Certifying the Synthesis of Heap-Manipulating Programs

YASUNARI WATANABE, Yale-NUS College, Singapore and National University of Singapore, Singapore
KIRAN GOPINATHAN, National University of Singapore, Singapore
GEORGE PÎRLEA, National University of Singapore, Singapore
NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore

Automated deductive program synthesis promises to generate executable programs from concise specifications, along with proofs of correctness that can be independently verified using third-party tools. However, an attempt to exercise this promise using existing proof-certification frameworks reveals significant discrepancies in how proof derivations are structured for two different purposes: *program synthesis* and *program verification*. These discrepancies make it difficult to use certified verifiers to validate synthesis results, forcing one to write an ad-hoc translation procedure from synthesis proofs to correctness proofs for each verification backend.

In this work, we address this challenge in the context of the synthesis and verification of heap-manipulating programs. We present a technique for principled translation of deductive synthesis derivations (*a.k.a. source* proofs) into deductive *target* proofs about the synthesised programs in the logics of interactive program verifiers. We showcase our technique by implementing three different certifiers for programs generated via SuSLik, a Separation Logic-based tool for automated synthesis of programs with pointers, in foundational verification frameworks embedded in Coq: Hoare Type Theory (HTT), Iris, and Verified Software Toolchain (VST), producing concise and efficient machine-checkable proofs for characteristic synthesis benchmarks.

CCS Concepts: • **Theory of computation → Logic and verification**; • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Separation Logic, Proof Assistants, Mechanised Proofs

ACM Reference Format:
Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP, Article 84 (August 2021), 29 pages. https://doi.org/10.1145/3473589

1 INTRODUCTION

As projects like CompCert (Leroy 2006) and CertiKOS (Gu et al. 2016) demonstrate, developing verified software is an extremely laborious process. In addition to writing code, it requires writing matching formal specifications, as well as proofs that the former satisfies the latter. A promising approach to reducing the amount of effort involved is to automate parts of the development using *deductive program synthesis* (Kneuss et al. 2013; Manna and Waldinger 1980; Polikarpova and Sergey 2019), which allows the programmer to focus on writing the specifications, letting the synthesiser search for a corresponding program *together with its proof*.

Authors' addresses: Yasunari Watanabe, Yale-NUS College, Singapore and National University of Singapore, Singapore, yasunari@u.yale-nus.edu.sg; Kiran Gopinathan, National University of Singapore, Singapore, kirang@comp.nus.edu.sg; George Pîrlea, National University of Singapore, Singapore, gpirlea@comp.nus.edu.sg; Nadia Polikarpova, University of California, San Diego, USA, npolikarpova@eng.ucsd.edu; Ilya Sergey, Yale-NUS College, Singapore and National University of Singapore, Singapore, ilya.sergey@yale-nus.edu.sg.

2475-1421/2021/8-ART84
https://doi.org/10.1145/3473589

As a simple example, consider implementing a verified bank account data structure that stores a user identifier along with the user's balance in a C-like imperative language with pointers. The constructor procedure `mk_acc` of this data structure might be given the following specification in Separation Logic (SL) (O'Hearn et al. 2001; Reynolds 2002):

$$\{\mathtt{r} \mapsto -\}\ \mathtt{mk\_acc}(\mathtt{r}, \mathtt{id}, \mathtt{bal})\ \{\mathtt{r} \mapsto x * \mathtt{account}(x, \mathtt{id}, \mathtt{bal})\} \tag{1}$$

In the spec above, the precondition asserts that the parameter `r` is a location that initially stores some unimportant value (−). The postcondition asserts that, after a call to `mk_acc(r, id, bal)` terminates, `r` should point to some location x that contains an account data structure initialised with `id` and `bal` (and the memory of the data structure does not include `r`, as indicated by the *separating conjunction* connective $*$). The definition of account from the postcondition is given by the following predicate:

$$\mathtt{account}(x, id, bal) \triangleq [x, 2] * x \mapsto id * (x + 1) \mapsto bal \tag{2}$$

That is, account is a record of size two starting at a location x, whose first field stores the value id, and the second stores bal. Given the specification (1), the deductive synthesiser SuSLik (Polikarpova and Sergey 2019) is able to automatically *derive* an implementation satisfying this spec (Fig. 1).[1]

```
1  void mk_acc (r, id, bal) {
2    {r ↦ −}
3    let z = malloc(2);
4    {r ↦ −∗[z, 2]∗z ↦ −∗(z+1) ↦ −}
5    *r = z;
6    {r ↦ z∗[z, 2]∗z ↦ −∗(z+1) ↦ −}
7    *z = id;
8    {r ↦ z∗[z, 2]∗z ↦ id∗(z+1) ↦ −}
9    *(z + 1) = bal;
10   {r ↦ z∗[z, 2]∗z ↦ id∗(z+1) ↦ bal}
11 }
12 {r ↦ x ∗ account(x, id, bal)}
```

Fig. 1. `mk_acc`: code and proof outline.

The Challenge: certifying automated program synthesis. But can we trust the correctness proof produced by SuSLik? Automated program synthesisers are quite complex, and it is hard to guarantee that they are free of bugs. Moreover, a realistic verification project will likely have portions that are beyond the capabilities of any synthesiser, and hence we need to make sure that synthesis results integrate well with parts of the system that are implemented and verified manually. A promising approach to addressing both of these concerns is to make the synthesiser produce *certificates*, allowing for independent checking of its results in foundational verification tools, which are embedded into proof assistants such as Coq and provide the highest assurance guarantees due to their minimal trusted code base.

In theory, it should be easy to generate certificates for deductive synthesisers: after all, they synthesise programs together with their "proofs"! In practice, however, it is far from straightforward, because these proofs are structured very differently from the proofs in foundational verifiers. SL verifiers typically work by using symbolic execution to propagate the *symbolic state* from the precondition forward through the program (see even lines 2–10 in Fig. 1), and only at the end do they check that the final symbolic state (line 10) entails the given postcondition (line 12). On the other hand, a deductive synthesiser is forced to use information from both the pre- and the post-condition, manipulating both in the process, because it cannot rely on the program to guide the search. Apart from this fundamental discrepancy, there are many low-level differences in the structure of the proofs, both between synthesis and verification, as well as between different foundational verifiers. As a result, our first attempt at translating SuSLik proofs into three different Coq-based verifiers resulted in three custom proof traversal procedures, intertwined with bookkeeping machinery and rule translation logic, leading to massive code duplication and a very brittle codebase overall.

In this paper, we present *modular proof interpreters*, a technique for certifying deductive synthesis in custom verification backends. The technique is inspired by the continuation-passing programming style, and supports writing proof translation logic in a uniform and modular way.

[1] For now, let us ignore the assertions of the form $\{\ldots\}$ on the even-numbered lines of the listing in Fig. 1.

Key insights. In the simplest case, translating a *source proof* generated by the synthesiser into a *target proof* understood by the verifier would involve mapping each individual application of a source inference rule in to a sequence of applications of target inference rules. This, however, does not work for our problem due to non-local differences in the rule application order employed by the synthesiser and the verifiers. Our first key idea is to equip the proof translator with a way to *define the order* in which the translated results of synthesis rule applications are executed in the script of the back-end verifier. Our approach requires traversing the source proof *just once*: the translator needs to define, for each source rule, in a modular fashion, which parts of its translation should be applied in the target verification script *immediately*, and which parts should be *deferred* until later stages. Intuitively, this dichotomy corresponds to partitioning forward-style program verification into symbolic execution (immediate steps) and entailment checking (deferred steps). Pragmatically, this technique manifests in collecting deferred target proof steps in the *proof interpreter state*, and applying those steps at leaves of the generated target proof tree.

As it turns out, simply deferring certain target proof steps is not enough: any changes in the target *proof context* since the moment certain deferred steps were scheduled, can render those steps invalid at the point of their application. Our second key idea is to allow the translator to define the logic for maintaining a backend-specific proof context, and also to *pass it as an argument* to the deferred proof steps at the time of their applications, making their treatment similar to that of composed continuations in interpreters for programs written in continuation-passing style.

We define the traversal of source proof trees to be *parametric wrt.* the target proof context and deferred steps. This way, the translator only needs to provide a "small-step" modular interpreter for each of the source rules. Our generic traversal will execute that interpreter in each of the source tree nodes, passing its state (*i.e.*, the proof context and deferred steps), thus constructing the target proof tree in a single pass.

Our contributions. The main conceptual contribution of this work is a realisation of the idea of modular proof interpreters—an approach for implementing the translation of synthesis proof trees into verification proof trees via a uniform source proof traversal, parameterised by the notion of the target proof context and the execution order of target proof steps. We applied this idea to SuSLik—a deductive program synthesis tool based on Separation Logic (Polikarpova and Sergey 2019), allowing for generation of proof certificates from its program derivations.

Our main practical contribution is instantiation of SuSLik's support for modular proof interpreters for *three different foundational verification backends*, implemented as embeddings into the Coq proof assistant: Hoare Type Theory (HTT) (Nanevski et al. 2010), Iris (Jung et al. 2018; Krebbers et al. 2017), and Verified Software Toolchain (VST) (Appel 2011). We provide an extensive comparison of our three proof interpreter implementations, outlining the main challenges of translating SuSLik-produced proofs in a synthesis-tailored version of Separation Logic to each of the backends and elaborating on the main implementation efforts in each case. Finally, we evaluate our implementation against a series of synthesis benchmarks for heap-manipulating programs, reporting on the qualitative and quantitative aspects of the produced proof scripts.

Paper outline. In the rest of the paper, we provide a brief background on deductive synthesis and verification with Separation Logic, building intuition for the addressed challenge (Sec. 2). We follow with an overview of our approach to proof translation (Sec. 3), zooming in on its integration with the SuSLik synthesis engine in Sec. 4. We then elaborate on the three specific instances of our technique in Sec. 5 (HTT), Sec. 6 (Iris), and Sec. 7 (VST). We report on our experiments with translating SuSLik proofs into each of the three supported backends in Sec. 8 and discuss our implementation experience and lessons learnt in Sec. 9. We conclude with a survey of related challenges and techniques, and an outline of the future work in Sec. 10.

2 SYNTHESIS PROOFS VS VERIFICATION PROOFS

Before we describe the key components of our work, it is essential to explain how exactly the program in Fig. 1 was synthesised, and what makes search for a program different from its verification.

Synthesising `mk_acc()`. A deductive SL-based synthesis takes specification (1) as its initial *goal* and searches for a series of inference rule applications which would reduce the initial goal to trivial subgoals, generating the desired program as a byproduct. A successful series of such rule applications is demonstrated by the following *proof trace*, which shows a progressive simplification of the goal until it becomes trivial (which means the synthesis is done). For our example, the starting point is the following goal requiring one to find a program that performs the transformation from a state satisfying the symbolic precondition $\{\mathsf{r} \mapsto -\}$ to the one satisfying the postcondition $\{\mathsf{r} \mapsto x * \mathsf{account}(x, \mathsf{id}, \mathsf{bal})\}$, assuming that `r`, `id`, and `bal` are program-level variables:

$$\{\mathsf{r}, \mathsf{id}, \mathsf{bal}\}\,; \{\mathsf{r} \mapsto -\} \rightsquigarrow \{\mathsf{r} \mapsto x * \mathsf{account}(x, \mathsf{id}, \mathsf{bal})\} \tag{3}$$

The first step of the trace discovered by the synthesis tool immediately elaborates the goal postcondition, by *unfolding* the predicate occurrence $\mathsf{account}(\ldots)$ and replacing it by its definition (2):

$$\{\mathsf{r}, \mathsf{id}, \mathsf{bal}\}\,; \{\mathsf{r} \mapsto -\} \rightsquigarrow \left\{\mathsf{r} \mapsto x * \boxed{[x, 2] * x \mapsto \mathsf{id} * (x + 1) \mapsto \mathsf{bal}}\right\} \tag{4}$$

This manipulation of the goal will not produce any immediate program commands, but the new postcondition makes it apparent that the way to proceed is to allocate two subsequent pointers, as those are missing from the precondition, hence the following goal transformation, resulting in the command at line 3 of Fig. 1 and capturing the newly allocated symbolic heap in the precondition:

$$\{\ldots, \boxed{\mathsf{z}}\}\,; \left\{\mathsf{r} \mapsto - * \boxed{[\mathsf{z}, 2] * \mathsf{z} \mapsto - * \mathsf{z} + 1 \mapsto -}\right\} \rightsquigarrow \{\mathsf{r} \mapsto x * [x, 2] * x \mapsto \mathsf{id} * (x + 1) \mapsto \mathsf{bal}\}$$

Given the similarity of the symbolic heap fragments in the pre- and postcondition, rooted in `z` and (existentially quantified) x respectively, the next step will issue a substitution $x \mapsto \mathsf{z}$, thus unifying the corresponding parts in the pre/post:

$$\{\ldots\}\,; \{\mathsf{r} \mapsto - * [\mathsf{z}, 2] * \mathsf{z} \mapsto - * \mathsf{z} + 1 \mapsto -\} \rightsquigarrow \left\{\mathsf{r} \mapsto \boxed{\mathsf{z}} * \boxed{[\mathsf{z}, 2]} * \boxed{\mathsf{z}} \mapsto \mathsf{id} * (\boxed{\mathsf{z}} + 1) \mapsto \mathsf{bal}\right\}$$

The remaining three writes at lines 5, 7, and 9 of Fig. 1 will be synthesised each by considering the pairs of *heaplets* of the form, *e.g.*, $\mathsf{r} \mapsto -$ and $\mathsf{r} \mapsto \mathsf{z}$ in the pre/postcondition respectively, and assigning the expected value to the source pointer variable (*i.e.*, to `r` in this case). This allows each pair of such identical heaplets to be *framed out* from the goal, as they will no longer be necessary to produce the rest of the program, simplifying the goal accordingly. For example:

$$\{\mathsf{r}, \mathsf{id}, \mathsf{bal}, \mathsf{z}\}\,; \{[\mathsf{z}, 2] * \mathsf{z} \mapsto - * \mathsf{z} + 1 \mapsto -\} \rightsquigarrow \{[\mathsf{z}, 2] * \mathsf{z} \mapsto \mathsf{id} * (\mathsf{z} + 1) \mapsto \mathsf{bal}\} \tag{5}$$

is the goal after removing the matching $\mathsf{r} \mapsto \mathsf{z}$ from the pre/postcondition. Therefore, after deriving the necessary "impedance-matching" assignments to the locations `z` and `(z + 1)` (lines 9 and 11) and framing out the corresponding assertions (as well as the administrative *block* assertion $[\mathsf{z}, 2]$) from the pre/postcondition, the goal will be reduced to a trivial one:

$$\{\mathsf{r}, \mathsf{id}, \mathsf{bal}, \mathsf{z}\}\,; \{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \tag{6}$$

This goal, which calls for transforming an empty symbolic heap `emp` to itself, can be satisfied without any code. Thus, it can be dismissed via a *terminal axiom* of the logic underlying the synthesis and emitting a program that does nothing.

Verifying `mk_acc()`. Let us now ensure that the synthesis result is indeed correct by utilising rules for SL-based symbolic execution (Berdine et al. 2005) to verify the obtained program. The even-numbered lines in Fig. 1 show intermediate assertions derived from the initial precondition by means of performing a forward-style (*a.k.a.* strongest-postcondition) symbolic execution of the program. While the correctness result indeed holds, one can notice that many of those assertions are quite different from the intermediate goals (4)–(6) from the deductive program derivation outlined above. Perhaps the most noticeable difference is the treatment of predicates in the postcondition. Specifically, a logic-based verifier keeps the instance $\mathsf{account}(x, \mathtt{id}, \mathtt{bal})$ "folded" with its existential variable x uninstantiated until the final symbolic state is produced at line 10, so the heap entailment

$$\{\mathtt{r} \mapsto \mathtt{z} * [\mathtt{z}, 2] * \mathtt{z} \mapsto \mathtt{id} * (\mathtt{z}+1) \mapsto \mathtt{bal}\} \vdash \{\mathtt{r} \mapsto x * \mathsf{account}(x, \mathtt{id}, \mathtt{bal})\} \tag{7}$$

could be discharged at the end of the proof. In contrast, the synthesis unfolds the predicate in the postcondition early to produce the subgoal (4), deriving the allocation command at line 3 of Fig. 1.

Most of the existing automated SL-based verifiers, such as Smallfoot (Berdine et al. 2006), VeriFast (Jacobs et al. 2011), HIP/SLEEK (Chin et al. 2011), and Viper (Müller et al. 2016), follow this strategy: execute the program symbolically, and then check a symbolic heap entailment. But in synthesis, there is *nothing to symbolically execute*: unlike a verifier, synthesis interleaves manipulations with pre- and postconditions to efficiently guide its search for a program—hence the mismatch.

3 OVERVIEW OF THE APPROACH

Now that we have shown deductive synthesis in action, let us present the main ideas for implementing a translation from synthesis derivations to proofs for a certifying back-end verifier in a uniform and modular way. For the remainder of the section, we will use a procedure for copying a pointer-based singly-linked list as our running example.

3.1 Background on Synthetic Separation Logic

We begin with a brief primer on (Cyclic) Synthetic Separation Logic (SSL) (Itzhaky et al. 2021)—the newly extended logical formalism underlying the deductive synthesis algorithm implemented by the current version of the SuSLik tool (Polikarpova and Sergey 2019). In SSL, a singly-linked list is defined by the following inductive predicate, a standard for Separation Logic (Reynolds 2002):

$$\begin{array}{rcl} \mathsf{sll}^{\alpha}(x, s) & \triangleq & x = 0 \land \{s = \emptyset; \mathsf{emp}\} \\ & \mid & x \neq 0 \land \{s = \{v\} \cup s_1 \land \beta < \alpha; [x, 2] * x \mapsto v * (x+1) \mapsto nxt * \mathsf{sll}^{\beta}(nxt, s_1)\} \end{array} \tag{8}$$

The two clauses in definition (8) correspond to the cases of an empty and non-empty list. In the former case, the "head" pointer of the list is null, and the heap allocated for the structure, as well as its payload, represented by the set s, are empty. In the latter case, the head pointer of the list x is non-empty, and the heap structure of the list is represented by two subsequent pointers, starting from x and storing a payload element v and the pointer nxt to the tail, which has the same structure, captured by the recursive occurrence of the same predicate sll. Following the choice made by the SuSLik implementation for more streamlined integration with third-party SMT solvers, we use mathematical *sets* to represent the linked list's payload instead of more traditional *algebraic lists*. We will elaborate on the consequences of this choice with regard to certification in Sec. 8.2. The only other unusual part of the predicate are the *cardinality variables* (α, β), as well as the constraints on them ($\beta < \alpha$), which are necessary to reason about termination of synthesised recursive programs and their auxiliary procedures via the mechanism of *cyclic proofs* (Rowe and Brotherston 2017).[2] For

[2]The exact usage of cyclic proofs for synthesis of provably terminating recursive programs is orthogonal to this work, and we refer the reader to the paper by Itzhaky et al. (2021) for the details. That said, cardinality variables will play an important role for the translation of SL predicates to the logics of Iris and VST, as we will discuss in Sec. 6.

the purpose of this work, one can think of cardinalities in inductive predicates as integer variables capturing the fact that the size of a heap, constrained by a recursive occurrence, is strictly smaller than that of the enclosing data structure, as in, *e.g.*, $\beta < \alpha$ in the predicate definition (8).

We can now specify a synthesis task for generating a copy of a singly-linked list as follows:

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}\ \mathtt{sll\_copy\ (r)}\ \{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\} \tag{9}$$

The procedure takes, as its parameter, a pointer variable r, which initially points to the head of the list to be copied, and at the end—to the head of a newly allocated list, holding the same payload s.[3]

Given a specification, SuSLik will generate a program in SusLang, a simple C-like language with pointers, function calls, and recursion, whose syntax is given in Fig. 2. SusLang values include booleans and integers, and a special type **loc** for pointer variables. Pointers are isomorphic to unsigned integers with a designated pointer constant, 0 (null). Expressions include variables, literal constants, equality checks and logical connectives. Additional *theory-specific expressions* are allowed depending on the underlying theory used for checking entailment in derivations; the most up-to-date version of SuSLik by Itzhaky et al. (2021) supports linear integer arithmetic and sets. The language allows pointer arithmetic in the form $x + \iota$.

Variable	x, y		Alpha-numeric identifiers
Size, offset	n, ι		Non-negative integers
Expression	e	::=	$0 \mid \mathsf{true} \mid x \mid e = e \mid e \wedge e \mid \neg e \mid d$
$\mathcal{T}$-expr.	d	::=	$n \mid x \mid d + d \mid n \cdot d \mid \{\} \mid \{d\} \mid \cdots$
Command	c	::=	$\mathtt{let}\ x = *(x + \iota) \mid *(x + \iota) = e \mid$ $\mathtt{let}\ x = \mathtt{malloc}(n) \mid \mathtt{free}(x) \mid \mathtt{err} \mid$ $f(\overline{e_i}) \mid c; c \mid \mathtt{if}\ (e)\ \{c\}\ \mathtt{else}\ \{c\}$

Fig. 2. SusLang syntax.

To synthesise an implementation of sll_copy in SusLang, the specification (9) is first transformed to a synthesis goal of the form $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$, where Γ is the set of currently available program-level and logical variables (in our example, it is initially just $\{\mathtt{r}, x, s, y\}$); $\{\mathcal{P}\}$ and $\{\mathcal{Q}\}$ are the corresponding ascribed pre- and postconditions; and c is an unknown program, yet to be synthesised. In general, both the pre- and postcondition of specifications and goals can feature a *pure* and spatial part, *e.g.*, $\{\mathcal{P}\} = \{\phi, P\}$. The pure part ϕ captures the logical constraints on variables and values involved in the specification. The spatial part P describes the heap shape using standard SL assertions, joined by the separating conjunction connective ($*$): emp for an empty heap, $(x + \iota) \mapsto e$ for an individual address x storing a value e at a (possibly zero) offset ι, a *block assertion* $[x, n]$ for a continuous segment of n elements starting at x, which can be deallocated, and $\mathsf{p}^{\alpha}(\overline{t_i})$ for a heap of size α (frequently omitted) described by an occurrence of a predicate p with arguments $\overline{t_i}$.

When searching for an implementation, the synthesis tries to apply the rules of SSL, building a derivation for the initial goal. Such a derivation will also contain the desired program c as its byproduct. Fig. 3 provides the selected rules. Within rules we use lower latin letters x, y for program variables (taken from the set ProgVars), e, t for program-level terms (of the syntactic class e in Fig. 2), Greek letters ν, ω for logical variables, and ϕ, ψ, χ for logical formulas. Assertions are interpreted in an environment Γ in which some of the variables are universally quantified and others existentially quantified, with a prefix of the form $\forall \overline{x}.\exists \overline{y}$. Program variables are *always* included in the universal prefix. Logical variables are split between universal (also called *ghost* variables and denoted GV(Γ)) and existential (EV(Γ)). We denote Vars(Γ) = $\{\overline{x}, \overline{y}\}$ for all quantified variables. The $\cup$ operator, used in some of the rules, joins two environments, so in the resulting environment the quantifiers follow the same $\forall.\exists$ quantifier pattern. We use $e[\Gamma]$ for the set of all expressions that can be constructed using program variables in Γ, and $\kappa[\Gamma]$—all logical terms that can be constructed with any variables from Γ. Finally, $[\sigma]P$ denotes an application of a substitution σ to all variables in a formula P.

[3] As we use sets for payloads, the specification (9) for singly-linked list copying is a bit more *loose* than an ideal one: for instance, it admits programs that drop duplicated elements from the copy of the list. That said, programs that abuse the specification in this or a similar way would be much more difficult to discover than a "morally correct" copying procedure.

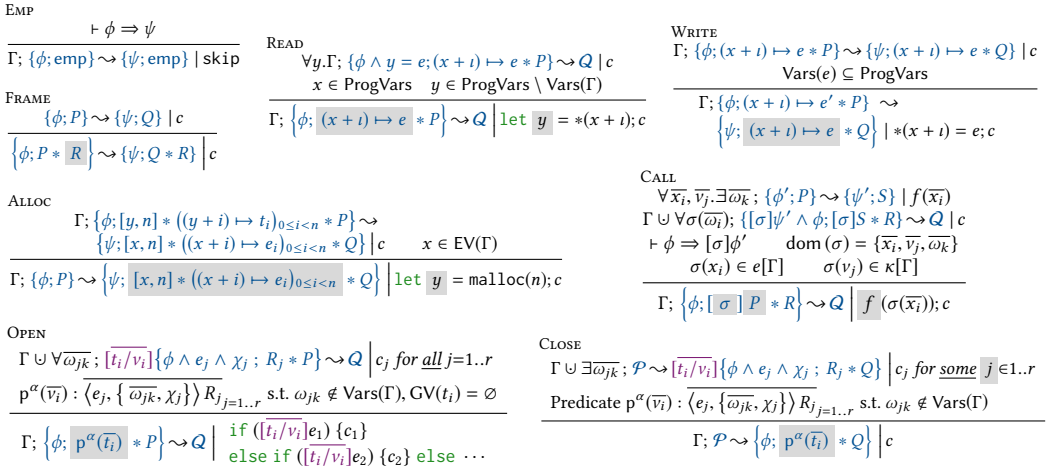

Fig. 3. Selected declarative rules of SSL. The choice of greyed parts is determined by a proof search strategy.

Most of the rules in Fig. 3 (Read, Write, Alloc, Open, and Call) are *operational*: when read bottom-up, they advance the synthesis by emitting parts of the program and reducing its goal to their premises. Perhaps the most interesting of those is the Call rule, which synthesises procedure calls. The rule combines SL-style framing, with R as the frame, and substitution of actual into formal parameters via σ, which is also applied to the procedure f's postcondition. Existential variables in the procedure's environment are renamed to fresh ghost variables in the second premise of the rule. Formal parameters x_i of f are mapped to program expressions $e[\Gamma]$ using program variables of Γ, and ghosts ν_j are mapped to logical terms κ using any variables of Γ. The rule Emp is a terminal one, and, when applied, corresponds to a successful synthesis of a program branch: empty heaps in both pre/postconditions mean that there is nothing more for a program to do, assuming the constraints $\phi \Rightarrow \psi$ accumulated in pure parts hold. Finally, the rules Frame and Close are structural ones: they do not emit a part of the program but rather change the shape of the goal, possibly making other rules applicable. For instance, Frame removes similar parts of the symbolic heap from the pre/postcondition, eventually enabling Emp, while Close unfolds a predicate instance in the goal's postcondition, replacing its occurrence $\mathrm{p}^{\alpha}(\overline{t_i})$ by p's j^{th} clause (for some j), thus revealing more information about the structure of the expected final heap and potentially enabling Alloc.

The left part of Fig. 4 shows an implementation of `sll_copy` in SuSLang synthesised via SSL rules as a byproduct of deriving a proof for specification (9).

3.2 From Proof Derivations to Proof Trees

The presentation of the rules in Fig. 3 is declarative and, thus, is intentionally non-deterministic: it does not immediately provide an exact synthesis *algorithm* that tries to apply the rules in a certain order. In fact, such an order can be determined by multiple decisions: for instance, which *heaplet* of the form $(x+\iota) \mapsto e$ is chosen for an application of a rule Read in a precondition of a current goal, or a call to what procedure f has been synthesised by applying Call. Indeed, an application order required for a valid derivation is exactly what a synthesis algorithm aims to discover. We refer the reader to the existing work on deductive program synthesis for a survey of existing techniques for finding valid derivations efficiently (Itzhaky et al. 2021; Kneuss et al. 2013).

Let us discuss how to encode a valid SSL derivation, once it is built by the synthesis algorithm. The greyed fragments in Fig. 3 (on the previous page) indicate, for each rule, in its conclusion, the

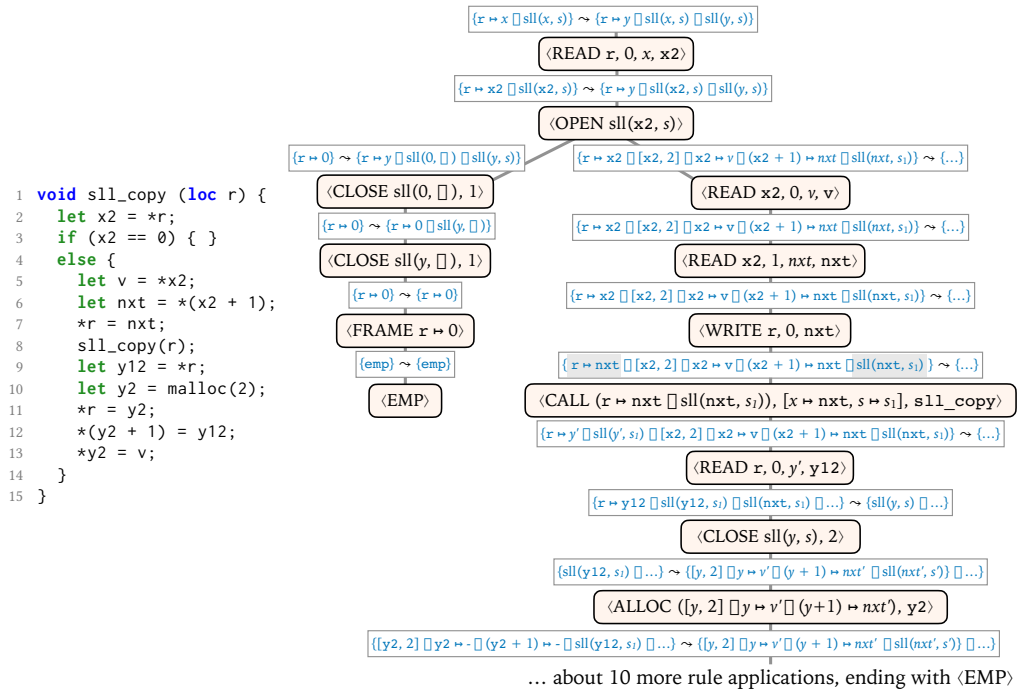

Fig. 4. Singly-linked list copying in SuSLik: synthesised code (left) and simplified proof tree (right).

components of the pre- and postcondition of the goal (and in the case of Close, in the rule's premise, a clause index) that need to be known in order to determine *how exactly* the rule's application has reduced a goal to concrete subgoals. In other words, knowing a tree of rule applications (*i.e.*, a *proof tree*) rooted in the initial goal, as well as the greyed fragments for each of the rule applications in it, makes it possible to fully restore the synthesis derivation for that goal and also the program that was generated. We can encode SSL proof trees as values of the following recursive data type $\tau_{\mathbf{ssl}}$:

$$\begin{array}{ll} \text{ProofTree } (\text{Step}_{\mathbf{ssl}}) & \tau_{\mathbf{ssl}} ::= \langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle \\ \text{Step}_{\mathbf{ssl}} & \mathcal{S}_{\mathbf{ssl}} ::= \langle \text{READ}, x, \iota, e, \mathrm{y} \rangle \mid \langle \text{CALL}, P, \sigma, f \rangle \mid \left\langle \text{OPEN}, \mathrm{p}^{\alpha}(\overline{t_i}) \right\rangle \mid \left\langle \text{CLOSE}, \mathrm{p}^{\alpha}(\overline{t_i}), j \right\rangle \mid \ldots \end{array} \tag{10}$$

A node in a proof tree $\tau_{\mathbf{ssl}}$ is a pair, composed of a proof step and a list of the node's children (an empty list means an application of a terminal rule, *e.g.*, Emp). We define a proof tree so it is parametric in the payload of its nodes; in the case of SSL, this payload is the individual proof steps of the synthesis logic ($\text{Step}_{\mathbf{ssl}}$). SSL proof steps $\mathcal{S}_{\mathbf{ssl}}$ couple the name of the applied rule with the components of the goal in its conclusion and other necessary parameters that control rule application non-determinism. The right part of Fig. 4 shows an example of a (simplified) valid proof tree for deriving an implementation of `sll_copy`, whose nodes contain concrete proof steps.[4] For the sake of demonstration, the tree in Fig. 4 also shows how applying the proof rules in the tree's nodes transforms the synthesis goal. For example, the left branch of the tree shows the derivation of the `then`-branch of `sll_copy`, which replaces an instance $\mathsf{sll}(\mathtt{x2}, s)$ with its first clause, thus deducing $\mathtt{x2} = 0 \land s = \emptyset$ 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 (8).

[4]We omit the synthesised program part "... | c" from the tree, as we will see shortly how it can be recovered from it.

The remaining derivation is completed via an application of the Frame and Emp rules. The most interesting part in the right branch is an application of the Call rule, which is "triggered" by the symbolic heap $\mathsf{r} \mapsto \mathsf{nxt} * \mathsf{sll}(\mathsf{nxt}, s_1)$ in the precondition (highlighted by grey boxes), as captured by the corresponding proof step.

Recovering the program from a proof tree. With an initial goal and its valid proof tree at hand, we can now easily generate the synthesised program by defining the following proof tree *evaluator*:

$$\begin{array}{lcll}
\mathcal{E}_{\mathbf{synt}} : \mathsf{Goal} \times \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{ssl}}) \to \mathsf{Prog} & & & \\
\mathcal{E}_{\mathbf{synt}}\ (\mathcal{G}, \langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle) & \triangleq & \mathbf{let}\ \left(\overline{\mathcal{G}}, k\right) & = \mathcal{I}_{\mathbf{synt}}\ \mathcal{S}_{\mathbf{ssl}}\ \mathcal{G} \qquad \mathbf{in} \\
& & \mathbf{let}\ \overline{c} & = \mathsf{map}\ \mathcal{E}_{\mathbf{synt}}\ (\mathsf{zip}\ \overline{\mathcal{G}}\ \overline{\tau_{\mathbf{ssl}}})\ \mathbf{in} \\
& & k\ \overline{c} & \\
\text{where} & & & \\
\mathcal{I}_{\mathbf{synt}} : \mathsf{Step}_{\mathbf{ssl}} \to \mathsf{Goal} \to \mathsf{Goal}^* \times (\mathsf{Prog}^* \to \mathsf{Prog}) & & &
\end{array} \tag{11}$$

In the definition above, the source proof tree evaluator takes a tree node and a goal and applies a *proof step interpreter* $\mathcal{I}_{\mathbf{synt}}$ to the proof step in the node and the goal. The result of this application is a sequence of subgoals $\overline{\mathcal{G}}$ and a program-constructing function k, whose arity matches the length of $\overline{\mathcal{G}}$. The evaluator then proceeds to generate the residual programs by processing the sub-goals $\overline{\mathcal{G}}$ with the corresponding subtrees $\overline{\tau_{\mathbf{ssl}}}$ (the lengths of the two sequences are assumed to be the same, which is the case for valid proof trees). Finally, it applies k to the resulting residual programs $\overline{c}$, obtaining the result. The type of $\mathcal{I}_{\mathbf{synt}}$ is indicative of its logic. As an example, this how it is implemented for the proof steps corresponding to applications of Emp and Read:

$$\begin{array}{lll}
\mathcal{I}_{\mathbf{synt}}\ \langle \mathsf{EMP} \rangle & _ & \triangleq ([\,], \lambda\,[\,].\mathsf{skip}) \\
\mathcal{I}_{\mathbf{synt}}\ \langle \mathsf{READ}, x, \iota, e, \mathsf{y} \rangle & (\Gamma; \{\phi; (x+\iota) \mapsto e * P\} \rightsquigarrow Q) \triangleq & \\
\quad ([\forall y.\Gamma; \{\phi \land y = e; (x+\iota) \mapsto e * P\} \rightsquigarrow Q], \lambda\,[c].\mathtt{let}\ \mathsf{y} = *(x+\iota); c) & &
\end{array}$$

That is, when applied to $\langle \mathsf{EMP} \rangle$ and any goal, $\mathcal{I}_{\mathbf{synt}}$ emits an empty list of subgoals to derive, as well as a 0-arity function (*i.e.*, a constant), which returns the program `skip`. For $\langle \mathsf{READ}, x, \iota, e, \mathsf{y} \rangle$, the step interpreter returns a single modified goal, as well as a function that prepends the read-operation $\mathtt{let}\ \mathsf{y} = *(x+\iota)$ to the residual program. Looking at the remaining rules in Fig. 3, it is easy to see that the arity of the function returned as a second component of $\mathcal{I}_{\mathbf{synt}}$'s result matches the number of subgoals in its first component and in the premise of the corresponding rule. Let us remark that 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 Towards Generating Target Proofs from Synthesis Proof Trees

Having seen how one can evaluate a synthesis proof tree to a resulting program, it is natural to attempt to use a similar approach to evaluate it to *another* proof tree that can be rendered into a proof script in a certified verifier. As a tentative certification target, let us take Hoare Type Theory (HTT) by Nanevski et al. (2010)—a foundational implementation in Coq of a Separation Logic for an idealised C-like language with higher-order functions. The left part of Fig. 5 shows an encoding of the singly-linked list predicate (8), specification (9) and an implementation of `sll_copy` from Fig. 4 in HTT. Leaving the explanation of most of the intricacies of HTT until Sec. 5, let us point out a few differences between logical assertions in the language of SSL and those of HTT. First, as a way to enable reasoning by reflection in Coq, HTT's SL-style assertions mention heaps *explicitly*, making them the subject of constraints that define their shapes. In this setting, SL's separating conjunction ($*$) is encoded as a disjoint union ($\bullet$) of explicit symbolic heaps, which can be additionally constrained via inductive predicates (as in, *e.g.*, `sll nxt s1 h1` at line 6 of Fig. 5).

```
1  (* Inductive heap predicate for SL lists *)
2  Inductive sll (x : ptr) s h : Prop :=
3  | sll_1 of x == null of s = [::] ∧ h = Unit
4  | sll_2 of (x == null) = false of
5    ∃ (v : nat) s1 nxt h1, s = v :: s1 ∧
6    h = x ↦ v • x+1 ↦ nxt • h1 ∧ sll nxt s1 h1.
7
8  (* Specification for SLL copying *)
9  Definition sll_copy_spec :=
10   ∀ (r: ptr), {(vghosts : ptr * seq nat)},
11   STsep(
12     (* Precondition *)
13     fun h => let: (x, s) := vghosts in
14     ∃ h1, h = r ↦ x • h1 ∧ sll x s h1,
15   [ (* Postcondition *)
16     vfun (_: unit) h =>
17     let: (x, s) := vghosts in
18       ∃ y h1 h2, h = r ↦ y • h1 • h2 ∧
19                 sll x s h1 ∧ sll y s h2]).
20
21 (* SLL copying implementation *)
22 Program Definition sll_copy : sll_copy_spec :=
23   Fix (fun (sll_copy : sll_copy_spec) r => Do (
24     x2 <-- @read ptr r;
25     if x2 == null
26     then ret tt (* return unit *)
27     else
28       v <-- @read nat x2;
29       nxt <-- @read ptr (x2+1);
30       r ::= nxt;;
31       sll_copy r;;
32       y12 <-- @read ptr r;
33       y2 <-- allocb null 2;
34       r ::= y2;;
35       (y2+1) ::= y12;;
36       y2 ::= v;;
37       ret tt)).
```

```
38 Next Obligation.
39 (* Initialise HTT proof context *)
40   apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
41 (* Read *) apply: bnd_readR=>/=.
42 (* Open (unfold) SLL instance in the precondition *)
43   case: Hsll; case: ifP; move=>IfCond//_;
44   [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
45   (* Case: empty list (x2 = 0) *)
46   - move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
47     (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
48     (* Close (unfold) SLL instance in postcondition *)
49       repeat split=>//=; do?[hhauto|constructor 1].
50   (* Case: non-empty list *)
51   - (* Read *) apply: bnd_readR=>//=.
52     (* Read *) apply: bnd_readR=>//=.
53     (* Write *) apply: bnd_writeR=>//=.
54     (* Call *)
55       rewrite (joinC _ h1) joinA; apply: bnd_seq.
56       apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
57       ∃ h1; split=>//=.
58       move=>h_call [y12][h11][h21][->][H2 H3]_.
59     (* Read *) apply: bnd_readR=>//=.
60     (* Alloc *) apply: bnd_allocbR=>y2//=.
61     (* Write *) apply: bnd_writeR=>//=.
62     (* Write *) apply: bnd_writeR=>//=.
63     (* Write *) apply: bnd_writeR=>//=.
64     (* Emp *)
65       apply: val_ret; rewrite defPtUn0; case/andP=>?.
66       ∃ y2, (x2 ↦ v • (x2+1) ↦ nxt • h11),
67             (y2 ↦ v • (y2+1) ↦ y12 • h21).
68     repeat split=>//=; first by hhauto.
69    + (* Close SLL instance 1 in postcondition *)
70       by constructor 2=>//=; ∃ v, s1, nxt, h11.
71    + (* Close SLL instance 2 in postcondition *)
72       constructor 2=>//=; first by apply negbTE.
73       by ∃ v, s1, y12, h21.
74 Qed.
```

Fig. 5. Copying a singly-linked list in HTT/Coq: definitions and specification (left), and proof (right).

Specifications distinguish between program-level variables (*e.g.*, `r` at line 10) and *ghost* (*i.e.*, logical) ones, with the latter passed as a single tuple (*cf.* `vghosts` at the same line). Similarly to SusLang, the language of HTT supports recursion via an explicit `Fix` operator, but requires the specification of a recursive function (*i.e.*, its *Hoare type*) to be declared explicitly, as shown at line 23 of Fig. 5.

The right part of Fig. 5 shows a hand-crafted proof script for verifying the implementation of `sll_copy` against its specification `sll_copy_spec`. The verification conditions in HTT are discharged by a forward-style symbolic execution, which is driven by the goal's precondition and the program structure, while maintaining assumptions about the shape of the heap and relations between symbolic values in Coq's native proof context and performing, when required, global substitutions in both the entire goal and the proof context. Without going any further into the details of HTT-powered verification, let us notice that the hierarchy of the proof (highlighted by offsets and bullet separators - and +) is similar to the structure of the proof tree in Fig. 4, and the proof's individual steps loosely resemble the corresponding steps in SSL proof tree, as indicated in the comments. Following this similarity, we start by defining the data type for HTT proof trees as follows:

$$\begin{array}{lll} \text{ProofTree}\,(\text{Step}_{\mathbf{htt}}) & \tau_{\mathbf{htt}} & ::= \langle \mathcal{S}_{\mathbf{htt}}, \overline{\tau_{\mathbf{htt}}} \rangle \\ \text{Step}_{\mathbf{htt}} & \mathcal{S}_{\mathbf{htt}} & ::= \text{a sequence of HTT/Coq tactic applications} \end{array}$$

Let us now define an evaluator, similar to (11), but for translating source trees to target trees:

$$\begin{aligned}
&\mathcal{E}_{\mathbf{htt}} : \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{ssl}}) \to \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{htt}}) \\
&\mathcal{E}_{\mathbf{htt}}\ \langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle \triangleq \mathbf{let}\ \mathcal{S}_{\mathbf{htt}} = \mathcal{I}_{\mathbf{htt}}\ \mathcal{S}_{\mathbf{ssl}}\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ \overline{\tau_{\mathbf{htt}}} = \mathrm{map}\ \mathcal{E}_{\mathbf{htt}}\ \overline{\tau_{\mathbf{ssl}}}\ \mathbf{in} \\
&\qquad\qquad \langle \mathcal{S}_{\mathbf{htt}}, \overline{\tau_{\mathbf{htt}}} \rangle \\
&\text{where} \\
&\mathcal{I}_{\mathbf{htt}} : \mathsf{Step}_{\mathbf{ssl}} \to \mathsf{Step}_{\mathbf{htt}}
\end{aligned} \tag{12}$$

HTT employs advanced proof automation to manage the proof context and the goal for each step of the forward symbolic execution, driven by the verification goal precondition and the program. For instance, the proof step advancing the symbolic execution past the `read` statement at line 24 of Fig. 5 corresponds to line 41 of the proof. With this in mind, we can attempt to start defining our proof step interpreter as follows:

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

The reason why the right-hand side of the definition above does not utilise any of the components of the source step is because the information conveyed by the program statement (*i.e.*, the location from which the read is performed) is sufficient for the verifier's automation to determine the remaining parts of the goal that need to be checked and transformed accordingly.

3.4 Contexts for Tracking Dependencies in Target Proofs

Consider the state of the HTT proof of `sll_copy` after executing the verification script from Fig. 5 up to line 43, as rendered in the display below.

```
r, x2 : ptr
h' : heap
Hsll : sll x2 s h'
============================================================================
verify (r ↦ x2 • h') (if x2 == null then ret tt else v <-- read x2; ...)
  [vfun _ h => ∃ y h1 h2, h = r ↦ y • h1 • h2 ∧ sll x2 s h1 ∧ sll y s h2]
```

HTT collects assumptions about the shape of the heap as named hypotheses in the native Coq context. Therefore, the Coq proof context (above the `====` line) will contain a hypothesis named `Hsll` asserting that the symbolic heap `h'` featured in the goal's precondition `r ↦ x2 • h'` is a singly-linked list starting at `x2` that has the sequence `s` as its payload (*i.e.*, `sll x2 s h'`). The proof script steps at lines 43 and 44 mimic the effect of the source step $\langle \mathsf{OPEN}, \mathsf{sll}(\mathsf{x2}, s) \rangle$ from the source proof tree (*cf.* Fig. 4), creating two subgoals with the preconditions adapted according to the two clauses of the predicate definition (8). It is easy to notice that nothing in the source step indicates that the predicate occurrence sll(x2, s) to be unfolded in the source poof corresponds to the Coq hypothesis named `Hsll` from the target proof: this knowledge is *backend-specific*.

To accommodate this and similar scenarios requiring the proof translator to keep track of definitions and dependencies between components in the source and target proofs, which are paramount in our case studies, we enhance our proof evaluator from Sec. 3.3, making it aware of the backend-specific proof context. The modified definition is now as follows:

$$\begin{aligned}
&\mathcal{E}_{\mathbf{htt}} : \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{ssl}}) \times \mathsf{Context}_{\mathbf{htt}} \to \mathsf{ProofTree}\ (\mathsf{Step}_{\mathbf{htt}}) \\
&\mathcal{E}_{\mathbf{htt}}\ (\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle, \mathit{ctx}) \triangleq \mathbf{let}\ (\mathcal{S}_{\mathbf{htt}}, \overline{\mathit{ctx}}) = \mathcal{I}_{\mathbf{htt}}\ \mathcal{S}_{\mathbf{ssl}}\ \mathit{ctx}\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ \overline{\tau_{\mathbf{htt}}} = \mathrm{map}\ \mathcal{E}_{\mathbf{htt}}\ (\mathrm{zip}\ \overline{\tau_{\mathbf{ssl}}}\ \overline{\mathit{ctx}})\ \mathbf{in} \\
&\qquad\qquad \langle \mathcal{S}_{\mathbf{htt}}, \overline{\tau_{\mathbf{htt}}} \rangle \\
&\text{where} \\
&\mathcal{I}_{\mathbf{htt}} : \mathsf{Step}_{\mathbf{ssl}} \to \mathsf{Context}_{\mathbf{htt}} \to \mathsf{Step}_{\mathbf{htt}} \times \mathsf{Context}^{*}_{\mathbf{htt}}
\end{aligned} \tag{13}$$

As the type of the new version of $\mathcal{I}_{\mathbf{htt}}$ indicates, the proof step interpreter now takes a target-specific proof context and a source step, and emits a target step and a sequence of proof contexts, the length of which matches the number of subgoals of the SSL rule corresponding to the source step. As an example, let us sketch the HTT-specific logic of $\mathcal{I}_{\mathbf{htt}}$ on the OPEN step from the proof tree in Fig. 4:

$$\mathcal{I}_{\mathbf{htt}}\ \langle \text{OPEN sll}(\texttt{x2}, s)\rangle\ \ ctx = \left(\left[\begin{array}{l}\texttt{case: Hsll; case: ifP; move=>IfCond//\_;}\\ \texttt{[move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].}\end{array}\right], [ctx, ctx']\right)$$

where $\texttt{Hsll} = ctx(\text{sll}(\texttt{x2}, s))$

$ctx' = ctx \cup [\text{sll}(\texttt{nxt}, s_1) \mapsto \texttt{H1}]$

As the display above shows, the generated proof steps in HTT perform case analysis on the Coq hypothesis named `Hsll`, which corresponds to unfolding the predicate occurrence $\text{sll}(\texttt{x2}, s)$ in the goal's precondition performed by the OPEN step in the source proof in Fig. 4. The witness `Hsll` for the occurrence has been recorded upon initialisation of the proof context *ctx* at the beginning of the target proof generation (*cf.* line 40 of Fig. 5), and it is retrieved when translating the OPEN step. Notice that the interpreter $\mathcal{I}_{\mathbf{htt}}$ generates two *different* proof contexts for the two subtrees. The latter one will be used for a "non-empty list" branch (lines 50–73), so it is updated with an entry $[\text{sll}(\texttt{nxt}, s_1) \mapsto \texttt{H1}]$. This new entry indicates that the predicate occurrence $\text{sll}(\texttt{nxt}, s_1)$ in the source proof is described by the hypothesis `H1 : sll nxt s1 h1` in the target proof. This piece of knowledge will become essential when generating HTT proof steps corresponding to the source step $\langle \text{CALL sll}(\texttt{nxt}, \texttt{s}_1),\ [x \mapsto \texttt{nxt}, s \mapsto s_1],\ \texttt{sll\_copy}\rangle$. Specifically, it will allow the target script to correctly use the heap `h1` (such that `sll nxt s1 h1` holds) as an existential witness at line 57 of Fig. 5 to satisfy the precondition of the recursive call to `sll_copy r` at line 31 of the implementation.

Expectedly, the mapping from source predicate occurrences to the target proof hypothesis is not the only piece of information that needs to be tracked for proof generation. We will elaborate on other components of HTT-specific proof contexts and their initialisation in Sec. 5.

3.5 Deferring Target Proof Steps

Consider lines 66-67, close to the end of the proof script in Fig. 5. 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 (9), as well as two heaps representing the linked lists: the original one $\text{sll}(\texttt{x2}, s)$ and the new one $\text{sll}(y, s)$, thus proving the postcondition by means of symbolic heap entailment. Constructing this target HTT proof step from the source proof is challenging. As the proof tree in Fig. 4 (right) shows, the information about the shape of the heap constrained by, *e.g.*, $\text{sll}(y, s)$ in the postcondition, is obtained *much earlier* (*wrt.* to the synthesised/verified program) in the source proof, by an application of CLOSE at the bottom of the right branch, preceding the synthesis of the `malloc` statement (via ALLOC) in the code shown in Fig. 4 (left). 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.

A careful reader may notice that this scenario is complicated even more by the fact that, by the time we need to provide the witness heap revealed by the CLOSE step 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. We address this challenge in the final version of our generic proof evaluator, defined in Fig. 6.

The evaluator $\mathcal{E}_{\mathbf{t}}$ is now parameterised by a target backend $\mathbf{t}$ (*e.g.*, HTT), which provides the definitions of a proof context $\text{Context}_{\mathbf{t}}$, proof step $\text{Step}_{\mathbf{t}}$, and source step interpreter $\mathcal{I}_{\mathbf{t}}$. As a minor

$$
\begin{array}{lll}
\mathcal{E}_{\mathbf{t}} : \mathsf{ProofTree}\,(\mathsf{Step}_{\mathbf{ssl}}) \times \mathsf{Context}_{\mathbf{t}} \times \mathsf{DeferredStep}_{\mathbf{t}} \to \mathsf{ProofTree}\,(\mathsf{Step}^*_{\mathbf{t}}) & & \\
\mathcal{E}_{\mathbf{t}}\,(\langle \mathcal{S}_{\mathbf{ssl}}, \overline{\tau_{\mathbf{ssl}}} \rangle, ctx, \mathcal{D}) \triangleq \mathbf{let}\ (\overline{\mathcal{S}_{\mathbf{t}}}, \overline{ctx}, \mathcal{D}') = \mathcal{I}_{\mathbf{t}}\ \mathcal{S}_{\mathbf{ssl}}\ ctx & & \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \mathcal{D}'' = \lambda ctx.\,((\mathcal{D}\ ctx) +\!\!+ (\mathcal{D}'\ ctx)) & & \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \overline{\tau_{\mathbf{t}}} = \mathsf{map}\ \mathcal{E}_{\mathbf{t}}\ (\mathsf{zip3}\ \overline{\tau_{\mathbf{ssl}}}\ \overline{ctx}\ (\mathsf{repeat}\ |\overline{\tau_{\mathbf{ssl}}}|\ \mathcal{D}'')) & & \mathbf{in} \\
\qquad\qquad \mathbf{if}\ |\overline{\tau_{\mathbf{t}}}| = 0\ \mathbf{then}\ \left\langle \overline{\mathcal{S}_{\mathbf{t}}} +\!\!+ (\mathcal{D}''\ ctx), [] \right\rangle\ \mathbf{else}\ \left\langle \overline{\mathcal{S}_{\mathbf{t}}}, \overline{\tau_{\mathbf{t}}} \right\rangle & &
\end{array}
$$

where

$$
\begin{array}{lll}
\mathcal{I}_{\mathbf{t}} & : & \mathsf{Step}_{\mathbf{ssl}} \to \mathsf{Context}_{\mathbf{t}} \to \mathsf{Step}^*_{\mathbf{t}} \times \mathsf{Context}^*_{\mathbf{t}} \times \mathsf{DeferredStep}_{\mathbf{t}} \\
\mathsf{DeferredStep}_{\mathbf{t}} & \triangleq & \mathsf{Context}_{\mathbf{t}} \to \mathsf{Step}^*_{\mathbf{t}}
\end{array}
$$

Fig. 6. Generic proof evaluator.

change to simplify the implementation, the payload in the target proof tree is now made to be a sequence of target steps rather than an individual step. The most significant change comes in a new parameter $\mathcal{D}$ of the evaluator—so-called *deferred proof steps*. As the type $\mathsf{DeferredStep}_{\mathbf{t}}$ 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 $\mathcal{I}_{\mathbf{t}}$) 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 $|\overline{\tau_{\mathbf{t}}}| = 0$), 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. 5, we will elaborate on a concrete scenario of using deferred steps in proof translation to implement late entailment checking in the generated HTT certificates for SSL proofs.

4 A FRAMEWORK FOR TRANSLATING SYNTHESIS PROOFS

In this section, we focus on the intricacies of implementing the general evaluator infrastructure for SuSLik, as well as some shared implementation challenges for all Coq-based backends.[5]

4.1 Constructing Proof Trees

SuSLik's deductive approach means that the sequence of proof steps needed to verify the constructed program is inherent in the synthesis itself. However, because SuSLik's original implementation was geared more towards synthesising the program and less towards accumulating the intermediate deductive steps in a recoverable format, we had to modify the synthesis procedure to more readily capture the run-time information needed to generate proofs.

A synthesis goal may be solved by any number of candidate rule applications, which in turn generate any number of subgoals. SuSLik uses an And/Or tree (Martelli and Montanari 1973) 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.*, candidate rule applications) 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. 3) we capture the names of the operation's source and

[5] A detailed overview of the framework is available in the first author's MSc thesis (Watanabe 2021).

```
// 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]
}
```

Fig. 7. Scala encoding of main components of evaluator for SSL proof trees in SuSLik.

destination variables, along with the name of the ghost variable that was instantiated by the read, as in definition (10). By collecting these nodes, we capture all paths explored in the set of possible rule application sequences, including those that failed.

In addition to capturing information from each step in the proof search, we also discard the steps that failed, keeping only those that contributed to the final synthesised 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 out of all candidates). We translate this to our source proof tree structure, collapsing these And/Or node pairs into single nodes.

The resulting tree is a compact representation of the steps taken to derive the program, with each node storing an instance of the datatypes (10) that contain proof information.

4.2 Common Elements of Backend-Specific Translators

SuSLik is implemented in Scala, and so is our certification framework for it. Fig. 7 shows our encoding of the type signatures for the main components of the translator from definition in Fig. 6. Our implementation provides concrete definitions of the `ProofTree` data type and the generic `Evaluator`. The clients of the framework are responsible for implementing the backend-specific components for generating a target proof `ProofTree[T]`, *i.e.*, a proof `Context[T]`, and an `Interpreter` instance. In the rest of this section, we highlight some implementation insights we observed consistently across all the backends and have factored out to common libraries. We focus on instrumenting program and proof generation; predicate definitions and program specifications are straightforwardly translated from their representations during the synthesis phase.

4.2.1 Translating Programs. For producing programs in the syntax of each verifier backend, our first instinct was to translate the synthesised SusLang programs into the desired syntax. However, we ultimately found it more reliable to reuse the infrastructure we developed for proof generation, that is, defining a *program-emitting proof interpreter*. This is a generalisation of an earlier technique we saw in Sec. 3.2; just as we showed how a SusLang program can be recovered from the proof tree, we can do the same for programs expressed in alternative backend-specific languages.

```
// 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)
```

Fig. 8. Two alternative translations of the Free rule.

Consider the Free rule applied to a memory block of size n referenced by a pointer x. While this corresponds to the SusLang statement free(x) (which makes no reference to n), in the language of HTT it corresponds to a *sequence* of n dealloaction statements. The difference is that memory is freed in SusLang at the level of entire allocated blocks, while HTT's program language does so at the level of individual cells. This example demonstrates that one cannot simply generate, *e.g.*, HTT programs simply by translating the SusLang program. Rather, the proof tree should be used as the source of the additional information, as shown in Fig. 8. Note how the source proof representation is more descriptive than either of the target representations. By generating both program and proof from the same source in this way, we ensure they are structurally aligned, and avoid relying on a specialised artefact of the synthesis implementation.

4.2.2 Backend-Specific Tactic Libraries. Since all three backends we examined are based on some version of Separation Logic, most of the operational SSL rules (*cf.* Read, Write, Alloc, Free in Fig. 3) have counterpart Coq tactics in the backend frameworks that we can map to directly when implementing the interpreter. However, the correspondence is not always so straightforward. For some rules, context-dependent preparation and clean-up is required before and after invoking the main lemma. We deal with these discrepancies by writing auxiliary Coq tactics in Ltac, Coq's Turing-complete tactic language (Delahaye 2000), to automate the pre/post-processing steps uniformly. Take the Call rule application in Fig. 5, whose HTT implementation starts in the following way:

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

The sequence of rewrites before the application of bnd_seq (which is the HTT implementation of the rule for decomposing sequential composition) rearranges 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, but the suitable rewrite sequence depends on each call instance. 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, via a tactic that applies this 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 interpreter need only invoke ssl_call_pre with the desired subheap as an argument. For each target backend, we follow this approach and design similar tactics for the relevant rules to create a small library. This collection of "wrapper tactics" serves as an interface to the underlying target framework that is tailored to the way SSL reasons about programs, leading to more predictable behaviour and readable proof scripts.

4.2.3 Discharging Pure Facts from SL Proofs. When performing search 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. 3) by invoking an SMT solver, which acts as a *validity oracle*. To use a fact returned by the oracle in the Coq proof, we first capture the entailment in the source proof tree, and decompose it so that, for every postcondition conjunct C, we have one entailment from the set of all precondition conjuncts whose variable set intersects with C's. Fig. 9 shows one such lemma, derived from an oracle-validated pure entailment from one of our benchmarks. 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 (Czajka and Kaliszyk 2018), Micromega (Besson and Makarov 2020), or SMTCoq (Ekici et al. 2017). For example, CoqHammer can solve the lemma from Fig. 9:

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

Fig. 9. A pure entailment lemma.

```
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. 8.2 discusses our handling of the lemmas in more advanced cases when CoqHammer 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_example: 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`. We similarly use the hint database to aid our automation tactics in Iris (*cf.* Sec. 6) and VST (*cf.* Sec. 7), by adding framework-specific assumptions, such as predicate inversion lemmas and valid integer ranges.

5 SYNTHESIS CERTIFICATION VIA HOARE TYPE THEORY

Our first concrete case study in certifying synthesised programs is via Hoare Type Theory (HTT) by Nanevski et al. (2010), a framework that represents SL specifications in terms of dependently-typed indexed monads (Nanevski et al. 2008).

5.1 HTT-Style Embedding of Separation Logic and Its Advantages

The task of defining a proof interpreter can be framed as one of bridging the *impedance* between SuSLik's mode of reasoning and that of the target backend. Naturally, the less impedance, the lower the implementation cost. In our experience, the primary cause of such impedance is the chosen style of SL embedding into Coq. For instance, both Iris and VST adopt a so-called *shallow* embedding of SL propositions, defining the corresponding data types (called `iProp` in Iris and `mpred` in VST) for their semantic models and encoding SL connectives as functions on those data types. In both Iris and VST, the object languages are *deeply* embedded: programs being verified are represented via instances of custom AST data types rather than as programs in Gallina, Coq's own language.

In contrast, HTT adopts a more lightweight embedding style, encoding SL propositions *directly in terms of Coq's propositions* of the sort `Prop`, thus reusing most of Coq's logical connectives. The HTT-style embedding lets us represent SSL predicates as Coq's inductive propositions, as shown in lines 2–6 of Fig. 5. This encoding of SSL propositions makes it possible to directly inject them into the spatial assertions of a specification (represented via dependent Coq types), simplifying the proof effort by giving us access to all of Coq's regular tactics for reasoning about propositions. For instance, recall, from Sec. 3.4, how unfolding a predicate occurrence in the precondition (the Open rule in SSL) can proceed by *case analysis* on an instance of the *inductive proposition* `sll x2 s h'` in Coq's native proof context. Programs in HTT are *shallowly* embedded and are represented as Gallina programs written in a monadic style. Thanks to this choice, in HTT boolean program expressions and pure assertions have the same type and can thus be freely interchanged.

We will see in Sec. 6 how that compares to the proof burden in the case of Iris/VST-style embedding of the logic and the object language.

5.2 Delayed Checking of Postcondition Entailment

While the HTT-style SL embedding makes proof generation pleasant for the most part, some details of HTT preclude a fully straightforward proof translation. We have already seen one scenario that requires tracking additional backend-specific knowledge via a *proof context*: Sec. 3.4 showed how later steps need to refer to Coq hypothesis names instantiated in earlier steps. We also introduced the need for *deferred proof steps* in Sec. 3.5, 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. We now elaborate further on this postcondition entailment checking process—first, we describe in

detail the information needed to check entailment; next, we show how using the proof context and deferred steps *in tandem* gives us a principled approach to instrumenting it.

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 the spec (9):

$$\{\mathsf{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\} \tag{14}$$

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 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, assertions 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. 4 on how to proceed, focusing on the SSL rules that pertain to this subheap. First, Close unfolds the predicate occurrence in the post 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 $[y \mapsto \mathsf{y2}, v' \mapsto \mathsf{v}, nxt' \mapsto \mathsf{y12}, s' \mapsto \mathsf{s1}]$. 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 aforementioned steps; doing so tells us that `h2` is eventually expanded to

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

Notice how this expansion (provided as the witness in line 67 of Fig. 5) 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 proof; we can also observe the related hypothesis `H3 : sll y12 s1 h21` in the Coq proof state shown above. This lets us 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 shows that solving the entailment requires us to track the SSL rules that *transform* the predicate occurrences in the postcondition. Furthermore, we observe how this information is useful for two purposes: (a) tracing the provenance of a predicate application so that we may readily identify 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 parametrised over a proof context* so that (a) can be done as a part of (b).

We 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 use the map as a lookup table to obtain the maximally expanded subheap of the corresponding predicate application. We implement (b) by emitting a deferred proof step on each encounter with a Close or Frame rule, 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. 3) to obtain the corresponding assertion, and then instantiates its existentials. Since the computation is parametrised over a proof context, it can refer to the lookup table to obtain heap witnesses, within the computation itself.

6 SYNTHESIS CERTIFICATION VIA IRIS

Our second certification backend is the Iris framework for higher-order concurrent separation logic (Jung et al. 2018). For the Iris backend, we translated SusLang programs to HeapLang (Jung 2020), an example language for heap-manipulating programs bundled with Iris.

6.1 Translating Predicates and Specifications

Iris has been designed as an extensible framework for developing domain-specific separation-style logics for various languages with state and side-effects. Because of this, it adopts a shallow-style embedding of SL and a deep embedding for the object language (*cf.* the discussion in Sec. 5.1). Due to this dichotomy in the logic/language embedding, Iris maintains a distinction between program-level expressions and specification assertions, and similarly between program variables, which must have type `val`, and specification variables, which can use native Coq types such as `Z`. This requires us to differentiate how we translate SSL expressions depending on the context in which they are used in Iris, but does not otherwise impact the structure of the generated proofs.

A much more significant difference is the encoding of SSL predicates. While HTT represents spatial assertions as Coq propositions of sort `Prop`, Iris uses an abstract type `iProp` to capture assertions on the state of the heap (Jung et al. 2016). Any spatial predicates must therefore be encoded as terms of this type. This distinction means that we can no longer encode heap predicates as Coq's native `Inductive` definitions. In particular, as `iProp` is not a valid Coq sort, we can no longer declaratively inject an inductively defined predicate (of type `Prop`) into the type of spatial assertions. One common way to implement inductive Iris predicates in Coq is by defining them as Gallina functions that take the parameters of the predicate and then *construct* a term of type `iProp` representing the complete assertion (Krebbers et al. 2017, § 3.2). Unfortunately, predicates in SSL are defined in a way that facilitates the synthesis of `if-else` statements via the Open rule (*cf.* Fig. 3), and encoding them as Gallina functions is not straightforward. Such functions must always terminate, and this constraint is typically enforced by means of a syntactic check on the function definition: recursive calls must always be on a structurally decreasing argument. A naïve translation of an SSL predicate will not satisfy this constraint and will be rejected.

To overcome this hurdle, we explored two alternative ways to represent SSL predicates in Iris:

(1) It is possible to engineer an Iris encoding of inductive SSL predicates as Coq's `Fixpoint` definitions by piggybacking on the mechanism of cardinality variables (*cf.* Sec. 3.1). Specifically, numeric relations on the cardinalities present in the clauses of SSL predicates (as, *e.g.*, in (8)) can be employed to provide a decreasing measure for well-formed recursive definitions in Coq.

```
Inductive sll_card : Set :=
| sll_card_0 : sll_card
| sll_card_1 : sll_card -> sll_card.

Fixpoint sll (x: loc) (s: list Z)
    (α: sll_card) { struct α } : iProp Σ :=
  match α with
  | sll_card_0 =>
      ⌜(x = null_loc) ∧ (s = [])⌝
  | sll_card_1 β =>
      ∃ v s1 nxt,
         ⌜¬(x = null_loc) ∧ (s = [v] ++ s1)⌝ ∗
         x ↦ #v ∗ (x + 1) ↦ #nxt ∗
         (sll nxt s1 β)
end.
```

(a) SLL predicate as a recursive Gallina function.

```
Definition sll_copy : val :=
rec: "sll_copy" "r" :=
   let: "x2" := ! ("r") in
   if: "x2" = #null_loc
   then ( #() )
   else (
      let: "v" := ! "x2" in
      let: "nxt" := ! ("x2" + #1) in
      "r" <- "nxt";;
      "sll_copy" "r";;
      let: "y12" := ! "r" in
      let: "y2" := AllocN #2 #() in
      "r" <- "y2";;
      ("y2" + #1) <- "y12";;
      "y2" <- "v").
```

(b) The `sll_copy` function in Iris HeapLang.

Fig. 10. Encoding of the sll^{α} heap predicate and `sll_copy` in Iris.

In order to make this approach work, though, it is necessary to prove some auxiliary lemmas that enable case-analysis of `if-else` statements via the assertions in a predicate's clauses.

(2) Another option is to exploit the fact that Iris is a *higher-order* separation logic (as is VST), so inductive predicates can be defined in it using the Knaster-Tarski fixpoint theorem, along with suitable induction principles. For instance, the sll predicate (8) can be encoded in Iris as:

```
Definition sllF (f : loc * list Z -> iProp Σ) (arg : loc * list Z) : iProp Σ :=
  let (x, s) := arg in
  ⌜x = null_loc ∧ s = []⌝ ∨
  (∃ v s1 nxt, ⌜¬(x = null_loc) ∧ (s = [v] ++ s1)⌝ ∗ x ↦ #v ∗ (x + 1) ↦ #nxt ∗ f (nxt, s1)).
Definition sll := fixpoint.bi_least_fixpoint sllF.
```

The downside of this approach is the need to prove monotonicity of a predicate-inducing functional passed to the fixpoint combinator (*e.g.*, `sllF` above), as well as lemmas for case-based reasoning about disjoint alternatives from the functional definition. While those proofs are not difficult, it is still non-trivial to derive them fully automatically from SSL predicate definitions.

Having experimented with both encoding strategies, we have found out that they result in very similar structure of the proofs for programs being certified. At the end, we have fully implemented the first strategy, as (a) we found the proofs of its auxiliary lemmas easier to automate (b) it works even for non-higher-order separation logics. We explain it in more detail below.

6.1.1 Translating Predicates to Recursive Functions. In order to reliably translate SSL predicates into terminating Gallina functions, we devise a technique to faithfully transform numeric cardinality constraints into syntactic ones. This translation is based on the following intuition: the constructor of an inductive data type enforces a relation between the size of a term and its arguments—forming a term by applying a constructor to some arguments ensures that the term itself is strictly larger than its arguments. Building on this intuition, we can map the relations on cardinalities in each clause of a SSL spatial predicate to a unique constructor of an inductive data type that enforces the same relation, and then use these constraints to ensure that our encoding of the predicate as a Gallina function passes Coq's syntactic termination checker. For example, we translate the sll^{α} predicate (8) into the inductive data type and recursive definition shown in Fig. 10a. The first clause of the original sll^{α} predicate, which enforces no constraints on its cardinality argument, is mapped to the 0-argument constructor `sll_card_0` of the inductive data type `sll_card`. Meanwhile the second clause, which requires a strictly decreasing cardinality ($\beta < \alpha$), maps to the 1-argument constructor `sll_card_1`. The Iris predicate is then defined as a recursive Gallina function that pattern-matches

on the cardinality parameter α and then constructs the corresponding `iProp` assertion, recursively calling itself on a syntactically smaller cardinality β when needed. This approach generalises to arbitrary SSL predicates. For instance, to encode cardinality constraints for a predicate that describes binary trees, we generate an inductive data type with two constructors, one that takes no arguments (corresponding to leaf nodes, which impose no cardinality constraints), and one that takes two arguments (corresponding to regular nodes, whose children must have strictly smaller heaps). In this way, we can faithfully and completely automatically translate SSL predicates into a form that is compatible with Coq's termination checker.

6.1.2 Recovering Predicate Selectors. The described translation of predicates comes at a cost. The original SSL predicates had clause selectors that were pure assertions on the predicate arguments, and the generated SusLang programs conditioned on these selectors to determine which branch to take. In sharp contrast, our translated Iris predicates have clause selectors determined purely by the cardinality constraints, and these are not reflected in any way in the translated programs, since cardinalities have no computational behaviour. Nonetheless, we want to verify the correctness of unmodified programs, which condition on unmodified selectors. We emphatically do *not* want the programs we generate to condition on the artificial cardinality data type, as our translated programs should match the original SusLang ones. This mismatch between the structure of the translated programs and, implicitly, the proof structure in SSL on one hand, and the structure of the Iris predicates on the other hand, poses a challenge. To overcome it, we must, given a program-level selector expression, *e.g.*, the one at line 4 in Fig. 10b, be able to, at the level of Iris proofs, *recover* the cardinality constructor corresponding to the selected clause. For example, given an assertion $\mathsf{sll}^{\alpha}(x, s)$ and the knowledge $\lceil$`¬(x = null_loc)`$\rceil$, obtained upon entering the **`else`**-branch of the program in Fig. 10b, we must be able to infer the fact $\lceil\exists\beta, \alpha =$ `(sll_card_1 β)`$\rceil$. We can then use this fact combined with the $\mathsf{sll}^{\alpha}(x, s)$ assertion to obtain the second clause of the Iris predicate, which lets us proceed with the proof in the fashion dictated by the SSL proof tree. To accomplish this recovery in a general fashion, whenever we translate an SSL predicate to Iris, we automatically prove *inversion lemmas* that let us infer the appropriate cardinality constructors given program-level selectors. For instance, the Iris `sll` predicate (Fig. 10a) comes bundled with the following lemmas:

```
Lemma sll_card_0_learn (x: loc) (s: list Z) α:
  sll x s α ⊢ sll x s α ∗ ⌜(x = null_loc) -> α = sll_card_0⌝.
Lemma sll_card_1_learn (x: loc) (s: list Z) α:
  sll x s α ⊢ sll x s α ∗ ⌜¬(x = null_loc) -> ∃β, α = (sll_card_1 β)⌝.
```

In our proofs, we use these lemmas in the interpretation of the Open rule (Fig. 3) to ensure the source and Iris proof trees remain synchronised despite the mismatch in predicate structures.

6.2 Differences in the Treatment of Pointer Assertions

While SusLang and HeapLang are quite similar, they differ in their representations of pointers. SusLang adopts a C-style model, where pointers are always nullable, whereas HeapLang has ML-style references and represents *nullable* pointers as terms of type `option val`, with the null pointer being `None`. Not wanting to modify generated programs to use references—which would have created a divergence in the proof structure—we instead modified HeapLang to support C-style pointers. Specifically, in the semantics of our modified HeapLang, we introduced a distinguished location `null_loc` corresponding to the null pointer, as seen at line 4 of Fig. 10b. Moreover, we modified and proved sound the logic rule for memory allocations to return, on top of the usual $\mathtt{l} \mapsto \mathtt{v}$ spatial assertion, a pure assertion $\lceil$`¬(l = null_loc)`$\rceil$. This additional assertion is important, as without it we would not be able to prove, for example, that newly-allocated memory locations can be the head of linked lists, as per the second clause of the predicate definition (Fig. 10a).

```
void sll_copy(loc r) {
  loc x2 = READ_LOC(r, 0);
  if (x2 == NULL) { return; }
  else {
    int v = READ_INT(x2, 0);
    loc nxt = READ_LOC(x2, 1);
    WRITE_LOC(r, 0, nxt);
    sll_copy(r);
    loc y12 = READ_LOC(r, 0);
    loc y2 = (loc) malloc(2 * sizeof(loc));
    WRITE_LOC(r, 0, y2);
    WRITE_LOC(y2, 1, y12);
    WRITE_INT(y2, 0, v);
    return;
  }
}
```

```
Definition sll_copy_spec :=
 DECLARE _sll_copy
  WITH r: val, x: val, s: (list Z), a: sll_card
  PRE [ (tptr ssl_val) ]
  PROP(is_pointer_or_null(r);is_pointer_or_null(x))
  PARAMS(r)
  SEP ((data_at (tarray ssl_val 1) [inr x] r);
       (sll x s a))
  POST[ tvoid ]
  EX y: val,
  EX b: sll_card,
  PROP( )
  LOCAL( )
  SEP ((data_at (tarray ssl_val 1) [inr y] r);
       (sll y s b);
       (sll x s a)).
```

Fig. 11. Definition of `sll_copy` in C (left) and corresponding specification in VST (right)

7 SYNTHESIS CERTIFICATION VIA VERIFIED SOFTWARE TOOLCHAIN

While our prior case studies have looked at verifying SusLang programs according to various "C-like" DSLs within Coq, we now ask: can we do the same for the real deal—*executable* C? With our final case study, we answer this question in the affirmative, implementing a certification backend for the Verified Software Toolchain (Appel et al. 2014), using it to translate SusLang programs into C and certify their correctness with regards to a simplified semantics of C.[6] As VST uses a shallow embedding of separation logic, defining a data type for the model of its assertions similarly to Iris, the strategy for the proof translation broadly follows the same steps, reusing a few of the same encoding techniques from the prior section (in particular the encoding of inductive predicates Sec. 6.1 and the associated inversion lemmas Sec. 6.1.2). In the rest of this section, we will provide an overview of this backend, focusing on the additional changes that had to be made to make SuSLik and SusLang conform to the constraints of real world executable code.

7.1 SusLang on Metal: Converting Programs to C and Specifications to VST

Before we set about certifying programs, we must first translate SusLang functions into C and their specifications to VST. This translation has some subtleties. For instance, Fig. 11 lists the translated program and specification for our running example `ssl_copy`, which follow closely from the original definitions, but make use of some custom data types (`loc, ssl_val`) and operations to read and write memory (`READ_LOC, WRITE_LOC`). As it turns out, these small modifications are crucial for faithfully realising the *simplified* memory model assumed by SusLang programs on real hardware.

In particular, a fact that we have glossed over in the prior sections has been the way in which SusLang arrays can freely contain pointer and integer values, side by side, without issue. More generally, by allowing these kinds of constructs, SusLang implicitly makes the assumption that integer and pointer values occupy the same amount of space on the heap, and while, in terms of C code, this might not be an uncommon assumption, it is an assumption nonetheless, introducing unsafe implementation-specific behaviour into the generated program.

```
typedef union sslval {
  int ssl_int;
  void *ssl_ptr;
} *loc;
```

Our solution then is quite natural: we encode this assumption within the C type system. In our translation, we enforce that all allocations, reads and writes to and from the heap will be done exclusively to terms of a custom type `sslval`, defined (left) as a union of integer and pointer values. Wrapping this up in a type alias, `typedef union sslval *loc`, and pairing it with corresponding read and write macros that transparently handle the coercion between types (*i.e.*, defining the operation to read locations as a compile-time macro `#define READ_LOC(x,y) (*(x+y)).ssl_ptr`), our

[6] Due to limitations of SuSLik's memory model, we must make the simplifying assumption that `malloc` never returns `NULL`.

generated programs thereby both *syntactically look* and *semantically behave* exactly as the SusLang programs they represent, simplifying the subsequent translation of VST specifications and proofs.

7.2 Getting Real: Impedance Mismatching Between SusLang and C Semantics

Having translated SusLang programs to C, the real question is whether we can actually verify these programs are correct. Thankfully, as VST uses the same forward-execution style of reasoning as all the prior frameworks, much of the form of these proofs still end up following the same broad strokes, requiring changes only to account for discrepancies in their semantic models. In the rest of this section, we comment on the most significant areas where we ran into issues.

7.2.1 Ternary Expressions. Consider the following program statement: `*m = (x < y ? 3 : 1);` In programming language with an absence of uncontrolled side-effects such as SusLang, it would always be safe to treat the evaluation of the entire statement as a single step, *i.e.*, a program operation writing the value of the expression $(x < y\ ?\ 3 : 1)$ directly into the memory at location m. In fact, SSL proofs take this even further, treating ternary expressions as single values within logical specifications, and using them as such within spatial assertions, as in $m \mapsto (x < y\ ?\ 3 : 1)$). Switching back to the semantics of C, where expressions can have arbitrary effects, such treatment of ternaries is clearly no longer valid. This is reflected in VST, where the evaluation of a ternary sub-expression is interpreted as an if statement, branching the proof into two separate control flow paths for each case, in contrast to direct execution in SSL proofs. To thereby keep the SSL and VST proof contexts synchronised and avoid divergence, we add additional logic to the proof interpreter to transparently handle such statements. Whenever an ternary expression is evaluated during a SSL proof, the generated VST proof branches on each cases of the ternary but also provides a unifying post-condition to the branch that then joins both cases together immediately afterwards using the fact that the result of the expression is equivalent to a logical ternary expression, using tactics provided by VST to automatically dispatch the generated obligations.

7.2.2 Splitting and Recombining Memory Blocks. Another translation aspect that required special care was in managing the particularly loose treatment of memory in SSL. Recall that a contiguous block of allocated memory in SSL is represented by the spatial assertion $[x, n]$, with the contents of this block captured separately as $(x + i) \mapsto -$ for each element. This encoding of allocations as separate blocks and mappings allows SSL proofs to easily mix between single cells in memory $x \mapsto -$ and individual elements of larger blocks $[x, n] * x \mapsto -$, so that synthesised programs can pass pointers *from the middle of blocks of memory* freely to procedures that expect lone pointers. While mixing these kinds of pointers is valid according to the semantics of C, we ran into difficulties when certifying such programs in VST, where blocks of memory and their contents are encoded as a single assertion and can not easily be split. Having experimented with a number of non-trivial logic memory transformations available in VST (*e.g.*, logically splitting memory blocks into individual segments and then recombining them back), we opted for a simpler and more principled solution: constraining SuSLik's proof search to reject programs that *mix pointers* from *different-size blocks*, allowing pointers to unify only if their associated blocks are of the same size. This restriction did not prevent any known SuSLik benchmarks from being synthesised.

7.2.3 Integer Semantics and Overflows. Being the only framework beholden to the constraints of real-world hardware, VST is uniquely challenged amongst the others in that it must necessarily reason about overflow semantics. This poses a fundamental problem when translating integer-manipulating programs from SusLang, as SuSLik proofs do not consider overflow, and synthesised programs may sometimes perform unsafe (bounded-)arithmetic operations. In particular, any numbers that arise during the execution of programs, intermediate or otherwise, in VST must

always be *guaranteed within the valid ranges* of integers before the proof may continue. This constraint of tracking overflow bounds causes issues even when verifying programs with numeric variables that only rely on comparisons, *i.e.*, having no chance of overflow—as the overflow bounds on these numeric variables must first be established before we can reason about the behaviour of any comparisons over them. While the differences in the overflow semantics between SusLang and C are too large to handle in the general case, by adjusting the synthesised program specs to include assumptions on numeric bounds, and using VST's native tactics to dispatch overflow obligations, we were able to certify programs that only rely on arithmetic comparisons, *e.g.*, finding the maximum or the minimum of a list of integers.

8 EVALUATION

We implemented the entire framework for translating SSL proof trees to Coq proofs on top of SuSLik in a combination of Scala (for proof evaluator and individual interpreters) and Coq (for backend-specific automation described in Sec. 4.2.2 and written primarily in Ltac). The table on the right summarises the overall implementation effort in terms of lines of code. Our implementation and benchmarks are open source and are publicly available (Watanabe et al. 2021). Each of the three backend-specific Coq automation libraries can be installed via OCaml's `opam` package manager. They are also available, along with the translated specifications and proofs for our benchmarks, in the supplementary material for this submission.

Component	Scala	Coq
Proof evaluator	1042	-
HTT support	1340	160
Iris support	1317	102
VST support	1887	166
The rest of SuSLik	5508	-

8.1 Evaluating Synthesis Certifiers on Standard Heap-Manipulating Benchmarks

Our translation works for unaltered SusLang programs, and the synthesis algorithm has only been slightly restricted to suit some particular patterns in the certification backends (*cf.* Sec. 7.2.2). With this regard, in the evaluation of our approach we aimed to answer the following questions:

(1) How efficient is the certification: what are the sizes of the generated Coq specs and proofs, and how long does it take to check them via the corresponding SL embeddings?
(2) What design choices in SusLang/SSL and the languages/logics of the verification backends might pose obstacles to automated certification of synthesised heap-manipulating programs?

Tab. 1 summarises our evaluation results on programs manipulating with individual pointers and integers, singly- and doubly-linked lists, and binary trees. The reported sizes of Coq artefacts do not include translated heap predicates and their inversion lemmas (in the case of Iris and VST), as those are shared between specs of multiple programs. All runtimes are obtained on a 1.90GHz Intel Core i7-8665U machine with 40GB RAM running Ubuntu 18.04 and Coq 8.11.2.

With regard to Question (1), Tab. 1 demonstrates that all generated proofs are relatively concise, with those for HTT being slightly longer due to a number of administrative renamings required to keep the proof in sync with the Coq context. The proof checking times for HTT and Iris are in the same ballpark, ranging from 2 to 20 seconds for all but three HTT examples. The three outliers for HTT (max, min, and sll-copy) are due to the use of CoqHammer for discharging pure entailment lemmas (*cf.* Sec. 4.2.3) via Ssreflect libraries for Peano numbers and sequences (Gonthier et al. 2009). In the case of Iris, those lemmas are handled for standard Coq numbers and lists via a more specialised (and hence more efficient) Micromega library (Besson and Makarov 2020). The significantly longer checking times for VST proofs are due to the generality of its `entailer!` tactic (Cao et al. 2018), whose performance can be improved by making proof scripts in VST follow a certain structure (which is not the case for our auto-generated ones).

Table 1. Statistics for synthesised programs with pointers from SuSLik benchmark suite. Sizes of generated Coq artefacts are in lines of code. For HTT, Iris, and VST we report the proof checking times (in seconds).

Group	Description	Synthesis Time	HTT			Iris			VST		
			Spec	Proofs	Time	Spec	Proofs	Time	Spec	Proofs	Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

As for Question (2), Tab. 1 indicates a few benchmarks that failed to verify in Iris and VST. The two case studies that are not handled in Iris (finding a minimum/maximum of a list of integers) rely on a ternary operator, which in Iris generates two proof obligations (one for each branch), significantly deviating from the SSL proof, which does not split on ternary operators. VST could not handle list length and tree size functions due to the possibility of integer overflows, as the corresponding checks were missing from the synthesised SusLang programs.

Both these shortcomings can be addressed by modifying SusLang and the corresponding synthesis rules to account for specific backend features, making the synthesiser language be more like an intermediate representation, serving multiple backends. This is an interesting research direction, which we are going to explore in the future.

8.2 Encoding Collection Payloads for Advanced Benchmarks

We did not include benchmarks that require extensive support for solving pure entailments via SMT, such as binary search trees and sorted lists, into the suite of programs in Tab. 1. The reason for that is in the challenge of encoding *payloads* for heap-based collections, which SuSLik and our backend provers address in very different ways, each opting for one suited to its unique requirements.

SuSLik represents collection contents using unordered *multi-sets*, a suitable choice for a synthesiser for 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.

The backends, in contrast, represent payloads as algebraic *lists*, as they are much easier to reason about in proofs. For example, because list equality assertions use Leibniz 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. 1. Nonetheless, for HTT we also experimented with more advanced benchmarks that rely on the collection payloads being implemented as multi-sets, by replacing Leibniz equality for lists with

the perm_eq relation from SSREFLECT/MATHCOMP library (Gonthier et al. 2009; Mahboubi and Tassi 2021; Sergey 2014), 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 *wrt.* predicate applications with otherwise identical arguments, *e.g.*, for sll from Fig. 5:

```
Axiom sll_perm_eq: ∀x h s1 s2, perm_eq s1 s2 -> sll x s1 h -> sll x s2 h.
```

In fact, we proved these axioms manually as lemmas for the corresponding predicates, but we didn't automate their proofs. An alternative approach would be to employ Coq's setoid rewriting mechanism with perm_eq taken as an ad-hoc equivalence relation on lists (Coen 2004; Sozeau 2009).

Several of the programs in these advanced benchmarks, summarised in Tab. 2, also feature non-trivial pure lemmas (discussed in Sec. 4.2.3) that our implementation's default proof strategy, COQHAMMER, 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 interpreter, attempt to compile them (with COQHAMMER), and only provide manual proofs of those pure lemmas that COQHAMMER fails to discharge. The figures in the Tab. 2 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.

Table 2. Benchmarks using multiset equality in HTT.

Group	Program	Synt. time	Coq time	Lemmas	Manual
Doubly-Linked Lists	copy	8.7	22.1	7	4
	two-element	0.6	11.6	3	3
	from-sll	1.2	18.2	5	2
Binary Search Trees	find-smallest	2.9	12.2	7	1
	insert	33.8	72.9	15	6
	rotate-left	6.3	16.9	4	2
	rmv-root-left	3.4	26.5	6	2
	rmv-root-right	34.7	25.1	6	2
	rotate-right	5.6	15.9	4	2
Sorted Lists	insertion-sort	1.8	10.7	7	0
	insert-sort-free	0.6	9.5	5	0
	insert	9.7	26.8	18	8
	prepend	0.3	6.6	2	0

9 LESSONS LEARNT

Extensibility of the proof translation. The sceptical reader might be wondering whether our strategy generalises—are our backend-specific proof interpreters simply overfitted for the benchmarks or does our approach actually capture some underlying patterns within each framework? Our experience from implementing the backends for this paper *strongly* suggests the latter.

During the development of the backends, we encountered several *milestones* at which large swathes of benchmarks were successfully certified. The first such milestone was the handling of READ and WRITE rules—with just these components, our interpreters were able to handle the certification of all integer-based benchmarks with no further changes. Following this, was the handling of OPEN and FREE rules which then made the verification of programs that free structures (sll_free, tree_free) possible. With the addition of CLOSE and ALLOC rules, the proof evaluators were then able to handle the verification of programs that allocate and copy structures. In this way, the process of implementing the translation was not strongly tied to the choice of benchmarks, with only a few examples needed to test the implementation of the rules before the entire milestone would be implemented. The fact that we saw the same milestones across all backends suggests that our experience will generalise to other targets (Charguéraud 2020; Chlipala 2011).

User experience of implementing backends. An interesting question is, from the perspective of our subjective user experience, which framework was the easiest to adopt as a certification backend?

We found out that implementing the HTT backend was the least challenging, with HTT's embedding style of SL and the object language placing the fewest constraints on context management. This simplified the implementation of the translator as there was no need to distinguish between program and proof-level terms, allowing for a straightforward implementation of the proof evaluator. Furthermore, HTT makes little use of custom notations, providing a simple "what-you-see-is-what-you-get" experience from the perspective of proof automation: writing domain-specific tactics on top of it can be done by simply matching on the exact structure of the Coq proof context.

Following HTT, VST came closely second, with its extensive proof automation handling much of the heap management. In particular, while VST's deep embedding of the language would, in theory, require carefully discriminating between program and proof-level terms, in practice, we found that VST's tactics would consistently handle this task, automatically discharging any obligations arising from their discrepancies. Our ranking of VST is influenced by the challenges of introducing SSL-specific automation to it, a necessity for managing impedance mismatches between synthesis and certification proof contexts. In contrast with HTT, VST makes *significant* use of custom notation to simplify the proof context. Because of this, writing non-trivial tactics on top of VST required us to "peek behind the curtain" of its notations, making the corresponding automation more fragile.

In the end, implementing the Iris backend was the most difficult. The main set of challenges arose from interacting with *Iris Proof Mode* (IPM) (Krebbers et al. 2017). Due to its human prover-oriented nature, IPM is geared for proofs that perform explicit context and goal management. In a typical IPM proof, SL hypotheses (*i.e.*, heaplets) are given explicit names, by which they are referred later, when used. To keep the same proof style, a proof interpreter for Iris would have to keep track of which heaps and pure assertions in the SSL proof context correspond to which IPM hypotheses names and ensure this correspondence is maintained as tactics are applied. This is possible, but would be burdensome. Instead, we opted for a proof style with "nameless" hypotheses, which closely matches the SSL proofs but heavily relies on heap unification to dispatch obligations. Sadly, IPM's `iFrame` tactic for heap unification is not particularly robust (compared to VST's `entailer!`, which seems almost magical), so we had to be careful when managing the IPM context to ensure that the unification will succeed. Another challenge we encountered when developing the proof interpreter for Iris was in deciphering the error messages that are produced when proofs go wrong. In particular, Iris's heavy use of Coq's coercions and canonical structures (Mahboubi and Tassi 2013) often means that when proofs go wrong, the resulting unification errors are not always clear.

To conclude, we don't claim that Iris and VST are inherently less suitable for synthesis certification than HTT. Rather, we hope that our observations will be helpful for enhancing those frameworks in the future to provide better support for interfacing with program synthesisers.

10 RELATED AND FUTURE WORK

Certifying compilers and proof-carrying code. Our work is a spiritual successor to a 25 year-long line of research on Proof-Carrying Code (PCC) started by Necula and Lee (1996). The original PCC proposal by Necula (1997) was to supply proofs, in a logic embedded into the Edinburgh Logical Framework (Harper et al. 1993), for executable binaries in the DEC Alpha assembly language. This would allow users of the binaries to independently check that, upon execution, the code does not violate basic type and memory safety properties. In follow-up work, Necula and Lee (1998) designed a *certifying* compiler, which would automatically generate such proofs when producing low-level assembly from code written in a high-level language. The ideas of PCC and certifying compilation have been studied extensively in the past two decades, in application to, *e.g.*, validation of temporal properties of systems code (Henzinger et al. 2002), static analysis via abstract interpretation (Besson et al. 2006), refinement types (Chen et al. 2010), information flow control (Barthe et al. 2013), security policies in software-defined networks (Skalka et al. 2019), and other classes of statically enforceable

program properties. Further research has been conducted on minimising the size of the trusted code base required to validate the proofs (Appel 2001).

Our work presents a novel application of the main ideas of PCC and certifying compilation—coupling generation of code and a machine-checked proof of its safety specification—to the area of automated program synthesis. Unlike the original work on PCC (Necula 1997), which targeted basic type- and memory safety properties of untrusted programs, our proposal focuses on a richer class of full functional correctness specifications.

In this work, we are not addressing the challenge of controlling the sizes of the obtained certificates or the time it takes to check them, leaving those proof aspects for future work. That said, we acknowledge the importance of those properties in a security context. For instance, if the time it takes to check a certificate coming from an untrusted party is not linear in its size, this could lead to denial-of-service attacks, in which an adversary produces short proofs that take longer to check.

Certified interactive program synthesis. The Fiat framework (Chlipala et al. 2017; Delaware et al. 2015) implements a certified *interactive* program synthesiser, by embedding the synthesis framework directly into the Coq proof assistant. Specifications in Fiat are represented by high-level non-deterministic functional programs and data types, which are then refined (He et al. 1986), in a step-wise fashion, into executable implementations in an imperative low-level language with explicit memory management and, eventually, to assembly (Pit-Claudel et al. 2020). Fiat is aimed to facilitate certified synthesis by refinement by providing (a) tactics for automating refi nement proofs in particular restricted domains and (b) a library of lemmas that can be used by the clients for verifying complex derivations. The main advantage of Fiat is it extensibility: it allows the users to add new verified compilation rules. In contrast with Fiat's approach to synthesis, which requires one to interactively verify a sequence of semantics-preserving optimisations as well as any new added rules, SuSLik's synthesis is based on a *fully automated* proof search in a *fixed* set of SSL's proof rules. That said, our certification experiments (*cf.* Sec. 8.2) indicated the need to occasionally ask the user for help with proofs of pure facts that are outside the reach of Coq's automation.

11 CONCLUSION

The mission of program synthesis is to raise the level of abstraction in specifying tasks for computers to execute, from the idioms of general-purpose programming languages handled by present-day compilers, to concise logical descriptions and examples, thus making programming more accessible. Given this *synthesis-as-a-next-generation-compiler* view (Yahav 2020), one may hope to have fully verified program synthesisers with machine-checked correctness guarantees, akin to the CompCert (Leroy 2006) or CakeML (Kumar et al. 2014) compilers, at some point in the future. While building a fully *certified* automated program synthesiser still appears to be a daunting task, in this work we have shown that implementation of a *certifying* automated synthesiser for interesting heap-manipulating programs using the technology at hand is already within our reach.

ACKNOWLEDGMENTS

We thank Adam Chlipala, Olivier Danvy, Aquinas Hobor, Shachar Itzhaky, and Anton Trunov for their feedback on drafts of this paper. We also thank the ICFP'21 PC and AEC reviewers as well as our shepherd Steve Zdancewic for their constructive and insightful comments. In particular, we thank Reviewer B for suggesting to consider translating inductive predicates from SSL to higher-order separation logics (*i.e.*, Iris and VST) using the Knaster-Tarski fixpoint combinator.

This research was supported by the National Science Foundation under Grant No. 1911149, by Singapore MoE Tier 1 Grant No. IG18-SG102, and by the Grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS).

REFERENCES

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. IEEE Computer Society, 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS)*, Vol. 6602. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. https://doi.org/10.1017/CBO9781107256552

Gilles Barthe, David Pichardie, and Tamara Rezk. 2013. A certified lightweight non-interference Java bytecode verifier. *Math. Struct. Comput. Sci.* 23, 5 (2013), 1032–1081. https://doi.org/10.1017/S0960129512000850

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *APLAS (LNCS)*, Vol. 3780. Springer, 52–68. https://doi.org/10.1007/11575467_5

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (LNCS)*, Vol. 4111. Springer, 115–137. https://doi.org/10.1007/11804192_6

Frédéric Besson, Thomas P. Jensen, and David Pichardie. 2006. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.* 364, 3 (2006), 273–291. https://doi.org/10.1016/j.tcs.2006.08.012

Frédéric Besson and Evgeny Makarov. 2020. Micromega: solvers for arithmetic goals over ordered rings. Online documentation available at https://coq.inria.fr/refman/addendum/micromega.html.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*. ACM, 412–423. https://doi.org/10.1145/1806596.1806643

Wei-Ngan Chin, Cristina David, and Cristian Gherghina. 2011. A HIP and SLEEK verification system. In *OOPSLA (Companion)*. ACM, 9–10. https://doi.org/10.1145/2048147.2048152

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245. https://doi.org/10.1145/1993498.1993526

Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *SNAPL (LIPIcs)*, Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.3

Claudio Sacerdoti Coen. 2004. A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In *TYPES (LNCS)*, Vol. 3839. Springer, 98–114. https://doi.org/10.1007/11617990_7

Lukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1-4 (2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

David Delahaye. 2000. A Tactic Language for the System Coq. In *LPAR (LNCS)*, Vol. 1955. Springer, 85–95. https://doi.org/10.1007/3-540-44404-1_7

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*. ACM, 689–700. https://doi.org/10.1145/2676726.2677006

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *CAV (LNCS)*, Vol. 10427. Springer, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2009. *A Small Scale Reflection Extension for the Coq system*. Technical Report 6455. Microsoft Research – Inria Joint Centre.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. 1986. Data Refinement Refined. In *ESOP (LNCS)*, Vol. 213. Springer, 187–196. https://doi.org/10.1007/3-540-16442-1_14

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. 2002. Temporal-Safety Proofs for Systems Code. In *CAV (LNCS)*, Vol. 2404. Springer, 526–538. https://doi.org/10.1007/3-540-45657-0_45

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *PLDI*. ACM, 944–959. https://doi.org/10.1145/3453483.3454087

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language.* Ph.D. Dissertation. Saarland University.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *OOPSLA*. ACM, 407–426. https://doi.org/10.1145/2509136.2509555

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the Working Coq User. In *ITP (LNCS)*, Vol. 7998. Springer, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. https://doi.org/10.5281/zenodo.4457887 Available at https://math-comp.github.io/mcb.

Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. https://doi.org/10.1145/357084.357090

Alberto Martelli and Ugo Montanari. 1973. Additive AND/OR Graphs. In *IJCAI*. William Kaufmann, 1–11.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*, Vol. 9583. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *ICFP*. ACM, 229–240. https://doi.org/10.1145/1411204.1411237

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. ACM, 261–274. https://doi.org/10.1145/1706299.1706331

George C. Necula. 1997. Proof-Carrying Code. In *POPL*. ACM Press, 106–119. https://doi.org/10.1145/263699.263712

George C. Necula and Peter Lee. 1996. Safe Kernel Extensions Without Run-Time Checking. In *OSDI*. ACM, 229–243. https://doi.org/10.1145/238721.238781

George C. Necula and Peter Lee. 1998. The Design and Implementation of a Certifying Compiler. In *PLDI*. ACM, 333–344. https://doi.org/10.1145/277650.277752

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, Vol. 2142. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR (LNCS)*, Vol. 12167. Springer, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *PACMPL* 3, POPL (2019), 72:1–72:30. https://doi.org/10.1145/3290385

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Reuben N. S. Rowe and James Brotherston. 2017. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*. ACM, 53–65. https://doi.org/10.1145/3018610.3018623

Ilya Sergey. 2014. Programs and Proofs: Mechanizing Mathematics with Dependent Types. https://doi.org/10.5281/zenodo.4996238 Lecture notes with exercises.

Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. 2019. Proof-Carrying Network Code. In *CCS*. ACM, 1115–1129. https://doi.org/10.1145/3319535.3363214

Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *J. Formaliz. Reason.* 2, 1 (2009), 41–62. https://doi.org/10.6092/issn.1972-5787/1574

Yasunari Watanabe. 2021. *A Framework for Certified Program Synthesis.* Master's thesis. National University of Singapore.

Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certified SuSLik (ICFP 2021 Artifact): Code and Benchmarks. https://doi.org/10.5281/zenodo.5005829

Eran Yahav. 2020. "A synthesizer is a compiler that doesn't work". https://twitter.com/IsilDillig/status/1287125716761542656. Attributed to Eran Yahav by Işil Dillig in a tweet on July 26, 2020.