Structuring the Synthesis of Heap-Manipulating Programs

NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore

This paper describes a deductive approach to synthesizing imperative programs with pointers from declarative specifications expressed in Separation Logic. Our synthesis algorithm takes as input a pair of assertions—a pre- and a postcondition—which describe two states of the symbolic heap, and derives a program that transforms one state into the other, guided by the shape of the heap. Our approach to program synthesis is grounded in proof theory: we introduce the novel framework of Synthetic Separation Logic (SSL), which generalises the classical notion of heap entailment $\mathcal{P} \vdash \mathcal{Q}$ to incorporate a possibility of transforming a heap satisfying an assertion $\mathcal{P}$ into a heap satisfying an assertion $\mathcal{Q}$. A synthesized program represents a proof term for a *transforming entailment* statement $\mathcal{P} \rightsquigarrow \mathcal{Q}$, and the synthesis procedure corresponds to a proof search. The derived programs are, thus, correct by construction, in the sense that they satisfy the ascribed pre/postconditions, and are accompanied by complete proof derivations, which can be checked independently.

We have implemented a proof search engine for SSL in a form of the program synthesizer called SuSLik. For efficiency, the engine exploits properties of SSL rules, such as invertibility and commutativity of rule applications on separate heaps, to prune the space of derivations it has to consider. We explain and showcase the use of SSL on characteristic examples, describe the design of SuSLik, and report on our experience of using it to synthesize a series of benchmark programs manipulating heap-based linked data structures.

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

Additional Key Words and Phrases: Program Synthesis, Separation Logic, Proof Systems, Type Theory

ACM Reference Format:
Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (January 2019), 30 pages. https://doi.org/10.1145/3290385

1 INTRODUCTION

Consider the task of implementing a procedure `swap(x, y)`, which swaps the values stored at two distinct heap locations, `x` and `y`. The desired effect of `swap` can be concisely captured via pre/postconditions expressed in Separation Logic (SL)—a Hoare-style program logic for specifying and verifying stateful programs with pointers (O'Hearn et al. 2001; Reynolds 2002):

$$\{x \mapsto a * y \mapsto b\}\ \texttt{void swap(loc x, loc y)}\ \{x \mapsto b * y \mapsto a\} \tag{1}$$

This specification is *declarative*: it describes *what* the heap should look like before and after executing `swap` without saying *how* to get from one to the other. Specifically, it states that the program takes as input two pointers, `x` and `y`, and runs in a heap where `x` points to an unspecified value `a`, and `y` points to `b`. Both `a` and `b` here are *logical* (*ghost*) variables, whose scope captures both pre- and postcondition (Kleymann 1999). Because these variables are ghosts, we cannot use them

Authors' addresses: Nadia Polikarpova, University of California, San Diego, USA, nadia.polikarpova@ucsd.edu; Ilya Sergey, Yale-NUS College, Singapore, National University of Singapore, Singapore, ilya.sergey@yale-nus.edu.sg.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART72
https://doi.org/10.1145/3290385

directly to update the values in x and y as prescribed by the postcondition; the program must first "materialize" them by reading them into local variables, a2 and b2 (*cf.* lines 2–3 of the code below). In our minimalistic C-like language, **loc** denotes untyped pointers, and **let** introduces a local variable. Unlike in C, both formals and locals are *immutable* (the only mutation is allowed on the heap).

```
1  void swap(loc x, loc y) {
2    let a2 = *x;
3    let b2 = *y;
4    *y = a2;
5    *x = b2; }
```

As a result of the two reads, the ghost variables in the post-condition can now be substituted with equal program-level variables: $x \mapsto \mathtt{b2} * y \mapsto \mathtt{a2}$. This updated postcondition can be realized by the two writes on lines 4–5, which conclude our implementation, so the whole program can now be verified against the specification (1).

Having done this exercise in program derivation, let us now observe that the SL specification has been giving us guidance on what effectful commands (*e.g.*, reads and writes) should be emitted next. In other words, the *synthesis* of swap has been governed by the given specification in the same way *proof search* is guided by a goal in ordinary logics. In this work, we make this connection explicit and employ it for efficiently synthesizing imperative programs from SL pre- and postconditions.

Motivation. The goal of this work is to advance the state of the art in synthesizing provably correct heap-manipulating programs from *declarative functional* specifications. Fully-automated program synthesis has been an active area of research in the past years, but recent techniques mostly targeted simple DSLs (Gulwani et al. 2011; Le and Gulwani 2014; Polozov and Gulwani 2015) or purely functional languages (Feser et al. 2015; Kneuss et al. 2013; Osera and Zdancewic 2015; Polikarpova et al. 2016). The primary reason is that those computational models impose strong *structural constraints* on the space of programs, either by means of restricted syntax or through a strong type system. These structural constraints enable the synthesizer to discard many candidate terms *a-priori*, before constructing the whole program, leading to efficient synthesis.

Low-level heap-manipulating programs in general-purpose languages like C lack inherent structural constraints *wrt.* control- and data-flow, and as a result the research in synthesizing such programs has been limited to cases when such constraints can be imposed by the programmer. From the few existing approaches we are aware of, Simpl (So and Oh 2017) and ImpSynt (Qiu and Solar-Lezama 2017) require the programmer to provide rather substantial *sketches* of the control-flow structure, which help restrict the search space; Jennisys by Leino and Milicevic (2012) does not require sketches, but also does not scale to functions with destructive heap updates.

Key Ideas. Our theoretical insight is that the structural constraints missing from an imperative language itself, can be recovered from the *program logic* used to reason about programs in that language. We observe that synthesis of heap-manipulating programs can be formulated as a proof search in a generalized proof system that combines entailment with Hoare-style reasoning for *unknown programs*. In this generalized proof system, a statement $\mathcal{P} \rightsquigarrow \mathcal{Q}$ means that there *exists* a program c, such that the Hoare triple $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ holds; the *witness* program c serves as a proof term for the statement. In order to be useful, the system must satisfy a number of practical restrictions. First, it should be expressive enough to (automatically) verify the programs with non-trivial heap manipulation. Second, it should be restrictive enough to make the synthesis problem tractable. Finally, it must ensure the termination of the (possibly recursive) synthesized programs, to avoid vacuous proofs of partial correctness.

In this paper we design such a generalized proof system based on the symbolic heap fragment of *malloc/free* Separation Logic[1] with inductive predicates, to which we will further refer as just Separation Logic or SL (O'Hearn et al. 2009; Reynolds 2002). Separation Logic has been immensely

[1]This nomenclature is due to Cao et al. (2017), who provide it as a rigorous alternative to the folklore notion of *classical* SL.

successful at specifying and verifying many kinds of heap-manipulating programs, both interactively and automatically (Appel et al. 2014; Berdine et al. 2011; Charguéraud 2010; Chen et al. 2015; Chin et al. 2012; Chlipala 2011; Distefano and Parkinson 2008; Nanevski et al. 2010; Piskac et al. 2014a), and is employed in modern symbolic execution tools (Berdine et al. 2005; Rowe and Brotherston 2017). We demonstrate how to harness all this power for program synthesis, devise the corresponding search procedure and apply it to synthesize a number of non-trivial programs that manipulate linked data structures. Finally, we show how to exploit laws of SL and properties of our proof system to prune the search space and make the synthesis machinery efficient for realistic examples.

Contributions. The central theoretical contribution of the paper is *Synthetic Separation Logic* (SSL): a system of deductive synthesis rules, which prescribe how to decompose specifications for complex programs into specifications for simpler programs, while synthesizing the corresponding computations compositionally. In essence, SSL is a proof system for a new *transforming entailment* judgment $\mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$ (reads as "the assertion $\mathcal{P}$ transforms into $\mathcal{Q}$ via a program c"), which unifies SL entailment $\mathcal{P} \vdash \mathcal{Q}$ and verification $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$, with the former expressible as $\mathcal{P} \rightsquigarrow \mathcal{Q} \mid \texttt{skip}$.

The central practical contribution is the design and implementation of SuSLik—a deductive synthesizer for heap-manipulating programs, based on SSL. SuSLik takes as its input a library of inductive predicates, a (typically empty) list of auxiliary function specifications, and an SL specification of the function to be synthesized. It returns a—possibly recursive, but loop-free—program (in a minimalistic C-like language), which *provably* satisfies the given specification.

Our evaluation shows that SuSLik can synthesize all structurally-recursive benchmarks from previous work on heap-based synthesis (Qiu and Solar-Lezama 2017), *without any sketches* and in most cases much faster. To the best of our knowledge, it is also *the first synthesizer* to automatically discover the implementations of copying linked lists and trees, and flattening a tree to a list.

The essence of SuSLik's synthesis algorithm is a backtracking search in the space of SSL derivations. Even though the structural constraints (*i.e.*, the shape of the heap) embodied in the synthesis rules already prune the search space significantly (as shown by our `swap` example), a naïve backtracking search is still impractical, especially in the presence of inductive heap predicates. To eliminate redundant backtracking, we develop several principled optimizations. In particular, we draw inspiration from *focusing proof search* (Pfenning 2010) to identify *invertible* synthesis rules that do not require backtracking, and exploit the *frame rule* of SL, observing that the order of rule applications is irrelevant whenever their subderivations have disjoint footprints.

Paper outline. In the remainder of the paper we give an overview of SSL reasoning principles (Sec. 2), describe its rules and the meta-theory (Sec. 3), outline the design of our synthesis tool (Sec. 4), present the optimizations and extensions of the basic search algorithm (Sec. 5), and report on the evaluation of the approach on a set of case studies involving various linked structures (Sec. 6), concluding with a discussion of limitations (Sec. 7) and a comparison with the related work (Sec. 8).

2 DEDUCTIVE SYNTHESIS FROM SEPARATION LOGIC SPECIFICATIONS

In Separation Logic, assertions capture the program state, represented by a symbolic heap. An SL assertion (ranged over by symbols $\mathcal{P}$ and $\mathcal{Q}$ in the remainder of the paper) is customarily represented as a pair $\{\phi; P\}$ of a *pure* part ϕ and a *spatial* part P. The *pure* part (ranged over by ϕ, ψ, and χ) is a quantifier-free boolean formula, which describes the constraints over symbolic values (represented by variables x, y, *etc*). The *spatial* part (denoted P, Q, and R) is represented by a collection of primitive heap assertions describing disjoint symbolic heaps (*heaplets*), conjoined by the *separating conjunction* operation $*$, which is commutative and associative (Reynolds 2002). For example, in the assertion $\{a \neq b; x \mapsto a * y \mapsto b\}$ the spatial part describes two disjoint memory cells that store symbolic values a and b, while the pure part states that these values are distinct.

Our development is agnostic to the exact logic of pure formulae, as long as it supports standard boolean connectives and equality, and comes equipped with a *validity oracle* and a *synthesis oracle* (we elaborate on both later in this section). The soundness of our development depends only on the soundness of the validity oracle. Our implementation uses the quantifier-free logic of arrays, uninterpreted functions, and linear integer arithmetic, which is sufficient to express all examples in this paper, and is efficiently decidable by SMT solvers, thus providing a sound and complete validity oracle.

To begin with our demonstration, the only kinds of heaplets we are going to consider are the *empty heap* assertion emp and *points-to* assertions of the form $\langle x, \iota \rangle \mapsto e$, where x is a variable or a pointer constant (*e.g.*, 0), ι is a non-negative integer offset (necessary to represent records), and e is a symbolic value stored in the memory cell addressed via the value of $(x + \iota)$.[2] In most cases, the offset is 0, so we will abbreviate heap assertions $\langle x, 0 \rangle \mapsto e$ as $x \mapsto e$.

Our programming component (to be presented formally in Sec. 3) is a simple imperative language that supports reading from pointers to (immutable) local variables (`let x = *y`), storing values into pointers (`*y = e`), conditionals, recursive calls, and pure expressions. The language has no `return` statement; instead, a function stores its result into an explicitly passed pointer.

2.1 Specifications for Synthesis

A synthesis *goal* is a triple $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q}$, where Γ is an *environment*, *i.e.*, a set of immutable program variables, $\mathcal{P}$ is a *precondition*, and $\mathcal{Q}$ is a *postcondition*. Solving a synthesis goal amounts to finding a program c and a derivation of the SSL assertion $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$. To avoid clutter, we employ the following naming conventions:

(a) the symbols $\mathcal{P}$, ϕ, and P refer to the goal's *precondition*, its pure, and spatial part;
(b) similarly, the symbols $\mathcal{Q}$, ψ, and Q refer to the goal's *postcondition*, its pure and spatial part;
(c) whenever the pure part of a SL assertion is true ($\top$), it is omitted from the presentation.

In addition, we will use the following macros to express the *scope* and the *quantification* over variables of a goal $\Gamma; \{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\}$. First, by $\mathsf{Vars}(A)$ we will denote *all* variables occurring in A, which might be an assertion, a logical formula, or a program. *Ghosts* (universally-quantified logical variables), whose scope is both the pre- and the postcondition, are defined as $\mathsf{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \mathsf{Vars}(\mathcal{P}) \setminus \Gamma$. *Existentials* are defined as $\mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \mathsf{Vars}(\mathcal{Q}) \setminus (\Gamma \cup \mathsf{Vars}(\mathcal{P}))$. For instance, taking $\Gamma = \{x\}$, $\mathcal{P} = \{x \neq y; x \mapsto y\}$, $\mathcal{Q} = \{x \mapsto z\}$, we have the ghosts $\mathsf{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \{y\}$, the existentials $\mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \{z\}$, and the pure part of the postcondition $\mathcal{Q}$ is implicitly true.

2.2 Basic Inference Rules

To get an intuition on how to represent program synthesis as a proof derivation in SSL, consider Fig. 1, which shows four basic rules of the logic, targeted to synthesize programs with constant memory footprint (remember that we use $\mathcal{P}$ and $\mathcal{Q}$ for the entire pre/post in a rule's conclusion).

The Emp rule is applicable when spatial pre- and postcondition are both empty. It requires that no existentials remains in the goal, and the pure precondition implies the pure postcondition: the premise $\vdash \phi \Rightarrow \psi$ represents an invocation of the *pure validity oracle*. Emp has no subgoals in its premises (*i.e.*, it is a *terminal* rule), and no computational effect: its witness program is simply `skip`.

The Read rule turns a ghost variable a into a program variable y (fresh in the original goal). That is, the newly assigned immutable program variable y is added to the environment of the sub-goal, and all occurrences of a are substituted by y in both the pre- and postcondition. As a side-effect, the rule prepends the read statement **let** $y = *(x + \iota)$ to the remainder of the program to be synthesized.

[2] Further in the paper, we will extend the language of heap assertions to support dynamically allocated memory blocks and user-defined inductive predicates.

$$\textsc{Emp}\quad \frac{\mathsf{EV}(\Gamma, \mathcal{P}, Q) = \emptyset \qquad \vdash \phi \Rightarrow \psi}{\Gamma;\ \{\phi; \mathsf{emp}\} \rightsquigarrow \{\psi; \mathsf{emp}\} \mid \mathtt{skip}}$$

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

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

$$\textsc{Frame}\quad \frac{\mathsf{EV}(\Gamma, \mathcal{P}, Q) \cap \mathsf{Vars}(R) = \emptyset \qquad \Gamma;\ \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c}{\Gamma;\ \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R\} \mid c}$$

Fig. 1. Simplified basic rules of SSL.

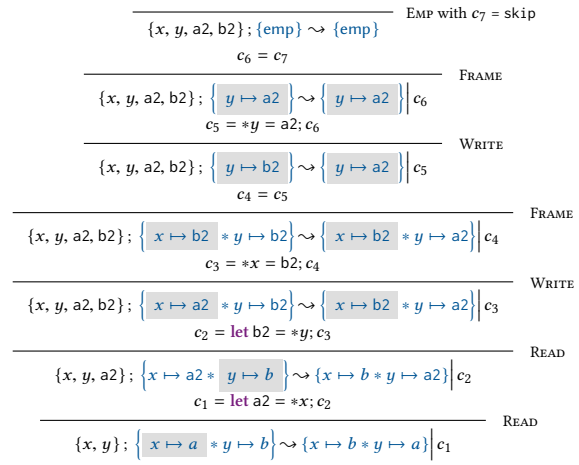

Fig. 2. Derivation of `swap(x,y)` as c_1.

The rule Write allows for writing a symbolic expression e into a memory cell $\langle x, \iota\rangle$, provided all e's variables are program-level. Notice that our Write rule, unlike the one in classical SL (O'Hearn et al. 2001), requires that a matching points-to heaplet be present in the postcondition, in order to decide which expression e to write. In the interest of modularity, the rule does not remove the resulting identical heaplets $\langle x, \iota\rangle \mapsto e$ from the sub-goal, leaving this to a more general rule Frame—SSL's version of Separation Logic's *frame rule*. This rule enables "framing out" a shared sub-heap R from the pre- and postcondition, as long as this does not create new existential variables. Notice, that unlike the classical SL's Frame rule, our version does not require a side condition saying that R must not contain program variables that are modified by the program c, because all variables in our language are immutable.

Synthesizing swap. Armed with the basic SSL inference rules from Fig. 1, let us revisit our initial example: the swap function (1). Fig. 2 shows the derivation of the program using the rules, and should be read bottom-up. For convenience, we name each subgoal's witness program, starting from c_1 (which corresponds to swap's body). Furthermore, each intermediate sub-goal highlights via gray boxes a part of the specification, which "triggers" the corresponding SSL rule. Intuitively, the goal of the synthesis process is to "empty" the spatial parts of the pre- and the postcondition, so that the derivation can eventually be closed via Emp; to this end, Read and Write work together to create matching heaplets between the two assertions, which are then eliminated by Frame.

2.3 Spatial Unification and Backtracking

Now, consider the synthesis goal induced by the following SL specification:

$$\{x \mapsto 239 * y \mapsto 30\}\ \mathbf{void}\ \mathtt{pick(loc\ x,\ loc\ y)}\ \{x \mapsto z * y \mapsto z\} \tag{2}$$

Since z does not appear among the formals or in the precondition, it is treated as an existential. The postcondition thus allows x and y to point to any value, as long as it is the same value.

To deal with existentials in the heap, we introduce the rule UnifyHeaps, which attempts to find a unifying substitution σ for some sub-heaps of the pre- and the

$$\textsc{UnifyHeaps}\quad \frac{[\sigma]R' = R \qquad \emptyset \neq \mathsf{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad \Gamma;\ \{P * R\} \rightsquigarrow [\sigma]\{\psi; Q * R'\} \mid c}{\Gamma;\ \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R'\} \mid c}$$

Fig. 3. SSL rule for heap unification.

postcondition. The domain of σ must only contain existentials. For example, applying UnifyHeaps to the spec (2) with $R \triangleq x \mapsto 239$ and $R' \triangleq x \mapsto z$ results in the substitution $\sigma = [z \mapsto 239]$, and the residual synthesis goal $\{x, y\}\ \{x \mapsto 239 * y \mapsto 30\} \rightsquigarrow \{x \mapsto 239 * y \mapsto 239\}$, which can be now synthesized by using the Frame,Write, and Emp rules.

Due to its freedom to choose a sub-heap and a unifying substitution, UnifyHeaps introduces non-determinism into the synthesis procedure and might require backtracking—a fact also widely observed in interactive verification community (Gonthier et al. 2011; McCreight 2009) *wrt.* SL assertions. For instance, consider the spec below:

$$\{x \mapsto a * y \mapsto b\}\ \texttt{void notSure(loc x, loc y)}\ \{x \mapsto c * c \mapsto 0\} \tag{3}$$

One way to approach the spec (3) is to first read from x, making a a program-level variable `a2` (via Read), then use UnifyHeaps and Frame on the $x \mapsto \bullet$ heaplets in the pre-/postcondition, substituting the existential c by $a2$.

```
void notSure(loc x, loc y) {
  *x = y;
  *y = 0;
}
```

That, however, leaves us with an unsolvable goal $\{x, y, \texttt{a2}\}\ \{y \mapsto b\} \rightsquigarrow \{\texttt{a2} \mapsto 0\}$. Hence we have to backtrack, and instead unify c with y, eventually deriving the correct program `notSure`.

2.4 Reasoning with Pure Constraints

So far we have only looked at SL specifications whose *pure* parts were trivially true. Let us now turn our attention to the goals that make use of non-trivial pure boolean assertions.

2.4.1 Preconditions. To leverage pure *preconditions*, we adopt a number of the traditional Smallfoot-style rules (Berdine et al. 2006), whose SSL counterparts are shown in the top part of Fig. 4. In the nomenclature of Berdine *et al.*, all those rules are *non-operational*, *i.e.*, correspond to constructing the proofs of symbolic heap entailment and involve no programming component. Note that the original rules in Berdine et al. (2005) assume a restricted pure logic with only equalities; we adapt these rules to our logic-agnostic style, relying on the oracle for pure validity instead of original syntactic premises. For instance, the rule SubstLeft makes use of a precondition that implies equality between two universal variables, $x = y$, substituting all occurrences of x in the subgoal by y. The rule StarPartial makes explicit the fundamental assumption of SL: disjointness of symbolic heaps connected by the $*$ operator. Most commonly, this rule's effect is observable in combination with another rule, Inconsistency, which identifies an inconsistent precondition,[3] and emits an always-failing program `error`.

These three SSL rules can be observed in action via the following example:

$$\{a = x \wedge y = a; x \mapsto y * y \mapsto z\}\ \texttt{void urk(loc x, loc y)}\ \{\text{true}; y \mapsto a * x \mapsto y\} \tag{4}$$

After applying SubstLeft, the goal transforms to $\{x, y\}\ \{x \mapsto x * x \mapsto z\} \rightsquigarrow \{x \mapsto x * x \mapsto x\}$, which is a trivial, as the precondition requires two *disjoint* points-to heaplets with the same source—a fact, which converted into a pure sub-formula $x \neq x$ by StarPartial (*i.e.*, false), resulting in the never-occurring `error` body (via Inconsistency), as no pre-state would satisfy its precondition.

2.4.2 Postconditions. In the presence of non-trivial pure *postconditions*, we face the problem of finding suitable terms to substitute for their existentials. This is essentially a program synthesis problem where both the specifications and the solution are terms from the pure logic. This is a challenging, yet well-studied problem (Alur et al. 2013), which we consider orthogonal to our agenda of deriving pointer-manipulating programs. Hence, SSL relies on a *pure synthesis oracle* to pick appropriate instantiations for existentials; formally, the oracle is represented by a non-deterministic

[3] Inconsistent preconditions are unlikely to appear in top-level goals, but frequently arise as a result of a case split, where they signify "dead" (*i.e.*, unreachable) branches.

$$\textsc{SubstLeft}\ \frac{\vdash \phi \Rightarrow x = y \quad x \in \Gamma \Rightarrow y \in \Gamma \quad \Gamma;\ [y/x]\{\phi; P\} \rightsquigarrow [y/x]\{Q\} \,|\, c}{\Gamma;\ \{\phi; P\} \rightsquigarrow \{Q\} \,|\, c}$$

$$\textsc{StarPartial}\ \frac{x + \iota \neq y + \iota' \notin \phi \qquad \phi' = \phi \wedge (x + \iota \neq y + \iota') \quad \Gamma;\ \{\phi'; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \rightsquigarrow \{Q\} \,|\, c}{\Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \rightsquigarrow \{Q\} \,|\, c}$$

$$\textsc{Inconsistency}\ \frac{\phi \Rightarrow \bot}{\Gamma;\ \{\phi; P\} \rightsquigarrow \{Q\} \,|\, \texttt{error}}$$

$$\textsc{Pick}\ \frac{x \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \quad \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [\psi/x]\{Q\} \,|\, c}{\Gamma;\ \{\mathcal{P}\} \rightsquigarrow \{Q\} \,|\, c}$$

$$\textsc{SubstRight}\ \frac{x \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \quad \Sigma; \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [\psi'/x]\{\psi \wedge x = \psi', Q\} \,|\, c}{\Sigma; \Gamma;\ \{\mathcal{P}\} \rightsquigarrow \{\psi \wedge x = \psi'; Q\} \,|\, c}$$

$$\textsc{UnifyPure}\ \frac{[\sigma]\psi' = \phi' \quad \emptyset \neq \mathrm{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, Q) \quad \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [\sigma]\{Q\} \,|\, c}{\Gamma;\ \{\phi \wedge \phi'; P\} \rightsquigarrow \{\psi \wedge \psi'; Q\} \,|\, c}$$

Fig. 4. Selected SSL rules for reasoning with pure constraints in the synthesis goal.

rule Pick (Fig. 4). In practice, the oracle can be realized, for example, by delegating to an existing pure synthesizer (Kuncak et al. 2010; Reynolds et al. 2015). In our implementation, however, we found a combination of three rules—the "one-point rule" (SubstRight), first-order unification (UnifyPure), and restricted enumerative search (Pick with ψ restricted to program variables)—to be quite effective, if theoretically incomplete, at discharging such synthesis goals.

As an example, consider the following goal (where S and S_1 are finite sets):

$$\{S = \{\mathrm{v}\} \cup S_1; \mathrm{x} \mapsto a\}\ \texttt{void elem(loc x, int v)}\ \{S = \{v_1\} \cup S_1; \mathrm{x} \mapsto v_1 + 1\} \tag{5}$$

Following the rule UnifyPure, one can unify the two facts about sets in the pre- and the postcondition, obtaining the substitution $[v_1 \mapsto v]$. The rest is accomplished by the rule Write, which emits the only necessary statement for `elem`'s body: `*x = v + 1`.

2.5 Synthesis with Inductive Predicates

The real power of Separation Logic stems from its ability to compositionally reason about linked heap-based data structures, such as lists and trees, whose shape is defined recursively via *inductive heap predicates*. The most traditional example of a data structure defined this way is a linked list segment (Reynolds 2002), whose definition is given by the two-clause predicate below:

$$\begin{aligned} \mathsf{lseg}(x, y, S) \triangleq\ & x = y \wedge \{S = \emptyset; \mathsf{emp}\} \\ \mid\ & x \neq y \wedge \{S = \{v\} \cup S_1; [x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto nxt * \mathsf{lseg}(nxt, y, S_1)\} \end{aligned} \tag{6}$$

The predicate $\mathsf{lseg}(x, y, S)$ describes a linked list that starts at location x, ends at location y, and contains a set of elements S.[4] The first clause states that, if x and y are equal, the list contains no elements and occupies an empty heap emp. The complementary second clause postulates the existence of an *allocated memory block* (or just *block*) of two consecutive locations rooted at x ($[x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto nxt$), such that the first location stores the payload v, while the second one stores a pointer nxt to the tail of the list, whose shape is defined recursively as $\mathsf{lseg}(nxt, y, S_1)$.

In general, a predicate definition $\mathcal{D} \triangleq p(\overline{x_i})\overline{\langle e_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N}$, starts with the name p and a vector or formal parameters $\overline{x_i}$, followed by a sequence of N *inductive clauses*. The j^{th} clause consist of a *guard* e_j—a *boolean* expression over the predicate's formals,[5] followed by the *clause body*—a SL assertion with a spatial part R_j and pure part χ_j, describing the shape of the heap and pure

[4] The predicate can be parametrized by different abstractions: the length of the list, its set/multiset of elements, or even an algebraic list datatype. The choice of abstraction is orthogonal to the story of this paper. In our examples, we chose the set abstraction, which supports an efficient SMT encoding and results in sufficiently strong specifications for synthesis.

[5] Program-level expressions are a subset of pure logic terms, free from non-executable constructs, such as set operations. In the case of logical overlap, the conditions for different clauses are checked in the order the clauses are defined.

constraints, correspondingly. Free variables of the clauses (*e.g.*, v, *nxt* above) are treated as ghosts or existentials, depending on whether the predicate instance is in a pre- or postcondition of a goal. We require that all predicates be *well-founded—i.e.*, have their recursive applications only on strictly smaller sub-heaps (Brotherston et al. 2008); we ensure well-foundedness via a syntactic check that each recursive clause contains at least one points-to heaplet. From now on, we extend the definition of the goal, with a *context* Σ, which stores the definitions of inductive predicates and specified functions, which are accessible in the derivation.

2.5.1 Dynamic Memory. In order to support dynamically allocated linked structures, as demonstrated by definition (6), we extend the language of symbolic heaps with two new kinds of assertions: *blocks* and *predicate instances*. Symbolic blocks are a well-established way to add to SL support for consecutive memory locations (Brotherston et al. 2017; Jacobs et al. 2011), which are allocated and disposed all together.[6] Two SSL rules, Alloc and Free (presented in Sec. 3), make use of blocks, as those appear in the post- and the preconditions of their corresponding goals. Conceptually, Alloc looks for a block in the postcondition rooted at an existential and allocates a block of the same size (by emitting the command `let x = malloc(n)`), adding it to the subgoal's precondition. Free is triggered by an un-matched block in the goal's precondition, rooted at some program variable `x`, which it then disposes by emitting the call to `free(x)`, removing it from the subgoal's precondition.

2.5.2 Induction. Let us now synthesize our first recursive heap-manipulating function, a linked list's *destructor* `listfree(x)`, which expects a linked list starting from its argument x and ending with the null-pointer, and leaves an empty heap as its result (we explain the meaning of the tag 0 in $\mathsf{lseg}^0(x, 0, S)$ below):

$$\{\mathsf{lseg}^0(x, 0, S)\}\ \texttt{void listfree(loc x)}\ \{\mathsf{emp}\} \tag{7}$$

The first synthesis step is carried out by the SSL rule Induction. We postpone its formal description until the next section, conveying the basic intuition here. Induction only applies to the *initial* synthesis goal whose precondition contains an inductive predicate instance, and its effect is to add a new *function symbol* to the goal's context, such that an invocation of this function would correspond to a recursive call. In our example (7) Induction extends the context Σ with a "recursive hypothesis" as follows:[7]

$$\Sigma_1 \triangleq \Sigma, \texttt{listfree}(x') : \{\mathsf{lseg}^1(x', 0, S')\}\{\mathsf{emp}\} \tag{8}$$

2.5.3 Unfolding Predicates. The top-level rule Induction is complemented by the rule Open (defined in Sec. 3), which *unfolds* a predicate instance in the goal's precondition according to its definition, and creates a subgoal for each inductive clause. For instance, invoked immediately on our goal (7), it has the following effect on the derivation:

(a) Two sub-goals, one for each of the clauses of lseg, are generated to solve:
   (*i*) $\Sigma_1; \{x\}; \{x = 0 \land S = \emptyset; \mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\}$
   (*ii*) $\Sigma_1; \{x\}; \{x \neq 0 \land S = \{v\} \cup S_1; [x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto nxt * \mathsf{lseg}^1(nxt, y, S_1)\} \rightsquigarrow \{\mathsf{emp}\}$
(b) Assuming c_1 and c_2 are the programs solving the sub-goals (*i*) and (*ii*), the final program is obtained by combining them as `if` $(x = 0)$ $\{c_1\}$ `else` $\{c_2\}$.

Thus, Open performs case-analysis according to the predicate definition. Note how the precondition of each generated sub-goal is *refined* by the corresponding clause's guard and body. The resulting sub-programs, once synthesized, are combined with a conditional statement that branches on the

[6] An alternative would be to adopt an object model with *fields*, which is more verbose (Berdine et al. 2005).
[7] In SSL, a context Σ can also store user-provided specifications of auxiliary functions synthesized earlier. We will elaborate on case studies relying on user-provided auxiliary functions in Sec. 6.2.

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

$$\textsc{Close}\quad \frac{\mathcal{D} \triangleq p(\overline{x_i})\overline{\langle e_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N} \in \Sigma \quad \ell < \mathsf{MaxUnfold} \quad 1 \le k \le N \quad \sigma \triangleq [\overline{x_i \mapsto y_i}] \quad R' \triangleq \lceil [\sigma]R_k \rceil^{\ell+1} \quad \Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\psi \wedge [\sigma]e_k \wedge [\sigma]\chi_k; Q * R'\} \,|\, c}{\Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\psi; Q * p^{\ell}(\overline{y_i})\} \,|\, c}$$

Fig. 5. Selected SSL rules for synthesis with recursive functions and inductive predicates.

predicate's guard (this is why we require the guards to be program expressions). It is easy to see that the first sub-goal (*i*) can be immediately solved via Emp rule, producing the program `skip`.

The second subgoal (*ii*), contains another instance of the inductive predicate—$\mathsf{lseg}^1(nxt, y, S_1)$. In principle, this instance can be unfolded again, yielding in turn a third instance, and so on. To avoid infinite unfolding of predicate instances we introduce *level tags* (natural numbers, ranged over by ℓ), which now annotate some predicate instances in the pre- and postcondition of the goal and the context functions. All predicates in the initial goal have their level tag set as $\ell = 0$. The rule Open only applies to instances with $0 \le \ell <$ MaxUnfold, incrementing their tags (hence, the number of "telescopic" unfoldings is bounded by the global parameter MaxUnfold).

2.5.4 Recursive calls. Synthesizing recursive programs requires extra care in order to avoid vacuously correct (in the sense of partial program correctness) *non-terminating* programs that simply call themselves. To ensure termination of synthesized programs, we require that the pre-heap P_f of any recursive call be *strictly smaller* than the pre-heap P of the top-level synthesis goal. Following ideas from Cyclic Termination Proofs (Brotherston et al. 2012), we enforce this restriction *syntactically*: since all inductive predicates are well-founded, it is sufficient to check that one of the predicate instances in P_f has been derived from the corresponding instance in P by unfolding. To perform this check, we piggy-back on level tags: intuitively, if every level tag in P starts out as 0 and is only incremented as a result of Open, a recursive call on an instance with $\ell > 0$ is guaranteed to terminate.

To see how tags control function calls, consider the rule Call in Fig. 5. It fires when the goal contains in its precondition a symbolic sub-heap R, which can be unified with the precondition P_f of a function symbol f from the goal's context Σ. This unification is similar to the effect of UnifyHeaps, with the difference that Call takes level tags into the account (reflected in the tag-aware equality predicate $=^{\ell}$), while UnifyHeaps (Fig. 3) and Frame (Fig. 1) ignore them when comparing sub-heaps for equality.

Returning to the `listfree` example, the remaining subgoal (*ii*)

$$\{x\}; \left\{x \neq 0 \wedge S = \{v\} \cup S_1; [x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto nxt * \mathsf{lseg}^1(nxt, y, S_1)\right\} \rightsquigarrow \{\mathsf{emp}\} \tag{9}$$

can be now transformed, via Read (focused on $\langle x, 1\rangle \mapsto nxt$), into

$$\{x, \mathtt{nxt2}\}; \left\{x \neq 0 \wedge S = \{v\} \cup S_1; [x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto \mathtt{nxt2} * \mathsf{lseg}^1(\mathtt{nxt2}, y, S_1)\right\} \rightsquigarrow \{\mathsf{emp}\} \tag{10}$$

The grayed fragment in (10) can now be unified with the precondition of `listfree` (8) following Call's premise. As the tags match (both indicate the *first* unfolding of the predicate), unification succeeds with the substitution $\sigma = [x' \mapsto \mathtt{nxt2}, S' \mapsto S_1]$ from f's parameters and ghosts to the goal variables. The same rule produces, from the f's postcondition, a new symbolic heap R', which replaces the targeted fragment in the precondition, recording the effect

```
void listfree(loc x) {
  if (x = 0) {} else {
    let nxt2 = *(x + 1);
    listfree(nxt2);
    free(x);
  } }
```

Fig. 6. Synthesized `listfree` (7).

of the call. In this example, the function's postcondition is $\{\mathsf{emp}\}$, so the goal becomes:

$$\{x, \mathtt{nxt2}\}\,; \{x \neq 0 \land S = \{v\} \cup S_1; [x,2] * x \mapsto v * \langle x, 1\rangle \mapsto \mathtt{nxt2} * \mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \tag{11}$$

The remaining steps are carried out by the rule Free, followed by Emp, with the former disposing the remaining block, thus, completing the derivation with program `listfree` shown in Fig. 6.

2.5.5 Unfolding in the postcondition. Whereas Open unfolds predicate instances in a goal's precondition, a complementary rule Close (Fig. 5) performs a similar operation on the goal's postcondition. The main difference is that instead of performing a case-split and emitting several subgoals, Close *non-deterministically picks* a single clause k from the predicate's definition (the intuition being that the required case split has already been performed by Open). Upon unfolding, the clause's adapted guard ($[\sigma]e_k$) and pure part ($[\sigma]\chi_k$) are added to the subgoal's postcondition, while its spatial part also gets its level tags increased by one ($\lceil [\sigma]R_k \rceil^{\ell+1}$), in order to account for the depth of unfoldings.

```
void listmorph(loc x, loc r) {
  if (x = 0) { } else {
    let v2 = *x;
    let nxt2 = *(x + 1);
    listmorph(nxt2, r);
    let y12 = *r;
    let y2 = malloc(3);
    free(x);
    *(y2 + 2) = y12;
    *(y2 + 1) = v2 + 1;
    *y2 = v2;
    *r = y2; } }
```

Fig. 7. Synthesized `listmorph` (13).

To showcase the use of Close, let us define a new predicate for a linked null-terminating structure lseg2, which stores in each node the payload v and $v+1$:

$$\begin{array}{l} \mathsf{lseg2}(x, S) \triangleq x = 0 \land \{S = \emptyset; \mathsf{emp}\} \\ \quad \mid \; x \neq 0 \land \left\{ \begin{array}{l} S = \{v\} \cup S_1; \\ [x,3] * x \mapsto v * \langle x, 1\rangle \mapsto v + 1 * \langle x, 2\rangle \mapsto nxt * \mathsf{lseg2}(nxt, S_1) \end{array} \right\} \end{array} \tag{12}$$

We now synthesize an implementation for the following specification, requiring to morph a regular list $\mathsf{lseg}(x, 0, S)$ to $\mathsf{lseg2}(y, S)$, both parameterized by the same set S:

$$\{r \mapsto 0 * \mathsf{lseg}^0(x, 0, S)\}\ \mathtt{void\ listmorph(loc\ x,\ loc\ r)}\ \{r \mapsto y * \mathsf{lseg2}^0(y, S)\} \tag{13}$$

The derivation starts with Induction, which produces the following function symbol (note that the postcondition always has all level tags *erased* in order to prevent chaining of recursive calls):

$$\mathtt{listmorph}(x', r') : \{r' \mapsto 0 * \mathsf{lseg}^1(x', 0, S')\}\{r' \mapsto y' * \mathsf{lseg2}(y', S')\} \tag{14}$$

Next, we Open $\mathsf{lseg}(x, 0, S)$, producing two sub-goals. The first one:

$$\{x, r\}\,; \{S = \emptyset \land x = 0; r \mapsto 0\} \rightsquigarrow \{r \mapsto y * \mathsf{lseg2}^0(y, S)\} \tag{15}$$

is easy to solve via Close, which should pick the first clause from lseg2's definition (12) (corresponding to emp), followed by Frame to $r \mapsto y$ in the postcondition. The second subgoal, after having read the value of $(x+1)$ into a program variable `nxt2`, looks as follows:

$$\begin{array}{l} \{x, r, \mathtt{nxt2}\}\,; \\ \{S = \{v\} \cup S_1 \land x \neq 0; r \mapsto 0 * [x,2] * x \mapsto v * \langle x, 1\rangle \mapsto \mathtt{nxt2} * \mathsf{lseg}^1(\mathtt{nxt2}, 0, S_1)\} \rightsquigarrow \\ \{r \mapsto y * \mathsf{lseg2}^0(y, S)\} \end{array} \tag{16}$$

Now, Close comes to the rescue, unfolding the grayed instance in (16)'s postcondition:

$$\begin{array}{l} \{x, r, \mathtt{nxt2}\}\,; \\ \{S = \{v\} \cup S_1 \land x \neq 0; r \mapsto 0 * [x,2] * x \mapsto v * \langle x, 1\rangle \mapsto \mathtt{nxt2} * \mathsf{lseg}^1(\mathtt{nxt2}, 0, S_1)\} \rightsquigarrow \\ \{S = \{v_1\} \cup S_2; r \mapsto y * [y,3] * y \mapsto v_1 * \langle y, 1\rangle \mapsto v_1 + 1 * \langle y, 2\rangle \mapsto nxt_1 * \mathsf{lseg2}^1(nxt_1, S_2)\} \end{array} \tag{17}$$

We can now use the Call rule, unifying the precondition of the induction hypothesis (14) with the grayed parts in the goal (17), obtaining the following subgoal:

$$\{x, r, \texttt{nxt2}\}\,;\quad \left\{S = \{v\} \cup S_1 \wedge x \neq 0;\ \boxed{[x,2] * x \mapsto v * \langle x, 1\rangle \mapsto \texttt{nxt2}} * r \mapsto y_1 * \mathsf{lseg2}(y_1, S_1)\right\} \rightsquigarrow$$
$$\left\{S = \{v_1\} \cup S_2; r \mapsto y * \boxed{[y,3] * y \mapsto v_1 * \langle y, 1\rangle \mapsto v_1 + 1 * \langle y, 2\rangle \mapsto nxt_1} * \mathsf{lseg2}^1(nxt_1, S_2)\right\}$$

The instances of lseg2 in the pre- and the postcondition can now be unified via UnifyHeaps, instantiating $[nxt_1 \mapsto y_1, S_2 \mapsto S_1]$, and then framed via Frame. The remaining derivation is done by Reading from r and x, subsequent disposing of a two-cell block (grayed in the precondition) and allocation of a three-cell block in order to match the grayed block in the postcondition. Finally, the exact payload for cells of the newly-allocated 3-pointer block is determined by unifying the set assertions in the pure pre- and postcondition (via UnifyPure), and then Write records the right values to satisfy the constraints imposed for the head of lseg2-like list by Definition (12). The resulted synthesized implementation of `listmorph` is shown in Fig. 7.

2.6 Enabling Procedure Calls by Means of Call Abduction

We conclude this overview with one last example—a recursive procedure for copying a linked list:

$$\left\{r \mapsto x * \mathsf{lseg}^0(x, 0, S)\right\}\ \texttt{void listcopy(loc r)}\ \left\{r \mapsto y * \mathsf{lseg}^0(x, 0, S) * \mathsf{lseg}^0(y, 0, S)\right\} \tag{18}$$

To make things more fun, we pass the pointer to the head of the list via another pointer `r`, which is also used to record the result of the function—an address y of a freshly allocated list copy. The synthesis begins by using Induction, producing the function symbol

$$\texttt{void listcopy(loc r')} : \left\{r' \mapsto x' * \mathsf{lseg}^1(x', 0, S')\right\}\left\{r' \mapsto y' * \mathsf{lseg}(x', 0, S') * \mathsf{lseg}(y', 0, S')\right\} \tag{19}$$

It follows by Read (from r into `x2`) and Open, resulting in two subgoals, the first of which (an empty list) is trivial. The synthesis proceeds, reading from `x2` into `v2` and from `x2 + 1` into `nxt2`, so after using Close (on either of $\mathsf{lseg}^0(x, 0, S)$ heaplets) in the postcondition, UnifyHeaps and Frame, we reach the following subgoal:

$$\{x, r, \texttt{x2}, \texttt{v2}, \texttt{nxt2}\}\,;\ \left\{S = \{\texttt{v2}\} \cup S_1 \wedge \texttt{x2} \neq 0;\ \boxed{r \mapsto \texttt{x2} * \mathsf{lseg}^1(\texttt{nxt2}, 0, S_1)}\right\} \rightsquigarrow \tag{20}$$
$$\left\{S = \{\texttt{v2}\} \cup S_2; r \mapsto y * \mathsf{lseg}^1(\texttt{nxt2}, 0, S_2) * \mathsf{lseg}^0(y, 0, S)\right\}$$

At this point of our derivation, we run into an issue. Ideally, we would like to use the grayed fragment of the goal (20)'s precondition, to fire the rule Call with the spec (19), *i.e.* to make a recursive call on the tail list. However, the (19)'s precondition requires r to point to the start of that list (`nxt2`), whereas in our case it still points to the start of the original list (`x2`).

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

Fig. 8. AbduceCall rule.

Any programmer would know a solution to this conundrum: we have to to write `nxt2` into r, in order to provide a suitable symbolic heap to make a recursive call. The Write rule, however, cannot make this change on its own, since its application is guided by the desired post-heap (in order to decide what to write), which in this case is determined by the callee's precondition and not by the current goal. To overcome this limitation, we introduce a novel rule AbduceCall (shown in Fig. 8), which emits a separate synthesis sub-goal to *prepare* the symbolic pre-heap for the recursive call.

First, the rule splits the precondition of the candidate callee $\mathcal{F}$ from Σ into two symbolic sub-heaps, P_f and F_f, such that all predicate instances are in P_f, while the rest of the heaplets (*i.e.*, blocks and points-to assertions) are in F_f. Next, it tries to unify P_f with some sub-heap P from the goal's precondition, finding a suitable substitution σ, such that $P = [\sigma]P_f$. While doing so, it does not account for the "remainder" $[\sigma]F_f$, which might not be immediately matched by anything in the goal's precondition. In order to make it match, the rule emits a subgoal $\Sigma; \Gamma; \{\phi; F\} \rightsquigarrow \{\phi, F'\} \mid c_1$, whose purpose is to synthesize a program c_1, which will serve as an impedance matcher between *some* symbolic subheap F from the original goal's precondition and $F' = [\sigma]F_f$.[8]

```
1  void listcopy (loc r) {
2    let x2 = *r;
3    if (x2 = 0) { } else {
4      let v2 = *x2;
5      let nxt2 = *(x2 + 1);
6      *r = nxt2;
7      listcopy(r);
8      let y12 = *r;
9      let y2 = malloc(2);
10     *y2 = v2;
11     *(y2 + 1) = y12;
12     *r = y2;
13   } }
```

Fig. 9. Synthesized `listcopy` (18).

For instance, in the specification (19), $P_f = \mathsf{lseg}^1(x', 0, S')$ and $F_f = r' \mapsto x'$, so an attempt to unify the former with the predicate instance in the grayed fragment of the goal (20) results in the substitution $\sigma = [x' \mapsto \mathtt{nxt2}, S' \mapsto S_1]$. Applying it to the remainder of the function's precondition, we obtain $F' = [\sigma]F_f = r' \mapsto \mathtt{nxt2}$. One of the candidates for the role of F from the goal's precondition is the heaplet $r \mapsto \mathtt{x2}$, so the corresponding subgoal will be of the form $\{\mathtt{r}, \ldots\}; \{\ldots; \mathtt{r} \mapsto \mathtt{x2}\} \rightsquigarrow \{r' \mapsto \mathtt{nxt2}\}$, which will produce the write `*r = nxt2`. Fig. 9 shows the eventually synthesized implementation, with the abduced call-enabling write on line 6.

3 SYNTHETIC SEPARATION LOGIC IN A NUTSHELL

Having shown SSL in action, we now proceed with giving a complete set of its inference rules, along with statements of the formal guarantees SSL provides *wrt.* synthesized imperative programs.

The syntax for the imperative language supported by SSL is given in Fig. 10. The set of values includes at least true and false, integers, and locations (isomorphic to non-negative integers). Expressions include variables, values, boolean equality checks and additional theory-specific expressions (*e.g.*, integer arithmetic). The language of commands does not include loops, which are modelled via recursive procedure calls ($f(\overline{e_i})$). For simplicity we do not provide a mechanism to return a variable from a procedure (so the language is missing the **return** command); result-returning discipline for a procedure can be encoded via passing a result-storing additional pointer, as, *e.g.*, in Example (18). A *function dictionary* Δ is simply a list of function definitions of the form $f\ (\overline{x_i})\ \{\ c\ \}$. For brevity, we omit the type annotations (*e.g.*, in function signatures) in the formalism.

The complete syntax of SSL assertions is shown in Fig. 11, and their meaning was explained in detail throughout Sec. 2. We only notice here that, syntactically, pure logic terms are a superset of program-level expressions e, and can contain theory-specific non-executable constructs, such as set operations. In a SL assertion $\{\phi, P\}$, the term ϕ must be *boolean*; our implementation enforces this through a simple type system, which we omit from the formalization for simplicity.

3.1 The Zoo of SSL Rules

Fig. 12 presents *all* rules of core SSL.[9] Since most of them have already made an appearance in Sec. 2, here, we only elaborate on the new ones, and highlight some important aspects of their interaction. It is convenient to split the set of rules into the following six categories:

[8] Our implementation is smarter than that: it ensures that F and F' have the same shape and differ only in pointers' values.
[9] Sec. 5 extends core SSL with several optimizations.

Variable	x, y		Alpha-numeric identifiers
Value	d		Theory-specific atoms
Offset	ι		Non-negative integers
Expression	e	$::=$	$d \mid x \mid e = e \mid e \wedge e \mid \neg e \mid \ldots$
Command	c	$::=$	$\textbf{let}\ x = *(x + \iota) \mid *(x + \iota) = e \mid$
			$\texttt{skip} \mid \texttt{error} \mid f(\overline{e_i}) \mid$
			$\textbf{if}\ (e)\ \{c\}\ \textbf{else}\ \{c\} \mid c; c$
Fun. dict.	Δ	$::=$	$\epsilon \mid \Delta, f\ (\overline{x_i})\ \{\ c\ \}$

Fig. 10. Programming language grammar.

Pure logic term	ϕ, ψ, χ	$::=$	$e \mid \ldots$
Symbolic heap	P, Q, R	$::=$	$\text{emp} \mid \langle e, \iota \rangle \mapsto e \mid$
			$[x, n] \mid p(\overline{x_i}) \mid P * Q$
Assertion	$\mathcal{P}, \mathcal{Q}$	$::=$	$\{\phi, P\}$
Heap predicate	$\mathcal{D}$	$::=$	$p\ (\overline{x_i})\ \overline{\langle e_j, \{\chi_j, R_j\}\rangle}$
Function spec	$\mathcal{F}$	$::=$	$f\ (\overline{x_i}) : \{\mathcal{P}\}\{\mathcal{Q}\}$
Environment	Γ	$:=$	$\epsilon \mid \Gamma, x$
Context	Σ	$:=$	$\epsilon \mid \Sigma, \mathcal{D} \mid \Sigma, \mathcal{F}$

Fig. 11. SSL assertion syntax.

C1 *Top-level rules* are represented by just one rule: Induction. This rule is only applicable at the very first stage of the derivation, and it produces a function symbol f, with the specification identical to the top-level goal, modulo the level tag of a predicate instance $p^0(\overline{e_i})$ in the pre-condition. In the case of *several* predicate instances $p^0(\overline{e_i})$ in the goal's precondition, the rule non-deterministically picks one recursion scheme, whereas other predicate instances in f's precondition are "sealed" via tag erasure $\lceil P \rceil$.

C2 *Terminals* include Emp and Inconsistency. These rules conclude a successful derivation by emitting `skip` or `error`, respectively.

C3 *Normalization rules* include NullNotLVal, SubstLeft, StarPartial, and Read. The role of these rules is to normalize the precondition: eliminate ghosts and equal variables, and make explicit the assumptions encoded in the spatial part. As we discuss in Sec. 5.1, a characteristic feature of those rules is that they cannot cause their subderivation to fail.

C4 *Unfolding rules* are the rules targeted specifically to predicate instances. The four basic unfolding rules are Open, Close, AbduceCall, and Call. As hinted before, the rule Open picks an instance $p^\ell(\overline{e_i})$ in the goal's precondition, and produces a number of subgoals, emitting a conditional that accounts for all of p's clauses. Close acts symmetrically on the postcondition. Both rules increment tags for the instances in the clause bodies, and are not applicable for $\ell \geq \mathsf{MaxUnfold}$. Having the search depth bounded by the global parameter MaxUnfold affects the completeness but not soundness of SSL; $\mathsf{MaxUnfold} \leq 2$ suffices for all examples in the paper. The rule AbduceCall prepares the heap for a call application (via Call), as described in Sec. 2.6. Note that the rules UnifyHeaps and Frame have been slightly generalized *wrt.* to what was shown in Fig. 1 and Fig. 3, and now include a parametric premise `frameable`. If we define `frameable` to return true *only* for predicate instances, we obtain *unfolding versions* of these rules, which deal specifically with predicate instances.

C5 *Flat* rules deal with with the flat part of the heap (blocks and points-to heaplets). They include Alloc, Free,[10] Write, as well as the *flat versions* of UnifyHeaps and Frame, with `frameable` defined to return true for flat heaplets.

C6 *Pure synthesis* rules are responsible for instantiating existentials. In Fig. 12 this category is represented by a single non-deterministic rule Pick, which replaces an existential with an arbitrary pure term ψ. Note that the choice of ψ does not affect the soundness of SSL, but enumerating all possible terms would make the proof search intractable. In practice, we assume that ψ is suggested by a *pure synthesis oracle*. We describe a concrete oracle used in our implementation in Sec. 5.6.

[10]The iterated $*$ operator in the premises of Alloc and Free is a syntactic sugar for records of fixed size.

Induction

$$\frac{\begin{array}{c} f \triangleq \text{goal's name} \quad \overline{x_i} \triangleq \text{goal's formals} \\ P_f \triangleq p^1(\overline{y_i}) * \lceil P \rceil \qquad Q_f \triangleq \lceil Q \rceil \\ \mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f\}\{\psi_f; Q_f\} \\ \Sigma, \mathcal{F}; \Gamma; \{\phi; p^0(\overline{y_i}) * P\} \rightsquigarrow \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma; \{\phi; p^0(\overline{y_i}) * P\} \rightsquigarrow \{Q\} \,|\, c}$$

Emp

$$\frac{\mathsf{EV}(\Gamma, \mathcal{P}, Q) = \emptyset \qquad \vdash \phi \Rightarrow \psi}{\Gamma; \{\phi; \mathsf{emp}\} \rightsquigarrow \{\psi; \mathsf{emp}\} \,|\, \mathtt{skip}}$$

Inconsistency

$$\frac{\vdash \phi \Rightarrow \bot}{\Gamma; \{\phi; P\} \rightsquigarrow \{Q\} \,|\, \mathtt{error}}$$

NullNotLVal

$$\frac{x \neq 0 \notin \phi \qquad \phi' \triangleq \phi \wedge x \neq 0 \qquad \Sigma; \Gamma; \{\phi'; \langle x, \iota\rangle \mapsto e * P\} \rightsquigarrow \{Q\} \,|\, c}{\Sigma; \Gamma; \{\phi; \langle x, \iota\rangle \mapsto e * P\} \rightsquigarrow \{Q\} \,|\, c}$$

SubstLeft

$$\frac{\vdash \phi \Rightarrow x = y \qquad x \in \Gamma \Rightarrow y \in \Gamma \qquad \Gamma; [y/x]\{\phi; P\} \rightsquigarrow [y/x]\{Q\} \,|\, c}{\Gamma; \{\phi; P\} \rightsquigarrow \{Q\} \,|\, c}$$

StarPartial

$$\frac{x + \iota \neq y + \iota' \notin \phi \qquad \phi' \triangleq \phi \wedge (x + \iota \neq y + \iota') \qquad \Sigma; \Gamma; \{\phi'; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \rightsquigarrow \{Q\} \,|\, c}{\Sigma; \Gamma; \{\phi; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \rightsquigarrow \{Q\} \,|\, c}$$

Read

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

Open

$$\frac{\begin{array}{c} \mathcal{D} \triangleq p(\overline{x_i})\overline{\langle e_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N} \in \Sigma \\ \ell < \mathsf{MaxUnfold} \qquad \sigma \triangleq [\overline{x_i \mapsto y_i}] \qquad \mathsf{Vars}(\overline{y_i}) \subseteq \Gamma \\ \phi_j \triangleq \phi \wedge [\sigma]e_j \wedge [\sigma]\chi_j \qquad P_j \triangleq \lceil [\sigma]R_j \rceil^{\ell+1} * \lceil P \rceil \\ \forall j \in 1 \ldots N, \quad \Sigma; \Gamma; \{\phi_j; P_j\} \rightsquigarrow \{Q\} \,|\, c_j \\ c \triangleq \mathtt{if}\ ([\sigma]e_1)\ \{c_1\}\ \mathtt{else}\ \{\mathtt{if}\ ([\sigma]e_2) \ldots \mathtt{else}\ \{c_N\}\} \end{array}}{\Sigma; \Gamma; \{\phi; P * p^\ell(\overline{y_i})\} \rightsquigarrow \{Q\} \,|\, c}$$

Close

$$\frac{\begin{array}{c} \mathcal{D} \triangleq p(\overline{x_i})\overline{\langle e_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N} \in \Sigma \\ \ell < \mathsf{MaxUnfold} \qquad \sigma \triangleq [\overline{x_i \mapsto y_i}] \\ \text{for some } k,\ 1 \le k \le N \qquad R' \triangleq \lceil [\sigma]R_k \rceil^{\ell+1} \\ \Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\psi \wedge [\sigma]e_k \wedge [\sigma]\chi_k; Q * R'\} \,|\, c \end{array}}{\Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\psi; Q * p^\ell(\overline{y_i})\} \,|\, c}$$

AbduceCall

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

Call

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

Alloc

$$\frac{\begin{array}{c} R = [z, n] * *_{0 \le i \le n}(\langle z, i\rangle \mapsto e_i) \qquad z \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \\ (\{y\} \cup \{\overline{t_i}\}) \cap \mathsf{Vars}(\Gamma, \mathcal{P}, Q) = \emptyset \\ R' \triangleq [y, n] * *_{0 \le i \le n}(\langle y, i\rangle \mapsto t_i) \\ \Sigma; \Gamma; \{\phi; P * R'\} \rightsquigarrow \{\psi; Q * R\} \,|\, c \end{array}}{\Sigma; \Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q * R\} \,|\, \mathtt{let}\ y = \mathtt{malloc}(n); c}$$

Free

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

Write

$$\frac{\mathsf{Vars}(e) \subseteq \Gamma \qquad e \neq e' \qquad \Gamma; \{\phi; \langle x, \iota\rangle \mapsto e * P\} \rightsquigarrow \{\psi; \langle x, \iota\rangle \mapsto e * Q\} \,|\, c}{\Gamma; \{\phi; \langle x, \iota\rangle \mapsto e' * P\} \rightsquigarrow \{\psi; \langle x, \iota\rangle \mapsto e * Q\} \,|\, *(x + \iota) = e; c}$$

Frame

$$\frac{\mathsf{EV}(\Gamma, \mathcal{P}, Q) \cap \mathsf{Vars}(R) = \emptyset \qquad \mathsf{frameable}\ (R') \qquad \Gamma; \{\phi; P\} \rightsquigarrow \{\psi; Q\} \,|\, c}{\Gamma; \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R\} \,|\, c}$$

UnifyHeaps

$$\frac{[\sigma]R' = R \qquad \mathsf{frameable}\ (R') \qquad \emptyset \neq \mathsf{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad \Gamma; \{P * R\} \rightsquigarrow [\sigma]\{\psi; Q * R'\} \,|\, c}{\Gamma; \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R'\} \,|\, c}$$

Pick

$$\frac{x \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad \Gamma; \{\mathcal{P}\} \rightsquigarrow [\psi/x]\{Q\} \,|\, c}{\Gamma; \{\mathcal{P}\} \rightsquigarrow \{Q\} \,|\, c}$$

Fig. 12. All core SSL rules. Grayed parts are parameters; instantiating them differently yields different rules.

Let us refer to rules that emit a sub-program as *operational* and to the rest (*i.e.*, to the rules that only change the assertions) as *non-operational*. The rules from Fig. 12 form the basis of SSL as a proof system, allowing for possible extensions for the sake of optimization or handling pure constraints. We make them intentionally *declarative* rather than *algorithmic*, which is essential for establishing the logic's soundness, leaving a lot of freedom for possible implementations. Such decisions have to be made, for instance, when engineering an implementation of AbduceCall or Pick. The algorithmic aspects of SSL, *e.g.*, non-deterministic choice of a frame or a unifying substitution, are handled by the procedures from Sec. 4.

3.2 Formal Guarantees for the Synthesized Programs

The programs resulting from the SSL derivations enjoy both (a) validity (*i.e.*, they obey their ascribed Hoare-style specifications in a sence of partial correctness) and (b) termination. Therefore, the soundness result for SSL entails the Hoare-style *total* correctness. This result is formally stated by Theorem 3.6, which builds on the two components, establishing the validity and termination separately, as we describe in Sec. 3.2.1 and 3.2.2.

Preliminaries. The definition of the operational semantics of the SSL language (Fig. 10) follows the standard RAM model. *Heaps* (ranged over by h) are represented as partial finite maps from pointers to values, with support for pointer arithmetic (via offsets). A function call is executed within its own stack frame (c, s), where c is the next command to reduce and s is a *store* recording the values of the function's local variables and parameters. A stack S is a sequence of stack frames, and a *configuration* is a pair of a heap and a stack. The small-step operational semantics relates a function dictionary Δ, and a pair of configurations: $\Delta; \langle h, S\rangle \rightsquigarrow \langle h', S'\rangle$, with $\rightsquigarrow^*$ meaning its reflexive-transitive closure. We elide the transition rules for brevity; similar rules can by found, *e.g.*, in the work by Rowe and Brotherston (2017).

3.2.1 Validity. Let us denote the *valuation* of an expression e under a store s as $[\![e]\!]_s$. $\mathcal{I}$ ranges over *interpretations*—mappings from user-provided predicates $\mathcal{D}$ to the relations on heaps and vectors of values. To formally define the *validity* of Hoare-style specs in SSL, we use the standard definition of the satisfaction relation $\models^{\Sigma}_{\mathcal{I}}$ on pairs of heaps and stores, contexts, interpretations, and SSL assertions without ghosts. For instance, the following SSL definitions are traditional for Separation Logics with interpreted predicates (Berdine et al. 2005; Nguyen et al. 2007):

- $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; \mathsf{emp}\}$ *iff* $[\![\phi]\!]_s = \mathsf{true}$ and $\mathrm{dom}\,(h) = \emptyset$.
- $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; [x, n]\}$ *iff* $[\![\phi]\!]_s = \mathsf{true}$ and $\mathrm{dom}\,(h) = \emptyset$.
- $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; \langle e_1, \iota\rangle \mapsto e_2\}$ *iff* $[\![\phi]\!]_s = \mathsf{true}$ and $\mathrm{dom}\,(h) = [\![e_1]\!]_s + \iota$ and $h([\![e_1]\!]_s + \iota) = [\![e_2]\!]_s$.
- $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; P_1 * P_2\}$ *iff* $\exists h_1, h_2, h = h_1 \cup h_2$ and $\langle h_1, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; P_1\}$ and $\langle h_2, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; P_2\}$.
- $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} \{\phi; p(\overline{x_i})\}$ *iff* $[\![\phi]\!]_s = \mathsf{true}$ and $\mathcal{D} \triangleq p(\overline{x_i})\overline{\langle e_j, \{\chi_j, R_j\}\rangle} \in \Sigma$ and $\left\langle h, \overline{[\![x_i]\!]_s}\right\rangle \in \mathcal{I}(\mathcal{D})$.

Therefore, blocks have no spatial meaning, except for serving as an indicator on the memory fragments that we allocated and can be disposed.

The notion of Hoare-style validity is standard and is given by Definition 3.1.

Definition 3.1 (Validity). We say that a Hoare-style specification $\Sigma; \Gamma; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is *valid wrt.* the function dictionary Δ *iff* whenever $\mathrm{dom}\,(s) = \Gamma$, $\exists \sigma_{\mathrm{gv}} = [\overline{x_i \mapsto d_i}]_{x_i \in \mathrm{GV}(\Gamma, \mathcal{P}, \mathcal{Q})}$ such that $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} [\sigma_{\mathrm{gv}}]\mathcal{P}$, and $\Delta; \langle h, (c, s)\cdot\epsilon\rangle \rightsquigarrow^* \langle h', (\mathtt{skip}, s')\cdot\epsilon\rangle$, it is also the case that $\langle h', s'\rangle \models^{\Sigma}_{\mathcal{I}} [\sigma_{\mathrm{ev}} \cup \sigma_{\mathrm{gv}}]\mathcal{Q}$ for some $\sigma_{\mathrm{ev}} = [\overline{y_j \mapsto d_j}]_{y_j \in \mathrm{EV}(\Gamma, \mathcal{P}, \mathcal{Q})}$.

That is, the definition assumes an initial stack frame (c, s) and a concrete heap h, both consistent with an environment Γ and the precondition $\mathcal{P}$ (with σ_{gv} providing concrete values for $\mathcal{P}$'s ghost

variables). In the case if the program terminates with the store s' and a heap h', they satisfy the postcondition Q, with σ_{ev} instantiating Q's existentials.

Proving validity in the absence of auxiliary and recursive function calls would be almost trivial, thanks to similarities between SSL and traditional Separation Logic. Unfortunately, dealing with (self-)recursion requires a well-founded inductive argument when reasoning about validity of nested function calls in a verified procedure.[11] Therefore, we conduct our proof by well-founded induction, establishing validity of a synthesized top-level recursive procedure in an assumption that all of its recursive sub-calls are valid and are "smaller" in some sense. To equip ourselves for such an argument, we provide a series of auxiliary definitions, the key one being of *sized* specification validity, which relies on the notion of a heap size $|h| \triangleq |\mathsf{dom}\,(h)|$:

Definition 3.2 (Sized validity). We say a specification $\Sigma; \Gamma; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is *n-valid wrt.* the function dictionary Δ whenever for any h, h', s, s', whenever (a) $|h| \leq n$, (b) $\Delta; \langle h, (c, s) \cdot \epsilon\rangle \rightsquigarrow^* \langle h', (\mathtt{skip}, s') \cdot \epsilon\rangle$, and (c) $\mathsf{dom}\,(s) = \Gamma$ and $\exists \sigma_{\text{gv}} = [\overline{x_i \mapsto d_i}]_{x_i \in \text{GV}(\Gamma, \mathcal{P}, \mathcal{Q})}$ such that $\langle h, s\rangle \models^{\Sigma}_{\mathcal{I}} [\sigma_{\text{gv}}]\mathcal{P}$, it is the case that $\exists \sigma_{\text{ev}} = [\overline{y_j \mapsto d_j}]_{y_j \in \text{EV}(\Gamma, \mathcal{P}, \mathcal{Q})}$, such that $\langle h', s'\rangle \models^{\Sigma}_{\mathcal{I}} [\sigma_{\text{ev}} \cup \sigma_{\text{gv}}]\mathcal{Q}$

Definition 3.2 is rather peculiar in that it defines a standard Hoare-style soundness *wrt.* pre-/postcondition, while doing that only for heaps of size smaller than n. This is an important requirement to stage a well-founded inductive argument for our soundness proof of SSL-based synthesis in the presence of recursive calls. By introducing the explicit sizes to the definition of validity, we can make sure that we only deal with calls on strictly decreasing subheaps *wrt.* the heap size when invoking functions (auxiliary ones or recursive self). To make full use of this idea, and also account for the possibility of having auxiliary functions, we lift Definition 3.2 to contexts, stratifying the shape of function dictionaries *wrt.* their specifications.

Definition 3.3 (Coherence). A dictionary Δ is *n-coherent wrt.* a context Σ ($\text{coh}\,(\Delta, \Sigma, n)$) *iff*

- $\Delta = \epsilon$ and $\text{functions}(\Sigma) = \epsilon$, or
- $\Delta = \Delta', f\ (\overline{t_i\ x_i})\ \{\ c\ \}$, and $\Sigma = \Sigma', f\ (\overline{x_i}) : \{\mathcal{P}\}\{\mathcal{Q}\}$, and $\text{coh}\,(\Delta', \Sigma', n)$, and $\Sigma'; \{\overline{x_i}\}; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is n-valid *wrt.* Δ', or
- $\Delta = \Delta', f\ (\overline{t_i\ x_i})\ \{\ c\ \}$, and $\Sigma = \Sigma', f\ (\overline{x_i}) : \{\phi; \lceil P\rceil * p^1(\overline{e_i})\}\{\lceil Q\rceil\}$, and $\text{coh}\,(\Delta', \Sigma', n)$, and $\Sigma; \{\overline{x_i}\}; \{\lceil P\rceil * p^1(\overline{e_i})\}\ c\ \{\lceil Q\rceil\}$ is n'-valid *wrt.* Δ for all $n' < n$.

That is, coherence is defined inductively on the dictionary/context shape (regarding specified functions and ignoring predicate definitions). A possibility of a (single) recursive definition f is taken into the account in its last option. In that last clause, recursive calls to the function f from Δ may only take place on heaps of size *strictly smaller* than n, whereas there is no such restriction for the calls to user-provided auxiliary functions, that can be invoked on heaps of sizes up to n.

Validity of synthesized programs is stated by Lemma 3.4, which defines validity of the synthesized program *for any* size of the input heap n, assuming that the validity of all recursive calls of the synthesized programs is only established for $n' < n$, where the case $n = 0$ means no calls take place, so it forms the base of the inductive reasoning.

Lemma 3.4 (Sized validity of SSL-derived programs). *For any n, Δ', if*

(i) $\Sigma'; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} \mid c$ *for a goal named f with formal parameters $\Gamma \triangleq \overline{x_i}$, and*

(ii) Σ' *is such that* $\text{coh}\,(\Delta', \Sigma', n)$, *and*

(iii) *for all $p^0(\overline{e_i}), \phi; P$, such that $\{\mathcal{P}\} = \{\phi; p^0(\overline{e_i}) * P\}$, taking $\mathcal{F} \triangleq f(\overline{x_i}) : \{\phi; p^1(\overline{e_i}) * \lceil P\rceil\}\{\lceil Q\rceil\}$, $\Sigma', \mathcal{F}; \Gamma; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is n'-valid for all $n' < n$ wrt. $\Delta \triangleq \Delta', f\ (\overline{t_i\ x_i})\ \{\ c\ \}$,*

then $\Sigma'; \Gamma; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is n-valid wrt. Δ.

[11] This challenge is well-known when reasoning about correctness of recursive programs (Le and Hobor 2018).

Proof. By the top-level induction on n and by inner induction on the structure of derivation $\Sigma'; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} \mid c$. We refer the reader to Appendix A of the extended version of the paper (Polikarpova and Sergey 2018) for the details of the proof. □

The assumptions (ii) and (iii) postulate the validity of calls to a function (auxiliary and the one being synthesized) on *strictly smaller* heaps, akin to the generalized principle of mathematical induction. Overall, Lemma 3.4 states that a program derived via SSL constitutes a *valid* spec with its goal (*i.e.*, all writes, reads and deallocations in it are *safe wrt.* accessing heap pointers), assuming that recursive calls, if present, are made on reduced sub-heaps.

3.2.2 Termination. The technique we used for stating and proving Lemma 3.4—allowing for safe calls done only on smaller sub-heaps—is reminiscent to the one employed in size-change termination-based analyses (Lee et al. 2001), which ensure that every infinite sequence of calls would cause infinite descent of some values, leading to the following result.

Lemma 3.5 (Termination of SSL-synthesized programs). *A program, which is derived via SSL rules from a spec that uses only well-founded predicates, terminates, assuming any of its auxiliary functions terminates.*

Proof. The only source of non-termination is self-recursive calls. Due to the tagging discipline, every recursive self-call is applicable after opening a well-founded instance predicate, and hence is done on a *smaller sub-heap*. Both self-recursive and auxiliary function calls *erase* the tags in the post-heap, preventing the "expansion" of the symbolic heaps that can be used for further calls. □

The argument for termination, supplied by the tagging discipline and used in the proof of Lemma 3.5, is somewhat similar to the syntactic termination criterion, used in proof assistants such as Coq (Coq Development Team 2018). In fact, by parameterizing inductive predicates with pure algebraic data types for describing the heap contents (*e.g.*, using algebraic lists instead of sets in the definition (6) of lseg), we could have employed them as a termination measure instead. We will explore this opportunity in the future work.[12]

3.2.3 Soundness. We conclude this section with the main SSL soundness result. We say that a function dictionary Δ is *consistent* with a context Σ *iff* any function $f \in \Delta$ is valid *wrt.* a corresponding specification in Σ with Γ taken to be f's formals.

Theorem 3.6. *If $\Sigma; \Gamma; \{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} \mid c$ for a goal named f with formal parameters $\Gamma \triangleq \overline{x_i}$, then:*
(a) *$\Sigma; \Gamma; \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ is valid* wrt. *any function dictionary Δ, consistent with Σ, and*
(b) *c terminates for any input store-heap pair, satisfying its precondition.*

Proof. Follows immediately from Lemma 3.4 taken for a heap of any size, and Lemma 3.5. □

4 BASIC SSL-POWERED SYNTHESIS ALGORITHM

In this section we show how to turn SSL from a declaratively defined inference system (Fig. 12) into an algorithm for deriving provably correct imperative programs. The core algorithm is a fairly standard goal-directed backtracking proof search (Kneuss et al. 2013; Mellish and Hardy 1984).

Encoding Rules and Derivations. The display on the right shows an algorithmic representation of SSL derivations and rules. To account for the top-level goal (which mandates the program synthesizer to generate a runnable function), we include the function name f

$$\begin{array}{rcll} \mathcal{G} & \in & \text{Goal} & ::= \langle f, \Sigma, \Gamma, \{\mathcal{P}\}, \{\mathcal{Q}\} \rangle \\ \mathcal{K} & \in & \text{Cont} & \triangleq (\text{Command})^n \to \text{Command} \\ \mathcal{S} & \in & \text{Deriv} & ::= \left\langle \overline{\mathcal{G}_i}, \mathcal{K} \right\rangle \\ \mathcal{R} & \in & \text{Rule} & \triangleq \text{Goal} \to \wp(\text{Deriv}) \end{array}$$

[12] We thank François Pottier for this suggestion.

Algorithm 4.1: synthesize ($\mathcal{G}$: Goal, *rules* : Rule*)

```
   Input: Goal G = ⟨f, Σ, Γ, {P}, {Q}⟩
   Input: List rules of available rules to try
   Result: Terminating program c, such that
           Σ; Γ; {P} c {Q} is valid
1  function synthesize (G, rules) =
2    withRules(rules, G)
3  function withRules (rs, G) =
4    match rs
5      case [ ] ⇒ Fail
6      case R :: rs' ⇒
7        match filterComm ( R(G) )
8          case ⊥ ⇒ withRules(rs')
9          case subderivs ⇒
10           tryAlts(subderivs, R, rs', G)
12 function tryAlts (derivs, R, rs, G) =
13   match derivs
14     case [ ] ⇒ if isInvert(R) then Fail else withRules(rs, G)
15     case ⟨goals, K⟩ :: derivs' ⇒
16       match solveSubgoals(goals, K)
17         case Fail ⇒ tryAlts(derivs', R, rs, G)
18         case c ⇒ c
19 function solveSubgoals (goals, K) =
20   cs := [ ]
21   pickRules = λG. phasesEnabled ? nextRules(G) : AllRules
22   for G ← goals; c = synthesize(G, pickRules(G)); c ≠ Fail do
23     cs := cs ++ [c]
24   if |cs| < |goals| then Fail else K(cs)
```

into the goal $\mathcal{G}$, whose other components are the context Σ, environment Γ (initialized with f's formals), precondition $\{\mathcal{P}\}$ and postcondition $\{\mathcal{Q}\}$. A successful application or a rule $\mathcal{R}$ results in one or more alternative *sub-derivations* $\mathcal{S}_k$. Several alternatives arise when the rule exhibits non-determinism (*e.g.*, due to choosing a sub-heap or a unifying substitution), and are explored by a search engine one by one, until it finds one that succeeds.

In its turn, each sub-derivation is a pair. Its first component contains zero (if $\mathcal{R}$ is a terminal) or more sub-goals, which *all* need to be solved (think of a conjunction of a rule's premises). The second component of the sub-derivation is a *continuation* $\mathcal{K}$, which combines the *list* of commands, produced as a result of solving subgoals, into a final program. The arity of a continuation (length of a list it accepts) is the same as a number of sub-goals the corresponding rule emits Zero-arity means that the continuation has been produced by a terminal, and simply emits a constant program. For non-operational rules (*e.g.*, Frame), $\mathcal{K} \triangleq \lambda[c].c$. For operational rules $\mathcal{K}$ typically prepends a command to the result (*e.g.*, Write), or generates a conditional statement (Open). Therefore, the synthesizer procedure constructs the program by applying the continuations of rules that have succeeded earlier, to the resulting programs of their subgoals, on the "backwards" walk of the recursive search, in the style of logic programming (Mellish and Hardy 1984).

The algorithm. The pseudocode of our synthesis procedure is depicted by Algorithm 4.1. Let us ignore the grayed fragments in the pseudocode for now and agree to interpret the code as if they were not present. Those fragments corresponds to optimizations, which we describe in detail in Sec. 5. The algorithm is represented by four mutually-recursive functions:

- synthesize ($\mathcal{G}$, *rules*) is invoked initially on a top-level goal, with *rules* instantiated with *AllRules*– all rules from Fig. 12. It immediately passes control to the first auxiliary function, withRules.
- withRules (rs, $\mathcal{G}$) iterates through the list rs of remaining rules, trying to apply each one to the goal $\mathcal{G}$. If a rule $\mathcal{R}$ is applicable, it emits a set of alternative sub-derivations *subderivs*, which are passed to tryAlts. Otherwise $\mathcal{R}(\mathcal{G})$ returns $\bot$, and the next rule form the list is attempted, until no more rules remain (line 5), at which point the synthesis for the current goal fails.
- tryAlts (*derivs*, $\mathcal{R}$, rs, $\mathcal{G}$) recursively processes the list of alternative sub-derivations *derivs*, generated by the rule $\mathcal{R}$. If the list is exhaused (line 14), withRules is invoked to try the rest of the rules rs. Otherwise, solveSubgoals is invoked for an alternative to solve all its sub-goals *goals* and apply the continuation $\mathcal{K}$. In the case of success (line 18), the resulting program c is returned.

- solveSubgoals ($goals, \mathcal{K}$) tries to solve all subgoals given to it, by invoking synthesize recursively with a suitable (full) list of rules, essentially, restarting the search problem "one level deeper" into the derivation. Unless some of the goals failed, their results are combined via $\mathcal{K}$.[13]

The algorithm explores the space of all valid SSL derivations rooted at the initial synthesis goal. The search proceeds in a *depth-first* manner: it starts from the root (the initial goal) and always extends the current incomplete derivation by applying a rule to its leftmost *open* leaf (*i.e.*, a leaf that is not a terminal application). The algorithm terminates when the derivation is *complete*, *i.e.*, it has no open leaves.

5 OPTIMIZATIONS AND EXTENSIONS

The basic synthesis algorithm presented in Sec. 4 is a naïve backtracking search in the space of all valid SSL derivations. Note that this is already an improvement over a blind search in the space of all programs: some incorrect programs are excluded from consideration a-priori, such as, *e.g.*, programs that read from unallocated heap locations. In this section we show how to further prune the search space by identifying unsolvable goals and avoiding their exploration.

5.1 Invertible Rules

Our first optimization relies on a well-known fact from proof theory (Liang and Miller 2009) that certain proof rules are *invertible*: applying such a rule to any derivable goal produces a goal that is still derivable. In other words, applying invertible rules eagerly without backtracking does not affect completeness. Algorithm 4.1 leverages this fact in line 14: when all sub-derivations of an invertible rule $\mathcal{R}$ fail, the algorithm need not backtrack and try other rules, since the failure cannot be due to $\mathcal{R}$ and must have been caused by a choice made earlier in the search.

In SSL, the normalization rules—Read, StarPartial, NullNotLVal, and SubstLeft—are invertible. The effect of these rules on the goal is either to change a ghost into a program-level variable or to strengthen the precondition; no rule that is applicable to the original goal can become inapplicable as a result of this modification, which is confirmed by inspection of all rules in Fig. 12.

5.2 Multi-Phased Search

Among the rules of SSL described in Sec. 3, the unfolding rules are focused on transforming (and eventually eliminating) instances of inductive predicates, while flat rules are focused on other types of heaplets (*i.e.*, points-to and blocks). We observe that if the unfolding rules failed to eliminate a predicate instance from the goal, there is no point to apply flat rules to that goal. It is easy to show that the flat rules can neither eliminate predicates from the goal, nor enable previously disabled unfolding rules: the only unfolding rule that matches on flat heaplets is Call, but those heaplets are taken care of separately by AbduceCall.

Following this observation, without loss of completeness, we can split the synthesis process into two phases: the *unfolding* phase, where flat rules are disabled, and the *flat* phase, which only starts when the goal contains no more predicate instances, and hence unfolding rules are inapplicable. This optimization is implemented in line 21 of Algorithm 4.1. As a result, some unsolvable goals will be identified early, in the unfolding phase. For example, the following goal:

$$\{y, a, b\}\,;\{y \mapsto b * a \mapsto 0;\} \rightsquigarrow \left\{y \mapsto u * u \mapsto 0 * \mathsf{lseg}^1(u, 0, S)\right\}$$

will fail immediately without exploring fruitless transformations on its flat heap, since no unfolding rules are applicable (assuming MaxUnfold = 1).

[13] The actual implementation is more efficient and breaks out of the loop as soon as one of the subgoals fails.

$$\textsc{PostInconsistent}\ \frac{\psi \neq \bot \qquad \vdash \phi \wedge \psi \Rightarrow \bot \qquad \Sigma; \Gamma;\ \{\mathsf{emp}\} \rightsquigarrow \{\bot, \mathsf{emp}\} \mid c}{\Sigma; \Gamma;\ \{\phi; P\} \rightsquigarrow \{\psi, Q\} \mid c}$$

$$\textsc{PostInvalid}\ \frac{P \text{ has no pred. instances} \qquad \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \emptyset \qquad \psi \neq \bot \qquad \vdash \neg(\phi \Rightarrow \psi) \qquad \Sigma; \Gamma;\ \{\mathsf{emp}\} \rightsquigarrow \{\bot, \mathsf{emp}\} \mid c}{\Sigma; \Gamma;\ \{\phi; P\} \rightsquigarrow \{\psi, Q\} \mid c}$$

$$\textsc{UnreachHeap}\ \frac{P, Q \text{ have no pred. instances or blocks} \qquad \mathsf{unmachedHeaplets}(P, Q) \qquad \Sigma; \Gamma;\ \{\mathsf{emp}\} \rightsquigarrow \{\bot, \mathsf{emp}\} \mid c}{\Sigma; \Gamma;\ \{\phi, P\} \rightsquigarrow \{\psi, Q\} \mid c}$$

Fig. 13. Failure rules.

5.3 Symmetry reduction

Backtracking search often explores all reorderings of a sequence of rule applications, even if they *commute*, *i.e.*, the order of applications does not change the end sub-goal. As an example, consider the following goal:

$$\{x, y, a, b\}\,; \{x \mapsto a * y \mapsto b * a \mapsto 0\} \rightsquigarrow \{x \mapsto a * y \mapsto b * b \mapsto 0\}$$

Framing out $x \mapsto a$ and then $y \mapsto b$, reveals the unsolvable goal $\{a \mapsto 0\} \rightsquigarrow \{b \mapsto 0\}$; upon backtracking, the naïve search would try the two applications of Frame in the opposite order, leading to the same result.

We implemented a *symmetry reduction* optimization to eliminate redundant backtracking of this kind. To this end, we keep track of the *footprint* of each rule application, *i.e.*, the sub-heaps of its goal's pre- and postcondition that the application modifies. This enables us to identify whether two sequential rule applications commute. Next, we impose a total order on rule applications; line 6 of Algorithm 4.1 rejects a new rule application $\mathcal{R}$ if it commutes with an earlier application $\mathcal{R}'$ in the current derivation, but comes before $\mathcal{R}'$ in the total order.

5.4 Early Failure Rules

Sometimes we can identify an unsolvable goal by analyzing its postcondition. For example, the goal

$$\{x, y\}\,; \{a = 0; x \mapsto a\} \rightsquigarrow \{a = u \wedge u \neq 0; x \mapsto u\}$$

is unsolvable because its pure postcondition is logically *inconsistent* with the precondition. To leverage this observation and eliminate redundant backtracking, we extend SSL with *failure rules*. Each failure rule fires when it identifies a certian type of unsolvable goal, and transforms it into a sub-goal $\{\mathsf{emp}\} \rightsquigarrow \{\bot, \mathsf{emp}\} \mid c$, to which no rule applies. All failure rules are also invertible, hence the effect is to backtrack an application of an earlier rule.

Our set of failure rules is shown in Fig. 13. The rule PostInconsistent identifies a goal where the the pure postcondition is inconsistent with the precondition. This is safe because during the derivation both assertions can only become stronger (as a result of unfolding rules); also, even if the postcondition still contains existentials, no instantiation of those existentials can produce a formula that is consistent with (let alone implied by) the precondition. The rule PostInvalid fires on a goal where the pure postcondition (which is free of existentials) is not implied by the precondition; this rule only applies when the precondition is free of predicate instances, and hence its pure part cannot be strengthened any further. The rule UnreachHeap fires when the spatial pre- and postcondition contain only points-to heaplets, but the *left-hand sides* of these heaplets cannot be unified; in this case neither UnifyHeaps nor Write can make the heaplets match. Note that it is important for completeness that failure rules are checked *after* Inconsistency: if the pure precondition is inconsistent, the derivation should not fail, but should instead emit `error`.

5.5 Auxiliary Functions

The version of Induction rule from Fig. 12 and the way it interactis with Call are harmful for the framework's completeness, due to the erasure of the tags from the postcondition of the function f

$$\textsc{SubstRight}\quad \frac{x \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad \Sigma; \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [\psi'/x]\{\psi \wedge x = \psi', Q\} \,|\, c}{\Sigma; \Gamma;\ \{\mathcal{P}\} \rightsquigarrow \{\psi \wedge x = \psi'; Q\} \,|\, c}$$

$$\textsc{UnifyPure}\quad \frac{[\sigma]\psi' = \phi' \qquad \emptyset \neq \mathsf{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [\sigma]\{Q\} \,|\, c}{\Gamma;\ \{\phi \wedge \phi'; P\} \rightsquigarrow \{\psi \wedge \psi'; Q\} \,|\, c}$$

$$\textsc{PickVar}\quad \frac{x \in \mathsf{EV}(\Gamma, \mathcal{P}, Q) \qquad y \in \Gamma \qquad \Gamma;\ \{\mathcal{P}\} \rightsquigarrow [y/x]\{Q\} \,|\, c}{\Gamma;\ \{\mathcal{P}\} \rightsquigarrow \{Q\} \,|\, c}$$

$$\textsc{Branch}\quad \frac{e \in \mathcal{L} \qquad \Sigma; \Gamma;\ \{\phi \wedge e; P\} \rightsquigarrow \{Q\} \,|\, c_1 \qquad \Sigma; \Gamma;\ \{\phi \wedge \neg e; P\} \rightsquigarrow \{Q\} \,|\, c_2}{\Sigma; \Gamma;\ \{\phi; P\} \rightsquigarrow \{Q\} \,|\, \textbf{if}\ (e)\ c_1\ \textbf{else}\ c_2}$$

Fig. 14. Pure synthesis rules.

to be applied recursively ($Q_f = \lceil Q \rceil$), so the subsequent applications of f on the symbolic heap, overlapping with the "sealed" part $\lceil Q \rceil$ are prohibited. While it is not unsafe to employ the predicates from the procedure call's postcondition in further procedure calls, one cannot ensure that they denote "smaller heaplets" and thus that the program terminates.[14]

To circumvent this limitation for common scenarios, we had to introduce one divergence between SuSLik and SSL as shown in Fig. 12. Specifically, in the tool, we implemented support for *stratified chained* auxiliary function calls, *i.e.*, calls on a heap resulted from another call (see flatten w/append and insertion sort benchmarks in Sec. 6). To allow for them, we had to implement a slightly different version of Induction rule, which instead of *erasing* tags of the corresponding part of the postcondition ($Q_f \triangleq \lceil Q \rceil$) would *increment* them: $Q_f \triangleq \lceil Q_f \rceil^{\bullet+1}$. This would prevent a call of the *same* function on the resulting heap fragment, but would enable calls of auxiliary functions, whose specs' preconditions feature predicates with matching *higher-level* tags. Specifically, each auxiliary function is assumed to have all predicate instances tagged with some $\ell > 0$ in its precondition, and with $\ell + 1$ in its postcondition. The discipline for assigning specific tag levels to auxiliary functions' specifications is fixed to follow the order in which their signatures are introduced in the goal. Eventually, due to incrementation of level tags with each call (to a recursive self or to an auxiliary function), no applicable functions would be left in the context Σ. While this extension is unlikely to break the SSL soundness and termination results (due to the limit of chained applications, enabled by growing tags), it would require us to generalize the well-foundedness argument in Sec. 3.2, and in the interest of time we did not carry out this exercise.

5.6 Pure Synthesis and Branch Abduction

The core SSL relies on a pure synthesis oracle to suggest suitable applications of the Pick rule. In our implementation, the declarative Pick rule is replaced with the four rules depicted in Fig. 14, which together play the role of an incomplete yet practically effective pure synthesis oracle. The first three rules have been explained in Sec. 2.4.2; it is easy to see that each of them is a specialization of Pick, and therefore is sound. The fourth rule, Branch, comes to the rescue whenever the pure postcondition cannot be satisfied without introducing a case split, and generates a conditional statement with some guard e. Although Branch is sound for any choice of guard, to curb the non-determinism, we pick e from the set of *learned guards* $\mathcal{L}$.

The set $\mathcal{L}$ is populated through a mechanism we call *branch abduction*, reminiscent of similar mechanisms in other synthesis techniques (Alur et al. 2017; Kneuss et al. 2013; Leino and Milicevic 2012; Polikarpova et al. 2016). The mechanism piggy-backs on the failure rule PostInvalid (Sec. 5.4), which detects a goal whose pure postcondition ψ does not follow from the precondition ϕ. While the goal is still rejected as unsolvable, branch abduction also searches a small set of expressions (all

[14] A similar issue is reported in the work by Rowe and Brotherston (2017) on verifying termination of procedural programs.

atomic boolean expressions over program variables in Γ) trying to find an expression e such that $\phi \wedge e \Rightarrow \psi$; if such an e exists, it is added to $\mathcal{L}$.

For a simple example, consider the following synthesis goal that corresponds to computing a lower bound of two integers x and y:

$$\{x, y\}\,; \{r \mapsto 0\} \rightsquigarrow \{m \leq x \wedge m \leq y; r \mapsto m\}$$

We first apply PickVar with $[m \mapsto x]$, arriving at the goal $\{r \mapsto 0\} \rightsquigarrow \{x \leq x \wedge x \leq y; r \mapsto x\}$. At this point PostInvalid fires, and branch abduction infers a guard $x \leq y$ and adds it to $\mathcal{L}$. Upon backtracking from the unsolvable goal, Branch fires since $\mathcal{L}$ is non-empty. Both of its subgoals can be solved by suitable applications of PickVar.

6 IMPLEMENTATION AND EVALUATION

We implemented SSL-based synthesis as a tool, called SuSLik, in Scala, using Z3 (de Moura and Bjørner 2008) as the back-end SMT solver via the ScalaSMT library (Cassez and Sloane 2017).[15] We evaluated our implementation with the goal of answering the following research questions:

(1) *Generality:* Is SuSLik general enough to synthesize a range of nontrivial programs with pointers?
(2) *Utility:* How does the size of the inputs required by SuSLik compare to the size of the generated programs? Does SuSLik require any additional hints apart from pre- and postconditions? What is the quality of the generated programs?
(3) *Efficiency:* Is it efficient? What is the effect of optimizations from Sec. 5 on synthesis times?
(4) *Comparison with existing tools:* How does SuSLik fare in comparison with existing tools for synthesizing heap-manipulating programs, specifically, ImpSynt (Qiu and Solar-Lezama 2017)?

6.1 Benchmarks

In order to answer these questions, we assembled a suite of 22 programs listed in Tab. 1. The benchmarks are grouped by the main data structure they manipulate: integer pointers, singly linked lists, sorted singly linked lists, binary trees, and binary search trees.

To facilitate comparison with existing work, most of the programs are taken from the literature on synthesis and verification of heap-manipulating programs: the ImpSynt synthesis benchmarks (Qiu and Solar-Lezama 2017), the Jennisys synthesis benchmarks (Leino and Milicevic 2012), and the Dryad verification benchmarks (Qiu et al. 2013). We manually translated these benchmarks into the input language of SuSLik, taking care to preserve their semantics. Dryad and ImpSynt use the Dryad dialect of separation logic as their specification language, hence the translation in this case was relatively straightforward. As an example, consider an ImpSynt specification and its SuSLik equivalent in Fig. 15. In addition to a pre-/postcondition, the former also contains a program *sketch*, where the constructs `??`, `cond`, and `statement` denote, respectively, an unknown expression, conditional guard, and statement, to be filled by the synthesizer. The main difference between the two pre-/postcondition pairs is that the Dryad logic supports recursive functions such as `len`, `min`, and `max`; in SuSLik this information is encoded in more traditional SL style: by passing additional ghost parameters to the inductive predicate `srtl`. The extra precondition $0 \leq n \wedge 0 \leq k \wedge k \leq 7$ in SuSLik corresponds to implicit axioms in ImpSynt (in particular, the condition on k is due to its encoding of list elements as unsigned 3-bit integers—there is no such restriction in SuSLik). In addition to benchmarks from the literature, we also added several new programs that show-case interesting features of SuSLik.

[15] The tool sources can be found at https://github.com/TyGuS/suslik.

Table 1. Benchmarks and SuSLik results. For each benchmark, we report the size of the synthesized *Code* (in AST nodes) and the ratio *Code/Spec* of code to specification; as well as synthesis times (in seconds): with all optimizations enabled (*Time*), without phase distinction (*T-phase*), without invertible rules (*T-inv*), without early failure rules (*T-fail*), without the commutativity optimization (*T-com*), and without any optimizations (*T-all*). *T-IS* reports the ratio of synthesis time in ImpSynt to *Time*. "-" denotes timeout of 120 seconds.

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

6.2 Results

Evaluation results are summarized in Tab. 1. All experiments were conducted on a commodity laptop (2.7 GHz Intel Core i7 Lenovo Thinkpad with 16GB RAM).

6.2.1 Generality and Utility. Our experiment confirms that SuSLik is capable of synthesizing programs that manipulate a range of heap data structures, including nontrivial manipulations that require reasoning about both the shape and the content of the data structure, such as insertion into a binary search tree. We manually inspected all generated solutions, as well as their accompanying SSL derivations, and confirmed that they are indeed correct.[16] Perhaps unsurprisingly, some of the solutions were not entirely intuitive: as one example, the synthesized version of list copy, in a bizarre yet valid move, *swaps the tails* of the original list and the copy at each recursive call!

Two of the programs in Tab. 1 make use of auxiliary functions: "insertion sort" calls "insert", and "tree flatten w/append" calls the "append" function on linked lists. The specifications of auxiliary functions have to be supplied by the user (while their implementations can, of course, be synthesized independently). Alternatively, tree flattening can be synthesized without using an auxiliary function, if the user supplies an additional list argument that plays the role of an accumulator (see "tree flatten w/acc"). As such, SuSLik shares a common limitation of existing synthesizers for recursive functions: they require the initial synthesis goal to be inductive, and do not try to discover recursive auxiliary functions (we discuss this in more detail in Sec. 7).

[16] In the future, we plan to output SSL derivations as SL proofs, checkable by a third-party system such as VST (Appel 2011).

```
loc srtl_insert(loc x, int k)
requires srtl(x)
ensures srtl(ret) ∧
  len(ret) = old(len(x)) + 1 ∧
  min(ret) = (old(k) < old(min(x))
    ? old(k) : old(min(x))) ∧
  max(ret) = (old(max(x)) < old(k)
    ? old(k) : old(max^(x))) {
  if (cond(1)) {
    loc ?? := new;
    return ??;
  } else {
    statement(1);
    loc ?? := srtl_insert(??, ??);
    statement(1);
    return ??;
  }
}
```

```
{
  0 ≤ n ∧ 0 ≤ k ∧ k ≤ 7 ;
  ret ↦ k ∗ srtl(x, n, lo, hi)
}
void srtl_insert(loc x, loc ret)
{
  n1 = n + 1 ∧
  lo1 = (k ≤ lo ? k : lo) ∧
  hi1 = (hi ≤ k ? k : hi) ;
  ret ↦ y ∗ srtl(y, n1, lo1, hi1)
}
```

Fig. 15. (left) The ImpSynt input for the sorted list insertion; (right) The SuSLik input for the same benchmark.

For simple programs specification sizes are mostly comparable with the size of the synthesized code, whereas more complex benchmarks is where declarative specifications really shine: for example, for all Tree programs, the specification is at most half the size of the generated code. Three notable outliers are "prepend", "rotate left", and "rotate right", whose implementations are relatively short, while the specification we inherited from ImpSynt describes the effects of the functions on the minimum and maximum of the list/tree. Note that the specification sizes we report exclude the definitions of inductive predicates, which are reusable, and are shared between the benchmarks.

6.2.2 Efficiency. SuSLik has proven to be efficient in synthesizing a variety of programs: all 22 benchmarks are synthesized within 40 seconds, and all but four of them take less than a second.

In order to assess the impact on performance of various optimizations described in Sec. 5, Tab. 1 also reports synthesis times with each optimization *disabled*: the column *T-phase* corresponds to eliminating the distinction between phases; *T-inv* corresponds to ignoring rule invertibility; *T-fail* corresponds to dropping all failure rules; *T-com* corresponds to disabling the symmetry reduction; finally, *T-all* corresponds to a variant of SuSLik with *all* the above optimizations disabled. The results demonstrate the importance of optimizations for nontrivial programs: 8 out of 22 benchmarks time out when all optimizations are disabled. The simpler benchmarks (*e.g.*, swap) do not benefit from the optimizations at all, since they do not exhibit a lot of backtracking. At the same time, all three BST benchmarks time-out as a result of disabling even a single optimization.

The copy benchmark is peculiar in that disabling any single optimization makes little difference, while disabling all of them together leads to a timeout. A closer examination of all possible configurations reveals that the speedup is due to interplay between phase distinction and invertible rules; this is not entirely surprising, since both optimizations prevent their fair share of particularly disastrous (wrt. performance) rule applications, such as early incorrect unification on flat heaps.

6.2.3 Comparison with Existing Synthesis Tools. We compare SuSLik with the most closely related prior work on ImpSynt (Qiu and Solar-Lezama 2017). Out of the 14 benchmarks from (Qiu and Solar-Lezama 2017) successfully synthesized by ImpSynt, we excluded 5 that are not structurally recursive;[17] the remaining 9 were successfully synthesized by SuSLik. The *qualitative difference* in terms of the required user input is immediately obvious from the representative example in Fig. 15:

[17] 3 out of 5 are iterative versions of their recursive benchmarks, so we synthesize an equivalent recursive program for each of those; the other 2 use an internal loop or non-structural recursion, and are currently beyond the scope of SuSLik (Sec. 7).

in addition to the declarative specification, ImpSynt requires the user to provide an implementation *sketch*, which fixes the control structure of the program, the positions of function calls, and the number of other statements. These additional structural constraints are vital for reducing the size of the search space in ImpSynt. Instead, SuSLik prunes the search space by leveraging the structure inherent in separation logic proofs, allowing for more concise, purely declarative specifications.

Despite the additional hints from the user, ImpSynt is also *less efficient*: as shown in the column *T-IS* of Tab. 1, on 6 out of 9 common benchmarks, ImpSynt takes at least an order of magnitude longer than SuSLik, even though it has been evaluated on a 10-core server with 96GB of RAM.

7 LIMITATIONS AND DISCUSSION

The synthesis problem tackled in this work is clearly undecidable, since it subsumes several well-known undecidable problems: termination of recursive programs, SL entailment in the presence of general inductive predicates (Antonopoulos et al. 2014), and pure synthesis from first-order logic specifications (Reynolds et al. 2015). Hence, no complete and terminating synthesis algorithm exists, and in SuSLik, we made a design decision to sacrifice completeness for the sake of termination.

Sources of Incompleteness. SSL and SuSLik currently cannot synthesize programs that are not structurally recursive *wrt.* some inductive predicate, such as, for instance, merging sorted lists or merge sort. To relax this restriction we could adopt a more flexible termination argument for the synthesized programs, such as showing that each recursive call decreases the value of a custom *termination metric*; this technique is used in several automated verifiers and synthesizers (Leino 2013; Polikarpova et al. 2016). A termination metric maps the tuple of function's arguments into an element of some set with a pre-defined well-founded order (usually a tuple of natural numbers), and can be either provided by the user or inferred by the synthesizer. Custom termination metrics would also enable support for mutually-recursive inductive predicates.

Another source of incompleteness is the limit MaxUnfold on the number of predicate unfoldings via Open and Close rules. This addresses the undecidability of reasoning with inductive predicates, but also precludes synthesis of some useful programs, *e.g.*, allocating a large constant-sized list.

Finally, SuSLik's pure synthesis oracle is obviously incomplete. This source of incompleteness can be mitigated by delegating to an off-the-shelf pure synthesizer (Kneuss et al. 2013; Polikarpova et al. 2016; Reynolds et al. 2015), or eliminated entirely—for a restricted fragment of pure logic—by leveraging complete synthesis procedures (Jacobs et al. 2013; Kuncak et al. 2010).

Termination. Algorithm 4.1 terminates, as long as the pure synthesis oracle suggest finitely many alternatives for a given goal. This can be established by considering the following tuples, ordered lexicographically, as a termination measure for a given goal $\mathcal{G}$: ⟨ # 0- or 1-tagged predicate instances; # heaplets in pre- and postcondition, for which there is no matching one in the post-/precondition; # existentials; # "flat" heaplets; # conjuncts in the precondition; # of points-to heaplets, whose disjointness or non-null-ness is not captured in the precondition ⟩. Notice that each rule from Fig. 12, except for Open and Close reduces this value for the emitted sub-goals. Applicability of those two rules is handled via MaxUnfold parameter.

Logic and Language Limitations. Although our programming component has no support for `while`-loops, this is not a fundamental limitation: any program with a top-level loop can be rewritten as a recursive program, and any internal loop can be rewritten as a *recursive auxiliary function.* Hence, the fundamental challenge is the synthesis of auxiliary functions, which SSL currently does not support; instead, the initial goal is considered as inductive and handled via Induction rule. Discovering specifications of auxiliary functions (or, equivalently, invariants of internal loops) significantly increases the search space. To our knowledge, all existing synthesis techniques that

support this feature are quite limited, and rely on built-in or user-provided templates (Eguchi et al. 2018; Qiu and Solar-Lezama 2017; Si et al. 2018; Srivastava et al. 2010).

Some of SSL limitations are inherent for Separation Logics in general: SLs are known to work well with disjoint tree-like linked structures, and programs whose recursion scheme matches the data definition, but not so well with ramified data structures, *e.g.*, graphs. To address those, one could integrate a more powerful, *ramified* version of Frame rule (Hobor and Villard 2013) into SSL, but this would likely require more hints from the user, thus reducing the utility of the approach.

8 RELATED WORK

There are two main directions in the area of program synthesis: synthesis from informal descriptions (such as examples, natural language, or hints) (Albarghouthi et al. 2013; Feng et al. 2017; Feser et al. 2015; Murali et al. 2018; Osera and Zdancewic 2015; Polozov and Gulwani 2015; Smith and Albarghouthi 2016; Yaghmazadeh et al. 2017) and synthesis from formal specifications. We will only discuss the more relevant latter direction. The goal of this type of program synthesis is to obtain a *provably correct* program.

In this area, there is a well-known trade-off between three dimensions: how complex the synthesized programs are, how strong the correctness guarantees are, and how much input is required form the user. On one end of the spectrum there are interactive synthesizers (Delaware et al. 2015; Itzhaky et al. 2016), which can be very expressive and provide strong guarantees, but the user is expected to guide the synthesis process (although, usually, with aid of dedicated proof *tactics*). On the other end, there is fully automated synthesis for loop- and recursion-free programs over simple domains, like arithmetic and bit-vectors (Alur et al. 2017; Gulwani et al. 2011). Our work lies in the middle of this spectrum, where synthesis is automated but programs are more expressive.

In the presence of loops or recursion, verifying candidates becomes nontrivial. Synthesizers like Sketch (Solar-Lezama 2013) and Rosette (Torlak and Bodík 2014) circumvent this problem by resorting to bounded verification, which only provides restricted guarantees and has scalability issues due to path explosion. In contrast, our work relies on unbounded deductive verification.

Among synthesis approaches that use unbounded verification, synthesizers like Leon (Kneuss et al. 2013) and Synquid (Polikarpova et al. 2016) focus on pure functional (recursive) programs, which are an easier target for unbounded verification. Proof-theoretic synthesis (Srivastava et al. 2010) is capable of synthesizing imperative programs with loops and arrays, but no linked structures; they pioneered the idea of synthesizing provably-correct programs by performing symbolic (SMT-based) search over programs *and their verification conditions* simultaneously.

Finally, the two pieces of prior work that are most closely related to ours in terms of scope are Jennisys (Leino and Milicevic 2012) and Natural Synthesis (Qiu and Solar-Lezama 2017), both of which generate provably-correct heap-manipulating programs. Both of them are essentially instances of proof-theoretic synthesis with a program logic for reasoning about the heap. To that end, Jennisys uses the Dafny verifier (Leino 2013), which supports expressive yet undecidable specifications, and often requires hints from the user, so in practice the tool doesn't scale to complex examples (for example, none of their benchmarks performs mutation). Natural Synthesis uses Dryad (Madhusudan et al. 2012; Qiu et al. 2013), an SL-style program logic for reasoning about heap-manipulating programs. The downside of this approach is that whole-program symbolic search doesn't scale to larger programs; to mitigate this, they require the user to provide sketches with substantial restrictions on the structure of the program. Our approach does not require sketches (but on the other hand, we do not support loops).

The recent tool FootPatch by van Tonder and Le Goues (2018) is very close in its methods and goals to SuSLik. FootPatch builds on Infer (Calcagno and Distefano 2011), an open-source SL-based static analyzer by Facebook, using it for *automated program repair*. It takes the intermediate

assertions, provided by Infer for programs with bugs, such as resource and memory leaks, and null dereferences, and constructs additive patches based on the observed discrepancy. In this, it acts similarly to our AbduceCall rule. FootPatch does not synthesize patches that would involve recursion or complex control flow.

Instead of whole-program symbolic search, like in proof-theoretic synthesis, our work follows the tradition of *deductive synthesis*, *i.e.*, backtracking search in the space of program derivation composed of synthesis rules, which gradually transform a specification into a program. This tradition originates from the work by Manna and Waldinger (1980), and similar ideas have been used in more recent synthesis work (Delaware et al. 2015; Kneuss et al. 2013; Polikarpova et al. 2016). In particular, the overall structure of our synthesis algorithm (backtracking and-or search) is similar to Leon (Kneuss et al. 2013), but our rules focus on heap manipulation, whereas their rules focus on synthesizing pure terms (so in fact Leon can be used as a component by our algorithm). Recent work on Optitian (Miltner et al. 2018) is very different in scope—they synthesize bijective string lenses from regular expression specifications and examples—but has interesting similarities in the technique. Their pre- and postcondition are regexes, and their technique tries to "align" them by *e.g.*, unfolding the Kleene star; this is similar to how SuSLik tries to align the spatial pre- and postcondition by unfolding predicates.

Deductive synthesis is closely related to proof search, and there has been recent resurgence in applying proof-theoretic techniques, like focusing, to program synthesis (Frankle et al. 2016; Scherer 2017). But none of them do it for a complex logic that can reason about stateful programs.

Despite the vast space of available tools for symbolic verification based on Separation Logic: Smallfoot (Berdine et al. 2006), HTT (Nanevski et al. 2010), Bedrock (Chlipala 2011), SLAyer (Berdine et al. 2011), HIP/SLEEK (Chin et al. 2011), VeriFast (Jacobs et al. 2011), SLAD (Bouajjani et al. 2012), GRASShopper (Piskac et al. 2014b), Viper (Müller et al. 2016), Cyclist (Rowe and Brotherston 2017), to name just a few, to the best of our knowledge none of them have been employed for deriving programs from specifications. It is certainly our hope that this work will bring new synergies between the research done in verification, theorem proving, and program synthesis communities.

For instance, in our approach to establish termination of SSL-synthesized programs, we used techniques close in spirit to the methods for proving total correctness in type/SL-based frameworks. *E.g.*, SSL's tags might be seen as a variant of resource capacities used in HIP/SLEEK (Le et al. 2014). Our use of Definition 3.2 of sized validity is similar to the induction on the finiteness of the heap used by Le and Hobor (2018) in their work on a logic for fractional shares. The way we use tagged predicates for establishing validity in Lemma 3.4 is reminiscent to the purpose of the $\triangleright$-modality in type theories for state and recursion (Appel et al. 2007).

9 CONCLUSION

In their seminal paper, Manna and Waldinger (1980) set forth an agenda for deductive synthesis of functional programs: *"theorem provers have been exhibiting a steady increase in their effectiveness, and program synthesis is one of the most natural application of those systems"*.

In this work, we moved this endeavour to an uncharted territory of stateful computations. For this, we employed a proof system which, instead of a pure type theory (Martin-Löf 1984), is based on Separation Logic—a *Type Theory of State* (Nanevski 2016). Taking this vision as a guiding principle, we designed Synthetic Separation Logic—a modest extension of Separation Logic, tailored for program synthesis, and implemented a proof search algorithm for it. In doing so, we took full advantage of the power of *local reasoning* about state (O'Hearn et al. 2001), which resulted in a *principled* and *fast* approach for synthesyzing provably correct heap-manipulating programs.

ACKNOWLEDGMENTS

We wish to thank Aquinas Hobor and Reuben Rowe for their insightful and extremely detailed comments on presentation, formalism, and examples. Their technical feedback has helped immensely to bring out the best of this paper. We also wish to thank Shachar Itzhaky and Ranjit Jhala for their comments on the draft. We are very grateful to Franck Cassez for his help with configuring and using ScalaSMT. We thank the POPL'19 PC and AEC reviewers for the careful reading and many constructive suggestions on the logic, algorithm, and the implementation.

Sergey's research was supported by a generous gift from Google and by the grant by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV (Part II) (LNCS)*, Vol. 9780. Springer, 934–950.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS (Part I) (LNCS)*, Vol. 10205. Springer, 319–336.

Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. 2014. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In *FOSSACS (LNCS)*, Vol. 8412. Springer, 411–425.

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS)*, Vol. 6602. Springer, 1–17.

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.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. ACM, 109–122.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *APLAS (LNCS)*, Vol. 3780. Springer, 52–68.

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.

Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. SLAyer: Memory Safety for Systems-Level Code. In *CAV (LNCS)*, Vol. 6806. Springer, 178–183.

Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2012. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In *ATVA (LNCS)*, Vol. 7561. Springer, 167–182.

James Brotherston, Richard Bornat, and Cristiano Calcagno. 2008. Cyclic proofs of program termination in separation logic. In *POPL*. ACM, 101–112.

James Brotherston, Nikos Gorogiannis, and Max I. Kanovich. 2017. Biabduction (and Related Problems) in Array Separation Logic. In *CADE (LNCS)*, Vol. 10395. Springer, 472–490.

James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. 2012. A Generic Cyclic Theorem Prover. In *APLAS (LNCS)*, Vol. 7705. Springer, 350–367.

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 459–465.

Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing Order to the Separation Logic Jungle. In *APLAS (LNCS)*, Vol. 10695. Springer, 190–211.

Franck Cassez and Anthony M. Sloane. 2017. ScalaSMT: Satisfiability Modulo Theory in Scala (Tool Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. 51–55.

Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *ICFP*. ACM, 321–332.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*. ACM, 18–37.

Wei-Ngan Chin, Cristina David, and Cristian Gherghina. 2011. A HIP and SLEEK verification system. In *OOPSLA (Companion)*. ACM, 9–10.

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* 77, 9 (2012), 1006–1036.

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245.

Coq Development Team. 2018. *The Coq Proof Assistant Reference Manual - Version 8.8*. Available at http://coq.inria.fr/.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

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.

Dino Distefano and Matthew J. Parkinson. 2008. jStar: Towards Practical Verification for Java. In *OOPSLA*. ACM, 213–226.

Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *APLAS*. To appear.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*. ACM, 422–436.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*. ACM, 229–239.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *POPL*. ACM, 802–815.

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to make ad hoc proof automation less ad hoc. In *ICFP*. ACM.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *PLDI*. ACM, 62–73.

Aquinas Hobor and Jules Villard. 2013. The ramifications of sharing in data structures. In *POPL*. ACM, 523–536.

Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles E. Leiserson, and Rezaul Alam Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *OOPSLA*. ACM, 145–164.

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.

Swen Jacobs, Viktor Kuncak, and Philippe Suter. 2013. Reductions for Synthesis Procedures. In *VMCAI (LNCS)*, Vol. 7737. Springer, 88–107.

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Formal Asp. Comput.* 11, 5 (1999), 541–566.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *OOPSLA*. ACM, 407–426.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *PLDI*. 316–329.

Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. 2014. A Resource-Based Logic for Termination and Non-termination Proofs. In *ICFEM (LNCS)*, Vol. 8829. Springer, 267–283.

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *PLDI*. ACM, 542–553.

Xuan Bach Le and Aquinas Hobor. 2018. Logical Reasoning for Disjoint Permissions. In *ESOP (LNCS)*, Vol. 10801. Springer, 385–414.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *POPL*. ACM, 81–92.

K. Rustan M. Leino. 2013. Developing verified programs with Dafny. In *ICSE*. ACM, 1488–1490.

K. Rustan M. Leino and Aleksandar Milicevic. 2012. Program Extrapolation with Jennisys. In *OOPSLA*. ACM, 411–430.

Chuck Liang and Dale Miller. 2009. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.* 410, 46 (2009), 4747–4768.

Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. 2012. Recursive proofs for inductive tree data-structures. In *POPL*. ACM, 123–136.

Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121.

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis.

Andrew McCreight. 2009. Practical Tactics for Separation Logic. In *TPHOLs (LNCS)*, Vol. 5674. Springer, 343–358.

Chris Mellish and Steve Hardy. 1984. Integrating Prolog in the POPLOG Environment. In *Implementations of Prolog*. 147–162.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *PACMPL* 2, POPL (2018), 1:1–1:30.

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.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *ICLR*. To appear.

Aleksandar Nanevski. 2016. Separation Logic and Concurrency. (June 2016). Lecture notes for Oregon Programming Languages Summer School (OPLSS). Available from http://software.imdea.org/~aleks/oplss16/notes.pdf.

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. 261–274.

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. 2007. Automated Verification of Shape and Size Properties Via Separation Logic. In *VMCAI (LNCS)*, Vol. 4349. Springer, 251–266.

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.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 11:1–11:50.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *PLDI*. ACM, 619–630.

Frank Pfenning. 2010. Lecture Notes on Focusing. (June 2010). Oregon Summer School on Proof Theory Foundations (OPLSS). Available from https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014a. Automating Separation Logic with Trees and Data. In *CAV (LNCS)*, Vol. 8559. Springer, 711–728.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014b. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (LNCS)*, Vol. 8413. Springer, 124–139.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI*. ACM, 522–538.

Nadia Polikarpova and Ilya Sergey. 2018. Structuring the Synthesis of Heap-Manipulating Programs – Extended Version. *CoRR* abs/1807.07022 (2018). arXiv:1807.07022 http://arxiv.org/abs/1807.07022

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *OOPSLA*. ACM, 107–126.

Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. 2013. Natural proofs for structure, data, and separation. In *PLDI*. ACM, 231–242.

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *PACMPL* 1, OOPSLA (2017), 65:1–65:28.

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *CAV (Part II) (LNCS)*, Vol. 9207. Springer, 198–216.

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

Reuben N. S. Rowe and James Brotherston. 2017. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*. ACM, 53–65.

Gabriel Scherer. 2017. Search for Program Structure. In *SNAPL (LIPIcs)*, Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 15:1–15:14.

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *ESEC/SIGSOFT FSE*. ACM, 515–527.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *PLDI*. ACM, 326–340.

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *SAS (LNCS)*, Vol. 10422. Springer, 364–381.

Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *POPL*. ACM, 313–326.

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*. ACM, 530–541.

Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. ACM, 151–162.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26.