A FRAMEWORK

FOR

CERTIFIED PROGRAM SYNTHESIS

YASUNARI WATANABE

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2021

Supervisor:

Associate Professor Ilya Sergey

Examiners:

Professor Abhik Roychoudhury

Associate Professor Martin Henz

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

April 2, 2021	Yasunari Watanabe
....	....
Date	*Name*

Acknowledgements

This work was made possible thanks to the help of many individuals—

Ilya Sergey, whose patient mentorship over the past two years has given me a deep appreciation for the field. I am very grateful for the valuable guidance that helped shape me into a better researcher.

Kiran Gopinathan and George Pîrlea, who implemented the certification for VST and Iris respectively. I owe much of the abstract evaluator design to their insightful observations about the synthesis/verification gap and subsequent discussions.

Everyone at the VERSE research group, for the many conversations shared over lunch at 99 Buona Vista Kitchen.

And my family, for remaining a steadfast presence across the ocean during these highly unusual times.

Contents

Summary

We present a generic framework for automated post-hoc certification of deductively synthesized programs. Our theoretical contribution is an abstract *proof evaluator*, which parses a synthesis derivation trace to generate a certificate of functional correctness for any trusted verifier. Our practical contributions are a Scala *implementation* of this evaluator for SuSLik [PS19], a state-of-the-art deductive synthesizer; an *instantiation* of it for Hoare Type Theory (HTT) [NVB10], a foundational program verification framework; and extensive evaluation on a set of indicative benchmarks.

Deductive synthesis produces programs from user-provided specifications, by conducting a *proof search* on a set of inference rules and emitting the program as a byproduct of the search. The deductive program synthesizer SuSLik can synthesize a broad class of imperative heap-manipulating programs from Hoare-style specifications by applying rules of *cyclic synthetic separation logic ($SSL_{\circlearrowleft}$)* [Itz+21].

Although such programs are said to be *correct by construction*, in practice the implementation of the synthesizer itself is large and complex—a recipe for introducing bugs. Verifying the correctness of the synthesizer's codebase is untenable; however, doing so for each synthesized program is both sound and achievable.

A first attempt at post-hoc certification might extract the successful rule derivations that produced a synthesis result, and execute the corresponding proof steps in the program logic of the target verifier to construct a certificate. Unfortunately, a fundamental gap separates the proof strategies of synthesis and verification: synthesis transforms a goal's pre- and postcondition through rule applications, while verification does forward-style symbolic execution to transform the precondition only, delaying the postcondition entailment check to the very end.

Our *proof evaluator* is designed to overcome this discrepancy. We isolate all verifier-specific reasoning to a *proof step interpreter* interface that each verifier can implement, to share generic traversal logic among all instantiations of the evaluator. The interpreter *locally* maps the synthesizer's rule derivations to equivalent proof steps of a target verifier, but we also equip it with two features to reason *non-locally* about derivations before and after the current one in a principled manner. A *proof context* accumulated throughout the evaluation allows the interpreter to store verifier-specific information for later access, while an interface for *deferred steps* allows it to emit steps corresponding to the current derivation in a delayed fashion.

We showcase the evaluator's features with an instantiation for HTT. Its shallow embedding makes for a pleasant implementation experience overall, but several non-trivialities arise from HTT's need to reason about heaps explicitly. We address them through a combined use of the evaluator's features and tactic engineering in Coq, the proof assistant into which HTT is embedded. We also present strategies to extract pure entailments and automate their proofs using certified solvers, and discuss their limitations.

Finally, we validate our approach to certifying programs in HTT against 18 standard benchmark programs that operate on various data structures. Additionally, we experiment with 11 advanced benchmarks that either require an alternative encoding of collection payloads or manual proofs of complex pure entailment lemmas.

List of Tables

List of Figures

1 Introduction

Testing shows the presence, not the absence of bugs.

—Dijkstra

Demand for provable program correctness has grown, as we entrust software systems to manage everything from stock exchanges to city power grids. Yet, developing certified software remains a laborious task. In order to demonstrate a program's correctness, a human must write a program specification, implementation, and proof that the latter satisfies the former. For this reason, well-known verification projects such as CompCert [Ler06] and CertiKOS [Gu+16] have each taken several person-years of effort to complete.

One promising way to alleviate this human burden is to automate parts of the task through *deductive program synthesis*. Using this approach, a programmer can write a program specification, and then delegate the derivation of a corresponding program implementation and correctness proof to a deductive *synthesizer*.

Consider a simple example of a procedure that copies a singly linked list.

$$\{r \mapsto x * \mathsf{sll}(x, S)\}\ \texttt{void sll\_copy(loc r)}\ \{r \mapsto y * \mathsf{sll}(x, S) * \mathsf{sll}(y, S)\} \qquad (1.1)$$

In this *declarative* specification, we observe a predicate sll that asserts the presence of a singly linked list. The precondition asserts that location r points to location x, and that x contains a singly linked list that stores the elements of multi-set S. The *separating conjunction* connective $*$ indicates that the memory location described by r is disjoint from the region of memory occupied by the linked list.

The postcondition asserts that there is still a linked list rooted at x, but also that location r now points to some location y which also stores a linked list containing the same elements.

SuSLik [PS19] is a deductive synthesizer that can automatically generate valid program implementations for specifications of this nature. For specification (1.1), it produces the imperative program shown in Fig. 1.1, expressed in C-style syntax.

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) { }
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

Figure 1.1: Code of `sll_copy`.

1.1 The Need For Automated Certification

But how can we guarantee that the program in Fig. 1.1 is *correct* with respect to specification (1.1)? After all, the synthesizer is itself a program written by humans. Its implementation is large and complex, so it is difficult to guarantee the total absence of bugs.

One approach is to make the synthesizer produce *certificates* of correctness that can be easily validated by *verification frameworks*, or *verifiers*. These frameworks are typically embedded into proof assistants such as Coq, which have a minimal trusted codebase, and are thus widely considered to have the highest possible correctness guarantees.

Since a deductive synthesizer derives a program via a proof search over the space of the program logic's inference rules, it should be easy enough to instrument this certificate extraction, given that the necessary proof steps are effectively encoded in the derivation of the program—or so it seems.

In reality, the process is far more complex, due to a discrepancy between a proof in a verification framework and a deductive synthesis "proof". Verifiers usually conduct proofs by *symbolically executing* through a program structure that is known ahead of time. Such a proof proceeds by forward-propagating the precondition's symbolic state at each step of the program, and then checking that the final symbolic state after completing the execution matches that of the postcondition. Meanwhile, a synthesizer does not have this program structure readily available to guide the proof (in fact, it is the very thing it needs to synthesize). It therefore resorts to manipulating both the pre- and postcondition to perform its proof search.

In an earlier effort [Wat20], we resolved this discrepancy in an ad-hoc manner, but this made the codebase rather brittle once we began to expand the certification coverage to a wider class of programs. Furthermore, we encountered extensive code duplication once we started to target multiple verification frameworks for the certificate generation, because adding support for a new verifier meant encoding another custom traversal of the SuSLik derivation tree with its own bookkeeping machinery. A more unified approach was needed.

1.2 Contributions and Overview

In the following chapters, we describe an *automated* and *modular* technique for producing correctness certificates of deductively synthesized programs. We present four key contributions:

1. A generic design for a *proof evaluator*, which parses a SuSLik derivation trace and generates a proof for a chosen verifier.

2. A practical implementation of this proof evaluator for SuSLik, written in Scala.

3. An instantiation of the proof evaluator for Hoare Type Theory (HTT) [NVB10], a foundational program verification framework.

4. An extensive evaluation of HTT translation on a series of characteristic benchmarks.

The rest of this work is structured as follows. Ch. 2 gives background knowledge on how SuSLik deductively synthesizes programs by application of $\mathrm{SSL}_{\circlearrowleft}$ rules, and how program correctness is proved in the HTT framework. Ch. 3 describes the proof evaluator and explains the observations that motivated our design choices. Ch. 4 shows the proof evaluator in action with HTT as the verification framework, outlining how our evaluator design and some Coq proof engineering allows us to overcome various nontrivialities. Ch. 5 evaluates our certification technique against a benchmark suite and highlights several particularly notable case studies. Ch. 6 compares our approach to existing literature on proof-carrying code, automated program synthesis, and certified solvers. Finally, Ch. 7 concludes by discussing the extensibility of this approach to other verification frameworks and future work.

2

Background: Synthesis and Verification

You make 'em, I amuse 'em.

—Dr. Seuss, on children

In this chapter, we describe state-of-the-art approaches to deductive synthesis (Sec. 2.1) and verification (Sec. 2.2) of imperative programs.

2.1 Synthesis With Cyclic Synthetic Separation Logic

In *deductive program synthesis*, the synthesizer takes a user-provided *declarative* specification as its initial *synthesis goal*. It then searches for a series of valid applications of inference rule that transform the initial goal until it is reduced to a trivial entailment. The desired program is obtained as a byproduct of this proof search. Here, we examine a particular synthesis tool, SuSLik [PS19], to see how this works in practice.

2.1.1 Specifications and Predicates

Let us revisit our `sll_copy` example from Ch. 1. Specification (1.1) is expressed in *cyclic synthetic separation logic* (SSL$_{\circlearrowleft}$) [Itz+21], a variant of Separation Logic (SL) [Rey02] used by SuSLik. Both the precondition and postcondition are *assertions*. An SL-style assertion $\{\mathcal{P}\}$ consists of a *pure* and *spatial* part, $\{\phi, P\}$. The pure part ϕ expresses logical constraints on the assertion's variables and values. The spatial part P describes the heap shape using standard SL assertions; a collection of *heaplets* is joined by the separating conjunction connective ($*$):

1. emp asserts an empty heap.

2. $(x + \iota) \mapsto e$ asserts a heaplet defined at one location, storing a value e at a (possibly zero) offset ι from an address x.

3. $[x, n]$ asserts a contiguous block of n elements starting at x that can be deallocated.

4. $\mathsf{p}^{\alpha}(\overline{t_i})$ constrains the shape of a size-α heap with a predicate occurrence with arguments $\overline{t_i}$ (the heap size is often abbreviated).

SSL$_{\circlearrowleft}$ defines the singly linked list predicate that appears in specification (1.1) as follows:

$$\begin{aligned} \mathsf{sll}^{\alpha}(x, s) \ &\triangleq\ x = 0 \land \{s = \emptyset; \mathsf{emp}\} \\ &\ \ \ \mid\ \ x \neq 0 \land \left\{s = \{v\} \cup s_1 \land \beta < \alpha; [x, 2] * x \mapsto v * (x + 1) \mapsto nxt * \mathsf{sll}^{\beta}(nxt, s_1)\right\} \end{aligned} \tag{2.1}$$

The first clause states that, if x is a null pointer, it contains no elements and consists of an empty heap. The second clause states that otherwise, x points to a contiguous two-cell memory block. The first location stores the payload v of the head of the linked list; and the second location stores the location nxt, which contains the remainder of the linked list, as indicated by the recursive assertion using the sll predicate. The annotations α, β are *cardinality variables* that are used in *cyclic proofs* [RB17]. SSL$_{\circlearrowleft}$ takes any constraint defined on these cardinalities as a termination measure for the synthesized recursive programs and their auxiliary procedures. That is, cardinalities represent the size of the heap on which a predicate is defined, and the constraints (such as $\beta < \alpha$) show that it is strictly decreasing in an inductive definition.

2.1.2 Synthesis By Proof Search

SuSLik transforms the user-provided specification (1.1) into an initial synthesis goal of the form $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} | \, c$, where $\{\mathcal{P}\}$ and $\{\mathcal{Q}\}$ are the pre- and postconditions of the specification; Γ is the set of program-level variables (which appear in the program's parameter list) and logical variables (which appear in the assertions); and c is the yet unknown program statement to be synthesized for this goal. Then, starting from the initial goal, SuSLik conducts a *proof search*, enumerating all SSL$_{\circlearrowleft}$ inference rules whose semantics enable it to be applied to the current

$$\textsc{Emp}\quad \frac{\vdash \phi \Rightarrow \psi}{\Gamma;\ \{\phi; \mathsf{emp}\} \rightsquigarrow \{\psi; \mathsf{emp}\} \,|\, \texttt{skip}}$$

$$\textsc{Read}\quad \frac{\forall y.\Gamma;\ \{\phi \wedge y = e; (x+\iota) \mapsto e * P\} \rightsquigarrow Q \,|\, c \qquad x \in PV \quad y \in \mathsf{ProgVars} \setminus \mathsf{Vars}(\Gamma)}{\Gamma;\ \left\{\phi;\ \colorbox{lightgray}{$(x+\iota) \mapsto e$} * P\right\} \rightsquigarrow Q \,\middle|\, \texttt{let}\ \colorbox{lightgray}{y} = *(x+\iota); c}$$

$$\textsc{Write}\quad \frac{\Gamma;\ \{\phi; (x+\iota) \mapsto e * P\} \rightsquigarrow \{\psi; (x+\iota) \mapsto e * Q\} \,|\, c \qquad \mathsf{Vars}(e) \subseteq \mathsf{ProgVars}}{\Gamma;\ \{\phi; (x+\iota) \mapsto e' * P\} \rightsquigarrow \left\{\psi;\ \colorbox{lightgray}{$(x+\iota) \mapsto e$} * Q\right\} \,|\, *(x+\iota) = e; c}$$

$$\textsc{Frame}\quad \frac{\{\phi; P\} \rightsquigarrow \{\psi; Q\} \,|\, c}{\left\{\phi; P * \colorbox{lightgray}{R}\right\} \rightsquigarrow \{\psi; Q * R\} \,\middle|\, c}$$

$$\textsc{Alloc}\quad \frac{\begin{array}{l}\Gamma;\ \{\phi; [y,n] * ((y+i) \mapsto t_i)_{0 \le i < n} * P\} \rightsquigarrow \\ \quad \{\psi; [x,n] * ((x+i) \mapsto e_i)_{0 \le i < n} * Q\} \,|\, c \end{array} \qquad x \in \mathsf{Existentials}(\Gamma)}{\Gamma;\ \{\phi; P\} \rightsquigarrow \left\{\psi;\ \colorbox{lightgray}{$[x,n] * ((x+i) \mapsto e_i)_{0 \le i < n}$} * Q\right\} \,\middle|\, \texttt{let}\ \colorbox{lightgray}{y} = \texttt{malloc}(n); c}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}\forall \overline{x_i}, \overline{v_j}.\exists \overline{\omega_k};\ \{\phi'; P\} \rightsquigarrow \{\psi'; S\} \,|\, f(\overline{x_i}) \\ \Gamma \mathbin{\dot\cup} \forall \sigma(\overline{\omega_i});\ \{[\sigma]\psi' \wedge \phi; [\sigma]S * R\} \rightsquigarrow Q \,|\, c \\ \vdash \phi \Rightarrow [\sigma]\phi' \\ \mathsf{dom}\,(\sigma) = \{\overline{x_i}, \overline{v_j}, \overline{\omega_k}\} \\ \sigma(x_i) \in e[\Gamma] \qquad \sigma(v_j) \in \kappa[\Gamma]\end{array}}{\Gamma;\ \left\{\phi; [\colorbox{lightgray}{σ}]\,\colorbox{lightgray}{P} * R\right\} \rightsquigarrow Q \,\middle|\, \colorbox{lightgray}{f}\,(\sigma(\overline{x_i})); c}$$

$$\textsc{Open}\quad \frac{\begin{array}{l}\Gamma \mathbin{\dot\cup} \forall \overline{\omega_{jk}};\ [\overline{t_i/v_i}]\left\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * P\right\} \rightsquigarrow Q \,\middle|\, c_j\ \textit{for}\ \underline{\textit{all}}\ j{=}1..r \\ \mathsf{p}^{\alpha}(\overline{v_i}) : \overline{\left\langle e_j, \left\{\overline{\omega_{jk}}, \chi_j\right\}\right\rangle R_j}_{j=1..r}\ \text{is s.t. } \omega_{jk} \notin \mathsf{Vars}(\Gamma), \mathsf{GV}(t_i) = \varnothing\end{array}}{\Gamma;\ \left\{\phi;\ \colorbox{lightgray}{$\mathsf{p}^{\alpha}(\overline{t_i})$} * P\right\} \rightsquigarrow Q \,\middle|\, \begin{array}{l}\texttt{if}\ ([\overline{t_i/v_i}]e_1)\ \{c_1\} \\ \texttt{else if}\ ([\overline{t_i/v_i}]e_2)\ \{c_2\}\ \texttt{else}\ \cdots\end{array}}$$

$$\textsc{Close}\quad \frac{\begin{array}{l}\Gamma \mathbin{\dot\cup} \exists \overline{\omega_{jk}};\ \mathcal{P} \rightsquigarrow [\overline{t_i/v_i}]\left\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * Q\right\} \,\middle|\, c_j\ \textit{for}\ \underline{\textit{some}}\ \colorbox{lightgray}{j} \in 1..r \\ \text{Predicate } \mathsf{p}^{\alpha}(\overline{v_i}) : \overline{\left\langle e_j, \left\{\overline{\omega_{jk}}, \chi_j\right\}\right\rangle R_j}_{j=1..r}\ \text{is s.t. } \omega_{jk} \notin \mathsf{Vars}(\Gamma)\end{array}}{\Gamma;\ \mathcal{P} \rightsquigarrow \left\{\phi;\ \colorbox{lightgray}{$\mathsf{p}^{\alpha}(\overline{t_i})$} * Q\right\} \,\middle|\, c}$$

Figure 2.1: Selected declarative rules of SSL$_{\circlearrowleft}$. Grayed parts indicate fragments instantiated non-deterministically by elements of the synthesis goal.

goal, and backtracking on failure (*i.e.*, when no synthesis rules are applicable). A successful synthesis builds a valid derivation from the initial goal, and emits a corresponding program c as a byproduct. Fig. 2.1 show some selected SSL$_{\circlearrowleft}$ rules. Their *declarative* style does not provide an exact *algorithm* to enforce a particular rule application order. That is, for a given synthesis goal, the choice of suitable instantiations of the grayed parts of each rule depends on the proof search strategy. This captures the inherent non-determinism of the synthesis approach. For instance, multiple candidate Read rule applications may be enabled for a synthesis goal if there is more than one heaplet of the form $(x + \iota) \mapsto e$ in the precondition. A deductive synthesis proof search aims to discover precisely the valid rule application order for each specification. Existing work on deductive program synthesis employs various techniques to implement the derivation search efficiently [Kne+13; Itz+21].

We now briefly describe the different categories of rules, and how each one is used to advance the synthesis.

Operational rules. These include Read, Write, Alloc, Open, and Call; they emit program statements as part of their application. A particularly intricate rule is Call, which synthesizes procedure calls. In the style of SL's frame rule, the goal precondition's sub-heap R that doesn't pertain to a procedure call f is *framed out* for local reasoning. The formal parameters $\overline{x_i}$ of f are matched with expressions $e[\Gamma]$ using Γ's program variables, and its ghosts are mapped to $\kappa[\Gamma]$ using any variables in Γ; this is done by a substitution map σ. Applying Call produces two subgoals, which correspond to the two premises in the rule description. The first is entailment of f's precondition from the goal's precondition after framing, which introduces f's existentials $\overline{w_k}$. The second is entailment of the goal's postcondition from f's postcondition, where σ renames f's existentials to fresh ghost variables in the new goal precondition.

Terminal rules. These are rules that conclude a successful synthesis of a program branch. Notably, Emp describes a trivial entailment from an empty heap to itself. Assuming that any remaining pure constraints $\phi \Rightarrow \psi$ hold, SuSLik can forego further rule applications.

Structural rules. Instead of emitting program statements, these rules manipulate the synthesis goal in other ways. For example, Frame applies SL-style framing to remove matching heaplets from the pre- and postcondition. Successive applications of this rule can gradually reduce the goal's heap sizes until the Emp rule is enabled. Close identifies a predicate instance p in the postcondition, and unfolds its occurrence $\mathsf{p}^{\alpha}(\overline{t_i})$ into some j^{th} clause of the predicate, provided that the j^{th} clause's selector is consistent with the goal's constraints. The assertion expanded from the clause can introduce additional facts about the expected postcondition heap shape. For example, the introduction of a block assertion of a certain size may enable the Alloc rule, an operational rule that allocates a memory block.

2.1.3 Synthesis of `sll_copy`

Fig. 2.2 compares the two complementary results of running SuSLik on the `sll_copy` specification (1.1). On the left is the synthesized implementation in SusLang, a toy C-like language of SuSLik. On the right are the $\mathrm{SSL}_{\circlearrowleft}$ rule derivations that contributed to the synthesis of this program. Note that a typical SuSLik run also produces many unsuccessful derivations that the proof search then backtracks from; these have been elided from the presentation in Fig. 2.2. An encoding of these successful derivations, expressed as a *proof tree*, and interleaved with the intermediate synthesis goals, is formally discussed in Sec. 3.1. For now,

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) { }
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

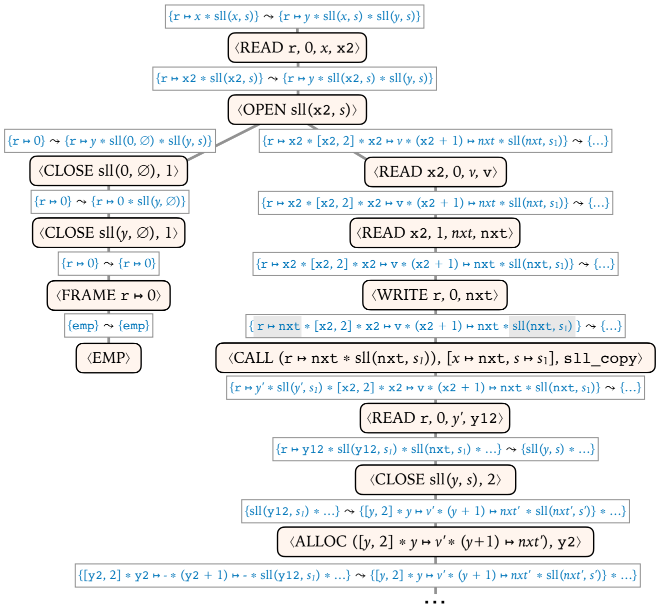

{r ↦ x ∗ sll(x, s)} ⇝ {r ↦ y ∗ sll(x, s) ∗ sll(y, s)}
⟨READ r, 0, x, x2⟩
{r ↦ x2 ∗ sll(x2, s)} ⇝ {r ↦ y ∗ sll(x2, s) ∗ sll(y, s)}
⟨OPEN sll(x2, s)⟩
{r ↦ 0} ⇝ {r ↦ y ∗ sll(0, ∅) ∗ sll(y, s)}
⟨CLOSE sll(0, ∅), 1⟩
{r ↦ 0} ⇝ {r ↦ 0 ∗ sll(y, ∅)}
⟨CLOSE sll(y, ∅), 1⟩
{r ↦ 0} ⇝ {r ↦ 0}
⟨FRAME r ↦ 0⟩
{emp} ⇝ {emp}
⟨EMP⟩
{r ↦ x2 ∗ [x2, 2] ∗ x2 ↦ v ∗ (x2 + 1) ↦ nxt ∗ sll(nxt, s1)} ⇝ {...}
⟨READ x2, 0, v, v⟩
{r ↦ x2 ∗ [x2, 2] ∗ x2 ↦ v ∗ (x2 + 1) ↦ nxt ∗ sll(nxt, s1)} ⇝ {...}
⟨READ x2, 1, nxt, nxt⟩
{r ↦ x2 ∗ [x2, 2] ∗ x2 ↦ v ∗ (x2 + 1) ↦ nxt ∗ sll(nxt, s1)} ⇝ {...}
⟨WRITE r, 0, nxt⟩
{ r ↦ nxt ∗ [x2, 2] ∗ x2 ↦ v ∗ (x2 + 1) ↦ nxt ∗ sll(nxt, s1) } ⇝ {...}
⟨CALL (r ↦ nxt ∗ sll(nxt, s1)), [x ↦ nxt, s ↦ s1], sll_copy⟩
{r ↦ y′ ∗ sll(y′, s1) ∗ [x2, 2] ∗ x2 ↦ v ∗ (x2 + 1) ↦ nxt ∗ sll(nxt, s1)} ⇝ {...}
⟨READ r, 0, y′, y12⟩
{r ↦ y12 ∗ sll(y12, s1) ∗ sll(nxt, s1) ∗ ...} ⇝ {sll(y, s) ∗ ...}
⟨CLOSE sll(y, s), 2⟩
{sll(y12, s1) ∗ ...} ⇝ {[y, 2] ∗ y ↦ v′ ∗ (y + 1) ↦ nxt′ ∗ sll(nxt′, s′)} ∗ ...}
⟨ALLOC ([y, 2] ∗ y ↦ v′∗ (y+1) ↦ nxt′), y2⟩
{[y2, 2] ∗ y2 ↦ - ∗ (y2 + 1) ↦ - ∗ sll(y12, s1) ∗ ...} ⇝ {[y, 2] ∗ y ↦ v′ ∗ (y + 1) ↦ nxt′ ∗ sll(nxt′, s′)} ∗ ...}
...

Figure 2.2: Singly-linked list copying in SuSLik: synthesized code (left) and simplified proof tree (right).

it suffices to informally observe the close *structural correspondence* between the SusLang program and SSL$_\circlearrowleft$ proof tree.

Following a value read from location `r`, the program diverges into two branches; this is captured by the Open rule, which unfolds the inductive predicate in the precondition because its clause selectors match the branch conditions. The "if" case of the program has no program statements, so the corresponding left branch of the proof tree has no operational steps; the branch is discharged by applying one Close rule each for the two inductive predicates in the postcondition and framing out `r`'s heaplet with Frame, which enables the Emp rule. The program's "else" case, corresponding to the right branch of the proof tree, issues a series of operational steps before applying similar reduction by framing to the goal's assertions to enable the Emp rule (the latter steps are not shown in Fig. 2.2). Midway through this proof branch, we can see Sec. 2.1.2's Call rule semantics in action. In the goal state immediately before the rule is applied, the symbolic heap $\mathtt{r} \mapsto \mathtt{nxt} * \mathsf{sll}(\mathtt{nxt}, s_1)$ (the grayed parts of the precondition) indicate heaplets that remain in focus after framing because they are used in the procedure call. Comparing that to the next goal state after applying Call, we observe the appearance of a new singly linked list rooted at a fresh ghost variable y', and a change of `r`'s referent from `nxt` to y'.

```
(* Inductive heap predicate for SL lists *)
Inductive sll (x : ptr) s h : Prop :=
| sll_1 of x == null of s = [::] ∧ h = Unit
| sll_2 of (x == null) = false of
  ∃ (v : nat) s1 nxt h1, s = v :: s1 ∧
  h = x ↦ v • x+1 ↦ nxt • h1 ∧ sll nxt s1 h1.

(* Specification for SLL copying *)
Definition sll_copy_spec :=
  ∀ (r: ptr), {(vghosts : ptr * seq nat)},
  STsep(
    (* Precondition *)
    fun h => let: (x, s) := vghosts in
    ∃ h1, h = r ↦ x • h1 ∧ sll x s h1,
  [ (* Postcondition *)
    vfun (_: unit) h =>
    let: (x, s) := vghosts in
      ∃ y h1 h2, h = r ↦ y • h1 • h2 ∧
              sll x s h1 ∧ sll y s h2]).

(* SLL copying implementation *)
Program Definition sll_copy : sll_copy_spec :=
  Fix (fun (sll_copy : sll_copy_spec) r => Do (
    x2 ← @read ptr r;
    if x2 == null
    then ret tt (* return unit *)
    else
      v ← @read nat x2;
      nxt ← @read ptr (x2+1);
      r ::= nxt;;
      sll_copy r;;
      y12 ← @read ptr r;
      y2 ← allocb null 2;
      r ::= y2;;
      (y2+1) ::= y12;;
      y2 ::= v;;
      ret tt)).
```

```
Next Obligation.
(* Initialize HTT proof context *)
  apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
  case: Hsll; case: ifP; move=>IfCond//_;
  [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
  (* Case: empty list (x2 = 0) *)
  - move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
    (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
    (* Close (unfold) SLL instance in postcondition *)
      repeat split=>//=; do?[hhauto|constructor 1].
  (* Case: non-empty list *)
  - (* Read *) apply: bnd_readR=>//=.
    (* Read *) apply: bnd_readR=>//=.
    (* Write *) apply: bnd_writeR=>//=.
    (* Call *)
      rewrite (joinC _ h1) joinA; apply: bnd_seq.
      apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
      ∃ h1; split=>//=.
      move=>h_call [y12][h11][h21][->][H2 H3]_.
    (* Read *) apply: bnd_readR=>//=.
    (* Alloc *) apply: bnd_allocbR=>y2//=.
    (* Write *) apply: bnd_writeR=>//=.
    (* Write *) apply: bnd_writeR=>//=.
    (* Write *) apply: bnd_writeR=>//=.
    (* Emp *)
      apply: val_ret; rewrite defPtUn0; case/andP=>?.
      ∃ y2, (x2 ↦ v • (x2+1) ↦ nxt • h11),
            (y2 ↦ v • (y2+1) ↦ y12 • h21).
      repeat split=>//=; first by hhauto.
     + (* Close SLL instance 1 in postcondition *)
       by constructor 2=>//=; ∃ v, s1, nxt, h11.
     + (* Close SLL instance 2 in postcondition *)
       constructor 2=>//=; first by apply negbTE.
       by ∃ v, s1, y12, h21.
Qed.
```

Figure 2.3: Copying a singly-linked list in HTT/Coq: definitions and specification (left), and proof (right).

2.2 Verification With Hoare Type Theory

Whereas synthesis is the derivation of a program that meets a user specification, *verification* is about checking that an already available program does indeed fulfill some specification.

To appreciate how these related notions differ, we now show how a program like `sll_copy` can be verified in a program verification framework.

2.2.1 Framework Overview

Hoare Type Theory (HTT) [NVB10] is a foundational framework for verifying correctness in the proof assistant Coq. It implements a SL-based program logic for an idealized C-style imperative programming language.

Fig. 2.3 shows the result of verifying the `sll_copy` program in HTT. It offers some important insights into the way HTT proofs reason about program state, and how that differs from SSL$_\circlearrowleft$ reasoning. The left hand side of the figure includes the HTT equivalents of the sll predicate (2.1), the specification of `sll_copy` (1.1), and the corresponding implementation as synthesized by SuSLik (Fig. 1.1), all expressed in Coq's specification language Gallina.

One immediate observation is that HTT predicates and specifications mention heaps explicitly. Specifically, the spatial assertions are expressed as *algebraic equalities*. The idea is that the framework encodes SL's separating conjunction ($*$) as a disjoint union ($\bullet$) of explicit symbolic heaps. In turn, each symbolic heap can be constrained by inductive predicates, which appear as conjunctions with the proposition asserting heap equality. For instance, the inductive predicate `sll nxt s1 h1` (line 6 of Fig. 2.3) constrains the heap `h1`, which is a subheap of the larger heap `h` constrained by the main predicate definition. In Ch. 4, we revisit these disparities between $\mathrm{SSL}_{\circlearrowleft}$ and HTT to discuss how our technique allows us to overcome them in a principled way. For now, let us focus on the proof.

The right side of Fig. 2.3 shows a handcrafted proof that shows the correctness of the `sll_copy` program implementation *wrt.* its specification `sll_copy_spec`. The proof methodology employs forward-style symbolic execution to transform the precondition incrementally until it matches the postcondition, where the proof direction is largely guided by the structure of the program. Some bookkeeping steps are interspersed between these operational steps; they label and retrieve any assumptions about the shape of the heap, pure constraints, or relations between symbolic values in Coq's native proof context. A walkthrough of Fig. 2.3's proof touches on each of these points.

2.2.2 Verification of `sll_copy`

The proof begins by initializing the Coq context; the program's ghost variables (logical variables that appear in the precondition) are given names after applying lemma `ghR` to instantiate them. The assumption representing the inductive predicate in the precondition is also given a name, `Hsll`. Note that this assumption name is local to Coq's context, and has no formal equivalent in SuSLik; Ch. 3 discusses this type of local context tracking.

The first line of the program (line 24 in Fig. 2.3) reads from a pointer `r`, so our corresponding first proof step after initialization (line 41) applies the `bnd_readR` lemma. The next line in the program (line 25) is a conditional statement, using the two selectors of the sll predicate as the branching conditions. The proof therefore unfolds the predicate instance in the precondition (line 43), by case analysis on the assertion we named `Hsll` earlier. In both branches, the selector is stored with the name `IfCond`, and the unfolded predicate clause's quantified variables are introduced and added to the Coq context. The proof steps for storing values and allocating memory proceed similarly to value reads—HTT has dedicated lemmas for them. The proof handles the recursive procedure call (line 31) by first reordering the precondition heaplets so that the ones pertaining to the footprint of the procedure call all appear on the left hand side of the rest (line 55). We have seen similar reasoning in the way the Call rule handles procedure calls by framing Sec. 2.1.2; the difference is the need for manual heaplet reordering in HTT. Next, `gh_ex` provides witnesses for the call's precondition ghosts and `val_do` symbolically executes the call (line 56); since the call's precondition heap contains an inductive predicate, we provide a corresponding heap witness $h1$, which we previously obtained by unfolding a predicate in line 44. Each proof branch ends by reconciling the postcondition with the precondition state after symbolically executing all statements in the corresponding program branch. This reconciliation turns out to be a characteristic step that verification frameworks expect but synthesis doesn't provide. For now, it suffices to observe that the Close rules that appeared in the middle of a proof branch in the $\mathrm{SSL}_{\circlearrowleft}$ proof tree (Fig. 2.2) has its corresponding steps in HTT appear at the end of the proof branch (lines 49, 70, 72). We defer more detailed discussion to Ch. 4, where we relate it to the proof evaluator design.

2.3 Synthesis and Verification: A Divide

We have taken a cursory look at the proof of `sll_copy`'s correctness, in terms of both $\mathrm{SSL}_{\circlearrowleft}$ (Sec. 2.1.3) and HTT's program logic (Sec. 2.2.2). While there is certainly a loose similarity between the general proof strategies, we also observed some notable differences, such as the need to track the names of symbolic values in the Coq context, and the out-of-order appearance of the Close rule. In the next chapter, we introduce a framework for *proof evaluation* that handles these discrepancies in a generic way.

3 A Framework For Evaluating Proofs

Like a bridge over troubled water, I will ease your mind.

—Simon and Garfunkel

This chapter proposes a method to overcome the gap between synthesis and verification observed in Sec. 2.3, by way of a generic framework for proof evaluation. We show how to represent $\text{SSL}_{\circlearrowleft}$ derivations as proof trees (Sec. 3.1), and extract such an encoding from SuSLik's synthesis (Sec. 3.2). Then, we describe the design for a *proof evaluator* that traverses this proof tree to generate a program certificate for any target verifier (Sec. 3.3).

3.1 Encoding Derivations As Proof Trees

We first describe the design of a *proof tree* encoding of the intermediate derivations that can be evaluated into a certificate for any target verification framework.

SuSLik synthesizes programs through a *proof search* strategy described in Sec. 2.1.2. Let us discuss how a valid $\text{SSL}_{\circlearrowleft}$ derivation can be encoded, once it is built by the synthesis algorithm. In Fig. 2.1, the grayed fragments of each rule's conclusion denote the parts of a synthesis goal (or other information, such as a predicate clause index in the rule premise for Close) that determine *how* a rule is applied to reduce it to concrete subgoals—in other words, they represent the parameters of each rule application instance. This means that by knowing a tree of the $\text{SSL}_{\circlearrowleft}$ rules applied to each synthesis goal starting from the initial one, along with the grayed fragments that instantiate each rule application, we can fully restore the derivation steps taken by the synthesizer, as well as the generated program. We encode SSL$_\circlearrowleft$ proof trees as values of the following recursive data type $\tau_{\textbf{ssl}}$:

$$
\begin{array}{lll}
\textsf{ProofTree}\ (\textsf{Step}_{\textbf{ssl}}) & \tau_{\textbf{ssl}} ::= \langle \mathcal{S}_{\textbf{ssl}}, \overline{\tau_{\textbf{ssl}}} \rangle & \\
\textsf{Step}_{\textbf{ssl}} & \mathcal{S}_{\textbf{ssl}} ::= \langle \textsf{READ}, x, \iota, e, \textsf{y} \rangle \mid \langle \textsf{CALL}, P, \sigma, f \rangle \mid \left\langle \textsf{OPEN}, \textsf{p}^{\alpha}(\overline{t_i}) \right\rangle \mid \left\langle \textsf{CLOSE}, \textsf{p}^{\alpha}(\overline{t_i}), j \right\rangle \mid \ldots &
\end{array}
\tag{3.1}
$$

A proof tree node $\tau_{\textbf{ssl}}$ is a pair consisting of a proof step $\mathcal{S}_{\textbf{ssl}}$ and a sequence of child nodes $\overline{\tau_{\textbf{ssl}}}$, where the latter is empty for terminal rule applications. The type of the node is parameterized by its payload type; for a SSL$_\circlearrowleft$ proof tree, these payloads are the individual proof steps ($\textsf{Step}_{\textbf{ssl}}$) corresponding to rule derivations. The proof steps encode the name of the applied rule and any information from the grayed fragments in Fig. 2.1 that control rule application non-determinism.

Recall, in Sec. 2.1.3, we informally walked through the derivation of a `sll_copy` implementation shown in the right part of Fig. 2.2. We now recognize this proof tree as a $\tau_{\textbf{ssl}}$ encoding of successful derivations. The nodes contain concrete proof steps; they are interleaved with the intermediate synthesis goal states, to demonstrate how each proof step transforms one goal into the next. For example, the left branch of the tree shows the derivation of the "then" branch of `sll_copy`, which replaces an instance $\textsf{sll}(\texttt{x2}, s)$ with its first clause, thus deducing $\texttt{x2} = 0 \land s = \emptyset$ and performing the corresponding substitutions (elided from the tree for brevity). What follows in the proof tree are the unfoldings, via the Close rule, of the two predicate occurrences in the postcondition, which are replaced by the first clause from definition (2.1). The remaining derivation is completed by an application of the Frame and Emp rules.

3.2 Constructing Proof Trees

We now explain how to extract this encoding from synthesis by augmenting the original SuSLik implementation to capture run-time deductive reasoning.

3.2.1 Extraction From And/Or Trees

The existing SuSLik implementation had a singular goal of producing a SusLang program, so it did not preserve run-time information in a recoverable format;

some modifications are needed to obtain a $\tau_{\mathbf{ssl}}$ encoding. We thus facilitate proof tree extraction by accumulating the intermediate derivations during synthesis, and then applying some additional postprocessing so it is easily consumed by the evaluator.

Extraction starts by accumulating the derivations made by SuSLik. The synthesizer's non-deterministic proof search was described in Sec. 2.1.2. At any point in the search, the state of the current synthesis goal can enable the application of a set of *candidate* $SSL_{\circlearrowleft}$ *rules*. In turn, each candidate rule application can advance the synthesis by transforming the goal into some number of *subgoals*. SuSLik uses an And/Or tree [MM73] to express these alternating layers, where proof goals are represented as *Or-nodes*, and candidate rule applications as *And-nodes*. An Or-node can be viewed as a disjunction, where the node succeeds if one of its children (*i.e.*, a candidate rule application) succeeds. An And-node can likewise be treated as a conjunction, where all of its children (*i.e.*, subgoals) must succeed for the node to succeed. Whenever an And-node is generated during synthesis, we capture it along with any knowledge generated by the corresponding rule application that would be useful when reconstructing a proof later. For example, for the Read rule (Fig. 2.1) we capture the names of the operation's source and destination variables, along with the name of the ghost variable that was instantiated by the read, as in definition (3.1). By collecting these nodes, we capture all paths explored in the set of possible rule application sequences.

After accumulating information from each step of the proof search, we discard the steps that failed, preserving only those that contributed to the final synthesized program. We remove failed branches by keeping track of which terminal rule applications failed during synthesis, and then for each one, retracing and pruning ancestors in bottom-up fashion until we reach an Or-node where one of the other candidates succeeded. After this pruning, we are left with an And/Or tree where every Or-node has a single corresponding And-node (*i.e.*, every subgoal was solved by applying a single rule chosen out of all candidates).

Finally, we translate this to our source proof tree structure, collapsing these And/Or node pairs into single nodes. Since the proof tree is meant to be a self-contained encoding that includes all necessary knowledge for certification, we also prefix every proof tree with a special initialization node whose payload is the initial goal (*i.e.*, the specification) instead of a $\mathrm{SSL}_{\circlearrowleft}$ rule application.

This completes the conversion of a tree structure designed for *exploring* the derivation search space (the And/Or tree) into a *retraceable* one designed to compactly represent the successful derivations only.

3.2.2 Finalizing Branch Abductions

We require one last transformation to ensure that the branching logic of the proof tree encoding is consistent with the actual goal transformations. In order to determine when and where it is appropriate to introduce a branch, SuSLik sometimes performs *branch abduction*. This is a strategy by which the synthesizer abduces the need for the program flow to diverge on a conditional statement. When applied, the rule generates a subgoal to synthesize a new branch (based on some condition) that is rooted at the current goal, *or its earliest valid ancestor* in the And/Or tree that has all of the variables used in the condition. A branch abduced to an ancestor location means that the rule application transforms an earlier goal, and not the current one. This is a natural behavior for synthesis, because some progress in the proof search may retroactively enable new insight about an action to take at a prior step. On the other hand, verification frameworks like HTT proceed by forward-propagating the precondition's symbolic state. Thus, a suitable encoding should ensure that each node's rule application incrementally transforms the latest synthesis goal, as we now demonstrate with an example.

```
{true; r ↦ null}
void min (loc r, int x, int y) {
  if (x ≤ y) {
    *r = x;
  } else {
    *r = y;
  }
}
{m ≤ x ∧ m ≤ y; r ↦ m}
```

Figure 3.1: Spec and code of `min`.

Fig. 3.1 shows the specification and code of the `min` program, which finds the minimum of two integers. Each branch in the implementation assigns a different value to the location `r`. Fig. 3.2 shows the corresponding abbreviated proof tree. A few steps into the derivation, SuSLik first applies the Pick rule, which "picks" program variable `x` as an existential witness for postcondition existential m. In the next step B, it abduces a branch on condition `x ≤ y` at an ancestor—the initial step A. Since a

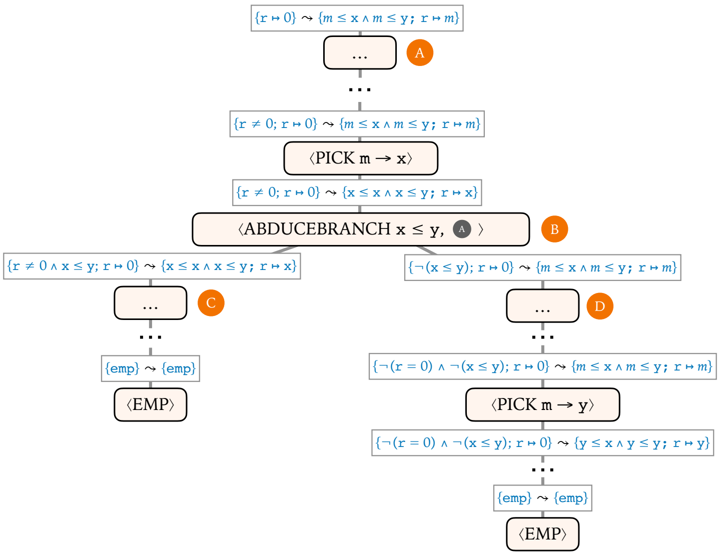
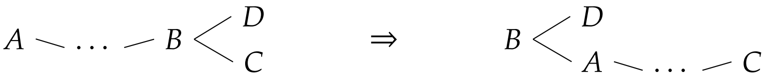

{r ↦ 0} ⇝ {m ≤ x ∧ m ≤ y; r ↦ m}
…
A
{r ≠ 0; r ↦ 0} ⇝ {m ≤ x ∧ m ≤ y; r ↦ m}
⟨PICK m → x⟩
{r ≠ 0; r ↦ 0} ⇝ {x ≤ x ∧ x ≤ y; r ↦ x}
⟨ABDUCEBRANCH x ≤ y, A ⟩
B
{r ≠ 0 ∧ x ≤ y; r ↦ 0} ⇝ {x ≤ x ∧ x ≤ y; r ↦ x}
…
C
{emp} ⇝ {emp}
⟨EMP⟩
{¬(x ≤ y); r ↦ 0} ⇝ {m ≤ x ∧ m ≤ y; r ↦ m}
…
D
{¬(r = 0) ∧ ¬(x ≤ y); r ↦ 0} ⇝ {m ≤ x ∧ m ≤ y; r ↦ m}
⟨PICK m → y⟩
{¬(r = 0) ∧ ¬(x ≤ y); r ↦ 0} ⇝ {y ≤ x ∧ y ≤ y; r ↦ y}
{emp} ⇝ {emp}
⟨EMP⟩

Figure 3.2: Simplified proof tree highlighting branch abduction for a program to find the minimum of two integers.

valid proof tree only encodes successful derivations, we know this abduction succeeded. Indeed, in the left branch (corresponding to the "if" case of the synthesized program), it adds the constraint to the precondition to the parent goal, and completes the derivation, with m instantiated with $\mathtt{x}$. However, in the right branch (corresponding to the "else" case), the negation of the constraint $\neg(\mathtt{x} \leq \mathtt{y})$ is added to the *initial goal*'s precondition, and m is subsequently instantiated with $\mathtt{y}$ by a *different* Pick rule application. As the synthesis goal before step B is not transformed by step B into the goal before step D, this proof tree does not accurately represent the actual proof a verifier would need to conduct; for example, the Pick rule should be applied with different existential witnesses for each branch.

We can resolve this discrepancy by rearranging the proof tree nodes that perform branch abduction to their correct location:



Here, the branch prefixed by step D is abduced at step B. The node for step B can be transplanted to its correct location before A, so that the proof tree encoding captures synthesis goal transformations that are consistent *wrt.* the SSL$_{\circlearrowleft}$ rule semantics.

The evaluator introduced in Sec. 3.3 uses this node as the bootstrapping point for the rest of the evaluation.

The resulting tree is a compact representation of the steps taken to derive the program, with each node storing an instance of the data types (3.1) that contain proof information.

3.2.3 Program Recovery From A Proof Tree

We now take a brief detour to reinforce the idea that this proof tree encoding captures the deductive steps taken during synthesis, before embarking on our main journey to generate proof certificates in the next section. Since a SusLang program is a *byproduct* of a deductive synthesis (Sec. 2.1) whose derivations are encoded by the proof tree, it is possible to *recover* the program after synthesis by simply traversing the extracted proof tree.

Let us define the following proof tree *evaluator*, which parses a source proof tree to recover a program:

$$\begin{aligned}
&\mathcal{E}_{\mathbf{synt}} : \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{ssl}}) \rightarrow \mathsf{Prog} \\
&\mathcal{E}_{\mathbf{synt}}\ (\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle) \ \triangleq\ \mathbf{let}\ k\ = \mathcal{I}_{\mathbf{synt}}\ \mathcal{S}_{\mathbf{ssl}} \quad \mathbf{in} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \overline{c}\ = \mathrm{map}\ \mathcal{E}_{\mathbf{synt}}\ (\overline{\tau_{\mathbf{ssl}}})\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\quad k\ \overline{c} \\
&\text{where} \\
&\mathcal{I}_{\mathbf{synt}} : \mathsf{Step}_{\mathbf{ssl}} \rightarrow (\mathsf{Prog}^{*} \rightarrow \mathsf{Prog})
\end{aligned} \tag{3.2}$$

In the definition above, the source proof tree evaluator takes a tree node and applies a *proof step interpreter* $\mathcal{I}_{\mathbf{synt}}$ to the proof step in the node. As described by its type, the interpreter returns a program-constructing continuation function k, whose arity is the length of $\overline{\tau_{\mathbf{ssl}}}$. The evaluator then proceeds to generate the residual programs by processing the child source nodes $\overline{\tau_{\mathbf{ssl}}}$. Finally, it applies k to the resulting residual programs $\overline{c}$, obtaining the result.

As an example, we provide a partial implementation of this interpreter for the proof steps corresponding to applications of Emp and Read:

$$\begin{aligned}
\mathcal{I}_{\mathbf{synt}}\ \langle \textsf{EMP} \rangle &\triangleq \lambda\,[\,].\texttt{skip}\\
\mathcal{I}_{\mathbf{synt}}\ \langle \textsf{READ}, x, \iota, e, \texttt{y} \rangle &\triangleq \lambda\,[c].\texttt{let}\ \texttt{y} = *(x+\iota); c
\end{aligned}$$

That is, when applied to $\langle \textsf{EMP} \rangle$, $\mathcal{I}_{\mathbf{synt}}$ emits a 0-arity function (*i.e.*, a constant), which returns the program `skip`. For $\langle \textsf{READ}, x, \iota, e, \texttt{y} \rangle$, the step interpreter returns a function that prepends the read-operation $\texttt{let}\ \texttt{y} = *(x+\iota)$ to the residual program. Looking at the remaining rules in Fig. 2.1, it is easy to see that the arity of the function returned by $\mathcal{I}_{\mathbf{synt}}$ matches the number of subgoals in the premise of the corresponding rule. Importantly, neither $\mathcal{I}_{\mathbf{synt}}$ nor $\mathcal{E}_{\mathbf{synt}}$ need to check that their arguments are well-formed, as they are assumed to be applied only to proof trees that are valid for the corresponding goals.

3.3 An Evaluator to Translate Synthesis Proofs

Having shown an instructive evaluator for converting proof trees to SusLang programs, let us now generalize it to an *abstract proof evaluator* that evaluates proof trees into correctness proofs of some target verification framework t:

$$\begin{aligned}
&\mathcal{E}_{\mathbf{t}} : \textsf{ProofTree}\ (\textsf{Step}_{\mathbf{ssl}}) \times \textsf{Context}_{\mathbf{t}} \times \textsf{DeferredStep}_{\mathbf{t}} \to \textsf{ProofTree}\ (\textsf{Step}^*_{\mathbf{t}})\\
&\mathcal{E}_{\mathbf{t}}\ (\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle, ctx, d) \triangleq \begin{array}{lll}
\mathbf{let}\ (\overline{\mathcal{S}_{\mathbf{t}}}, \overline{ctx}, d') & = \mathcal{I}_{\mathbf{t}}\ \mathcal{S}_{\mathbf{ssl}}\ ctx & \mathbf{in}\\
\mathbf{let}\ d'' & = \lambda ctx.\,((d\ ctx) \mathbin{+\!\!+} (d'\ ctx)) & \mathbf{in}\\
\mathbf{let}\ \overline{\tau_{\mathbf{t}}} & = \text{map}\ \mathcal{E}_{\mathbf{t}}\ (\text{zip3}\ \overline{\tau_{\mathbf{ssl}}}\ \overline{ctx}\ (\text{repeat}\ |\overline{\tau_{\mathbf{ssl}}}|\ d'')) & \mathbf{in}\\
\multicolumn{3}{l}{\mathbf{if}\ |\overline{\tau_{\mathbf{t}}}| = 0\ \mathbf{then}\ \left\langle \overline{\mathcal{S}_{\mathbf{t}}} \mathbin{+\!\!+} (d''\ ctx), [] \right\rangle\ \mathbf{else}\ \left\langle \overline{\mathcal{S}_{\mathbf{t}}}, \overline{\tau_{\mathbf{t}}} \right\rangle}
\end{array}\\
&\text{where}\\
&\mathcal{I}_{\mathbf{t}} \quad : \quad \textsf{Step}_{\mathbf{ssl}} \to \textsf{Context}_{\mathbf{t}} \to \textsf{Step}^*_{\mathbf{t}} \times \textsf{Context}^*_{\mathbf{t}} \times \textsf{DeferredStep}_{\mathbf{t}}\\
&\textsf{DeferredStep}_{\mathbf{t}} \quad \triangleq \quad \textsf{Context}_{\mathbf{t}} \to \textsf{Step}^*_{\mathbf{t}}
\end{aligned} \tag{3.3}$$

We intend this abstract mechanism to be instantiated by an engineer interested in supporting a particular verification framework. The design of the evaluator is guided by several usability goals:

1. It has a generic infrastructure that supports correctness proof generation for any verification framework.

2. It requires minimal effort to add support for each verification framework.

3. The interface provides the ideal amount of expressivity for an engineer to successfully add the necessary support for a framework and resolve any discrepancies between SSL$_{\circlearrowleft}$ and the framework's program logic.

The following sections explain each component of Eq. 3.3. In particular, we describe a pluggable interface for *proof step interpretation* (Sec. 3.3.1); a *context* for forward-propagating verifier-local information (Sec. 3.3.2); and *deferred steps* to delay the evaluation of certain proof steps (Sec. 3.3.3). Finally, we summarize how the components fit together to meet the usability goals in Sec. 3.4.

3.3.1 Modular Proof Step Interpreters

At a high level, Eq. 3.3 expresses a tree traversal, where evaluator $\mathcal{E}_{\mathbf{t}}$ maps a source proof tree $\tau_{\mathbf{ssl}} ::= \langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle$ (3.1) to a target proof tree $\tau_{\mathbf{t}} ::= \left\langle \overline{\mathcal{S}_{\mathbf{t}}}, \overline{\tau_{\mathbf{t}}} \right\rangle$. Crucially, the node-wise mapping from source step to target step—a verifier-specific operation—is delegated to a *proof step interpreter* $\mathcal{I}_{\mathbf{t}}$. By applying interpreter $\mathcal{I}_{\mathbf{t}}$ to $\mathcal{S}_{\mathbf{ssl}}$, the evaluator obtains a target proof step sequence $\overline{\mathcal{S}_{\mathbf{t}}}$, which it can then use to construct its return value. Finally, the evaluator recursively applies itself to each child source node $\overline{\tau_{\mathbf{ssl}}}$, obtaining the corresponding child target nodes $\overline{\tau_{\mathbf{t}}}$ (the lengths of the two sequences are assumed to be the same, which is the case for valid proof trees).

This separates concerns between generic traversal logic and verifier-specific logic. By defining the interpreter as a *modular* interface, support can be added for each verifier by defining a new implementation. Note how this is a generalization of the interpreter $\mathcal{I}_{\mathbf{synt}}$ that appeared in Sec. 3.2.3

Let us examine the interpreter in greater detail. Definition (3.3) describes the type of $\mathcal{I}_{\mathbf{t}}$ as follows:

$$\mathcal{I}_{\mathbf{t}} \;:\; \mathsf{Step}_{\mathbf{ssl}} \to \mathsf{Context}_{\mathbf{t}} \to \mathsf{Step}_{\mathbf{t}}^{*} \times \mathsf{Context}_{\mathbf{t}}^{*} \times \mathsf{DeferredStep}_{\mathbf{t}} \qquad (3.4)$$

For now we disregard $\mathsf{Context}_{\mathbf{t}}$ and $\mathsf{DeferredStep}_{\mathbf{t}}$, both auxiliary information that enhances the interpreter's capabilities, as we elaborate on each of them in

Sec. 3.3.2 and Sec. 3.3.3 respectively. Then, at its core, the interpreter maps a $\mathsf{SSL}_{\circlearrowleft}$ proof step $\mathsf{Step}_{\mathsf{ssl}}$ to an equivalent proof step sequence $\mathsf{Step}_{\mathsf{t}}^{*}$ for verifier t.

Without an evaluator framework, adding support for each new verifier would result in a lot of code duplication. In contrast, proof evaluation with a pluggable interpreter makes for an easily extensible design. By only requiring the engineer in charge of supporting the verifier to write a new interpreter instance, we reduce their scope of implementation, so that it suffices to reason *locally* about each source step in isolation.

For example, consider the first READ rule application in the proof tree on the left side of Fig. 2.2, and compare it to the corresponding HTT proof step in line 41 of Fig. 2.3. The encoded derivation:

$$\langle \mathsf{READ}, r, 0, x, \mathtt{x2} \rangle$$

has sufficient information to generate the target step:

```
apply: bnd_readR=>/=.
```

Thus, the interpreter simply needs to include this one-to-one mapping to support READ rule translation:

$$\mathcal{I}_{\mathsf{htt}}\, \langle \mathsf{READ}, x, \iota, e, \mathsf{y} \rangle \triangleq [\mathtt{apply{:}\ bnd\_readR{=}{>}/{=}.}]$$

But this approach does not work in every case, as hinted in Sec. 2.3. What if the interpretation of some source step needs additional insight on earlier or later $\mathsf{SSL}_{\circlearrowleft}$ rule applications to generate the correct target proof steps? In the following sections, we discuss enhancements to the basic design we covered, that equip the interpreter with two *standardized* ways of employing *non-local* reasoning.

3.3.2 Contexts For Tracking Verifier State

One way the interpreter might reason non-locally is if it needs to access some knowledge encountered or generated earlier in the evaluator's traversal of the source proof tree. Then, it is not enough for the interpreter to only be aware of the current source step; we must introduce a mechanism to "remember" information for later access. To accommodate the need to keep track of definitions and dependencies between components in the source and target proofs, which are paramount in our case studies, our proof evaluator (3.3) tracks a verifier-specific *proof context*.

Consider the Open rule, which unfolds a predicate occurrence in the precondition. From the `sll_copy` proof tree in Fig. 2.2, we know that the rule is applied with the predicate $\mathsf{sll}(x2, x)$. In Sec. 2.2.2, we observed that the corresponding HTT proof step (line 43 of Fig. 2.3) performs case analysis on that predicate, by referring to `Hsll`, the named hypothesis in the Coq context asserting that this predicate constrains some symbolic heap. The hypothesis was introduced with this name at the beginning of the proof (in line 40), along with the precondition ghost variables.

The evaluator's proof context can store such information. In definition (3.3), observe how the evaluator is parameterized by the proof context *ctx*. The interpreter $\mathcal{I}_{\mathsf{t}}$ consumes *ctx* and returns a sequence of updated child contexts $\overline{ctx}$ (whose length matches the number of child nodes $\overline{\tau_{\mathsf{ssl}}}$). These contexts are threaded through to each recursive application of the evaluator to the child nodes. In functional programming terms, this proof context is effectively an *accumulator* that is continually updated throughout the traversal's lifespan.

3.3.3 Deferring Target Proof Steps

The interpreter may also be applied to a $\mathrm{SSL}_{\circlearrowleft}$ rule, but then decide that the corresponding target proof step should appear at a later point in the evaluation. We observed this kind of non-local reasoning in lines 66-67 of Fig. 2.3.

As HTT implements forward symbolic execution, the goal of these steps is to provide existential witnesses for the head pointer y of the newly created list from the postcondition of the spec (1.1), as well as two heaps representing the linked lists: the original one $\mathsf{sll}(\mathtt{x2}, s)$ and the new one $\mathsf{sll}(y, s)$, thus proving the postcondition by means of symbolic heap entailment. Constructing this step in the target HTT proof from the source proof turns out to be challenging. The reason, as the proof tree in Fig. 2.2 shows, is that the information about the shape of the heap constrained by, *e.g.*, $\mathsf{sll}(y, s)$ in the postcondition, is obtained *much earlier* in the source proof, by an application of the Close rule, preceding the synthesis of the `malloc` statement. In contrast, in the target proof, a step that exploits this information (by instantiating the existential) takes place near the end of the proof branch, when proving the entailment.

This scenario is complicated even more by the fact that, by the time we need to provide the witness heap revealed by the Close rule from the source proof, the logical (ghost) variable y has been replaced, in all assertions involving it, by a program-level variable `y2`, storing the head pointer of the newly allocated list. This is why it is insufficient to simply defer some target proof steps until the later entailment checking stage: it also should be possible to adapt them to all changes in the proof context (*e.g.*, variable substitutions) that can take place after their emission but before their application in the target proof.

The discrepancy arises from the difference in the proof strategy taken by deductive synthesis and verification. Each $\mathrm{SSL}_{\circlearrowleft}$ rule derivation can transform both the pre- and postcondition of the synthesis goal (Sec. 2.1.2), whereas verifiers like HTT symbolically execute steps to transform the precondition, and only show its entailment to the postcondition at the end of a proof branch (Sec. 2.2.1). This means that for any $\mathrm{SSL}_{\circlearrowleft}$ rule that transforms the postcondition, the verifier needs to *delay* the execution of its equivalent target step.

We equip our evaluator with this functionality in the form of *deferred proof steps*, represented as the evaluator's third and final parameter d. As the type $\mathsf{DeferredStep}_{\mathsf{t}}$ of this component shows, deferred steps are encoded as functions from proof contexts to sequences of target proof steps. In a normal execution, as long as the source proof tree node still has children, the evaluator simply accumulates the deferred steps by composing already accumulated ones with those emitted by the interpretation (via $\mathcal{I}_{\mathsf{t}}$) of the node's payload in a way resembling *continuation-passing style*. Those accumulated deferred steps are all released once the proof tree branch reaches its end (detected as $|\overline{\tau_{\mathsf{t}}}| = 0$), which corresponds to the entailment checking stage in the forward execution proofs. Thanks to their type, deferred steps can access the most up-to-date proof context at the moment of their application, thus, circumventing the variable/hypothesis issue outlined above. Indeed, in order to bootstrap the source tree evaluation, the implementer needs to provide the initial deferred steps, which are most commonly just a trivial function $\lambda\_.[\,]$.

In Sec. 4.3.3, we elaborate on the instrumentation of the full postcondition entailment-checking procedure for HTT that utilizes the proof context and deferred steps.

3.3.4 Putting It All Together

This completes our exposition of the full evaluator definition (3.3). The evaluator $\mathcal{E}_{\mathbf{t}}$ takes three parameters: a source proof tree node $\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle$; a proof context *ctx*; and the accumulated deferred steps d. It applies the interpreter $\mathcal{I}_{\mathbf{t}}$ to the source node's payload step $\mathcal{S}_{\mathbf{ssl}}$ and the proof context, to obtain three results: the corresponding target step sequence $\overline{\mathcal{S}_{\mathbf{t}}}$, the transformed contexts $\overline{ctx}$ to evaluate with each child node (so that each child node can be evaluated with a different context), and any newly emitted deferred steps d', which are then composed with the existing deferred steps to form d''. The evaluator then recursively applies itself to each of the child source nodes $\overline{\tau_{\mathbf{ssl}}}$, with its corresponding transformed proof context in $\overline{ctx}$ and the updated deferred step sequence d''; this returns the sequence of target proof nodes $\overline{\tau_{\mathbf{t}}}$ corresponding to each child. Finally, if the evaluator has reached the end of a proof branch, then all accumulated deferred steps up to this point are released with the current proof context *ctx*, and its results are concatenated with the target step sequence $\overline{\mathcal{S}_{\mathbf{t}}}$ returned by the interpreter; a terminal target proof node is returned with these steps as the payload and with no children. Otherwise, the evaluator returns a target node with payload $\overline{\mathcal{S}_{\mathbf{t}}}$ and children $\overline{\tau_{\mathbf{t}}}$.

We now revisit the three design goals proposed at the beginning of this section, making references to Fig. 3.3, which shows an implementation of this evaluator in Scala.

1. There is a clear delineation between the generic source proof tree traversal and the verifier-specific components, with the latter handled by the proof step interpreter and proof context. Both are thus encoded as interfaces, or as Scala `trait`s, in Fig. 3.3; these can be extended to support each verifier.

```
// Proof trees
case class ProofTree[S](step: S, children: List[ProofTree[S]])
type Target
// Target proof context
trait Context[T <: Target]
// Deferred target proof step
type Deferred[T <: Target, C <: Context[T]] = C ⇒ List[T]
// Source step interpreter
trait Interpreter[T <: Target, C <: Context[T]] {
  def apply(value: SSLStep, ctx: C): (List[T], List[C], Deferred[T,C])
}
// Source proof tree evaluator (provided)
class Evaluator[T <: Target, C <: Context[T]] {
  val interpret: Interpreter[T,C]
  def compose(d1: Deferred[T,C], d2: Deferred[T,C]): Deferred[T,C]
  def apply(node: ProofTree[SSLStep], ctx: C, deferred: Deferred[T,C]): ProofTree[T]
}
```

Figure 3.3: Main components of the proof evaluator encoded in Scala.

2. The interpreter has a narrow job scope—node-wise translation of source to target proof steps. The simplicity, reflected in the interpreter's type signature, eases the burden of adding support for a new verifier.

3. The proof context allows the interpreter to store information generated during one application and then retrieving it in a later application. Deferred steps allow it to delay the appearance of a SSL$_{\circlearrowleft}$ rule to a later point in the target proof. Both equip the interpreter with tools to reason non-locally in a principled way.

3.4 Extension: Controlling the Release of Deferreds

To simplify the exposition, Sec. 3.3's design of the evaluator releases deferred steps at the terminal node of each proof branch, which captures the way verifiers delay reasoning about the postcondition until the end. However, more fine-grained control over the timing of release is possible by accumulating and releasing deferred steps in stackable, isolated environments.

Such a feature is useful in relation to how SuSLik synthesizes procedure calls via the Call rule (Sec. 2.1.2). Actual insight on the possibility of a procedure call is obtained by *abduction*, as expressed in the SSL$_{\circlearrowleft}$ rule CallSetup:

$$\textsc{CallSetup}\quad \frac{\Gamma;\ \{\phi; P\} \rightsquigarrow \{\psi; S\} \,|\, c_1 \qquad \Gamma \cup \mathsf{BV}(c_1);\ \{\psi; S * R\} \rightsquigarrow Q \,|\, f(\overline{e_i}); c_2}{\Gamma;\ \{\phi; P * R\} \rightsquigarrow Q \,|\, c_1; f(\overline{e_i}); c_2} \tag{3.5}$$

When applied to a procedure specification and synthesis goal, the rule compares the procedure's precondition to the goal to see if a call is applicable. If some preparatory steps c_1 are needed to bring the goal precondition to a heap S that can be unified with the call's precondition, the rule abduces that fact.[1] The rule then replaces the goal postcondition with S (remembering the original assertion so it can be restored later when the CALL rule executes the call), and tries to act on that abduction by synthesizing the steps needed to reach S. Since our proof tree encoding only stores successful derivations, any invocations of CALLSETUP are guaranteed to be accompanied by a subsequent application of CALL, possibly with some other preparatory steps in between.

This call-producing derivation sequence, bookended by the two procedure call rules, represents a *distinct synthesis environment* from the original one, as the entire postcondition of the goal is swapped out. Consequently, any postcondition-transforming $\mathrm{SSL}_{\circlearrowleft}$ rule applications (like FRAME in Fig. 2.1) invoked within the abduction refer to a different postcondition from those outside of the abduction. However, the interpreter is not aware of this, because for the most part, we restrict it to local reasoning about individual $\mathrm{SSL}_{\circlearrowleft}$ rules in isolation (aside from the exceptions discussed in Sec. 3.3.2 and Sec. 3.3.3).

One could devise an ad-hoc way to detect when the proof enters/exits call abduction by tracking the information in the verifier-specific proof context. But this is arguably a generic maneuver that is best handled at the evaluator-level, which we can do by extending our evaluator (3.3) to represent the layers of evaluation environments as a stack.

$$
\begin{array}{lll}
\mathcal{E}_{\mathsf{t}} : \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{ssl}}) \times \mathsf{Context}_{\mathsf{t}} \times \mathsf{DeferredStack}_{\mathsf{t}} \rightarrow \mathsf{ProofTree}\ (\mathsf{Step}^{*}_{\mathsf{t}}) & & \\
\mathcal{E}_{\mathsf{t}}\ (\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle, ctx, \mathcal{D}) \triangleq & \mathbf{let}\ (\overline{\mathcal{S}_{\mathsf{t}}}, \overline{ctx}, d) = \mathcal{I}_{\mathsf{t}}\ \mathcal{S}_{\mathbf{ssl}}\ ctx & \mathbf{in} \\
& \mathbf{let}\ \mathcal{D}' = \begin{cases} \lambda ctx.\ (d\ ctx) :: \mathcal{D} & \mathbf{if}\ \mathcal{A}(\mathcal{S}_{\mathbf{ssl}}) = push \\ \lambda ctx.\ (\mathrm{tl}\ \mathcal{D}) & \mathbf{if}\ \mathcal{A}(\mathcal{S}_{\mathbf{ssl}}) = pop \\ \lambda ctx.\ (((\mathrm{hd}\ \mathcal{D})\ ctx) +\!\!+ (d\ ctx)) :: (\mathrm{tl}\ \mathcal{D}) & \mathbf{if}\ \mathcal{A}(\mathcal{S}_{\mathbf{ssl}}) = noop \end{cases} & \mathbf{in} \\
& \mathbf{let}\ \overline{\tau_{\mathsf{t}}} = \mathrm{map}\ \mathcal{E}_{\mathsf{t}}\ (\mathrm{zip3}\ \overline{\tau_{\mathbf{ssl}}}\ \overline{ctx}\ (\mathrm{repeat}\ |\overline{\tau_{\mathbf{ssl}}}|\ \mathcal{D}')) & \mathbf{in} \\
& \mathbf{if}\ \mathcal{A}(\mathcal{S}_{\mathbf{ssl}}) = pop\ \mathbf{then}\ \left\langle \overline{\mathcal{S}_{\mathsf{t}}} +\!\!+ ((\mathrm{hd}\ \mathcal{D})\ ctx), \overline{\tau_{\mathsf{t}}} \right\rangle\ \mathbf{else}\ \left\langle \overline{\mathcal{S}_{\mathsf{t}}}, \overline{\tau_{\mathsf{t}}} \right\rangle & \\
\text{where} & & \\
\mathcal{I}_{\mathsf{t}} & : \mathsf{Step}_{\mathbf{ssl}} \rightarrow \mathsf{Context}_{\mathsf{t}} \rightarrow \mathsf{Step}^{*}_{\mathsf{t}} \times \mathsf{Context}^{*}_{\mathsf{t}} \times \mathsf{DeferredStep}_{\mathsf{t}} & \\
\mathcal{A} & \triangleq \mathsf{Step}_{\mathbf{ssl}} \rightarrow push \mid pop \mid noop & \\
\mathsf{DeferredStep}_{\mathsf{t}} & \triangleq \mathsf{Context}_{\mathsf{t}} \rightarrow \mathsf{Step}^{*}_{\mathsf{t}} &
\end{array}
\tag{3.6}
$$

$\mathcal{E}_{\mathsf{t}}$'s third parameter is now a stack of accumulated deferred steps $\mathcal{D}$, instead

[1] The synthesizer invokes a *call abduction oracle* to implement the necessary goal decomposition for abducing calls efficiently [Itz+21].

of just one; the stack is updated to $\mathcal{D}'$ according to the appropriate stack action corresponding to the current source step, provided by a classifying function $\mathcal{A}$. With this approach, accumulation of deferred steps is local to the current evaluation environment (*i.e.*, top layer of the stack). $\mathcal{A}$ can return three action types—*push* to push a new layer onto the stack (*i.e.*, start a new evaluation environment); *pop* to release all deferred steps accumulated in the current layer and pop it from the stack; and *noop* to continue accumulating in the current layer. The initialization (Sec. 3.2.1) and CallSetup rule steps emit the *push* action; the Emp and Call rule steps emit the *pop* action. Like the earlier design, the implementer must provide an initial deferred stack consisting of a single layer to bootstrap the evaluation; we also assume the well-formedness of the stack because only valid proof trees are meant to be evaluated. An application of this feature to verification in HTT is discussed in Sec. 4.3.3.

4 The Evaluator in Action: HTT

Earlier, we showed how program correctness proofs proceed in Hoare Type Theory (HTT) [NVB10], In this chapter, we apply the proof evaluation framework outlined in Ch. 3 to automate the certification of SuSLik programs using HTT as the target verifier. We describe the implementation of all components of the certification: programs (Sec. 4.1), predicates (Sec. 4.2), and proofs (Sec. 4.3). We also show how we extract and automate the proofs of pure entailments in the postcondition with the aid of certified solvers (Sec. 4.4).

4.1 Translating Programs

HTT certificates express program implementations in terms of its own idealized C-like language. Since all SusLang statements have a counterpart in this language, our first instinct was to obtain a program in HTT's language by directly translating from the synthesized SusLang output. However, we found it more reliable to reuse the evaluator infrastructure we originally developed for proof generation, repurposing it for program generation.

This stems from the fact that in deductive synthesis, the resulting program implementation is a *byproduct* of the derivation process (Sec. 2.1); the proof tree is the real source of truth. In fact, we previously demonstrated a procedure to recover a program from a proof tree encoding Sec. 3.2.3.

Consider the Free rule, which deallocates a whole memory block in the precondition. During synthesis, applying this rule to a location x that references a memory block generates a single SusLang statement `free(x)`, which captures

```
// Source proof representation
case class FreeStep(ptr: Var, block: Block) extends S
// Suslang representation
case class Free(ptr: Var) extends SusLang
def proofToSuslang(step: FreeStep) = Free(step.ptr)
// HTT representation
case class Dealloc(ptr: HTTVar, offset: Int) extends HTT
def proofToHTT(step: FreeStep) =
  for (i ← 0 until block.size) yield Dealloc(ptr, i)
```

Figure 4.1: Two alternative translations of the FREE rule.

the semantics of this deallocation. The equivalent statement in the language of HTT, however, slightly differs; unlike its SUSLANG counterpart, a single statement only deallocates one memory cell. That means that in order to deallocate an entire block of size n, the program must contain n instances of the statement. This information about block size, while clearly used at the time of rule application (and captured in the proof tree node in Fig. 4.1), is discarded when encoded as a SUSLANG statement. The example demonstrates why it is better to use the proof tree as the source of truth for *both program and proof*, rather than relying on a specialized byproduct of synthesis for the former. That way, we ensure that both are structurally aligned.

4.2 Translating Predicates

Fortunately, SSL$_{\circlearrowleft}$ predicates are structurally similar to HTT predicates in that they are both *relationally* defined, so the translation goes smoothly. The insight is that HTT encodes program-logic-level assertions as native Gallina propositions of type **Prop**. As a result, we can represent SSL$_{\circlearrowleft}$ predicates in HTT as *inductive propositions*, as shown in lines 2–6 of Fig. 2.3. Once defined, those propositions can be directly injected into the spatial assertions of a specification, such that when it comes time to reason about the specification in a proof, the effort is vastly simplified, because we can employ all of Coq's native tactics for reasoning about propositions. For instance, recall, from Sec. 2.2.2, how unfolding a predicate occurrence in the precondition (the OPEN rule in SSL$_{\circlearrowleft}$) can proceed by *case analysis* on an instance of the *inductive proposition* `sll x2 s h'` in Coq's native proof context.

HTT's *shallow embedding* underlies this insight. That is, the specifications and programs are encoded directly in terms of types and programs of the host language (*i.e.*, Coq's Gallina). In contrast, *deeply embedded* frameworks define their own languages and logics as a separate layer on top of the host language. Sec. 7.1 touches on work done by collaborators to adapt the certification procedure to support such frameworks, which face additional challenges in automating proofs due to the added layer of indirection.

4.3 Translating Proofs

The largest part of the implementation effort concerns proof translation, which is done by defining an instance of a *proof step interpreter* Sec. 3.3.1 that the evaluator can use.

4.3.1 Wrapper Tactics

Since both SSL$_{\circlearrowleft}$ and HTT are derived from Separation Logic, most of the operational SSL$_{\circlearrowleft}$ rules (*cf.* Read, Write, Alloc, Free in Fig. 2.1) have counterpart Coq tactics in the target frameworks that the interpreter can map to directly.

Other rules require additional effort to fully support. Earlier, we explained the semantics of the Call rule Sec. 2.1.2, which performs a number of steps:

1. Frame out the precondition subheap not affected by the call.
2. Map the procedure call's formal parameters and precondition ghost variables to expressions in the current goal.
3. Create a goal to show entailment from the current goal's precondition to the call's precondition.
4. Create a goal to show entailment from the current goal's precondition with the footprint replaced by the call's postcondition, to the current goal's postcondition.

The HTT proof step corresponding to (1) is:

```
rewrite (joinC _ h1) joinA ; apply: bnd_seq.
```

This performs a sequence of rewrites to the precondition heap, rearranging the heaplets in the Coq context so that those affected by the imminent function call are grouped together on the left-hand side of the rest. The challenge, however, is that the suitable rewrite sequence depends on each call instance—specifically, the rewrite sequence is determined by the current precondition heaplet order, and the subheap that needs to be grouped. Since SuSLik does not reason about heaplets in an order-sensitive way, it is best to handle such *fine-grained bookkeeping* at the Coq level.

Coq provides a Turing-complete tactic language, Ltac [Del00], to uniformly automate such pre/post-processing steps that don't correspond to one of the main lemmas. We can write one such tactic that applies the rewrite logic in a generic way to any call heap:

```
Ltac ssl_call_pre h := prepare_call_heap h; rewrite ?joinA -?(joinA h).
```

Here, `prepare_call_heap` is an auxiliary tactic that pattern-matches on the heap to perform the actual rearranging. The takeaway is that our proof step interpreter need only invoke `ssl_call_pre` with the desired subheap as an argument.

We follow this approach for any other rule whose equivalent main lemma in HTT requires extra preparation or clean-up before and after its invocation. This collection of "wrapper tactics" serves as an interface to the underlying target framework that is tailored to the way SSL$_{\circlearrowleft}$ reasons about programs, leading to more predictable behavior and readable proof scripts.

4.3.2 Tracking Named Hypotheses

The *proof context* is a feature of the evaluator that allows for verifier-specific state tracking (Sec. 3.3.2). Remembering the current names of Coq hypotheses is one important way HTT uses this feature. Recall how an inductive predicate was named `Hsll` at the beginning of the proof and then later used for case analysis in the step corresponding to the Open rule, in the proof walkthrough from Sec. 2.2.2. To emulate this reasoning uniformly, the HTT implementation of the proof step interpreter maintains a map from inductive predicate instances to Coq hypothesis names in the proof context. A unique name for an inductive predicate hypotheses is obtained by concatenating the predicate name, arguments, and cardinality. This mapping is updated whenever an argument value is substituted for another one, or when a new inductive predicate is introduced as a Coq hypothesis. To refer to an inductive predicate by its hypothesis name, the interpreter can simply look it up in this mapping.

4.3.3 Delayed Checking of Postcondition Entailment

In Sec. 2.2.2, we observed a complexity in the way HTT checks entailment of the specification's postcondition from the precondition after symbolically executing all program statements at the end of a proof branch. We now show how our proof evaluation framework facilitates the automatic handling of this entailment checking. The case study highlights two important outcomes of our proof evaluator design, which emphasizes isolating verifier-specific translation logic to a proof step interpreter:

1. The interpreter can reason about each $\text{SSL}_{\circlearrowleft}$ step *in isolation*. This is reflected in the simplicity of its type signature (Eq. 3.3), where the only inputs are a $\text{SSL}_{\circlearrowleft}$ step and context object.

2. Yet the interpreter can perform actions that are *non-local* to the current $\text{SSL}_{\circlearrowleft}$ step being examined. The *proof context* allows the current interpretation to access information from an earlier proof step. The *deferred steps* allow the current interpretation to take effect at a later proof step.

The previous section discussed one usage of the proof context. We also introduced the need for deferred steps in Sec. 3.3.3, pointing to the delay between the step where SuSLik applies the Close rule and the later step in the corresponding HTT proof that uses this knowledge. The postcondition entailment checking process requires both mechanisms to work *in tandem*. First, we describe in detail the information needed to check entailment; next, we show a principled approach to instrumenting it using the evaluator.

How HTT checks postcondition entailment At the end of a proof branch in HTT, we are asked to prove that the state of the heap after symbolically executing the program matches that of the specification's postcondition. Recall the shape of the postcondition heap from specification (1.1):

$$\{\mathsf{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\} \tag{4.1}$$

To get a fuller picture of what information from the source proof tree is needed and when, for the remainder of this section, let us reason from the HTT proof step interpreter's point of view. In particular, suppose the interpreter has reached the terminal Emp rule application in the non-trivial case of the proof, corresponding to the "else" branch of the program. The proof state is as follows, such that two predicate assertions are available to us as hypotheses, *i.e.*, as items in the precondition:

```
...
h11, h21 : heap
H2 : sll nxt s1 h11
H3 : sll y12 s1 h21
============================================================================
∃ (y : ptr) (h1 h2 : heap),
 r ↦ y2 • x2 ↦ v • (x2 +1) ↦ nxt • h11 • y2 ↦ v • (y2 +1) ↦ y12 • h21 =
 r ↦ y • h1 • h2 ∧ sll x2 s h1 ∧ sll y s h2
```

Let us hone in on the existential `h2` and the associated predicate application, `sll y s h2`. This `h2` is a *heap existential*, a unique feature of HTT. The framework exploits the observation that the class of heaps form a *partial commutative monoid* with the heap union operation, to encode heaplets *algebraically*. The consequence is that a spatial assertion is expressed in HTT as an algebraic heap equality. In particular, those that contain predicate applications must be existentially quantified over the subheaps they describe such that their witnesses make the algebraic equality hold.

We can consult the proof tree in Fig. 2.2 on how to proceed, focusing on the SSL$_{\circlearrowleft}$ rules that pertain to this subheap. First, Close unfolds the predicate occurrence in the postcondition using the second clause:

$$\{[y, 2] * y \mapsto v' * (y + 1) \mapsto nxt' * \mathsf{sll}(nxt', s')\}.$$

Then (at later stages of the proof, omitted from the figure), each of those heaplets is *unified* and framed out with a matching heaplet in the precondition, hence the substitution $[y \mapsto \texttt{y2}, v' \mapsto \texttt{v}, nxt' \mapsto \texttt{y12}, s' \mapsto \texttt{s1}]$. Knowledge of these steps gives us everything we need to proceed.

Let us return to the HTT proof. First, we can derive a suitable existential heap witness for `h2` by fully applying the two-step sequence. Doing so tells us that `h2` is eventually expanded to:

$$\texttt{y2} \mapsto \texttt{v} \bullet (\texttt{y2}+1) \mapsto \texttt{y12} \bullet \texttt{h21}$$

Notice how this expansion (provided as the witness in line 67 of Fig. 2.3) still references a heap variable `h21`. This is the label for the subheap that corresponds to the nested sll occurrence that has started as $\mathsf{sll}(nxt', s')$ in our SSL$_{\circlearrowleft}$ proof; we can also observe the related hypothesis `H3 : sll y12 s1 h21` in the Coq proof state shown above. This allows us to make progress on the assertion `sll y s h2` and eventually solve it by evaluating, *in a delayed fashion*, the steps (expansion and unification) in the order we encountered them in the source proof tree earlier.

Using the proof context and deferred steps The example tells us that solving the entailment requires us to track the SSL$_{\circlearrowleft}$ rules that *transform* the predicate occurrences in the postcondition. Furthermore, we observe how this information is useful in two ways: (a) tracing the provenance of a predicate application so that we may readily identify correct heap existential witnesses; and (b) determining the appropriate proof steps that have to be applied to a Coq entailment goal (*e.g.*, the one above) to either solve it or make progress on it. These observations lead us to make use of the proof context for (a), and deferred steps for (b), all the while ensuring that *the deferred step computations are parameterized over a proof context* so that (a) can be done as a part of (b).

We can provide for (a) by keeping a map in the proof context, from predicate applications to either their expanded clause assertion form (on encountering a Close) or their heap variable name (on unification/frame steps). Then, whenever a witness is needed for some heap existential, we simply need to use the map as a lookup table to obtain the maximally expanded subheap of the corresponding predicate application. Next, we can implement (b) by emitting a deferred proof step on each encounter with either of the pertinent rules, and composing them in the order they were encountered. When these deferred steps are released at the end of a proof branch, they perform the necessary moves to discharge the entailment subgoals. For instance, a deferred Close application applies a *concrete* j^{th} constructor of the inductive predicate (*cf.* Fig. 2.1) to obtain the corresponding assertion, and then instantiates its existentials. Since the computation is parameterized over a proof context, it can refer to the lookup table to obtain heap witnesses, within the computation itself.

Using stacked deferred step evaluation environments Other than at the end of a branch, an HTT proof also does a similar entailment-check as part of a procedure call (Sec. 2.1.2). Here, it needs to check that the precondition of the current synthesis goal is unifiable with that of the call.

Recall from Sec. 3.4 that SuSLik synthesizes procedure calls by abducing that it is possible, emitting any preparatory steps to adjust the synthesis goal to a state where the call can take place, and then symbolically executing the call. We noticed that the synthesizer operates in a distinct evaluation environment during call abduction, and developed an enhanced evaluator design (3.6) that allows deferred steps to be accumulated in a separate stack layer during call abduction, and then released when the call is made. This allows us to directly reuse the same method we devised in this section for entailment-checking at the Emp rule, to also handle it at the Call rule.

Incidentally, the latter task is strictly simpler than the former in the current version of SuSLik, due to an implementation detail of the synthesizer: only Frame rule steps appear during call abduction because the synthesis proof search algorithm does not try to apply the Close rule when in that mode. As a result, we do not observe the sort of nested predicate unfoldings that is common for regular postcondition entailment. This is also why the manual proof of `sll_copy` in Fig. 2.3 is able to discharge the call entailment check on line 57 using only a regular simplification tactic—the Coq context has the postcondition assertion that needs to be framed out so it is solved trivially, without relying on insights from synthesis.

Nonetheless, the resilience of an evaluator design should be judged not only on whether it can generate valid proofs (which, as we have shown, ours does), but also on how closely the generated proofs align with SuSLik's reasoning process, so that proof support does not break irreconcilably when the synthesizer is upgraded with more powerful reasoning capabilities. Our approach based on stackable deferred steps would remain viable if, for instance, SuSLik were later equipped with a more insightful call abduction oracle such that it became necessary to emit the Close rule inside a call abduction. Such extensibility is indicative of the generality of our evaluator design.

4.4 Handling Postcondition Pure Constraints

So far, we have emphasized the spatial part of the entailment-checking process. We conclude this chapter by describing how pure facts are proved in HTT using knowledge from the synthesis. This aspect is somewhat distinct from the preceding discussion, as the $\text{SSL}_{\circlearrowleft}$ rule derivations (and therefore proof tree evaluation) do not tell us much about how to reason about pure constraints. For that, we turn to oracles.

When searching for a proof, SuSLik frequently checks whether the pure part of a goal's precondition entails that of its postcondition (as in, *e.g.*, rule Emp in Fig. 2.1) by invoking an SMT solver, which acts as a *validity oracle*. We now show how to incorporate this oracular insight into our proofs by extracting them as standalone Coq lemmas (Sec. 4.4.1) and then solving them automatically using certified solvers (Sec. 4.4.2).

4.4.1 Extracting Pure Lemmas

To use an oracle-validated fact in the Coq proof, we first capture the entailment as a node in the proof tree, and then generate a Coq lemma from it during evaluation.

Certified solvers in Coq (discussed more in the next section) rely on all facts available to them in their invoked context to find a valid proof for an entailment. It is thus desirable to aid their search by *limiting* the statement of the lemma to only include antecedents that can contribute to the particular entailment, so that finding a solution is tractable. Unfortunately, the oracle used by SuSLik only returns boolean answers with no further insight on the proof construction.

One way to immediately prune some unnecessary conjuncts from the antecedent is to discard those that refer to cardinality variables. This works for HTT, which does not use cardinality variables at all; it would not be suitable for other frameworks that do (Sec. 7.1).

```
Lemma pure_example1 k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).
```

Figure 4.2: A pure entailment lemma.

Another, more generally applicable pruning strategy is through syntactic analysis on the variables—if an antecedent conjunct shares no variables with any of the other conjuncts that contribute to the proof of the entailment, we can safely elide it. Formally, we decompose the full entailment into individual lemma statements such that for every postcondition conjunct $i \in C_{post}$, we have one entailment from the set of all precondition conjuncts $j \in C_{pre}$ whose variable set intersects with i's by transitive closure. We implement this as a reachability check. Let $Var(n)$ be the set of variables used in a conjunct expression n. For every postcondition conjunct $i \in C_{post}$, we construct an undirected graph G_i of nodes $\{i\} \cup C_{pre}$ such that an edge connects two nodes $n_1,\ n_2$ if $Var(n_1) \cap Var(n_2) \neq \emptyset$. Then, we can find the subset of precondition conjuncts that excludes those with no possibility of contributing to the entailment proof. Fig. 4.2 shows one such extracted and pruned lemma, derived from an oracle-validated pure entailment from one of our benchmarks.

Though this is a mere syntactic analysis that may not find the most minimal conjunct set (*e.g.*, the lemma in Fig. 4.2 is provable using only the first and fifth antecedents), it has been mostly sufficient for simplifying the lemmas to a level that is tractable for automated solving, as we describe in the next section.

4.4.2 Pluggable Automation

For the most part, the extracted lemmas (which, for our benchmarks, largely consist of arithmetic equalities and inequalities) can be solved by a certified solver, such as CoqHammer [CK18], Micromega [BM20], or SMTCoq [Eki+17]. For example, CoqHammer's `hammer` tactic is a powerful automation tool that finds proofs with an external first-order *automated theorem prover* (ATP), and then reconstructs them in Coq. In fact, it can solve the lemma from Fig. 4.2:

```
Proof. intros. hammer. Qed.
```

In our implementation, by default we emit all such lemmas with accompanying proofs that invoke CoqHammer, which suffices for our standard benchmarks. Sec. 5.2 discusses our handling of the lemmas in more advanced cases when `hammer` is unable to solve them.

Once we obtain our lemmas, we use a Coq feature that allows the user to extend the hint database used for automated proof search with additional lemmas. For instance, we can provide the above lemma as a hint to the database `ssl_pure`:

```
Hint Resolve pure_example1: ssl_pure.
```

This lets us use it in a proof search while verifying the correctness of the main program, *e.g.*, `eauto with ssl_pure`.

5 Evaluation and Case Studies

We implemented the SSL$_{\circlearrowleft}$ proof tree extraction logic, proof evaluator, and HTT proof step interpreter in Scala. The source code is publicly available.[1]

We also developed HTT's verifier-specific automation (Ch. 4) tactics in Coq's tactic language Ltac. A lightweight Coq library that bundles these tactics can be installed via OCaml's `opam` package manager. It is also open source, along with all proof scripts generated as part of the benchmark suite.[2]

Tab. 5.1 summarizes the overall implementation effort in terms of lines of code. Our evaluation focuses on two criteria. The first is the efficiency of verifying synthesized certificates, in terms of Coq specification/proof sizes and proof-checking times. The second is the presence of observable gaps between the language/logic of the synthesizer and verifier that complicate our automated certification efforts.

Table 5.1: Scala implementation size in lines of code.

Component	Scala	Coq
Proof evaluator	1042	-
HTT support	1340	160
The rest of SuSLik	5508	-

We discuss the first item *wrt.* our standard benchmarks (Sec. 5.1), and the second item *wrt.* our advanced benchmarks (Sec. 5.2).

5.1 Standard Heap-Manipulating Benchmarks

Tab. 5.2 summarizes our evaluation results on programs manipulating individual pointers and integers, singly- and doubly-linked lists, and binary trees. All certificates are generated for unaltered SusLang programs. The reported sizes

[1] https://github.com/TyGuS/suslik/tree/certification

[2] https://github.com/TyGuS/ssl-htt

Table 5.2: Statistics for synthesized programs with pointers from SuSLik benchmark suite. Sizes of generated Coq artifacts in lines of code; proof checking times in seconds.

Group	Description	Synthesis Time	Spec	Proofs	Proof Checking Time
Integers	max	0.6	55	18	57.6
	min	0.3	55	18	52.6
	swap2	0.5	49	15	3.4
	swap4	0.4	53	23	8.1
Singly-Linked Lists	maximum	0.7	68	99	3.0
	minimum	1.6	68	99	3.2
	length	0.8	68	100	3.1
	append	0.4	60	89	6.2
	copy	0.8	70	103	50.8
	two-element	0.6	57	50	2.8
	dispose	0.1	55	46	1.8
	singleton	0.1	54	33	1.8
DLLs	append	2.7	74	154	6.5
	singleton	0.1	55	37	2.5
Trees	copy	1.3	73	135	6.8
	flatten	0.4	92	138	6.2
	dispose	<0.1	58	62	2.4
	size	0.5	64	92	4.0

of Coq artifacts do not include translated heap predicates, as those are shared between specifications of multiple programs. All results are obtained on a 3.10GHz Intel Core i7-5557U machine with 8GB RAM running macOS 10.15.7 and Coq 8.11.1.

Tab. 5.2 demonstrates that all generated proofs are relatively concise. Some are more lengthy than others, due to a number of administrative renamings required to keep the proof in sync with the Coq context (Sec. 4.3.2). The proof checking times range from 2 to 8 seconds for all but three HTT examples. The three outliers (integer minimum/maximum, and singly linked list copying) are due to the use of CoqHammer for discharging pure entailment lemmas that rely on a formalization of Peano numbers and sequences imported from the Ssreflect library [GMT09].

5.2 Advanced Benchmarks: Encoding Collection Payloads

In the suite of programs in Tab. 5.2, we did not include benchmarks that require extensive support for solving pure entailments via SMT, such as binary search trees and sorted lists. The reason for that lies in the challenge of encoding *payloads* for heap-based collections. SuSLik and HTT address this in very different ways, each opting for one suited to its unique requirements.

Table 5.3: Benchmarks using multi-set equality in HTT.

Group	Program	Synt. time	Coq time	Lemmas	Manual
Doubly-Linked Lists	copy	5.2	12.1	7	4
	two-element	0.5	6.2	3	3
	from-sll	1.0	8.6	5	2
Binary Search Trees	rotate-left	3.9	8.7	4	2
	rotate-right	3.6	8.6	4	2
	insert	18.5	33.9	15	6
	rmv-root-left	1.9	14.2	6	2
	rmv-root-right	16.8	16.4	6	2
	find-smallest	1.7	6.7	7	1
Sorted Lists	prepend	0.1	4.0	2	0
	insert	6.2	17.6	18	5
	insertion-sort	1.0	6.2	7	0
	insert-sort-free	0.5	5.2	5	0

SuSLik represents collection contents using unordered *multi-sets*, a suitable choice for a synthesizer for the purpose of performing framing and unification. It is not uncommon, say, for a partitioned set to later be unified in a different order, so the unorderedness of sets is accommodating of such cases.

```
Lemma sll_perm_eq x h s t :
  perm_eq s t -> sll x s h -> sll x t h.
Proof.
  move=>Hpermeq Hsll1.
  case: Hsll1=>cond.
  - move=>[Hsll1_pure ->].
    constructor 1=>//; sslauto.
  - move=>[v] [s2] [nxt] [h'].
    move=>[Hsll1_pure [-> Hssl2]].
    constructor 2=>//.
    ∃ v, s2, nxt, h'.
    sslauto.
    assumption.
Qed.
```

Figure 5.1: Manual proof of a lemma asserting that two predicate applications parameterized by equivalent multi-sets are equally valid.

HTT, in contrast, represents payloads as algebraic *lists*, as they are much easier to reason about in proofs. For example, because list equality assertions use the propositional equality (=), Coq's rewriting tactics can take advantage of them; multi-set equality cannot be encoded this way. For this reason, we have selected programs that are agnostic to those differences for our "standard" benchmarks in Tab. 5.2. Nonetheless, we also experimented with more advanced benchmarks, summarized in Tab. 5.3, that rely on the collection payloads being implemented as multi-sets, by replacing regular list equality with the perm_eq predicate from the Ssreflect/MathComp library [MT17; GMT09], which asserts that one list is a permutation of the other. As discussed, this means that Coq's rewriting tactics can no longer handle these assertions, so we must make a trade-off to introduce a non-trivial axiom about the congruence of perm_eq across predicate applications with otherwise identical arguments. For our advanced benchmark programs, we proved these axioms manually as lemmas. Fig. 5.1 shows one such lemma for the sll predicate from Fig. 2.3, proved by unfolding the same constructor in both predicate applications, and then showing that the pure part of one entails the pure part of the other.

Several of the programs in these advanced benchmarks also feature non-trivial pure lemmas (discussed in Sec. 4.4.1) that our implementation's default proof strategy, CoqHammer's `hammer` tactic, fails to solve. To make the evaluation tenable, we chose to first auto-generate the proof scripts for the benchmark programs using our HTT proof step interpreter, attempt to compile them (with `hammer`), and only provide manual proofs of those pure lemmas that `hammer` fails to discharge. The figures in the Tab. 5.3 were obtained by benchmarking these modified proof scripts. The last column indicates the number of pure lemmas that required manual proofs. While this deviates from the idea of full proof automation, it is still consistent with the intent of the proof scripts, in that they are generated in a way that the user can easily revisit and verify the output.

```
Lemma pure_example2 lo2x vx2 k2 hi1x lo1x :
  vx2 <= lo2x -> 0 <= vx2 -> hi1x <= vx2 ->
  k2 <= vx2 -> vx2 <= 7 -> 0 <= k2 -> k2 <= 7 ->
  (if k2 <= ((if vx2 <= lo1x then vx2 else lo1x))
   then k2 else (if vx2 <= lo1x then vx2 else lo1x)) =
  (if vx2 <= (if k2 <= lo1x then k2 else lo1x)
   then vx2 else (if k2 <= lo1x then k2 else lo1x)).
Proof.
  move=>H1 H2 H3 H4 H5 H6 H7;
  case (vx2 <= lo1x) eqn:H8;
  case (k2 <= lo1x) eqn:H9;
  case (k2 <= vx2) eqn:H10;
  sauto.
Qed.
```

Figure 5.2: Manual proof of an extracted pure entailment lemma that the `hammer` tactic failed to solve.

In our experience, it was not too difficult to manually prove pure lemmas that CoqHammer failed to handle. Fig. 5.2 shows one such lemma whose automated proof failed, extracted from the certificate of one of the advanced benchmark programs that inserts an element into a binary search tree. While being relatively complex with seven antecedents, and some with nested ternary expressions, the proof proceeded by exhaustively enumerating the conditions, and then discharging the rest with `sauto`, a tactic in CoqHammer's toolset that provides an enhanced alternative to Coq's native `auto` tactic.

We suspect the `hammer` tactic fails to automate this reasoning because the correct assertions to case-analyze on are located deep inside the ternary expressions, rather than being accessible as standalone facts. In fact, in discussing CoqHammer's limitations, the authors recognize that they currently take a relatively simplistic approach to detecting propositions in subterms [CK18], so this may be an instance of that observation. Sec. 6.4 describes CoqHammer (and hammers in general) in greater detail.

Among our advanced benchmarks, this example is fairly representative of those lemmas whose proofs need human intervention. Correspondingly, the structure of their respective manual proofs also heavily rely on case analyzing the proper conditions.

6 Related Work

In this chapter, we survey existing techniques related to our work. Sec. 6.1 and Sec. 6.2 discuss our work's roots in proof-carrying code and translation validation, respectively; Sec. 6.3 contrasts our approach to interactive certified synthesis; and Sec. 6.4 provides further exposition on our usage of CoqHammer and on certified solvers in general.

6.1 Certifying Compilers and Proof-Carrying Code

Our work is the latest application of the widely studied technique of *proof-carrying code* (PCC). PCC was first proposed in 1996 as a way to produce certified binaries that supply a formal proof of the code along with the native code, so that an operating system kernel could independently verify type and memory safety properties about an untrusted binary before executing it [NL96]. The method [Nec97] expressed safety proofs in terms of a logic embedded in the Edinburgh Logical Framework [HHP93] and targeted the DEC Alpha assembly language. A subsequent effort designed a *certifying compiler* [NL98] that compiles programs written in a high-level language into these PCC binaries, whose safety properties can then be easily validated. Further work reduced the size of the trusted codebase by extracting a certified verifier from a Coq proof, using the PCC approach itself [App01].

Since its conception, PCC has been extended and applied to a number of domains, such as:

- Verifying temporal safety properties in systems code [Hen+02].

- Automatically generating safety invariant certificates for a PCC framework via abstract interpretation [BJP06].
- A type-preserving compiler from a source language with a rich type system to an untyped low-level language by carrying proof terms throughout the compilation [CCS10].
- Certifying and validating information-flow properties at the bytecode level [BPR13].
- Enforcing behavioral properties *wrt.* a program's interaction with software-defined networks [Ska+19].

Our work presents a new application of the PCC approach to automated program synthesis, continuing in the tradition of pairing program code generation with proofs of desirable properties. Whereas the original proposal [Nec97] supplies proofs of a program's safety properties, our work certifies their *full functional correctness*, by leveraging the deductive reasoning capabilities of the synthesizer.

6.2 Translation Validation

Pnueli, Siegel, and Singerman [PSS98]'s work on *translation validation* is an alternative post-hoc technique for verifying translators such as compilers and code generators. It involves defining an *analyzer* which, given a source program and generated target program, produces a correctness proof if the translation is valid, similar to our abstract proof evaluator design.

Our work differs from translation validation in the way verification proceeds. In translation validation (which supports certification of *general* translators, such as compilers), correctness is expressed in terms of the target system's refinement of the source system. Thus, the correctness of target programs are verified using simulation techniques. Meanwhile, our technique is specific to certifying deductive synthesizers. By narrowing our focus, we remove the need to obtain a refinement mapping, as the deductive synthesis itself gives us a proof "for free", which then need only be checked in a system with a smaller trusted codebase such as Coq.

6.3 Certified Interactive Program Synthesis

The FIAT framework [Del+15; Chl+17] implements certified *interactive* program synthesis. It encodes programs as abstract data types (ADTs), and embeds the synthesis procedure directly into the Coq proof assistant to obtain the highest correctness guarantee. The synthesis is done by *refinement*, where a high-level specification that may admit multiple implementations is incrementally refined [HHS86] via a user-guided search aided by a set of automation tactics, until a satisfactory ADT is obtained; the resulting ADT can be extracted to executable OCaml code. The framework provides refinement automation tactics for certain restricted domains, as well as a library of lemmas to handle other domains for more advanced use cases. While synthesis using FIAT requires a user to interactively choose the desired refinement sequence at each step, SUSLIK performs fully automated synthesis by proof search.

6.4 Certified Solvers

We match SUSLIK's delegation of pure entailments to a validity oracle (realized in the implementation as a third-party SMT solver) with a similar delegation to certified solvers to discharge the equivalent proof obligations in the verification frameworks.

On its own, a standard Coq installation already comes equipped with robust proof automation tactics. The `auto` tactic does a proof search by enumerating basic tactics `reflexivity`, `assumption`, and `apply` with the lemmas accessible by it. A variant tactic `eauto` is capable of unifying the goal with a lemma even when there are no immediate instantiations available, by using existential variables and substituting witnesses later. Micromega [BM20], a module bundled with Coq, is a collection of *decision procedures* for solving proof goals that have a particular form relying on no heuristics. Its tactic `lia` supports linear integer arithmetic and inequality goals. These suffice for simple cases.

Hammers are suitable for more difficult proofs in large projects that have many lemmas in scope (and thus more potential proof sequence combinations). These tools employ advanced techniques to trigger, with one command, an efficient three-part proof search [Bla+16]:

1. *Premise selection.* Choosing which available lemmas are potentially useful for proving the goal.

2. *Translation.* Converting the goal and selected premises in the proof assistant's logic to an expression in the logic of an automated theorem prover (ATP).

3. *Reconstruction.* Converting a solution found by ATP back to the logic of the proof assistant, so it can be accepted as a valid proof.

We previously discussed our usage of CoqHammer [CK18] in HTT (Sec. 4.4.2) and some limitations we encountered (Sec. 5.2). Its `hammer` tactic takes this three-step approach: it learns a suitable set of premises with the aid of machine learning algorithms trained on past proofs from the Coq standard library, translates expressions from the Calculus of Inductive Constructions (CIC)—the type theory implemented by Coq—to a first-order logic problem, and reproves the goal in Coq using the ATP proof trace as a hint. The reasoning during the last step is aided by the other component of CoqHammer, `sauto`. This tactic, which implements a proof search strategy by type inhabitation [Cza20], empirically outperformed Coq's own `auto` tactic, and can be used as a standalone replacement to it independently of `hammer`.

Whereas CoqHammer works with general ATPs, SMTCoq [Eki+17] is a separate tool that operates specifically in the problem domain of satisfiability modulo theories (SMT). The tool integrates Coq with external SMT solvers. It increases the confidence in the result produced by an external solver by checking the generated proof witness in Coq; the checking procedure itself is certified as it is also implemented in Coq.

For our HTT implementation, we chose to handle pure entailments with CoqHammer and obtained promising results. We discussed our usage of it in Sec. 4.4.2, and some limitations we encountered in Sec. 5.2.

7 Discussion and Conclusion

We conclude this work by noting parallel efforts to instantiate the evaluator with other verifiers (Sec. 7.1) and discussing future work (Sec. 7.2).

7.1 Extensibility To Other Verification Frameworks

Other than HTT (Ch. 4), the abstract evaluator has been successfully instantiated by collaborators to two additional foundational verification frameworks, Iris [Jun+18] and Verified Software Toolchain (VST) [App+14]. A separate work [Wat+21] summarizes the findings from Ch. 3 and Ch. 4, discusses the experience of adding support for Iris and VST, and compares the three frameworks in terms of the relative ease of adapting each one as a certification target.

Both Iris and VST are deeply embedded frameworks, in contrast to HTT's shallow embedding, which is cause for some additional discrepancy between SusLang/SSL$_\circlearrowleft$ and the target language and logic. For example, while HTT's encoding of spatial assertions as native Gallina propositions enables a relational definition of predicates (Sec. 4.2), both Iris and VST use a custom abstract type to do the same. This means that instead of encoding inductive predicates as propositions, they must be defined as computations (*i.e.*, functions) that return the custom type—a nontrivial conversion. It is resolved by using the cardinality variables (Sec. 2.1) to specify the termination measure for the recursive functions.

Nonetheless, the results obtained for the two frameworks indicate the generality of the abstract evaluator design. With a few exceptions, both Iris and VST successfully handle most of the standard benchmarks in Tab. 5.2. Iris does not certify two case studies, finding the minimum and maximum of a list of singly linked list of integers, whose SSL$_{\circlearrowleft}$ specification uses a ternary operator; Iris needs to generate proof obligations for each branch, so the proof diverges from the one encoded in the proof tree. VST does not handle list length and tree size; the framework verifies C programs that run on real hardware, so it needs to account for additional error modes like integer overflow, which SusLang does not. All of these exceptions arise due to verifier-specific details that are too far removed from the SusLang/SSL$_{\circlearrowleft}$ representation of programs and proofs; thus they are beyond the scope of the evaluator design.

7.2 Future Work

In Sec. 6.4, we contrasted Fiat's interactive synthesis approach with SuSLik's automated one. However, our experiments (*cf.* Sec. 5.2) also demonstrate some cases where the certification of a SuSLik-synthesized program needs manual assistance, when faced with complex pure entailments beyond the scope of Coq's proof automation capabilities. We believe that by combining fully automated deductive synthesis with interfaces for a user to provide manual proofs for non-trivial facts, we may be able to achieve an optimal balance between minimizing the proof burden on the user and expanding the domain of certifiable programs.

Controlling certificate sizes and proof checking times could be another potential area of exploration. Our proof-tree-guided certification technique does considerably minimize the amount of additional proof search needed by the verifier (on top of what was already done by the synthesizer). Nonetheless, our work emphasizes full functional correctness for a broad range of programs, and so it does not explicitly optimize for concise certificates or instant verification. As our results show, checking time still suffers even for some simple programs due to the challenge of automating pure lemma proofs. Yet we recognize the importance of such properties in a security context that demands an efficient and scalable verification methodology—in fact, minimizing run-time certification overhead was a central concern in the original PCC proposal [NL96]—so we leave this for future work.

7.3 Conclusion

Our results hint at a more accessible and secure future where users can obtain correct programs by providing concise descriptions. While a fully verified synthesizer is a heavy undertaking, in this work we have shown the viability of a post-hoc approach to automatically synthesizing certified programs with currently available technology. Furthermore, we have demonstrated that the technique can be made verifier-agnostic by designing a suitable abstraction to match the impedance between synthesis and verification.

Bibliography

[App+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014 (cit. on p. 48).

[App01] Andrew W. Appel. "Foundational Proof-Carrying Code". In: *LICS*. IEEE Computer Society, 2001, pp. 247–256 (cit. on p. 44).

[BJP06] Frédéric Besson, Thomas P. Jensen, and David Pichardie. "Proof-carrying code from certified abstract interpretation and fixpoint compression". In: *Theor. Comput. Sci.* 364.3 (2006), pp. 273–291 (cit. on p. 45).

[Bla+16] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. "Hammering towards QED". In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 101–148 (cit. on p. 47).

[BM20] Frédéric Besson and Evgeny Makarov. *Micromega: solvers for arithmetic goals over ordered rings*. Online documentation available at `https://coq.inria.fr/refman/addendum/micromega.html`. 2020 (cit. on pp. 37, 46).

[BPR13] Gilles Barthe, David Pichardie, and Tamara Rezk. "A certified lightweight non-interference Java bytecode verifier". In: *Math. Struct. Comput. Sci.* 23.5 (2013), pp. 1032–1081 (cit. on p. 45).

[CCS10] Juan Chen, Ravi Chugh, and Nikhil Swamy. "Type-preserving compilation of end-to-end verification of security enforcement". In: *PLDI*. ACM, 2010, pp. 412–423 (cit. on p. 45).

[Chl+17] Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. "The End of History? Using a Proof Assistant to Replace Language Design with Library Design". In: *SNAPL*. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 3:1–3:15 (cit. on p. 46).

[CK18] Lukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for Dependent Type Theory". In: *J. Autom. Reason.* 61.1-4 (2018), pp. 423–453 (cit. on pp. 37, 43, 47).

[Cza20] Łukasz Czajka. "Practical proof search for Coq by type inhabitation". In: *International Joint Conference on Automated Reasoning*. Springer. 2020, pp. 28–57 (cit. on p. 47).

[Del+15] 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. 46).

[Del00] David Delahaye. "A Tactic Language for the System Coq". In: *LPAR*. Vol. 1955. LNCS. Springer, 2000, pp. 85–95 (cit. on p. 31).

[Eki+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq". In: *CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 126–133 (cit. on pp. 37, 47).

[GMT09] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Tech. rep. 6455. Microsoft Research – Inria Joint Centre, 2009 (cit. on pp. 40, 41).

[Gu+16] 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*. USENIX Association, 2016, pp. 653–669 (cit. on p. 1).

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. “Temporal-Safety Proofs for Systems Code”. In: *CAV*. Vol. 2404. LNCS. Springer, 2002, pp. 526–538 (cit. on p. 44).

[HHP93] Robert Harper, Furio Honsell, and Gordon D. Plotkin. “A Framework for Defining Logics”. In: *J. ACM* 40.1 (1993), pp. 143–184 (cit. on p. 44).

[HHS86] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. “Data Refinement Refined”. In: *ESOP*. Vol. 213. LNCS. Springer, 1986, pp. 187–196 (cit. on p. 46).

[Itz+21] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben Rowe, and Ilya Sergey. *Cyclic Program Synthesis*. Conditionally accepted for publication at PLDI’21. 2021 (cit. on pp. v, 5, 7, 26).

[Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *J. Funct. Program.* 28 (2018), e20 (cit. on p. 48).

[Kne+13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. “Synthesis modulo recursive functions”. In: *OOPSLA*. ACM, 2013, pp. 407–426 (cit. on p. 7).

[Ler06] Xavier Leroy. “Formal certification of a compiler back-end or: programming a compiler with a proof assistant”. In: *POPL*. ACM, 2006, pp. 42–54 (cit. on p. 1).

[MM73] Alberto Martelli and Ugo Montanari. “Additive AND/OR Graphs”. In: *IJCAI*. Ed. by Nils J. Nilsson. 1973, pp. 1–11 (cit. on p. 15).

[MT17] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at `https://math-comp.github.io/mcb`. 2017 (cit. on p. 41).

[Nec97] George C. Necula. “Proof-Carrying Code”. In: *POPL*. ACM Press, 1997, pp. 106–119 (cit. on pp. 44, 45).

[NL96] George C. Necula and Peter Lee. "Safe Kernel Extensions Without Run-Time Checking". In: *OSDI*. ACM, 1996, pp. 229–243 (cit. on pp. 44, 49).

[NL98] George C. Necula and Peter Lee. "The Design and Implementation of a Certifying Compiler". In: *PLDI*. ACM, 1998, pp. 333–344 (cit. on p. 44).

[NVB10] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. "Structuring the verification of heap-manipulating programs". In: *POPL*. 2010, pp. 261–274 (cit. on pp. v, 3, 10, 28).

[PS19] Nadia Polikarpova and Ilya Sergey. "Structuring the Synthesis of Heap-Manipulating Programs". In: *PACMPL* 3.POPL (2019), 72:1–72:30 (cit. on pp. v, 2, 5).

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation validation". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1998, pp. 151–166 (cit. on p. 45).

[RB17] Reuben N. S. Rowe and James Brotherston. "Automatic cyclic termination proofs for recursive procedures in separation logic". In: *CPP*. ACM, 2017, pp. 53–65 (cit. on p. 6).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. IEEE Computer Society, 2002, pp. 55–74 (cit. on p. 5).

[Ska+19] Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. "Proof-Carrying Network Code". In: *CCS*. ACM, 2019, pp. 1115–1129 (cit. on p. 45).

[Wat+21] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. *Certifying the Synthesis of Heap-Manipulating Programs*. Conditionally accepted at ICFP'21. 2021 (cit. on p. 48).

[Wat20] Yasunari Watanabe. "Building a Certified Program Synthesizer". Bachelor's thesis. Yale-NUS College, 2020 (cit. on p. 3).