YaleNUSCollege

Building
A Certified Program Synthesizer

Yasunari Watanabe

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Dr. Ilya Sergey

AY 2019/2020

Yale-NUS College Capstone Project

DECLARATION & CONSENT

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

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

ACCESS LEVEL

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

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

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

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

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

Yasunari Watanabe / Elm College
Name & Residential College of Student

Y. Watanabe
Signature of Student

13 April 2020
Date

Dr. Ilya Sergey
Name & Signature of Supervisor

10 April 2020
Date

Acknowledgements

I am tremendously grateful to the following people who have shaped my experience at Yale-NUS College.

My capstone advisor Dr. Ilya Sergey, whose prior work on SuSLik sowed the seeds for this project. My capstone would not exist without his generous guidance in implementing the certification procedure, as well as his timely feedback on multiple drafts of this manuscript.

Dr. Olivier Danvy, whose lectures were formative for my computer science training at the College. In particular, his course *Functional Programming and Proving* introduced me to the isomorphism of programs and proofs, an idea that runs deeply throughout this project.

Dr. Cathay Liu, who taught me to write coherent arguments with proper warrants.

My parents for supporting my education and always welcoming me home, and to Kosuke for being the best sibling I could ask for.

Finally, Momoka and my wonderful (suite)mates—Celeste, Celia, Chengpei, Grady, Hansel, Kian Hao, and Ryan—for an unforgettable four years.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Building A Certified Program Synthesizer

by Yasunari WATANABE

Deductive program synthesis produces programs that are *correct by construction*. This project provides a fully automated interface for certified synthesis. The procedure generates formal definitions, a verifiable program, and a correctness certificate, all from concise user-provided specifications. This paper describes the design of the certification task, as well as its implementation that extends SUSLIK, a state-of-the-art program synthesizer, and uses the interactive theorem prover Coq as the certification backend.

Keywords: Program Synthesis, Formal Verification, Separation Logic

Contents

List of Figures

Chapter 1

The Thesis

One can have fully automated synthesis of certified programs.

Chapter 2

Background and Motivation

Software bugs are costly. Their presence in mission critical code can spell trouble for a business's bottom line, or in severe cases cost human lives.

The discrepancy between a program's specification and implementation is one major source of such bugs, especially for stateful programs written in imperative style. Often, a program's *declarative* specification (describing *what* the program should do) is more intuitive than its *imperative* implementations (describing *how* to perform the task, from the machine's perspective).

For example, consider the following program specification.

> "Given an unordered list of alphabetic characters, *SORT* produces a list of the same characters in lexicographic order."

A layperson can easily check whether a sample pair of program inputs and outputs meets this specification. On the other hand, it is considerably more difficult to devise a candidate imperative program that implements the specification. Even after developing a candidate program, determining whether *all* of its execution paths satisfy *SORT* is yet another challenge. Such non-trivialities are expounded as the specification grows in complexity, and are therefore hotbeds for human error (i.e., bugs).

The field of *program synthesis* seeks to address these concerns by developing techniques to synthesize programs from user-provided information. In this project, we enhance an existing *deductive* synthesizer for heap-manipulating programs with an post-hoc certification procedure. As a result, we achieve the fully automatic construction of *provably correct* programs amenable to third-party verification.

2.1 On Automatic Program Synthesis

Program synthesis is the construction of a program that satisfies a given specification, commonly classified into *deductive* and *inductive* approaches. *Deductive synthesis* constructs a program from a logical specification. Deductive rules are applied to formally prove the existence of a program that meets the specification. Exploiting an isomorphism between programs and constructive proofs, a program is *extracted* from the proof derivation. While this requires a formalization of both the synthesis rules and specification language, the end product is a *provably correct* program.

In *inductive synthesis*, the specification is a set of example program executions that specify behavior for a subset of possible inputs, and the task is to generalize them into a program that can handle all possible inputs. One consequence is that such a specification is incomplete, and suceptible to an erroneous synthesis result. However, it is useful in domain-specific scenarios where defining a formal specification is intractable but imperfect synthesis is tolerable. The automated synthesis of string transformations in spreadsheets is a successful application of domain-specific inductive synthesis; it has since been commercialized as the FlashFill feature in Microsoft Excel [10].

We focus on *deductive* synthesis, which is suitable for generic low-level procedures that demand a strict correctness guarantee. For imperative programs, a *declarative functional* specification describes the key properties of the target program in terms of a formally defined language.

One of the earliest formalizations is *Hoare logic* [7, 11]. Let $\mathcal{P}$, $\mathcal{Q}$ be logical assertions about a machine's state. Then, $\mathcal{P} \vdash \mathcal{Q}$ denotes that $\mathcal{P}$ *entails* $\mathcal{Q}$. In the context of a *Hoare triple* $\{\mathcal{P}\}c\{\mathcal{Q}\}$ for a program c, we refer to $\mathcal{P}$ and $\mathcal{Q}$ as the *pre-* and *post-conditions*. The triple denotes that given a state satisfying $\mathcal{P}$, the operation c transforms the state such that $\mathcal{Q}$ holds as a result, if the operation terminates.

While Hoare logic allows for reasoning about stateful programs with mutable variables, it does not handle those involving pointer manipulations. Subsequently, *separation logic* (SL) was developed as an extension of Hoare logic, to enable reasoning about heap-manipulating programs [18].

This project enhances SuSLik, a state-of-the-art deductive program synthesizer [17] that applies SL to the task of synthesis.

2.2 On Interactive Verification

To *verify* a program means to formally check its correctness with respect to a specification. A *certificate* is a guarantee of such correctness, typically in the form of a proof, which be verified by any independent party later.

In *interactive verification*, a human prover guides the proof search, and a *proof assistant* software mechanically checks the soundness of the prover's steps. Proof assistants have been used to formalize pen-and-paper proofs of many seminal mathematical theorems.

We present a method to produce, from user-friendly synthesis specifications, a set of formal definitions, a fully verifiable program, and a certificate. These deliverables can be interpreted by the proof assistant Coq, and thus interactively verified by any independent third party.

2.3 Contributions

There is a gap to bridge between automated synthesis and interactive verification—the latter demands human intervention the former does not provide. On one hand, automated synthesis reduces the human burden of constructing imperative programs. On the other hand, interactive program certification is premised on the availability of human ingenuity to guide the proof. We fill the gap from both sides—code generation performed as part of the synthesis procedure, and proof engineering in the language of the interactive verifier.

A related prior work is Fiat, a project implemented with the Coq proof assistant [6]. Fiat is an interactive certified synthesizer, that provides a strong correctness guarantee but requires the user to provide hints to guide the synthesis along. We present a *fully automated* certification procedure with no need for user intervention, and the same strong guarantee provided by Coq (but with slightly less expressivity).

In summary, we present the following contributions.

1. Conceptual contributions

    (a) A fully automated approach to produce formal specifications suitable for proof assistants, from more concise user-friendly synthesis specifications;

(b) an approach to faithfully reconstruct a verifiable program from the synthesis language;

(c) a set of methods allowing one to harness the synthesis search procedure to produce certificates for the generated programs.

2. Practical contributions

(a) Implementation of the approaches described above in the context of state of the art tools: SuSLik and Coq.

2.4 Paper Outline

In the following chapters, we summarize the rationale for our tool choices (Chapter 3); outline SuSLik's program search algorithm and its dual role as a proof search (Chapter 4); describe how SuSLik specifications, programs, and proofs are translated into their formal counterparts suitable for proof assistants (Chapter 5); describe the result of applying the approach to a case study (Chapter 6); and conclude with a discussion of future work (Chapter 7).

Chapter 3

Prerequisites and Preliminary Investigation

In this chapter, we describe our tool choices and rationale.

3.1 Setup

3.1.1 Synthesis with SuSLik

SuSLik is a program synthesizer written in Scala that provides a declarative interface for synthesizing imperative programs by only specifying the desired effect in separation logic (SL) [17].

SuSLik performs deductive synthesis. This approach has its roots in the Curry-Howard correspondence, which states the isomorphic relationship between programs and proofs [19]. In the same spirit, SuSLik frames the synthesis procedure as a proof construction problem, where the synthesizer provides a proof term (candidate program) that facilitates the entailment of the postcondition from the precondition. The synthesis procedure is built atop a system of rules called *Synthetic Separation Logic*

(SSL), which extends SL for the task of synthesis. SuSLik repeatedly enumerates and applies SSL rules until it reaches its goal.

3.1.2 Verification with Coq

While deductive synthesis produces *correct-by-construction* programs, SuSLik falls short of making a formal correctness *guarantee* of its outputs. To claim so, we would have to verify the soundness of SuSLik's SSL rule implementation. As the Scala source code for SuSLik spans thousands of lines, this would require several person-years of dedicated effort.

Instead, we take a *post-hoc* approach to program verification; rather than formally certifying the entire synthesizer itself, we devise an automated procedure to certify the correctness of its outputs. Consequently, we transfer the burden of trust to a smaller, well-studied codebase by conducting post-hoc analysis in the third-party prover Coq.

We choose Coq [20] as our certification target. It is a proof assistant used widely for interactive verification. Notable applications of Coq to computing systems includes verification of a file system [3] and OS kernel [9]. Coq supports a backward reasoning style, which works from a proof's desired conclusion back to its premises. In a Coq proof, the conclusion, displayed as an initial *goal*, is solved iteratively using commands called *tactics*. Each tactic, which represents a deductive rule, reduces the current goal into a set of subgoals. Those generated subgoals are the propositions that imply the current goal. Goals are reduced until they match either a premise or a previously proven theorem.

While Coq's standard library includes essential tactics, users can also

```
predicate lseg(loc x, set s) {
 | x == 0 ⇒
    { s =i {} ; emp }
 | not (x == 0) ⇒
    { s =i {v} ++ s1 ;
      [x, 2] ** x ↦ v ** (x + 1) ↦ nxt ** lseg(nxt, s1) }
}
```

FIGURE 3.1: SUSLIK's `lseg` predicate implementation.

define their own tactics with LTAC, Coq's tactic language. This is an important feature that enables some proof automation in our project.

3.2 Candidate Program Verification Frameworks

This project requires a Coq framework designed for proving correctness with respect to Hoare-style specifications. Coq implements a higher-order functional specification language called Gallina. This facilitates the creation of domain-specific proof frameworks. We survey two options, the Verified Software Toolchain (VST) and Hoare Type Theory (HTT), and find the latter to be better suited for our project.

Both frameworks support reasoning about heap-manipulating programs and their specifications. However, they choose different abstractions to represent this information. For our purposes, we judge the frameworks based on their interoperability with SUSLIK; namely, how translatable the framework's representation of program logic is from SUSLIK.

Throughout this section, we use an example SUSLIK predicate to guide our discussion of each framework's translation behavior. The inductive predicate `lseg` (Figure 3.1) asserts the presence of a linked list segment that contains a set s of elements, whose head is at location x. The first clause asserts that if x is a null pointer, the set s contains no elements and

the heap is empty. The second clause asserts that otherwise, x points to a consecutive memory block of length two, whose first cell stores a payload v that belongs to the set s, and whose second cell stores a pointer nxt to a linked list segment of the remaining elements of s, whose shape is described by the recursive predicate application `lseg(nxt, s1)`. Details of SUSLIK's predicate language syntax are provided in Section 5.1.

3.2.1 VST

VST is a framework for certifying C programs [1]. Its program logic Verifiable C is accompanied by a proof system, VST-FLOYD [2]. The toolchain remains in active development, and was recently enhanced with techniques to handle graph-manipulating programs [21]. The certification guarantee of assertions at the program level extends down to the machine-language level via CompCert [14], a C compiler proven correct in Coq with respect to the assembly language of several processor types. However, VST's guarantee of supporting cross-architecture C language semantics adds complexity. To use the framework, one must handle many OS-level nuances to even conduct the simplest of program proofs.

Figure 3.2 shows a translation of `lseg` into VST. VST is a *deeply embedded* framework that circumvents Coq's native higher-order logic, and a myriad of custom bindings are required to differentiate between the context of VST's logic and that of native Coq, such as a proprietary propositional type (`mpred`) for its assertions, and a `!!` binding to denote every program-level assertion.

SUSLIK uses a *relational* definition of the predicate (Figure 3.1), so a

```
Fixpoint lseg (x: val) (s: list val) (size: nat) : mpred :=
  match size with
  | S size' ⇒
    !!not(x = nullval) &&
      EX nxt:val, EX v:val, EX s': list val,
      data_at Tsh (tarray (tptr tint) 2) (v :: nxt :: nil) x * lseg nxt s' size'
  | O ⇒ !!(x = nullval) && !!(s = nil) && emp
  end.
```

FIGURE 3.2: VST's `lseg` predicate implementation.

faithful translation would encode the predicate as an inductive proposition with constructors. This would also reduce the proof search procedure to a largely mechanical process of pattern-matching on the constructors. Yet, due to the multiple layers of indirection necessitated by VST's deep embedding, the predicate is instead given a *functional* definition, encoded using a `Fixpoint`. An additional `size` parameter indicates to Coq that the function's recursion is well-founded via an explicit strictly decreasing term.

Once defined, using the predicate presents further obstacles. SUSLIK does not represent a real memory model, so differences in data type size are not considered. This is also why a SUSLIK memory block may hold elements of different types, which enables a `lseg` definition like Figure 3.1. On the other hand, C programs specify behavior for real hardware, and therefore maintain a strict account of the footprint of all data in memory. As a result, it is difficult to faithfully represent `lseg` as a C program by substituting mixed-type SUSLIK memory blocks (`loc` and the payload type) with C arrays that expect every element to have the same type.

The authors of VST do provide an alternative linked list segment predicate that represents nodes as C structures, as well as program proofs that use this alternative predicate definition. Yet to complete even these proofs, they first develop a general theory of the user-defined predicate

with auxiliary lemmas spanning over a thousand lines. This indicates we would have to replicate the effort for our own list segment definition that does not rely on C structures, and for any new predicate definitions (such as those involving tree structures). This is not a trivial task for a human to complete, let alone to automate.

The root cause of these issues is a mismatch between SuSLik's goal, which is to showcase an implementation of a novel deductive synthesis technique, and VST's goal, which is to certify real software systems with their own idiosyncrasies. VST is better suited for projects that require robust certification of C programs in particular. To ensure our primary effort remains focused on the certification task at hand, we ultimately decided not to move forward with VST for our certification procedure.

3.2.2 HTT

Hoare Type Theory (HTT) is a framework for reasoning about SL via native Coq propositions [15], notable for its representation of heap data structures. HTT assigns an abstract type to heaps, which are referenced by heap variables, and which are described by predicates.

HTT's key insight is that the class of heaps form a *partial commutative monoid* (PCM) with the heap union operation, from which we obtain algebraic properties to simplify a complex assertion about a large heap into assertions about its disjoint sub-heaplets. Formally, the set $heap$, binary operator $\bullet$, and identity element $empty$ form a PCM described by the triple $(heap, \bullet, empty)$, where:

1. The heap union operator $\bullet$ is partially associative on defined heap operands.

2. The identity element $empty \in heap$ represents the empty heap, such that $h \bullet empty = empty \bullet h = h$ for all $h \in heap$.

3. $\bullet$ is commutative on defined heap operands.

4. The union of two non-disjoint heaps produces an undefined heap.

By preserving these algebraic properties over disjoint heaps, HTT gracefully handles the reordering of heaplets without needing a disjointedness proof at every juncture. This is essential, as most SL rules defined in HTT require a particular ordering of the target heaplets. In Section 5.3, we design Coq tactics to automate the reordering.

Another characteristic of HTT is its monadic representation of the Hoare triple $\{\mathcal{P}\}c\{\mathcal{Q}\}$, which allows Coq to functionally reason about commands that produce side-effects. After reformulating the SL inference rules so that they can be encoded as Coq types, we have a SL framework that reuses Coq's infrastructure wherever possible. For example, the rule of variable substitution in SL is inherited directly from Coq's substitution rules. Likewise, side-effectful branching in HTT statements uses Coq's pattern-matching (`if`/`else`) operators.

Figure 3.3 shows an HTT translation of SuSLik's `lseg` predicate. We observe none of the difficulties we faced in the VST translation: a relational definition of an inductive proposition is possible because HTT reuses Coq's native `Prop` type, as well as its equality and conjunction operators. We observe identical usage of the $\mapsto$ operator, while the $**$ operator is merely replaced by \+. The precise semantics of these operators are discussed in Section 5.1.

One notable difference is that we parametrize the predicate with a list of payload elements (in the order they appear in the linked list), rather

```
Inductive lseg (x : ptr) (s : seq nat) (h : heap) : Prop :=
| lseg0 of x == 0 of
    s = nil ∧ h = empty
| lseg1 of x != 0 of
    ∃ nxt s1 v, (* Value existentials *)
    ∃ h', (* Auxiliary heap existentials *)
       s = [:: v] ++ s1 ∧ (* Pure part *)
       h = x ↦ v \+ x .+ 1 ↦ nxt \+ h' ∧ (* Spatial part *)
       lseg nxt s1 h'. (* Elaborating on recursive sub-heaps *)
```

FIGURE 3.3: HTT's `lseg` predicate implementation.

than with a set as SUSLIK does. Also, without loss of generality we constrain the payload type, which is polymorphic in SUSLIK's specification, to natural numbers (represented in Coq as `nat`).

One notable difference from SUSLIK is that HTT predicates range over an additional heap parameter. Consequently, HTT assertions require explicit *heap existentials*, which are the target of nested predicate applications and facilitate inductive reasoning about recursive sub-heaplets. These existentials are implicit in SUSLIK assertions, so there is a challenge of doing the necessary bookkeeping to make heap variables explicit during synthesis. Section 5.1 discusses our approach to solve this problem.

3.2.3 Levels of Embedding

We observe that expressing SUSLIK programs and their proofs is easier in HTT than in VST. This corresponds to the degree of *embedding* that each framework's program logic has within the host infrastructure (Coq's higher order logic).

VST is a *deeply embedded* logic, wherein its logical formulas are expressed using custom data types that deviate from Coq's native logic.

Deep embedding generally benefits users of the framework, who can directly use the framework's domain-specific abstractions and avoid those of the underlying host. However, the tradeoff is high implementation cost; every aspect of the framework must be written with respect to the host infrastructure, entirely from scratch. To import SuSLik's abstractions into VST's framework-specific abstraction, we must extend the framework with the new information (Section 3.2.1).

In contrast, HTT is a *shallowly embedded* logic that expresses logical formulas in terms of Coq's native logic. At the cost of general expressivity of the framework's domain, shallow embedding expedites the implementation of the framework, as it reuses the host infrastructure. This aligns with our objective of achieving interoperability with SuSLik.

In summary, we evaluate the shallowly embedded HTT to be more suitable than the deeply embedded VST as an initial certification target.

Chapter 4

Deconstructing Synthesis

Consider the following SL specification for a procedure to swap the referents of two pointers.

$$\{x \mapsto a * y \mapsto b\} \texttt{ void swap(loc x, loc y) } \{x \mapsto b * y \mapsto a\} \quad (4.1)$$

Figure 4.1 shows the program that SuSLik synthesizes from this specification, as well as an HTT proof that certifies its correctness.

In this chapter, we describe SuSLik's existing procedure to synthesize such a program (Section 4.1), and highlight the steps to adapt the synthesis so that it may also emit an HTT correctness proof (Section 4.2). Finally, we identify the presence of a representational "gap" between program synthesis and certification (Section 4.3) that we bridge in Chapter 5.

4.1 Deductive Synthesis in SuSLik

In adapting SuSLik for HTT certification, we first describe the derivation steps involved (Section 4.1.1), and then show how the derivation steps entail a program search (Section 4.1.2).

```
swap (loc x, loc y) {
  let a2 = *x;
  let b2 = *y;
  *x = b2;
  *y = a2;
}
```

```
apply: ghR; move⇒h [a b]→HValid//=.
apply: bnd_readR⇒//=.
apply: bnd_readR⇒//=.
apply: bnd_writeR⇒//=.
apply: bnd_writeR⇒//=.
apply: val_ret⇒//=.
```

FIGURE 4.1: A SUSLIK `swap` program (left) and its HTT proof (right).

```
loc x, loc y |-
{true ; x ↦ a ** y ↦ b} ?? {true ; x ↦ b ** y ↦ a}
...
[Op: read]: SUCCESS at depth 2, 1 alternative(s) [1 sub-goal(s)]
| int a2, loc x, loc y |-
| {true ; y ↦ b ** x ↦ a2} ?? {true ; x ↦ b ** y ↦ a2}
| [Op: read]: SUCCESS at depth 3, 1 alternative(s) [1 sub-goal(s)]
| | int b2, int a2, loc x, loc y |-
| | {true ; x ↦ a2 ** y ↦ b2} ?? {true ; y ↦ a2 ** x ↦ b2}
| | [Op: write-frame]: SUCCESS at depth 4, 1 alternative(s) [1 sub-goal(s)]
| | | int b2, int a2, loc x, loc y |-
| | | {true ; y ↦ b2} ?? {true ; y ↦ a2}
| | | [Op: write-frame]: SUCCESS at depth 5, 1 alternative(s) [1 sub-goal(s)]
| | | | int b2, int a2, loc x, loc y |-
| | | | {true ; emp} ?? {true ; emp}
| | | | [Sub: emp]: SUCCESS at depth 6, 1 alternative(s) [0 sub-goal(s)]
```

FIGURE 4.2: The derivation log emitted by SUSLIK (some steps omitted).

4.1.1 Synthesis in Action

SUSLIK is a *deductive* program synthesizer. Its input is an initial *synthesis goal* that describes the top-level program specification. For example, in Equation 4.1's precondition, x points to a and y points to b, while the postcondition reverses the referents.

SUSLIK operates on a set of rules called *Synthetic Separation Logic* (SSL). An extension of SL, SSL rules provide semantics for how to decompose a complex synthesis goal into simpler subgoals, and how to compose the computations corresponding to those simpler subgoals so that they produce a program statement for the parent goal. To capture this behavior, we say that applying a SSL rule to a synthesis goal produces a *derivation*.

$$\textsc{Call}\quad \frac{\begin{array}{c}\mathcal{F} \triangleq f(\overline{x_i}) : \left\{\phi_f, P_f\right\}\left\{\psi_f, Q_f\right\} \in \Sigma \\ R =^{\ell} [\sigma]P_f \qquad \phi \Rightarrow [\sigma]\phi_f \qquad \phi' \triangleq [\sigma]\psi_f \qquad R' \triangleq \lceil [\sigma]Q_f \rceil \\ \overline{e_i} = [\sigma]\overline{x_i} \qquad \mathsf{Vars}\,(\overline{e_i}) \subseteq \Gamma \qquad \Sigma; \Gamma; \{\phi \wedge \phi'; P * R'\} \rightsquigarrow \{\mathcal{Q}\} \,|\, c\end{array}}{\Sigma; \Gamma; \{\phi; P * R\} \rightsquigarrow \{\mathcal{Q}\} \,|\, f(\overline{e_i}); c}$$

FIGURE 4.3: The CALL rule in SSL.

In SUSLIK, a derivation stores this collection of subgoals and a statement-producing computation.

We can get a sense of how SUSLIK conducts synthesis by examining its runtime log, which is emitted by enabling a command-line flag. The information logged includes the rules that are applied, the resulting subgoals, and whether they succeed. Figure 4.2 shows the information emitted during synthesis of the `swap` program from Figure 4.1. Only successful rule applications are shown for brevity.

We highlight the first SSL rule application, READ, which reads the value stored at a location into a program variable. This step applies READ to the variable y, which initializes a new program variable a_2 with y's referent, a. The synthesis goal (pre/postcondition pair) is transformed accordingly, and the new goal becomes the target of synthesis occurring at the next level of depth. This process continues until there are no subgoals left to solve, at which point the precondition (an empty heap) entails the postcondition (also an empty heap).

4.1.2 Program Search

Now that we have observed a sample execution of synthesis with SUSLIK (Figure 4.2), we elaborate on how such a derivation is obtained.

The synthesizer performs a backtracking search over the domain of SSL derivations. At each step, a set of candidate SSL rules are successively applied to the current goal until one succeeds. Rules that have no chance of succeeding for the current goal are excluded altogether. For example, the CALL rule (Figure 4.3) unifies a subheap R in the goal's precondition with a subheap P_f for some function symbol f that is present in the goal context. If a goal contains no predicates, invoking this rule is futile, so we prune it from the search space.

While each rule application in Figure 4.2 produces one subderivation, this is not necessarily true in general. More than one possible derivation may arise from a rule application, in case there are multiple ways to apply the rule to the goal. For instance, if multiple function symbols are present in the goal context, CALL may be applicable to any one of them, so each possible derivation is tried nondeterministically. If a derivation fails, SUSLIK backtracks and tries the next one if it exists.

A derivation produces a collection of synthesis subgoals that must all be syntehsized for the derivation to succeed. In general, a program search proceeds recursively resulting in a tree-shaped footprint. A derivation may produce multiple subgoals if, for instance, the program being synthesized has conditional branching. Figure 4.2 illustrates the degenerate case where each derivation produces at most one subgoal.

A derivation also specifies a synthesis computation, which composes a list of program statements to produce a single statement. A key feature

of SuSLik's synthesis algorithm is that the *need* for a synthesis computation becomes known long before the appropriate time to *perform* the computation. Once again, let us consider the first Read application of Figure 4.2. The derivation specifies a synthesis computation that prepends a single program statement (which reads a value into a variable) to the rest of the synthesized program. At the time of derivation, the body to which we prepend the new program statement is still unknown. We only gain access to this information, and are able to perform the computation, after successfully synthesizing the rest of the program.

SuSLik delays execution of these synthesis computations with *continuation-passing style* (CPS). For each derivation of a given goal, the corresponding synthesis computation is encoded as a *continuation*, and passed along to the next depth level for recursive synthesis of its subgoals. If any of the subgoals fail, the continuation is not executed, and the algorithm backtracks to the most recent successful step. Otherwise, the continuation executes with the list of program statements resulting from the successful synthesis of all subgoals.

4.2 Towards Deductive Proofs in HTT

We have seen that during synthesis, SuSLik produces all information needed to certify the synthesized program in HTT. SSL provides a set of axioms and inference rules that relate program statements to the computations they perform on the heap. A formal proof of the Hoare triple $\{\mathcal{P}\}c\{\mathcal{Q}\}$ consists in transforming the heap according to the program logic semantics of the synthesized statements. Thus, the runtime information from the program search procedure of Section 4.1.2 inherently

contains a correctness proof of the synthesized program.

However, the transient representation of this information (as log messages) limits its utility for post-hoc analysis. We require a persistent and unified representation of this synthesis data for our purpose.

In this section, we improve the interface between SuSLik and HTT by persisting the synthesis information in a *derivation tree* (Section 4.2.1) and defining the mapping from SSL rules to HTT lemmas (Section 4.2.2).

4.2.1 A Derivation Tree

In Section 4.1.2, we showed how the synthesis footprint, which consists of recursive rule applications that give rise to nested subgoals, forms a tree structure. We capture this footprint as a *derivation tree* that persists information emitted during synthesis (i.e., the rules applied and their corresponding computations).

We implement our derivation tree in Scala with a `Trace` class that stores `TraceNodes`. These `TraceNodes` are co-inductive case classes that capture the mutually recursive relationship of each action in the synthesis algorithm:

1. A `GoalTrace` node stores the current *goal context* and a list of `RuleAppTraces`.

2. A `RuleAppTrace` node stores a *SSL rule application* (to the goal stored in the parent `GoalTrace`) and a list of `SubderivationTraces`.

3. A `SubderivationTrace` node stores a *subderivation* resulting from a rule applied to a goal; it also stores a list of `GoalTraces` corresponding to the list of subgoals generated by this subderivation.

READ

$$\frac{a \in \mathsf{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) \qquad y \notin \mathsf{Vars}(\Gamma, \mathcal{P}, \mathcal{Q}) \qquad \Gamma \cup \{y\}\,;\, [y/a]\{\phi; \langle x, \iota\rangle \mapsto a * P\} \rightsquigarrow [y/a]\{\mathcal{Q}\} \mid c}{\Sigma; \Gamma;\, \{\phi; \langle x, \iota\rangle \mapsto a * P\} \rightsquigarrow \{\mathcal{Q}\} \mid \textbf{let}\ y = *(x + \iota); c}$$

```
bnd_read :
  verify ([x → v] • i) (s₂ v)• r →
    def ([x → v] • i) →
      verify ([x → v] • i) (bind_s (read_s A x) s₂)• r
```

FIGURE 4.4: The "read" rule in SSL (left) and HTT (right).

We then populate this derivation tree in tandem with the main synthesis task. By collecting the successful `RuleAppTraces` from the tree, we know exactly the sequence of SSL rule applications and their parameters. Now, a "sketch" of the correctness proof in SuSLik's program logic is accessible for post-hoc proof generation in Coq.

4.2.2 Mapping Rules

We must also establish a mapping between SSL rules and HTT lemmas. Throughout a proof, HTT maintains a goal context of the form `verify` $i\ s\ q$, where i is the initial heap, s is the program statements, and q is the postcondition assertion about the final heap [15]. This is how the `verify` predicate represents Hoare triples in the context of a Coq proof. HTT implements its structural rules as lemmas about this `verify` predicate. When a lemma is applied, HTT updates the heap accordingly, and pops off a command from s. A multi-statement program is proved by applying a sequence of such lemmas until there are no more commands to process.

Most SSL rules have a counterpart lemma in HTT, so we construct a one-to-one mapping from SSL rules to their HTT counterparts. For instance, we observe a close similarity between SSL and HTT representations of the "read" rule as shown in Figure 4.4. The SSL version replaces

all occurrences of a ghost variable a in the goal with a fresh program variable y. The HTT version does likewise with helper lemmas `bind_s` and `read_s`. We observe this Figure 4.1, where an operation to read from a pointer destination is synthesized by the SSL rule READ, which corresponds to the HTT lemma `bnd_readR`.

4.3 Mind the Gap

Our strategy for certification consists of the two adaptations described in Section 4.2— traverse the derivation tree to collect successful SSL rule applications, and map each one to its corresponding HTT lemma.

However, a representational "gap" remains. This becomes apparent when comparing the HTT lemmas invoked in Figure 4.1 and the SSL rules of Figure 4.2. The first line of the HTT proof invokes `ghR`, a lemma about ghost existential instantiation. There is no equivalent SSL rule, as this step is implicit in SUSLIK; thus, the synthesizer must use the derivation tree to deduce the necessary information for the proof generation. We discuss this in depth in Section 5.1.

This is one of several instances where SUSLIK's derivation tree is a necessary but insufficient tool to certify a synthesized program. While the derivation tree contains all required information, it does not immediately yield the information to construct a formal HTT proof. The next chapter describes the legwork required to bridge the gap between these two mediums.

Chapter 5

Bridging the Gap

In this chapter, we introduce a number of procedures to transform the SuSLik representation of each component into its HTT representation, thereby bridging the gap identified in Section 4.3.

- Section 5.1 discusses inductive predicates and assertions.
- Section 5.2 discusses program specifications.
- Section 5.3 discusses proofs. We defunctionalize the synthesis's continuations in Section 5.3.1, and describe proof generation using both the derivation tree (Section 5.3.2) and Ltac tactics (Section 5.3.3).
- Section 5.4 discusses programs.

We translate all components of SuSLik the same way. We first define the primitives needed to construct a component's abstract syntax tree (AST) representation in HTT. Then, we define a functional translation procedure, wherein the AST expression in SuSLik is pattern-matched, and its sub-expressions are evaluated recursively, to produce an AST expression in HTT. Finally, during Coq script generation, we traverse the AST and emit a pretty-printed string representation of each node.

5.1 Predicates and Assertions

First, we discuss the translation of *inductive predicates*. Figures 3.1 and 3.3 each show the SuSLik and HTT expressions of the `lseg` predicate discussed in Section 3.2. Both representations share the common features of a predicate name, (typed) parameters, and *inductive clauses*. Each clause consists of a selector (a boolean expression) and a corresponding *assertion*. HTT predicates range over an additional heap variable that represents the local fragment of the heap that this predicate reasons about.

Next, we discuss the translation of *assertions*. Each assertion consists of a *pure* part (a boolean formula ranging over symbolic values) and a *spatial* part (a collection of descriptions about disjoint *heaplets*). In SuSLik, the pure part is encoded as an expression, whose sub-expressions are conjoined by a $\wedge$ operator. The spatial part is a formula consisting of heaplets that are conjoined by a *separating conjunction* operator `**`. In HTT, the entire assertion is encoded as a function parametrized over a heap variable representing the current heap context. Both the pure and spatial parts are `Prop` expressions combined into a single proposition by Coq's native conjunction operator $\wedge$. The spatial part equates the heap parameter variable to the union of disjoint heaplets by the operator `\+`.

In SuSLik, a heaplet may be any one of:

1. An *empty heap* assertion (`emp`).

2. A *points-to* assertion $\langle x, \iota \rangle \mapsto e$, which denotes that value $x + \iota$ (where x is a variable/pointer constant and ι is an optional integer offset) references a location storing payload element e.

3. An *allocated memory block* of the form $[x, sz]$, which expresses a block of size sz rooted at location x.

4. A *predicate application*.

Some of these details differ in HTT. For instance, the empty heap is represented by the unit type. Also, the presence of allocated memory blocks is implicitly conveyed by the complementary points-to assertions, so there is no explicit syntax for it.

Another difference from SuSLik is that HTT expects value and heap existentials to be provided. We obtain value existentials by collecting variables that appear in the assertion, and removing any that were declared in an outer scope. (For example, an assertion that is part of a predicate definition should not treat the predicate's parameters as existentials.) Meanwhile, heap existentials are used to elaborate on recursive sub-heap structures; they are passed as the last argument in predicate applications, as observed in Figure 3.3's `lseg` predicate.

We observe both types of existentials in the following assertion excerpted from the HTT `lseg` predicate's second constructor (Figure 3.3):

```
∃ nxt s1 v, (* Value existentials *)
∃ h', (* Auxiliary heap existentials *)
  s = [:: v] ++ s1 ∧ (* Pure part *)
  h = x ↦ v \+ x .+ 1 ↦ nxt \+ h' ∧ (* Spatial part *)
  lseg nxt s1 h'. (* Elaborating on recursive sub-heaps *)
```

We need to generate a heap existential for every predicate application present in the assertion. Since this assertion has one predicate application, we generate one heap existential.

```
void swap(loc x, loc y) []
{true ; x ↦ a ** y ↦ b}
{true ; x ↦ b ** y ↦ a}
```

```
Definition swap_type (x:ptr) (y:ptr) :=
  {(a : nat) (b : nat)},
    STsep (
      fun h ⇒
        true ∧ h = x ↦ a \+ y ↦ b,
      [vfun (_: unit) h ⇒
        true ∧ h = x ↦ b \+ y ↦ a]
    ).
```

FIGURE 5.1: The `swap` specification in SUSLIK (left) and HTT (right).

```
void listfree(loc x) []
{true ; lseg(x, S)}
{true ; emp}
```

```
Definition listfree_type :=
  ∀ (x: ptr), {s : seq nat},
    STsep (
      fun h ⇒ lseg x s h,
      [vfun (_ : unit) h ⇒ h = empty]).
```

FIGURE 5.2: The `listfree` specification in SUSLIK (left) and HTT (right).

5.2 Program Specifications

A *specification* describes the behavior of a program in terms of its precondition and postcondition assertions. In HTT, the placement of program variables differs between inductive and non-inductive program specifications. For non-inductive specifications like `swap` (Figure 5.1), they are passed in as dependent type parameters, while for inductive ones like `listfree` (Figure 5.2), they are universally quantified. This is a subtle implementation detail of HTT needed to satisfy Coq's typechecking. We decide by checking SUSLIK's derivation tree for any successful applications of SSL rules that pertain to induction.

5.3 Proofs

Earlier in Section 4.2.1, we showed how the derivation tree stores synthesis steps that provide a rough *proof "sketch"* for constructing the *formal*

$$\textsc{Free}\quad \frac{R = [x,n] * *_{0\le i\le n}\,(\langle x,i\rangle \mapsto e_i) \quad \mathsf{Vars}\,(\{x\}\cup\{\overline{e_i}\}) \subseteq \Gamma \quad \Sigma;\Gamma;\ \{\phi;P\} \rightsquigarrow \{\mathcal{Q}\}\,|\,c}{\Sigma;\Gamma;\ \{\phi;P*R\} \rightsquigarrow \{\mathcal{Q}\}\,|\,\texttt{free}(n);c}$$

```
bnd_dealloc :
    (def i → verify i (s2 ())• r) →
    def ([x → v] • i) →
        verify ([x → v] • i) (bind s (dealloc s x) s2)• r
```

FIGURE 5.3: The "free" rule in SSL (left) and HTT (right).

proof of correctness in HTT.

We approach the task of proof generation by mapping the SSL rules to HTT lemmas. Then, we follow the rule applications on the *successful branches* of the derivation tree, obtaining the HTT representation of each derivation. Some SSL rules do have a direct equivalent in HTT; for example, READ corresponds exactly to the HTT lemma `bnd_readR`. However, not all translations are so straightforward. As we noted in Section 4.3, the interface is imperfect and representational "gaps" remain.

In some cases, a loose mapping exists, but with a slight difference in behavior that requires more contextual information to resolve than is provided in SUSLIK's rule application. For example, the FREE rule in SSL prescribes behavior for freeing an entire memory block rooted at a program variable x, but `bnd_dealloc`, its closest equivalent in HTT, only prescribes behavior for point-wise deallocation (Figure 5.3). Thus, `bnd_dealloc` must be invoked multiple times to match a single FREE.

For this particular proof generation to succeed, the corresponding synthesis computation stored in the derivation tree must be aware of the size of the memory block to free, so that it can issue that number of `bnd_dealloc` lemmas. Yet, crucial information about the synthesis computations is *present* but not *accessible* in the current form of the derivation tree (Section 4.2.1), preventing its use as the source of truth.

```
case class Prepend(s) extends StmtComputation {
  def apply(stmts) = SeqComp(s, stmts.head)
}

case class MakeOpen(selectors) extends StmtComputation {
  def apply(stmts) = {
    def mkNestedIf(branches) = branches match {
      case b :: Nil ⇒ b.stmt
      case b :: rest ⇒ If(b.selector, b.stmt, mkNestedIf(rest))
    }
    mkNestedIf(selectors.zip(stmts))
  }
}
```

FIGURE 5.4: Two defunctionalized continuations in abbreviated Scala.

5.3.1 Interlude: Defunctionalizing Continuations

As discussed in Section 4.1.2, CPS passes along a continuation with every function call, and specifies how the function's result should be handled. SUSLIK uses continuations to compute synthesis steps in a backtracking search, where evaluation order must be inside out. The issue is that continuations encoded as functions are not traversable. We have no way to "unwrap" the layers of computation if we wish to reason about them at a later stage (namely, for the task of HTT proof generation).

We address this by *defunctionalizing* the continuations, which transforms higher-order functions into data types [5]. Inhabitants of this data type implement a first-order `apply` function that inherits the signature of the continuation function, and performs the computation of interest. Through defunctionalization we obtain an enumeration of the possible synthesis action types, which lays bare the exact *form* of the computation. Each defunctionalized computation is a "snapshot" of the synthesis procedure that may now be stored in the derivation tree's nodes, and executed and re-executed at will [5].

We encode our synthesis continuation as an enumerable set of Scala case classes representing several types of synthesis action. Two synthesis

actions of type `StmtComputation` are shown in Figure 5.4. `Prepend` represents the basic action of prepending a statement to a sequence of previously synthesized statements. `MakeOpen` produces a nested conditional statement from the synthesized subgoals.

Each of these actions becomes a concrete synthesis computation when its class is instantiated with the appropriate parameters. Suppose we have two selectors b_1 and b_2, and for each selector, we wish to store the value 10 or 20 in the variable x with zero offset. Then, the computation `MakeOpen`(b_1, b_2) applied to the sequence of statements `[Store(`$x, 0, 10$`), Store(`$x, 0, 20$`)]` synthesizes the program statement:

$$\texttt{If}(b_1\texttt{, Store}(x,0,10)\texttt{, Store}(x,0,20))$$

where the three parameters of `If` are the selector, *then* branch, and *else* branch, and where it is assumed that b_2 is matched if b_1 is not.

5.3.2 Using the Derivation Tree

Equipped with a defunctionalized derivation tree, we can generate the HTT correctness proofs for our SuSLik programs. For example, at the start of this section, we encountered the problem of storing the size of the deallocated heap in the synthesis computation. This is now resolved by defining a variant of `Prepend` (Figure 5.4) that stores an additional `size` parameter for later access.

We also touch briefly on another problem, the instantiation of value and heap existentials required of all HTT program proofs, as described

in Section 5.1. This is something we can handle at the start of the translation procedure, by searching SUSLIK's goal context for ghost variables and predicate applications, and determining the appropriate existential variables to instantiate. Then, we emit proof steps that extract the variables into the proof context with the lemma `ghR`. We observe this in the first line of the proof in Figure 4.1:

```
apply: ghR; move⇒h [a b]→HValid//=.
```

5.3.3 Using LTAC

We consider one final translation problem that concerns the WRITE rule of SSL, and the ordering of heaplets expected by its corresponding HTT lemma `bind_writeR`. We find Coq's tactic language LTAC to be well-suited for this task.

To convey the issue, we take a short detour to discuss the *frame rule* of SL, which states that we can safely extend local reasoning about the direct footprint of a program to an assertion about a larger heap [16]. Consider the following SL assertion.

$$\{y \mapsto b2 * x \mapsto b2\}\ \texttt{*y = a2;}\ \{y \mapsto a2 * x \mapsto b2\} \tag{5.1}$$

We notice that the heaplet $x \mapsto b2$ stays constant before and after program execution. By the frame rule, if assertion

$$\{y \mapsto b2\}\ \texttt{*y = a2;}\ \{y \mapsto a2\} \tag{5.2}$$

holds, then Equation 5.1 must also hold.

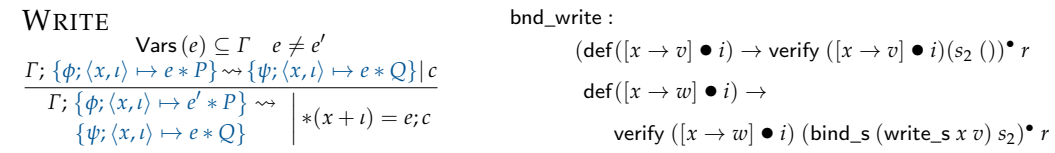

FIGURE 5.5: The "write" rule in SSL (left) and HTT (right).

Now, we return to discuss WRITE. The standalone rule (Figure 5.5) writes an expression e into the memory location $\langle x, \iota \rangle$. For the sake of efficiency, SUSLIK's implementation of WRITE combines it with a SL frame rule application. We confirm an occurence of the latter in the synthesis log for `swap` (Figure 4.2), where the WRITE rule produces a subgoal in the style of Equation 5.2.

On the other hand, HTT's `bnd_writeR` lemma is not combined with the frame rule, and its usage is better described by Equation 5.1. Therefore, the HTT proof in Figure 4.1 is valid, but it does not faithfully represent SUSLIK's goal. A more accurate HTT translation of SUSLIK's derivation requires us to explicitly invoke the `frame` lemma (which implements the frame rule), in addition to the main lemma `bnd_writeR`. Crucially, `frame` expects the heaplets to be in a particular order, and we do so with `joinC` and `unitL`, two helper lemmas that reorder the heaplets. This reordering problem is unique to HTT; the need arises from its representation of heaps as PCMs (Section 3.2.2). SUSLIK, on the other hand, represents heaps as order-agnostic sets, so there is no need to provide this machinery explicitly.

With the framing step included, the first WRITE application can be translated to:

```
apply: bnd_writeR⇒//=; rewrite !(joinC (x ↦ b)); apply: frame.
```

```
Ltac ssl_write_post x :=
  match goal with
  | [ |- verify (x ↦ _ \+ _) _ _ ] ⇒
    rewrite !(joinC (x ↦ _))
  | [ |- verify (x ↦ _) _ _ ] ⇒
    rewrite -(unitL (x ↦ _))
  end;
  apply frame.
```

FIGURE 5.6: A LTAC tactic to reorder the heap for the frame rule.

Likewise, the second application translates to:

```
apply: bnd_writeR⇒//=; rewrite -(unitL (y ↦ a)); apply: frame.
```

This is a process we can automate with LTAC, a *tactic language* for Coq. It provides the necessary utilities to build proof automation tactics in Coq, such as the ability to pattern-match on both Coq terms and the proof context (goal and hypotheses). Notably, HTT does perform extensive proof automation on its own. It is designed to be maintainable and composable, by defining procedures in terms of Coq's native type system [8]. However, we may defer to LTAC to resolve our small scale ad-hoc automation needs.

Figure 5.6 shows an LTAC tactic that prepares the proof context for invoking one of the main lemmas, by defining tactics that perform auxiliary heap reordering tasks. We pattern match to invoke different reordering tactics depending on the size of the heap (namely, `joinC` versus `unitL`).

5.4 Programs

The defunctionalized derivation tree (Section 5.3.1) facilitates proof generation (Section 5.3.2, but it can also be used to generate programs in the syntax of HTT.

```
swap (loc x, loc y) {
  let a2 = *x;
  let b2 = *y;
  *x = b2;
  *y = a2;
}
```

```
Program Definition swap : swap_type :=
  fun x y ⇒
    Do (a2 := @read nat x;
        b2 := @read nat y;
        x ::= b2;;
        y ::= a2;;
        ret tt).
```

FIGURE 5.7: The `swap` program in SUSLIK (left) and HTT (right).

Both SUSLIK and HTT support a simple imperative language. The most straightforward way of translating program statements into HTT is by deriving a one-to-one mapping from SUSLIK's language constructs to those of HTT. Though this method suffices for simple programs, a program is but an artifact. It is a specialized manifestation of the rich context that was available during synthesis, so some loss of information is unavoidable.

Rather than translating one specialized form into another, we can obtain the information needed to construct the HTT representation directly from the context under which the SUSLIK program was synthesized. In other words, we can use the derivation tree to directly generate not only proofs (Section 5.3), but also programs.

We recall the `StmtComputation` case classes we defined in Figure 5.4. Since these computations emit SUSLIK AST code, we can simply define corresponding computations to emit HTT AST representations. With this method of HTT program generation, it becomes more accurate to say we are *synthesizing a program in the flavor of* HTT, rather than *translating a synthesized* SUSLIK *program into* HTT.

Chapter 6

Case Studies

In this chapter, we demonstrate our findings from Chapter 5 with the correctness proofs of two programs.

6.1 The CALL Rule in `listfree`

During the synthesis of recursive functions, the CALL rule in SSL (Figure 4.3) applies an inductive hypothesis that was previously added to the synthesis context via another SSL rule. Translating the application of the CALL rule to an HTT proof requires information from both the derivation tree (Section 5.3.2) and LTAC tactics (Section 5.3.3). Here, we demonstrate this with a proof of a recursive *destructor* for linked lists, `listfree` (Figure 6.1) that frees the memory occupied by a list.

First, we instantiate the ghost variables. This function has one ghost variable, the sequence of numbers `S`. The rule alone does not tell us what should be the correct instantiation. However, the synthesis computation associated with this rule application provides a *substitution map* from the specification's placeholder variables to the current context's variables to

```
Definition listfree_type :=
  ∀ (x: ptr),
    {s : seq nat}, STsep (
      fun h ⇒ lseg x s h,
      [vfun (_ : unit) h ⇒ h = empty]).

Program Definition listfree : listfree_type :=
  Fix (fun (listfree : listfree_type) x ⇒
    Do (
      if x == 0
      then ret tt
      else nxt2 := @read ptr (x .+ 1);
           listfree nxt2;;
           dealloc (x .+ 1);;
           dealloc x;;
           ret tt
    )).
```

```
(* Existential elimination *)
ssl_ghostelim_pre.
move⇒S//=.
move⇒H_lseg HValid.
(* Open rule *)
case: ifP⇒cond; case H_lseg;
rewrite cond//=⇒_.
- (* Case 1: x == 0 *)
  move⇒[E] [→]; ssl_emp.
- (* Case 2: x != 0 *)
  move⇒[nxt] [s1] [v].
  move⇒[h'].
  move⇒[E] [→] H_rec_lseg.
  (* nxt2 := !(x .+ 1) *)
  ssl_read.
  (* listfree nxt2 *)
  put_to_head h'.
  apply: bnd_seq.
  apply: (gh_ex s1).
  apply: val_do⇒//= _ ? →;
  rewrite unitL⇒_.
  (* dealloc (x .+ 1) *)
  ssl_dealloc.
  (* dealloc x *)
  ssl_dealloc.
  ssl_emp.
```

FIGURE 6.1: An HTT specification, program, and proof of `listfree`.

```
Ltac put_to_head h :=
  repeat match goal with
  | [|- context[_ \+ h]] ⇒ rewrite joinC
  | [|- context[_ \+ (h \+ _)]] ⇒ rewrite joinCA
  end.
```

FIGURE 6.2: A LTAC tactic to reorder the heap for the call rule.

instantiate with. This substitution map tells us that the appropriate instantiation for this variable `S` in the context of the `listfree` function body is the existential variable `s1`. Our derivation tree provides this map.

Next, we prepare the proof context for using the lemma `val_do`, which expects the heaplet passed to the recursive call to be at the beginning of the heap. We define a LTAC tactic (Figure 6.2), which takes a heaplet h as a parameter, and repeatedly executes a combination of commutative and associative rules to reorder the heap so that h becomes the leftmost heaplet. Finally, we call `val_do` and end with some other auxiliary simplifications and reordering.

```
Inductive tree (x : ptr) (s : seq nat) (h : heap) : Prop :=
| tree0 of x == 0 of
    s = nil ∧ h = empty
| tree1 of x != 0 of
    ∃ v s1 s2 r l,
    ∃ h' h'',
    s = [:: v] ++ s1 ++ s2 ∧
    h = x ↦ v \+ x.+1 ↦ l \+ x.+2 ↦ r \+ h' \+ h'' ∧
    tree l s1 h' ∧ tree r s2 h''.

Definition treefree_type :=
  ∀ (x : ptr),
  {(s : seq nat)},
    STsep (
      fun h ⇒ tree x s h,
      [vfun (_: unit) h ⇒ h = empty]).

Program Definition treefree : treefree_type :=
  Fix (fun (treefree : treefree_type) x ⇒
    Do (
      if x == 0
      then ret tt
      else l2 := @read ptr (x .+ 1);
           r2 := @read ptr (x .+ 2);
           treefree l2;;
           treefree r2;;
           dealloc (x .+ 2);;
           dealloc (x .+ 1);;
           dealloc x
    )).
```

```
(* Existential elimination *)
ssl_ghostelim_pre.
move⇒s//=.
move⇒H_tree HValid.
case: ifP⇒cond; case H_tree;
rewrite cond//=⇒_.
- (* Case 1: x == 0 *)
  move⇒[E] [→].
  ssl_emp.
- (* Case 2: x != 0 *)
  move⇒[v] [s1] [s2] [r] [l].
  move⇒[h'] [h''].
  move⇒[E] [→].
  move⇒[H_rec_tree H_rec_tree'].
  (* l2 := !(x .+ 1) *)
  ssl_read.
  (* r2 := !(x .+ 2) *)
  ssl_read.
  (* treefree l2 *)
  put_to_head h'.
  apply: bnd_seq. apply: (gh_ex s1).
  apply: val_do⇒//= _ ? →;
  rewrite unitL⇒_.
  (* treefree r2 *)
  put_to_head h''.
  apply: bnd_seq. apply: (gh_ex s2).
  apply: val_do⇒//= _ ? →;
  rewrite unitL⇒_.
  (* dealloc (x .+ 2) *)
  ssl_dealloc.
  (* dealloc (x .+ 1) *)
  ssl_dealloc.
  (* dealloc x *)
  ssl_dealloc.
  ssl_emp.
```

FIGURE 6.3: A generated HTT certificate for `treefree`.

6.2 Other Data Structures With `treefree`

We can also extend our certified synthesis to programs that reason about other data structures. In Figure 6.3, we define a `tree` predicate to assert the existence of a binary tree on some local heap, as well as the specification, program, and proof of `treefree`, a *destructor* for trees.

There are two recursive calls in `treefree` (one for each branch of the tree), and correspondingly two subheaps. With the method described in Section 6.1, our certification procedure successfully synthesizes these function calls and their proofs. It also correctly detects that it must emit *three* deallocation statements in HTT, using the method of Section 5.3.2.

Chapter 7

Discussion

7.1 Running the Project

The certification procedure for SuSLik described in this project has been tested with Coq 8.9.0. Users should verify generated certificates by adding HTT (source) to their `$COQPATH`, as well as the tactics stored in the file `tactics.v`. Instructions and example certificates are available at:

https://github.com/yasunariw/suslik/tree/coq-translation/examples/certificates.

7.2 Future Work

The focus of this project has been to establish the infrastructure to make the diagram in Figure 7.1 commute, by establishing an intial pipeline for the automated construction of certified programs from user-provided specifications. Further work would focus on two areas—broadening our pipeline, and supporting other third-party certification targets.

Our immediate next task is to incorporate information from SuSLik's SMT solver into the certification procedure. While we chose examples

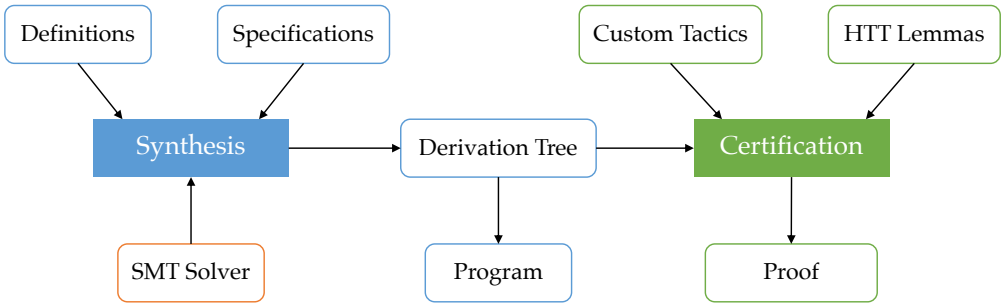

FIGURE 7.1: The machinery in a nutshell.

that can be certified without a SMT solver in this project, the proof generation of other programs rely on the solver for reasoning about a specification's pure constraints. For example, consider a `listcopy` procedure that copies a linked list defined in terms of the `lseg` predicate. The SMT solver emits certain facts about the `lseg` predicate that aid this program's synthesis. However, automating the certification of this procedure in HTT would require us to incorporate a number of `lseg`-related lemmas that can replicate the SMT solver's contribution to synthesis. This is certainly feasible, but was not fully carried out in the interest of time.

Along the same lines, we may also expand the certification repertoire to additional data structures. So far, we have shown the results for the synthesis of recursive programs that manipulate singly linked lists and trees. However, SUSLIK has also been demonstrated to efficiently synthesize non-trivial programs that operate on other data structures such as binary search trees and sorted lists, so we may naturally extend our post-hoc certification to the same.

Finally, given that we were able to use the derivation tree to generate both proofs (Section 5.3.2) and programs (Section 5.4), the derivation tree presents itself as a promising intermediate representation for SUSLIK.

This forms the basis for supporting more than one certification backend. One such alternative third-party certification target is Isabelle, a generic theorem prover that can encode different object logics, including higher order logic [13].

7.3 Conclusion

The result of Chapter 5, that a program and its correctness proof can be computed from a single derivation tree, underscores the Curry-Howard isomorphism, which treats proofs as programs, and programs as proofs [4, 12].

In this work, we presented a fully automated approach to synthesizing certifiably correct programs. We demonstrated how the footprint of a deductive synthesis procedure contains all of the information necessary to provide a strong correctness guarantee. By defunctionalizing the procedure, we captured the rich contextual information produced during synthesis, and made it accessible for post-hoc generation of both programs *and* proofs in a format suitable for third party verification. Finally, we implemented these steps by extending the deductive program synthesizer SuSLik to produce verifiable Coq certificates. Together with prior work on SuSLik, we achieved an end-to-end pipeline that can parse a concise program specification to produce a fully formal definition, implementation, and certificate.

Bibliography

[1] Andrew W. Appel. “Verified Software Toolchain - (Invited Talk)”. In: *ESOP*. Vol. 6602. LNCS. Springer, 2011, pp. 1–17 (cit. on p. 10).

[2] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. “VST-Floyd: A separation logic tool to verify correctness of C programs”. In: *Journal of Automated Reasoning* 61.1-4 (2018), pp. 367–422 (cit. on p. 10).

[3] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. “Using Crash Hoare logic for certifying the FSCQ file system”. In: *SOSP*. 2015, pp. 18–37 (cit. on p. 8).

[4] Haskell B. Curry. “Functionality in combinatory logic”. In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pp. 584–590 (cit. on p. 40).

[5] Olivier Danvy and Lasse R. Nielsen. “Defunctionalization at work”. In: *PPDP*. ACM, 2001, pp. 162–174 (cit. on p. 29).

[6] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. “Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant”. In: *POPL*. ACM, 2015, pp. 689–700 (cit. on p. 5).

[7] Robert W. Floyd. "Assigning meanings to programs". In: *Proceedings of the Symposium on Applied Mathematics*. Vol. 19. 1967, pp. 19–31 (cit. on p. 4).

[8] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. "How to make ad hoc proof automation less ad hoc". In: *ICFP*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 163–175 (cit. on p. 33).

[9] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *OSDI*. 2016, pp. 653–669 (cit. on p. 8).

[10] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-Output Examples". In: *SIGPLAN Notices* 46.1 (2011), pp. 317–329 (cit. on p. 3).

[11] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580 (cit. on p. 4).

[12] William A. Howard. "The formulae-as-types notion of construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Original paper manuscript from 1969. Academic Press, 1980, pp. 479–490 (cit. on p. 40).

[13] *Isabelle*. URL: `https://isabelle.in.tum.de/` (cit. on p. 40).

[14] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert – A formally verified optimizing compiler". In: SEE, 2016 (cit. on p. 10).

[15] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. "Structuring the verification of heap-manipulating programs". In: *POPL*. 2010, pp. 261–274 (cit. on pp. 12, 22).

[16] Peter W. O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)." In: *Software Safety and Security* 33 (2012), pp. 286–318 (cit. on p. 31).

[17] Nadia Polikarpova and Ilya Sergey. "Structuring the synthesis of heap-manipulating programs". In: *PACMPL* 3.POPL (2019), 72:1–72:30 (cit. on pp. 4, 7).

[18] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. IEEE Computer Society, 2002, pp. 55–74 (cit. on p. 4).

[19] Ilya Sergey. *Programs and Proofs. Mechanizing Mathematics with Dependent Types*. Lecture notes with exercises, 2019. Available at `http://ilyasergey.net/pnp` (cit. on p. 7).

[20] *The Coq Proof Assistant*. URL: `https://coq.inria.fr/` (cit. on p. 8).

[21] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. "Certifying graph-manipulating C programs via localizations within data structures". In: *PACMPL* 3.OOPSLA (2019), 171:1–171:30 (cit. on p. 10).