WRITE and CALL rules will be repeated for the recursively synthesising the right sub-tree. At this stage, we are left with two flattened singly-linked-list stored inside y and z. And therefore, at this point of derivation, we now have the synthesis goal of:

$$\left\{\begin{array}{c} s = \{\mathtt{v}\} \cup s_l \cup s_r; \mathtt{r} \mapsto y_r * \mathsf{sll}(\mathtt{yl}, s_l) * \mathsf{sll}(y_r, s_r) * \\ [\mathtt{x}, 3] * \mathtt{x} \mapsto \mathtt{v} * \langle \mathtt{x}, 1\rangle \mapsto \_ * \langle \mathtt{x}, 2\rangle \mapsto \_ \end{array}\right\} \tag{2.10}$$
$$\rightsquigarrow \{\mathtt{r} \mapsto y * \mathsf{sll}(y, s)\}$$

Here, we have two lists y_l and y_r in the preheap while a single list of y which contains all the elements of both heaplets along with v in the postheap. The tool will proceed by unfolding the predicate instances of `sll` by using the OPEN rule. This would again results in two lists: y_r, which is the same as before, and y'_l, the tail of y_l. As this can be unified with the predicate instances in Eq. 2.10, the CALL rule will be triggered, emitting an auxiliary recursive procedure `append(y,v, x,z)`. The result of this synthesis is as shown in Figure 2.6.

```
void flatten(z) {
  let x = *z;
  if (x == 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    *z = l;
    flatten(z);
    let y = *z;
    *z = r;
    flatten(z);
    append(y, v, x, z);
  }
}
void append(y1, v, x, z){
  let y2 = *z;
  if (y1 == 0) {
    let y = malloc(2);
    free(x);
    *z = y;
    *(y + 1) = y2;
    *y = v;
  } else {
    let n = *(y1 + 1);
    append(n, v, x, z);
    let y = *z;
    *z = y1;
    *(y1 + 1) = y;
  } }
```

FIGURE 2.6: Tree flattening program synthesised by SUSLIK.

However, as mentioned in Section 1.4, we know that this approach of flattening a tree will incur a quadratic time complexity and can be reduced to a linear one by merely introducing an accumulator. As such, in the next sections, we will attempt to provide a pipeline where users can request the tool to use an accumulator in the synthesis process.

Chapter 3

Synthesising Code With An Accumulator

3.1 Overview of the Pipeline

Previously, we have demonstrated the importance of accumulators in optimizing the efficiency of synthesised programs. In order for the tool to be able to use the accumulator in synthesising recursive functions, we have attempted to implement a pipeline entailing the three sub-processes as below:

- Algorithm to generate accumulator-based specification from the client specification
- Verifying the correctness of the generated accumulator-based specification
- Synthesising the accumulator-based function

The implementation of this entire pipeline could be checked out here in our GitHub Repository[1]. In the following sections, we will delve into

[1]https://github.com/NayChi-7/suslik/tree/accumulators

the specifics of each process, in order to provide a comprehensive understanding of the process of introducing accumulators to the synthesiser.

3.2 Algorithm for Generating Accumulator Based Specifications

One way of enforcing the tool to use an accumulator is for the user to include it in the specification. However, automating this process would greatly enhance user experience. To this end, we have implemented a pipeline where the user can transform their original specification into one with an accumulator. This functionality can be assessed by running the tool via the terminal with an additional parameter "-g true".

To generate an accumulator-based specification from the original one, we would need to alter all three components of the specification, namely the precondition: $\mathcal{P}$, postcondition: $\mathcal{Q}$, and the function parameters. Each of these components requires different variations of hints for accumulator usage. The crux of this process lies in determining the correct accumulator type from the original specification.

3.2.1 Determining the Correct Accumulator Type

As mentioned before, the nature of an accumulator is to accumulate results over time so that the function can reuse such results and return these accumulated results at the end (Danvy, 2023). As such, one way to examine the correct type of accumulator to be used is by analysing the postcondition of the specification, which entails the final result the program is expected to return. To understand this better, we will revisit the example of flattening a tree into a singly linked list as described in Figure. 2.5. Below is the declarative functional specification for the task that needs to be parsed into the tool:

In the precondition, a tree is allocated in the memory at address r. After flattening it, we want to return a singly linked list carrying the same payload set s as the tree, at a new memory address y. In order to determine the correct type of accumulator, we can first take a look at the synthesised code without an accumulator. In the example of a tree being flattened to a list, we can see that the program will recursively flatten the left and right trees and store these results in a singly linked list. Therefore, each intermediate result would essentially be the result of appending the root of the subtree at each recursive call into the singly linked list. Finally, we want to return the final result where the payload set s is stored in the form of a singly linked list. From this analysis, if we recall the purpose of the accumulators as described in Section 1.4, we can conclude that the accumulator needs to be a singly linked list. In the context of this paper, we want to use accumulators for accumulation across the intermediate state as well as storing the final result of the program. We can simply take a look at the final element to be returned in the original specification to acquire a correct instance of the accumulator for the program.

3.2.2 Transformation of the Pre-Condition, $\mathcal{P}$

As mentioned in Chapter 2, the pre-condition of a specification has pure and spatial parts, represented as $\{\phi, P\}$. However, to transform the pre-condition of the client specification (original specification) into one with an accumulator, we only need to transform the spatial part in the precondition. This is because the pure part, ϕ, represents the logical constraints of the program state. We want the addition of the accumulator to enhance the program's efficiency while preserving the original purpose. As the accumulator should not interfere with existing logical constraints if any, there is no need to transform the pure part of the precondition.

However, we need to transform the spatial component, P, of the precondition – i.e. we need to include an instance of the correct accumulator in the precondition so that the tool can recognise it and manipulate it throughout the synthesis. After determining the correct type of accumulator as mentioned above, we then create an instance of such accumulator in the memory at a new address, namely *original_loc*, carrying a payload of set *acc*, which is usually empty but could be of any value. Then, we allocate a new address in the memory, namely *ans*, which stores a pointer to the accumulator. This pointer will also serve as an explicit result-storing pointer, i.e. the return to the program following the design of SuSLik . Given that each program should only have one return value, any existing explicit result-storing pointers in the original precondition will be removed. However, any existing heap allocation associated with it would be preserved.

The result of the transformation is shown below:

```
{ tree(r, s) ** ans :-> original\_loc ** sll(original\_loc ,
    accumulator)}
```

3.2.3 Transformation of the Function Parameters

Through the specification, the tool also takes in immutable program variables, i.e. function parameters that it needs to utilise. As such, to enable the tool to recognise the accumulator instance, a reference to the previously allocated accumulator must be added to these parameters. This is simply achieved by parsing in *locans* where *loc* is an untyped pointer used for location references, while *ans* is the address of the accumulator. The new function signature should look like this:

```
tree_flatten(loc r, loc ans)
```

3.2.4 Transformation of the Post-Condition, $\mathcal{Q}$

Unlike the precondition, this time, we are concerned with both the pure and spatial parts of the assertion to capture the expected state of the program upon termination accurately.

Transformation of the Spatial Part, Q

As mentioned previously, the accumulators serve both purposes of "carrying" intermediate computations across recursion and storing the final result. Therefore, we need to transform the current postcondition into one that would explicitly return the accumulator element with the help of the result-storing pointer, which we allocated as *ans* in Section 3.2.2. After the program execution, *ans* should store a pointer to a new address in the memory, which we have instantiated as *new_loc*. While it is possible to just append to the existing accumulator at the original address when we manually program, the tool needs to instantiate a new one as it is essential for the tool to recognize that this is a newly processed result. Additionally, as we are accounting for all general instances of accumulators that are not necessarily empty, the payload that the accumulator will now carry would be a combination of its existing payload *acc* and the payload of the heaplet in the memory before program execution, i.e. payload set s in our examples. Thus, we have introduced a new payload variable, namely $final\_result$, to capture this.

The result of the transformation on spatial part of the post condition is shown below (the pure part is omitted):

```
{ ...; ans :-> new_loc ** sll(new_loc , final_result)}
```

Transformation of the Pure Part, P

In the section above, we claimed that the final payload at the return address would be $final\_result$. However, without any logical constraints, it could just be any set. It, therefore, would not necessarily capture the essence of the payload that the accumulator should carry, and thus, will fail to fulfill the original program requirements. We can solve this problem by making some changes to the existing pure formulas in the postcondition, i.e. inflicting some logical constraints on this variable.

First of all, we need to remember that the payload that the accumulator should carry upon program termination is essentially the combination of the original payload that the accumulator instance was carrying and the final payload in the original specification. As such, we will take first a close look at the postcondition of the original specification. First, we will zoom into the spatial part and determine the return element associated with the result-storing pointer. From this element, we can figure out the final payload, which in example **??** would be set `s`. Then, we will examine if there are any existing logical constraints associated with this payload set `s`. If there are some existing pure formulas, we will filter out the one that utilises the logical operator, $OverloadedEq$. This operator is used to express information regarding the elements/value in the final payload set `s` in the original postcondition. Therefore, we need to utilise this information when creating a logical constraint to $final\_result$ as well.

For example, if the pure formula extracted is $\texttt{s} = \texttt{s}_1 \cup \texttt{s}_2$, then, the logical constraint to be generated would take the form of $\texttt{final\_res} = \texttt{s} \cup \texttt{acc}$, and when unpacked, would be $\texttt{final\_res} = \texttt{s}_1 \cup \texttt{s}_2 \cup \texttt{acc}$. If there is no pure formula extracted according to the aforementioned criteria, then, there is no need for us to "unpack" the contents of `s`, and therefore, the constraint would simply take the form of $\texttt{final\_res} = \texttt{s} \cup \texttt{acc}$.

We now need to represent these logical constraints in the correct syntax, by utilising the logical and binary operators that SUSLIK provides. There are multiple binary operators responsible for various purposes such as addition, subtraction, and multiplication. While we could generate all possible logical constraints on the logical variable: $final\_result$ by utilising all of these binary operators, it is impractical considering the time and energy needed to generate and verify the correctness of each of them through the tool, just to be filtered out to a single correct instance. Therefore, we will filter these operations down to just one which is most appropriate in this context.

As mentioned previously, the purpose of the accumulator in our programs is to aggregate results over time, storing the final return value upon termination. As such, the changes to the accumulators would be monotone: i.e. always increasing in value or size of the payload set. Thus, we can filter it down to a single operator, namely *OverloadedPlus*. This operator can be used to perform both numerical addition and set unions based on the type of payloads that it is dealing with. Additionally, since the addition of elements and values are commutative and associative, we do not consider the order of the logical variables in the formula. As such, the logical constraint generated could be filtered down to a single formula.

3.3 Verifying the Correctness of the Guessed Specification with an Accumulator

In the previous section, we introduced an algorithm that could help us generate an accumulator-based specification candidate from the client specification. However, we need to ensure that this generated candidate is correct, i.e. when synthesised, it satisfies the same program requirements as the client function. Toward this end, we will provide the generated accumulator-based specification as an auxiliary to the tool along with the client function specification. Through this, we aim to direct the tool to synthesise the client function by invoking the accumulator-based function with an empty accumulator instance. It is capable of performing in the expected manner in synthesising the client functions of `binary-tree-size` and `sll-length`. However, such is not always the case.

When dealing with accumulators which are instances of linked data structures, the tool simply ignores the accumulator-based specification and starts to synthesise the client function on its own. This could be due to two main reasons: 1) it is cheaper for the tool to synthesise the client specification on its own, and 2) the synthesis rules fail to "unify" the client goal with that of the accumulator-based one. The latter could imply that the generated specification is not the correct one.

Let us first assume that the generated candidate is indeed the correct one. Then, we will examine the cause of the problem by manually synthesise the targeted versions of `tree-flattening` and `sll-length`, with the help of their accumulator-based functions to compare the steps needed. As mentioned previously, the process essentially is parsing an empty accumulator instance as an argument to the accumulator-based function as shown in Figure 3.1a.

```
void flatten_with_helper(loc
    z)
    let x = *z ;
    *z = 0;
    flatten_acc(x, z);
```

(A) Tree Flattening

```
void sll_len_with_helper (loc
    x, loc ret) {
  *ret = 0;
  sll_len_acc(x, ret);
}
```

(B) SLL Length

FIGURE 3.1: Targeted Synthesis of Tree Flattening and SLL Length with the Help of Accumulator-based Functions.

$$\{\mathsf{ans} \mapsto ori\_loc * \mathsf{sll}(ori\_loc, acc) * \mathsf{tree}(x, s)\}$$
$$\texttt{void flatten}(\texttt{loc } x, \texttt{loc } ans)$$
$$\{final\_res = s + +acc; \mathsf{ans} \mapsto new\_loc * \mathsf{sll}(new\_loc, final\_res)\}$$

FIGURE 3.2: Generated Specification with an Accumulator for Flattening a Tree into SLL

At first glance, it is surprising to see that the tool is only able to synthesise `sll_len_with_helper` but not `flatten_with_helper` as the steps required seem to be identical. To understand the problem better, we have analysed the steps needed for deducing `flatten_with_helper` in Figure 3.1a. It will first need to apply the READ rule to transform the logical variable `z` into program variable `x`. Then, we want the tool to apply the CALL rule and create a recursive call on the auxiliary function, which in this case would be `flatten_acc`. Here, we need to parse in the `tree` heaplet along with an additional parameter which should be a `sll` instance as described in Figure 3.2. However, there is no such `sll` instance in our current preheap. Then, let us create an empty `sll` heaplet to parse in the result. Then, we will store the address of this heaplet, which would be `0`, inside `z` via the WRITE call rule. Such preparation of the symbolic heap will now result in the trigger of the CALL rule application on `flatten_acc`.

The derivation of this entire process is shown in Fig. **??** .

$$
\textsc{Read}\ \dfrac{\textsc{AbductionNeeded}\ \dfrac{\textsc{Write}\ \dfrac{\textsc{Call}\ \dfrac{\textsc{Frame}\ \dfrac{\textsc{Emp}\ \dfrac{}{\{\mathsf{emp}\} \leadsto \{\mathsf{emp}\} \mid \texttt{skip}}}{\{\texttt{final\_res == s + emp;z} \mapsto \mathtt{x}' * \mathsf{sll}(x', \texttt{final\_res})\} \leadsto \{\mathtt{z} \mapsto \mathtt{y} * \mathsf{sll}(y,s)\} \mid \texttt{skip}}}{\{\mathtt{z} \mapsto 0 * \mathsf{tree}(x,s) * \mathsf{sll}(0, emp)\} \leadsto \{\mathtt{z} \mapsto \mathtt{y} * \mathsf{sll}(y,s)\} \mid \texttt{flatten\_acc(x,z)}}}{\left\{\mathtt{z} \mapsto \mathtt{x} * \mathsf{tree}(x,s) * sll(0, emp)\right\} \leadsto \{\mathtt{z} \mapsto y * \mathsf{sll}(y,s)\} \,\Big|\, \begin{array}{c}\texttt{*z = 0;}\\ \texttt{flatten\_acc(x,z)}\end{array}}}{\{\mathtt{z} \mapsto \mathtt{x} * \mathsf{tree}(x,s)\} \leadsto \{\mathtt{z} \mapsto y * \mathsf{sll}(y,s)\} \mid \begin{array}{c}\texttt{*z = 0;}\\ \texttt{flatten\_acc(x,z)}\end{array}}}{\{\mathtt{z} \mapsto r * \mathsf{tree}(r,s)\} \leadsto \{\mathtt{z} \mapsto y * \mathsf{sll}(y,s)\} \mid \begin{array}{c}\texttt{let x = *z;}\\ \texttt{*z = 0;}\\ \texttt{flatten\_acc(x,z)}\end{array}}
$$

FIGURE 3.3: Derivation of Tree Flattening to SLL with accumulator-based function as helper.

3.3.1 Addition of Identity Element and Safety Check

To resolve this issue, we will need to add a "zero"/ empty instance of the accumulator to the precondition of the client specification. The zero instance will differ based on the type of accumulator we are dealing with. If the accumulator stores an integer element, the zero instance will be simply 0. On the other hand, if the accumulator is a dynamically-allocated memory like a singly-linked list, it will take the form of an empty list, having the structure of `sll(0,{})`.

After determining the zero instance of the accumulator, we will then add it to the precondition of the client specification $\mathcal{P}_1$. Let this newly transformed precondition be $\mathcal{P}_2$. We then construct a new synthesis goal G' with $\mathcal{P}_2$ as its pre-condition, keeping the function parameters and postcondition of the client specification. We then parse the new synthesis goal G' along with the accumulator-based specification. We observe that the tool is then able to synthesise G' as intended, i.e. no recursive function except the accumulator-based function is invoked in the synthesized code.

Even though this introduction of an empty instance "magically" resolves the verification process, we need to ensure that the client goal did not get transformed during the process, i.e. the client goal G and the new goal G' are indeed the same. Toward this end, we have implemented a safety check procedure as follows. First, we create another sub-synthesis goal G_1, where its pre- and postcondition are $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. If the two assertions are identical, the pre-and postheap could be cancelled out as per FRAME rule, leaving EMP on both sides. Then, EMP rule will kick in to terminate the program, by emitting a trivial program `skip`. Otherwise, the tool will produce a safety goal error. We have tested our examples with this procedure and the results support our argument that the client goal G and the newly transformed goal G' being identical.

3.4 Synthesising the Accumulator-Based Function

Previously, we have generated a candidate specification for an accumulator-based function from the client specification in Section 3.2 and also ensured that our guess is indeed a correct one in Section 3.3. Now, we need to ensure that the tool is capable of synthesising the accumulator-based function from the guessed specification. Thus, we provided this guessed specification to the tool for synthesis, and all of them were successfully synthesised as expected. However, these synthesised accumulator-based functions are not more efficient than the client functions.

The main source of this inefficiency stems from the tool utilising the accumulator instance in a "wrong" way. As mentioned previously, we would like to use an accumulator to remove recursive auxiliary functions which attributed to the expensive runtime complexity and also in synthesising tail-recursive functions whenever possible. However, the tool performed opposite to our intentions: synthesised recursive auxiliary functions whenever possible and did not use the accumulator to carry the information across recursive calls to enable tail call optimisation. These two inefficiencies are evident in the examples of flattening a tree into a singly-linked list and calculating the length of a singly-linked list correspondingly. We will try to explore each issue in detail in the following subsections 3.4.1 and 3.4.2.

3.4.1 Removing Recursive Auxiliary Functions

The synthesis of the function to flatten a tree into a singly linked list (according to the specification in Figure 3.2) results in the inefficient program shown in Figure 3.4.

```
void flatten (loc x, loc ans) {
    let o = *ans;
    if (o == 0) {
      flatten_helper(x, ans);
    } else {
      let nx = *(o + 1);
      *ans = nx;
      flatten(x, ans);
      let n = *ans;
      *ans = o;
      *(o + 1) = n;
    }
  }
```

```
void flatten_helper (loc x, loc a) {
    if (x == 0) {
    } else {
      let v = *x;
      let l = *(x + 1);
      let r = *(x + 2);
      flatten_helper(l, a);
      flatten(r, a);
      let ne = *a;
      let n = malloc(2);
      free(x);
      *a = n;
      *(n + 1) = ne;
      *n = v;
    }
}
```

FIGURE 3.4: Inefficient Synthesis of Tree Flatten Function with an Accumulator

By tracing the results of the synthesis, we noticed that the heuristic costs associated with synthesising a recursive auxiliary function are cheaper than an attempt at trying to utilise the accumulator instance correctly in the invocation of the recursive call on the main function itself. As such, by limiting the space for possible candidate goals to be unified with, to just the top-level specification, we can achieve the desired synthesis of `flatten_w_acc` as shown in Figure 3.5.

```
void flatten_w_acc (loc x, loc
   ans) {
  if (x == 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    flatten(l, ans);
    flatten(r, ans);
    let ne = *ans;
    let n = malloc(2);
    free(x);
    *ans = n;
    *(n + 1) = ne;
    *n = v;
  }
}
```

FIGURE 3.5: Synthesis of Tree Flatten with Accumulator without Recursive Auxiliary Function

3.4.2 Exploring the Synthesis of Tail Recursive Calls

Another important use of accumulators is to enable tail call optimisation to transform recursive functions into tail-recursive ones. Such transformation is prevalent in various arithmetic functions ranging from simple functions such as computing the size of the tree, the sum of the elements in a list to more complex ones such as factorial or Fibonacci numbers. Let us focus on the synthesis of relatively simpler functions such as calculating the length and sum of a list and size and sum of a tree. While the current tool utilises the accumulator in synthesising these functions, it is not capable of using the accumulator in the way we want. The essence of the tail recursive function as mentioned before is the fact that the recursive call to the function is the last action performed in the program. However, we realised that the tool modifies the accumulator after the recursive call in the version of the program that is produced as shown in

```
void sll_len_current (loc x,
    loc a) {
  let ans = *a;
  if (x == 0) {
  } else {
    let n = *(x + 1);
    *a = 0;
    sll_len(n, a);
    let l = *a;
    *a = ans + l + 1;
  }
}
```

(A) Inefficient Synthesis

```
void sll_len_desired (loc x,
    loc a) {
  let ans = *a;
  if (x == 0) {
  } else {
    let n = *(x + 1);
    *a = ans + 1;
    sll_len(n, a);
  }
}
```

(B) Desired Synthesis

FIGURE 3.6: Inefficient Synthesis vs Desired Synthesis of Singly-linked List Length Function with an Accumulator

Figure 3.6a. In order to better understand the failure, let us compare it to the ideal tail recursive version, which we have manually synthesised, as shown in Figure 3.6b.

The main difference lie in line 6 of both functions, where the tool needs to apply the WRITE rule to the accumulator instance `ans`. As such, we have tried to analyse the intermediate assertions of Figure 3.6b as shown here. Through this analysis, we found out that if the tool were given a pure formula `a == a1+ 1` in the precondition, it would synthesise the version in Figure 3.6b. To understand the reason behind this, we have printed the traces of both versions of the synthesis (obtained by running the tool with '-r 1'), we realised that it is because without this hint, the tool is unable to determine the modifications that it needs apply to the accumulator and therefore, will try to invoke the CALL rule to synthesise the recursive function and add in the accumulator value finally to satisfy the logical constraint in the postcondition.

We can provide the same hint to synthesise a function which calculates the binary tree size. This is because, at each recursive call, the accumulator is incremented by one. However, this hint unfortunately is not the magical solution to synthesising tail recursive functions. Let us consider the case of implementation of `sll-sum` function which sums all the elements in a list as the name suggests. Then, at each recursive call, the accumulator is modified by adding the head of the list to it. As elements in a list are variables and can take any value, we can no longer create a trivial hint as at each recursive call, the value needed to add to the accumulator needs to change. Therefore, with the current capabilities of the tool, we are not able to provide a generalised solution to ensure the synthesis of tail-recursive functions whenever possible.

Chapter 4

Evaluation

4.1 Cost and Efficiency

In order to understand the benefits and cost of our work, let us take a look at Table 4.1 .

Function Name	Synthesis Time (s)	Time Complexity
`tree_flatten_to_sll`	6.326	$O(n^2)$
`tree_flatten_to_sll_acc`	2.049	$O(n)$
`tree_flatten_to_dll`	20.667	$O(n^2)$
`tree_flatten_to_dll_acc`	5.205	$O(n)$
`sll_length`	2.044	$O(n)$
`sll_length_acc`	3.688	$O(n)$
`sll_length_acc_with_extra_pure_formula`	2.821	$O(n)$
`binary_tree_size`	2.145	$O(n)$
`binary_tree_size_acc`	13.794	$O(n)$
`binary_tree_size_acc_with_extra_pure_formula`	4.909	$O(n)$

TABLE 4.1: Synthesis Time and Complexity for Functions with and without (correct) use of accumulators

Here, we can see that the introduction of accumulators is greatly beneficial in the cases of flattening a tree into a singly-linked list (`sll`) and doubly-linked list (`dll`) both in terms of time taken to synthesise and the time complexity of the resulting function. In the cases of calculating the length and size of the binary tree, it takes longer for the tool to synthesise, especially in the case of "incomplete" specification which does not hint at the changes needed to be made to the accumulator instance. At a first glance, one might think that this attempt at introducing the accumulator might be decreasing the overall efficiency as the synthesis time increases while the overall time complexity remains the same. However, we should note that when provided with the correct hint, the increase in the synthesis time is trivial, yet we gain more efficiency through the tail call optimisation whose benefit we have mentioned previously in Section 1.4.

However, we should also note that there is a cost (time and space) associated with the generation of the accumulator-based specification. It approximately takes a user 30 seconds to transform each specification via the tool, verify the correctness of the specification, and then replace the client specification with an accumulator-based one. This current cost alone is longer than all of the examples that we have tried to synthesise. While this could be automated to enhance user convenience, it, nonetheless, will still have some costs associated with time and space.

While this is true, we believe that any improvement in the time and space complexity will outweigh these costs. Any programmer would agree that the improvement of time complexity from $O(n^2)$ to $O(n)$ is crucial especially dealing with large datasets where the time taken could scale extremely over time. As such, a good programmer would always try to optimise their solutions and the time taken to manually do so would be significantly bigger than 30 seconds in almost all the cases.

4.2 Limitations

While our current tool can generate a correct accumulator-based specification when given a working specification, it is not necessarily the version where the tool can synthesise the most efficient version. Such instances are evident in cases where the synthesised function could be optimised into a tail-recursive one as discussed before. Additionally, accumulators do not necessarily improve the efficiency of the function in certain instances such as copying an instance of `sll`. In such instances, the function without an accumulator would only traverse the list once to copy each element into a new list, incurring only the cost of linear time complexity $O(n)$. Additionally, in most cases of copying a list, we would like our list to be in the same order as the original list. However, even if we could make the accumulator-based copy function tail-recursive, it would still result in a reversed version. Even though the introduction of the accumulator would not incur significant costs, it also would not improve the existing efficiency, and therefore, there is no use of one in these cases.

4.3 Future Work

As mentioned previously, the inclusion of “correct" pure formula hints plays a significant role in assisting the tool to use the accumulator instances in a correct manner. Therefore, we could develop a mechanism where the tool is capable of analysing the exact steps that the accumulator should take either by inspecting the client specification or by having a procedure to analyse the synthesised code and reuse those analysis in making more informed decisions. Additionally, we could also implement a mechanism which could automate the examination of the necessity of an accumulator in a function and only synthesise with the use of such accumulators only in those instances.

Bibliography

Danvy, Olivier (2023). “Folding left and right matters: Direct style, accumulators, and continuations”. In: *Journal of Functional Programming* 33, e2.

Itzhaky, Shachar et al. (2021). “Cyclic program synthesis”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 944–959.

Manna, Zohar and Richard Waldinger (1980). “A deductive approach to program synthesis”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1, pp. 90–121.

O’Hearn, Peter W (2012). “A Primer on Separation Logic (and Automatic Program Verification and Analysis).” In: *Software safety and security* 33, pp. 286–318.

Polikarpova, Nadia and Ilya Sergey (2019). “Structuring the synthesis of heap-manipulating programs”. In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–30.

Reynolds, John C (2002). “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, pp. 55–74.