Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities
(Invited Paper)

Shachar Itzhaky[1], Hila Peleg[2], Nadia Polikarpova[2], Reuben N. S. Rowe[3], and Ilya Sergey[4]

[1] Technion, Israel (shachari@cs.technion.ac.il)
[2] University of California, San Diego, USA (hpeleg,npolikarpova@eng.ucsd.edu)
[3] Royal Holloway, University of London, UK (reuben.rowe@rhul.ac.uk)
[4] Yale-NUS College and National University of Singapore, Singapore (ilya.sergey@yale-nus.edu.sg)

Abstract. This paper presents the main ideas behind deductive synthesis of heap-manipulating program and outlines present challenges faced by this approach as well as future opportunities for its applications.

1 Introduction

Just like a journey of a thousand miles begins with a single step, an implementation of a working operating system, cryptographic library, or a compiler begins with writing a single function. This is not quite so for verified software, whose development starts with three "steps": a function *specification* (or, *spec*), followed by its *implementation*, and then by a *proof* that the implementation satisfies the spec. Although recent years have seen an explosion of increasingly diverse and sophisticated verified systems [14, 20, 26, 31, 41, 48, 71, 73, 96], their cost remains high, owing to the effort required to write formal specifications and proofs in addition to writing the code.

The good news is that in many cases the aforementioned three steps can be replaced by just one of them: writing the spec. The rest can be delegated to *deductive program synthesis* [52]—an emerging approach to automated software development, which takes as input a specifications, and searches for a corresponding program *together with its proof*.

Past approaches to deductive synthesis typically avoided low-level programs with pointers [43,69,83], which are notoriously difficult to reason about, making these approaches inapplicable to automating the development of verified systems code. The few techniques that did handle the heap [47,72] had significant limitations in terms of expressiveness and/or efficiency. Our prior work on the SuSLik synthesizer [70], has introduced an alternative approach to synthesis of pointer-manipulating programs, whose key enabling component is the use of Separation Logic (SL) [66, 75] as the specification formalism. Due to its proof scalability, Separation Logic enabled modular verification of low-level imperative code and has been implemented in a large number of automated and interactive program verifiers [4,7,18,37,57,62,64,68]. The main novelty of SuSLik was an observation

that the structure of SL specifications can be used to efficiently guide the search for a program and its proof. Since then, our follow-up work has explored automatic discovery of recursive auxiliary functions [34], generating independently checkable proof certificates for synthesized programs [93], and giving the user more control over the synthesis using concise mutability annotations [19].

As an appetizer for SL-powered deductive program synthesis consider the problem of flattening a binary tree data structure into a doubly-linked list. Assume also that the programmer would prefer to perform this transformation *in-place*, without allocating new memory, which they conjecture is possible because the nodes of the two data structures have the same size (both are records with a payload and two pointers). With SuSLik, the programmer can describe this transformation using the following Hoare-style SL specification:

$$\{\mathsf{tree}(\mathtt{x}, S)\}\ \texttt{flatten (loc x)}\ \{\mathsf{dll}(\mathtt{x}, y, S)\} \tag{1}$$

Here the precondition asserts that initially `x` points to the root of a tree, whose contents are captured by a set S. The postcondition asserts that after the execution of `flatten`, *the same location* `x` is a head of a doubly-linked list, with the same elements S as the initial tree (y denotes the existentially quantified back-pointer of the list). The definitions of the two predicates, tree and dll, which constrain the symbolic heaps in the pre- and postcondition are standard for SL [75] and will be shown in Sec. 2.

Given the spec (1), SuSLik takes less than 20 seconds to generate the program in Fig. 1, written in a core C-like language, as well as a formal proof that the program satisfies the spec. Several things are noteworthy about this program. First, the code indeed does not perform any allocation, and instead accomplishes its goal by switching pointers (in lines 17, 18, 23, and 25); this makes it economical in terms of memory usage as only a low-level program can be: similar code written in a functional language like OCaml would inevitably rely on garbage collection. Second, the main function `flatten` relies on an auxiliary recursive function `helper`, which the programmer did not anticipate; in fact the need for this auxiliary—and its specification—is discovered by SuSLik completely automatically. All the programmer has to do to obtain a provably correct implementation of `flatten` is to write the spec (1) and define the two SL predicates it uses, which are, however, reusable across different programs.

```
1  flatten(loc x) {
2    if (x == 0) {
3    } else {
4      let l = *(x + 1);
5      let r = *(x + 2);
6      flatten(l);
7      flatten(r);
8      helper (r, l, x);
9    }
10 }
11
12 helper(loc r, loc l,
13                loc x) {
14   if (l == 0) {
15     if (r == 0) {
16     } else {
17       *(r + 2) = x;
18       *(x + 1) = r;
19     }
20   } else {
21     let v = *l;
22     let w = *(l + 1);
23     *(l + 2) = r;
24     helper(r, w, l);
25     *(l + 2) = x;
26   }
27 }
```

Fig. 1: Flattening a tree into a DLL.

At this point, a critical reader might be wondering whether this technology is mature enough to move past hand-crafted benchmarks and assist them in developing the next CompCert [48] or CertiKOS [31]. For one, the program in Fig. 1 does not seem optimal: a closer look reveals that the role of `helper` is to concatenate the lists obtained by flattening the two subtrees, resulting in the overall $O(n^2)$ complexity *wrt.* the size of the original tree.[1] Apart from performance of synthesized programs, the reader might have the following concerns:

- What is the class of programs this approach is fundamentally capable of synthesizing? How picky is it to the exact shape of input specifications?
- Is the proof search predictably fast across a wide range of problems?
- Will the synthesized code be concise and easy to understand?
- Finally, what are the "killer apps" for this technology and in which domains can we hope for its adoption for practical need?

The goal of this manuscript is precisely to illustrate the remaining challenges in SL-based synthesis of heap-manipulating programs and outline some future research directions towards addressing these challenges. In the remainder of this paper we provide the necessary background and a survey of the results to date (Sec. 2); we then zoom in on the promising techniques for improving proof search (Sec. 3); in Sec. 4 we discuss the *completeness* of synthesis, outlining the work that needs to be done in order to formally characterize the class of programs that can and cannot be generated; in Sec. 5 we talk about possible extensions to the synthesis procedure for improving the quality of synthesized programs; finally, in Sec. 6 we discuss possible applications of SL-based synthesis, such as program repair, data migration, and concurrent programming.

2 State of the Art

2.1 Specifications

SuSLik takes as input a Hoare-style specification, *i.e.* a pair of a pre- and a post-condition. Consider, for example, a specification for a function `swap`,[2] which swaps the values of two pointers:

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\}\ \texttt{swap(loc x, loc y)}\ \{\mathtt{x} \mapsto b * \mathtt{y} \mapsto a\} \tag{2}$$

The precondition $\mathtt{x} \mapsto a * \mathtt{y} \mapsto b$ states that the relevant part of the heap contains two memory locations, `x` and `y`, which store values a and b, respectively. We also know that and $\mathtt{x} \neq \mathtt{y}$, because the semantics of *separating conjunction* ($*$) require that the two heaps it connects be *disjoint*. The postcondition $\mathtt{x} \mapsto b * \mathtt{y} \mapsto a$ demands that after executing the function, the values stored in `x` and `y` be swapped.

[1] In Sec. 4 we show what it takes to derive an alternative, linear-time solution.

[2] Our language has no `return` statement, hence all functions have return type `void`, which is omitted from the spec; return values are emulated by writing to the heap.

This specification also implicitly guarantees that `swap` always terminates and executes without memory errors (*e.g.*, null-pointer dereferencing). Note that `x` and `y` also appear as parameters to `swap`, and hence are *program variables*, *i.e.*, can be mentioned in the synthesized program; the payloads a and b, on the other hand, are *logical variables*, implicitly universally quantified, and must not appear in the program. In the rest of this paper, we distinguish program variables from logical variables by using monotype font for the former.

In general, in a specification $\{\mathcal{P}\}$ `f(...)` $\{\mathcal{Q}\}$, assertions $\mathcal{P}$, $\mathcal{Q}$ both have the form $\phi; P$, where the *spatial* part P describes the shape of the heap, while the *pure* part ϕ is a plain first-order formula that states the relations between variables (in (2) the pure part in both pre- and postcondition is trivially `true`, and hence omitted). For the spatial part, SuSLik employs the standard *symbolic heap* fragment of Separation Logic [66, 75]. Informally, a symbolic heap is a set of atomic formulas called *heaplets* joined with separating conjunction ($*$). The simplest kind of heaplet is a *points-to* assertion $x \mapsto e$, describing a single memory location with address x and payload e. An empty symbolic heap is represented with emp.

To capture linked data structures, such as lists and trees, SuSLik specifications use *inductive heap predicates*, which are standard in Separation Logic. For instance, the `tree` predicate used in (1) is inductively defined as follows:

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

The predicate is parametrized by the root pointer x and the set of tree elements S. This definition consists of two guarded *clauses*: the first one describes the empty tree (and applies when the root pointer is null), and the second one describes a non-empty tree. In the second clause, a tree node is represented by a three-element record starting at address x. Records are represented using a generalized form of the points-to assertion with an *offset*: for example, the heaplet $\langle x, 1\rangle \mapsto l$ describes a memory location at the address $x + 1$. The *block* assertion $[x, 3]$ is an artifact of C-style memory management: it represents a memory block of three elements at address x that has been dynamically allocated by `malloc` (and hence can be de-allocated by `free`). The first field of the record stores the payload v, while the other two store the addresses l and r of the left and right subtrees, respectively. The two disjoint heaps $\mathsf{tree}(l, S_l)$ and $\mathsf{tree}(r, S_r)$ store the two subtrees. The pure part of the second clause indicates that the payload of the whole tree consists of v and the subtree payloads, S_l and S_r.

2.2 The Basics of Deductive Synthesis

The formal underpinning of SuSLik is a deductive inference system called Synthetic Separation Logic (SSL). Given a pre-/postcondition pair $\mathcal{P}$, $\mathcal{Q}$, deductive synthesis proceeds by constructing a derivation of the SSL *synthesis judgment*, denoted $\{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} \mid c$, for some program c. In this derivation, c is the output program, constructed while searching for the proof of the synthesis goal

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

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

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

Fig. 2: Selected SSL rules (simplified).

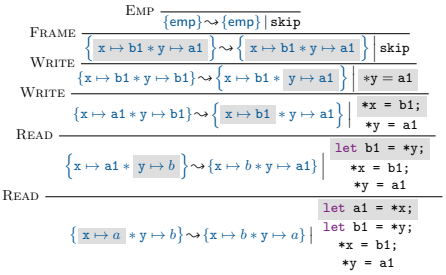


Fig. 3: Derivation of `swap`.

$\{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\}$. Intuitively, the output program c should satisfy the Hoare triple $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$. The derivation is constructed by applying inference rules, a subset of which is presented in Fig. 2, and every inference rule "emits" a program fragment corresponding to this deduction.

Fig. 3 shows an SSL derivation for `swap`, using inference rules of Fig. 2. The derivation, read bottom-up, starts with the pre/post pair from (2) as the synthesis goal; each rule application simplifies the goal until both the pre- and the post-heap are empty, and might also prepend a statement (highlighted in grey) to the output program. In the initial goal, the Read rule can be applied to the heaplet $\mathtt{x} \mapsto a$ to read the logical variable a from location `x` into a fresh program variable `a1`; the second application of Read similarly reads from the location `y`. At this point, the Write rule is applicable to the post-heaplet $\mathtt{x} \mapsto \mathtt{b1}$ because its right-hand side only mentions program variables and can be directly written into the location `x`; note that this rule equalizes the corresponding heaplets in the pre- and post-condition. After two applications of Write, the pre- and the post-heap become equal and can be simply cancelled out by the Frame rule, leaving emp on either side of the goal; the terminal rule Emp then concludes the derivation. Although very simple, this example demonstrates the secret behind SuSLik's efficiency: the shape of the specification restricts the set of applicable rules and thereby guides program synthesis.

2.3 Synthesis with Recursion and Auxiliary Functions

We now return to our introductory example—flattening a binary tree into a doubly-linked list—whose specification (1) we repeat here for convenience:

$$\{\mathsf{tree}(\mathtt{x}, S)\}\ \mathtt{flatten(loc\ x)}\ \{\mathsf{dll}(\mathtt{x}, y, S)\}$$

The definition of the tree predicate has been shown above (3); the predicate $\mathsf{dll}(x, y, S)$ describes a doubly-linked list rooted at x with back-pointer y and payload set S:

$$\begin{aligned} \mathsf{dll}(x, y, S) \triangleq\ & x = 0 \Rightarrow \{S = \emptyset; \mathsf{emp}\} \\ \mid\ & x \neq 0 \Rightarrow \{S = \{v\} \cup S';\ [x, 3] * x \mapsto v * \langle x, 1\rangle \mapsto n * \langle x, 2\rangle \mapsto y * \mathsf{dll}(n, x, S')\} \end{aligned} \tag{4}$$

Note that in the spec (1) both the set S and the back-pointer y are logical variables, but S is implicitly universally quantified (a so-called *ghost* variable),

```
flatten (loc x) {
  if (x == 0) {
  } else {
    let l = *(x + 1);   // tree(l, S_l)
    let r = *(x + 2);   // tree(r, S_r)
    flatten(l);         // dll(l, y_l, S_l)
    flatten(r);         // dll(r, y_r, S_r)
    .. ?? ..            // ⇝ dll(x, y, {v} ∪ S_l ∪ S_r)
  }
}
```

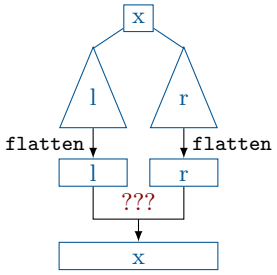


Fig. 4: Intermediate synthesis state when deriving `flatten`.

because it occurs in the precondition, while y is existentially quantified (a so-called *existential* variable), because it only occurs in the postcondition. The reader might be wondering why use an existential here instead of a null pointer: as we show below, such weakening is required to obtain the solution in Fig. 1; we discuss the alternative spec and corresponding solution in Sec. 4.

At a high level, the synthesis of `flatten` proceeds by eagerly making recursive calls on the left and the right sub-trees, `l` and `r`, as illustrated in Fig. 4, which leads to the following synthesis goal:

$$\{[\mathtt{x},3] * \mathtt{x} \mapsto v * \langle \mathtt{x},1\rangle \mapsto \mathtt{l} * \langle \mathtt{x},2\rangle \mapsto r * \mathsf{dll}(\mathtt{l}, y_l, S_l) * \mathsf{dll}(\mathtt{r}, y_r, S_r)\} \\ \rightsquigarrow \{\mathsf{dll}(\mathtt{x}, y, \{v\} \cup S_l \cup S_r)\} \tag{5}$$

Now the synthesizer must concatenate the two doubly-linked lists, rooted at `l` and `r`, together with the parent node `x` into a single list. Since the spec gives us no access to the last element of either of the two lists, this concatenation requires introducing a *recursive auxiliary function* to traverse one of the lists to the end. We now demonstrate how SuSLik synthesizes recursive calls and discovers the auxiliary using a single mechanism we call *cyclic program synthesis* [34], inspired by cyclic proofs in Separation Logic [11,76]. The main idea behind cyclic proofs is that, in addition to reaching a terminal rule like Emp, a sub-goal can be "closed off" by forming a cycle to an identical *companion* goal earlier in the derivation; in SSL these cycles give rise to recursive calls.

Fig. 5 depicts a cyclic derivation of `flatten`. For now let us ignore the applications of the Proc rule, which do not modify the synthesis goal; their purpose will become clear shortly. Given the initial goal (1), SuSLik first applies the Open rule, which unfolds the definition of tree in the precondition and emits a conditional with one branch per clause of the predicate. The first branch ($x = 0$) is trivially solved by `skip`, since a null pointer is both an empty tree and an empty list. The second branch is shown in Fig. 5: its precondition contains two predicate instances $\mathsf{tree}(\mathtt{l}, S_l)$ and $\mathsf{tree}(\mathtt{r}, S_r)$ for the two sub-trees of `x`.

Now SuSLik detects that either of those instances can be unified with the precondition $\mathsf{tree}(\mathtt{x}, S)$ of the top-level goal, so it fires the Call rule, which uses cyclic reasoning to synthesize recursive calls. More specifically, Call has two premises: the first one synthesizes a recursive call and the second one the rest of the program after the call. The spec of the first premise must be identical to

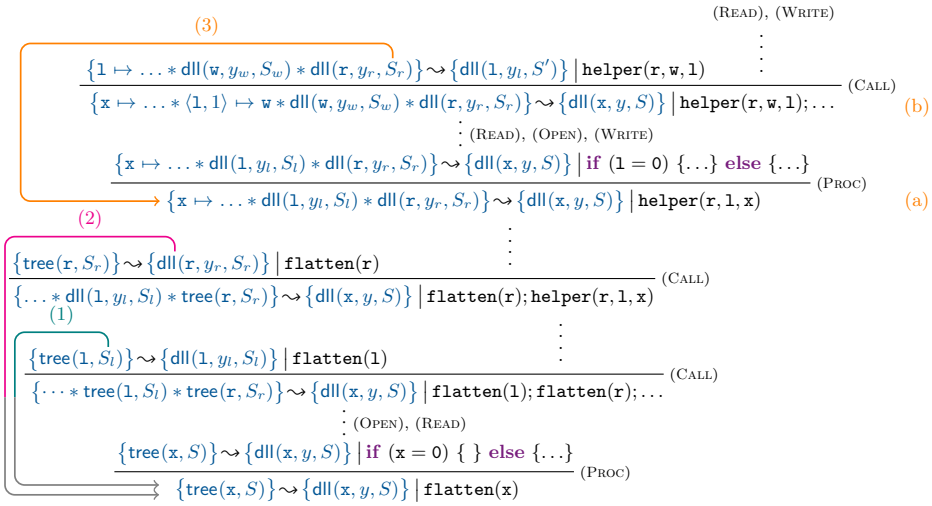


Fig. 5: Derivation of `flatten` and its recursive auxiliary `helper`.

some earlier goal, so that it can be closed off by forming a cycle; in our example, the back-link (1) connects the first premise back to the top-level goal. Once a companion goal is identified, SuSLik inserts an application of Proc right above it: its purpose is to delineate procedure boundaries, or, in other words, give a name to the piece of code that the Call rule is trying to call. To ensure that recursion is terminating, we must prove that $\mathsf{tree}(\mathtt{l}, S_l)$ in the precondition of the Call's premise is strictly smaller than $\mathsf{tree}(\mathtt{x}, S)$ in the precondition of the companion (see [34] for more details about our termination checking mechanism).

After the second application of Call (to $\mathsf{tree}(\mathtt{r}, S_r)$), SuSLik arrives at the goal (5), with two lists in the precondition (marked (a) in Fig. 5). Ignoring again the application of Proc, which will be inserted later, SuSLik proceeds by unfolding the list $\mathsf{dll}(\mathtt{l}, y_l, S_l)$ via Open, eventually arriving at the goal (b): this goal again has two lists in the precondition but one of them is now *smaller* (it is the tail of $\mathsf{dll}(\mathtt{l}, y_l, S_l)$). At this point Call detects that (a sub-heap of) goal (b) can be unified,[3] with goal (a) thus forming the cycle (3), which this time links to an *internal* goal in the derivation instead of the top-level goal. As before SuSLik inserts an application of the Proc rule just above the companion goal (a), thereby abducing an auxiliary procedure with a fresh name.

2.4 Implementation and Empirical Results

The most up-to-date implementation of SuSLik is publicly available at:

`https://github.com/TyGuS/suslik`

[3] This is where we rely on the existential back-pointer in (1): if we replace y_l with 0, then $\mathsf{dll}(\mathtt{l}, 0, S_l)$ and $\mathsf{dll}(\mathtt{w}, y_w, S_w)$ would not unify.

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*	*TimeSC*	*TimeNB*
Integers	1	swap two	1	4	1.0x	0.2	1.2	0.2
	2	min of two[1]	1	3	1.1x	0.8	3.0	1.1
Singly Linked List	3	length[2]	1	6	1.4x	0.4	0.5	0.6
	4	max[2]	1	11	1.9x	3.0	7.0	4.7
	5	min[2]	1	11	1.9x	2.9	6.7	4.1
	6	singleton[1]	1	4	0.9x	0.2	0.2	0.2
	7	deallocate	1	4	5.5x	0.2	0.2	0.2
	8	initialize	1	4	1.6x	0.4	0.4	0.6
	9	copy[3]	1	11	2.7x	0.6	1.0	393.3
	10	append[3]	1	6	1.1x	0.4	0.4	0.6
	11	delete[3]	1	12	2.6x	1.2	0.9	2.0
	12	deallocate two	2	9	6.2x	0.2	0.2	0.2
	13	append three	2	14	2.3x	1.0	2.5	1.7
	14	non-destructive append	2	21	3.0x	8.0	51.5	-
	15	union	2	23	5.5x	4.3	20.6	36.0
	16	intersection[4]	3	32	7.0x	101.1	121.2	-
	17	difference[4]	2	21	5.1x	4.7	55.0	29.5
	18	deduplicate[4]	2	22	7.3x	1.8	2.5	5.5
Sorted list	19	prepend[2]	1	4	0.4x	0.2	0.3	0.3
	20	insert[2]	1	19	3.1x	1.0	16.2	1.2
	21	insertion sort[2]	1	7	1.2x	0.7	2.7	42.7
	22	sort[4]	2	13	4.9x	1.0	1.5	2.9
	23	reverse[4]	2	11	4.0x	0.7	0.7	1.4
	24	merge[2]	2	30	4.4x	55.6	10.1	-
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2	0.2	0.5
	26	copy	1	22	4.3x	7.2	9.9	-
	27	append[3]	1	10	1.6x	1.7	27.2	-
	28	delete[3]	1	19	3.7x	3.4	3.5	-
	29	single to double	1	23	6.0x	0.7	0.8	4.6
List of Lists	30	deallocate	2	11	10.7x	0.2	0.3	0.3
	31	flatten[4]	2	17	4.4x	0.6	0.6	1.9
	32	length[5]	2	21	5.5x	22.8	-	-
Binary Tree	33	size	1	9	2.5x	0.4	0.6	185.8
	34	deallocate	1	6	8.0x	0.2	0.2	0.2
	35	deallocate two	1	16	11.8x	0.4	0.5	0.5
	36	copy	1	16	3.8x	2.5	42.9	-
	37	flatten w/append	1	17	4.8x	0.4	0.6	0.7
	38	flatten w/acc	1	12	2.1x	0.6	0.9	1.9
	39	flatten	2	23	7.1x	1.5	1.0	5.5
	40	flatten to dll in place	2	15	9.6x	11.3	-	23.2
	41	flatten to dll w/null[5]	2	17	11.2x	106.1	1418.3	46.5
BST	42	insert[2]	1	19	2.8x	14.6	21.7	518.0
	43	rotate left[2]	1	5	0.2x	6.2	7.0	-
	44	rotate right[2]	1	5	0.2x	4.9	5.6	-
	45	find min[5]	1	11	1.4x	66.3	80.2	-
	46	find max[5]	1	18	2.2x	58.0	80.8	-
	47	delete root[2]	1	18	1.3x	13.9	-	-
	48	from list[4]	2	27	5.7x	10.0	10.7	-
	49	to sorted list[4]	3	32	7.7x	20.8	11.7	-
Rose Tree	50	deallocate	2	9	12.0x	0.2	0.3	0.2
	51	flatten	3	25	8.0x	11.0	6.3	-
	52	copy[5]	2	32	7.9x	-	-	-
Packed Tree	53	pack[5]	1	16	1.6x	-	-	-
	54	unpack[5]	1	23	2.9x	21.0	-	-

[1] Jennisys [47] [2] ImpSynth [72] [3] Dryad [50] [4] Eguchi *et al.* [24] [5] New

Table 1: SuSLik benchmarks and results. We report the number of *Proc*edures generated, total number *Stmt* of statements in those procedures, the ratio *Code/Spec* of code to specification (in AST nodes), and the synthesis time in seconds for standard SuSLik (*Time*), with a simpler cost function (*TimeSC*) and with no bounds on predicate unfolding and calls (*TimeNB*). "-" denotes timeout after 30 minutes. Footnotes indicate the sources of benchmarks.

Tab. 1 collects the results of running SuSLik on benchmarks from our prior work [19,34,70,93] as well as seven new benchmarks, which we added to illustrate various challenges discussed in subsequent sections.[4] Most existing benchmarks had been adapted from the literature on verification and synthesis [24,47,50,72]. In addition to standard textbook data structures, our benchmarks include operations on two less common data structures, which to the best of our knowledge cannot be handled by other synthesizers. A *rose tree* [51] is a variable-arity tree, where child nodes are stored in a linked list; it is described in SL by two mutually recursive predicates (rtree for the tree and children for the list of children), and our synthesized operations on rose trees are also mutually recursive. A *packed tree* is a binary tree serialized into an array; it is interesting because operations on packed trees use non-trivial pointer arithmetic (we discuss them in Sec. 6).

Apart from the size of each program (in statements), we also report the ratio of code size to spec size (both in AST nodes) as a measure of synthesis utility. For the majority of the benchmarks the generated code is larger than the specification, sometimes significantly (up to $12x$); the only exceptions are benchmarks with very convoluted specs, such as BST rotations (benchmarks 43 and 44), or extremely simple programs, such `swap` from Fig. 3 (benchmark 1) and prepending an element to a sorted list (benchmark 19).

A number of benchmarks generate more than one procedure: those programs require *recursive auxiliaries* [34], such as our running example `flatten` from Fig. 1 (benchmark 40). It is worth mentioning that benchmarks 37 through 41 encode different versions of flattening a binary tree into a singly or doubly-linked list: 37 and 38 are simplified versions that do not require discovering auxiliaries because they contain additional hints from the user (a library function for appending lists in 37 and an inductive specification for `flatten` with a list accumulator in 38); 39 is similar to 40 but returns a singly-linked list (and hence requires allocation). Finally 41 is a version of 40 that uses 0 instead of y as the back-pointer of the output list; this precludes SuSLik from generating an auxiliary for appending two lists, and instead it discovers a slightly more complex, but *linear-time* solution, which we discuss in Sec. 4.

The missing synthesis times for some benchmarks indicate that they could not be synthesized automatically after 30 minutes, but were possible to solve in an "interactive" mode, where the search has been given hints on how to proceed in the case of multiple choices. We elaborate on the possibility of generating those programs automatically in subsequent sections. Apart from regular SuSLik time we also report time for two variations discussed in Sec. 3.

3 Proof Search

Similarly to existing deductive program synthesizers [43], SuSLik adopts *best-first* And/Or *search* [54] to search for a program derivation. The search space is represented as a tree with two types of nodes. An Or-node corresponds to a synthesis goal, whose children are alternative derivations, *any* of which is

[4] The code and benchmarks accompanying this paper are available online [35].

sufficient to solve the goal. An AND-node corresponds to a rule application, whose children are premises, *all* of which need to be solved in order to build a derivation. Each goal has a *cost*, which is meant to estimate how difficult it is to solve. The search works by maintaining a *worklist* of OR-nodes that are yet to be explored. In each iteration, the node with the least cost is dequeued and expanded by applying all rules enabled according to a *proof strategy*; the node's children are then added back to the worklist.

The proof strategy and the cost function are crucial to the performance of the proof search. In current SUSLIK implementation both are ad-hoc and brittle; in the rest of the section we outline possible improvements to their design.

3.1 Pruning via Proof Strategies

A *proof strategy* is a function that takes in a synthesis goal and its ancestors in the search tree, and returns a list of rules enabled to expand that goal. Without strategies, the branching factor of the search would be impractically large. SUSLIK's strategies are based on the observation that some orders of rule applications are redundant, and hence can be eliminated from consideration without loss of completeness. Identifying redundant orders is non-trivial and is currently done informally, increasing the risk of introducing incomplete strategies.

For example, SUSLIK's proof strategy precludes applying CALL if CLOSE (a rule that unfolds a predicate in the postcondition) has been applied earlier in the derivation. The reasoning is that CALL only operates on the precondition, while CLOSE only operates on the postcondition, hence the two rule applications must be independent, and can always be reordered so that CALL is applied first. But it gets more complicated once we let CALL abduce auxiliaries: now applying CALL after CLOSE could be useful to give it access to more companion goals, whose postconditions differ from that of the top-level goal. Consider for example copying a rose tree with the following spec:

$$\{\mathtt{r} \mapsto x * \mathsf{rtree}(x, S)\}\ \mathtt{void\ rtcopy(loc\ r)}\ \{\mathtt{r} \mapsto y * \mathsf{rtree}(y, S) * \mathsf{rtree}(x, S)\} \quad (6)$$

Copying a rose tree seems to require two mutually-recursive procedures: the main one (6) that copies an rtree and an auxiliary one that copies the list of its children, and hence has children instead of rtree in its postcondition. To our surprise, however, our proof strategy does not preclude the derivation of `rtcopy` (see benchmark 52 in Tab. 1): in this derivation, the auxiliary returns two rtrees, which are then unfolded after the call to extract the relevant children.

Future Directions. To develop more principled yet efficient strategies, we need to turn to the *proof theory* community, which has accumulated a rich body of work on efficient proof search. One technique of particular interest—*focusing* [53]—defines a *canonical representation* of proofs in linear logic [29] (more precisely, a canonical ordering on the application of proof rules, which can be enforced during the search by tracing local properties). Existing program synthesis work [27, 79] has leveraged ideas from focusing, but only in the setting of type inhabitation for pure lambda calculi. SUSLIK takes advantage of some of these ideas, too: it

designates some rules, such as READ and logical normalization rules, to be *invertible*; these rules can be applied eagerly and need not be backtracked. Beyond focusing, we might explore the applicability of more advanced canonical representations of programs and proofs [1, 33, 79]. We believe that these techniques will help us formalize and leverage inherent SSL symmetries, such as that two programs operating on disjoint parts of the heap can be executed in any order.

3.2 Prioritization via a Cost Function

When selecting the next goal to expand, SUSLIK's best-first search relies on a heuristic cost function of the form (with $p, w > 1$):

$$\mathsf{cost}(\{\phi, P\} \rightsquigarrow \{\psi, Q\}) = p * \mathsf{cost}(P) + \mathsf{cost}(Q) \quad \mathsf{cost}(\mathsf{p}(\overline{e})^{u,c}) = w * (1 + u + c)$$
$$\mathsf{cost}(P * Q) = \mathsf{cost}(P) + \mathsf{cost}(Q) \quad \mathsf{cost}(\_) = 1$$

In other words, a cost of a synthesis goal is a (weighted) total cost of all heaplets in its pre- and postcondition. The intuition is that the synthesizer needs to eliminate all these heaplets in order to apply the terminal EMP rule, so each heaplet contributes to the goal being "harder to solve". Predicates are more expensive than other heaplets, because they can be unfolded and produce more heaplets. In addition, for each predicate instance $\mathsf{p}(\overline{e})^{u,c}$ SUSLIK keeps track of the number of times it has been unfolded (u) or has gone through a call (c); factoring this into the cost prevents the search from getting stuck in an infinite chain of unfolding or calls. Finally, it can be useful to give a higher weight to the heaplets in the precondition, because many rules that create expensive search branches (most notably CALL) operate on the precondition.

Our implementation currently uses $p = 3, w = 2$, which is a result of manual tuning. Column *TimeSC* in Tab. 1 shows how synthesis times change if we set $p = 1$. As you can see, SUSLIK's performance is quite sensitive even to this small change: four benchmarks, which originally took under 30 seconds, now time out after 30 minutes, while benchmark 24, on the contrary, is solved five times faster. These results suggest that different synthesis tasks benefit from different search parameters, and that we might need a mechanism to tune SUSLIK 's search strategy for a given synthesis task.

In addition, because the cost heuristic is not efficient enough at guiding the search, we introduce hard bounds on the number of unfoldings and calls u and c for a predicate instance. Column *TimeNB* in Tab. 1 shows the results of running SUSLIK without these bounds: as you can see, 19 benchmarks time out (compared to only two in the original setup). The requirement to guess sufficient bounds for each benchmark hampers the usability of SUSLIK, hence in the future we would like to replace them with a better cost function.

Future Directions. To guide the search in a more intelligent and flexible way, we turn to extensive recent work on using learned models to guide proof search [8, 28, 49, 78, 95] and program synthesis [5, 15, 39, 46, 55, 82]. Guiding *deductive synthesis* would most likely require a non-trivial combination of these two lines of work.

In the area of proof search, existing techniques are used to select the next strategy in a proof assistant script [59, 60, 78, 95], or select a subset of clauses to

use in a first-order resolutions proof [9, 49]. Although these techniques are not directly applicable to our context, we can likely borrow some high-level insights, such as two-phased search [49], which applies a slow neural heuristic to make important decisions in early stages of search (*e.g.*, which predicate instances to unfold), and then less accurate but much faster hand-coded heuristics take over. Among the many techniques for guiding program synthesis, neural-guided deductive search (NGDS) [39] might be the natural place to start, since it shows how to condition the next synthesis step on the current synthesis sub-goal.

At the same time we also expect the limited size of the available dataset (*i.e.*, the benchmarks from Tab. 1) would hamper the application of deep learning to SuSLik. An alternative approach is to encode feature extractors [58] and apply machine learning algorithms to the result of such feature extractors. Another approach is to learn a coarse-grained model from available data and then adjust it during search, based on the feedback from incomplete derivations, as in [6,15,82].

4 Completeness

Soundness and completeness are desirable properties of synthesis algorithms. In our case, it is natural to formalize these properties relative to an underlying verification logic, which defines Hoare triples $\{P\}\ c\ \{Q\}$, with the total correctness interpretation "starting from a state satisfying P, program c will execute without memory errors and terminate in a state satisfying Q". This logic can be defined in the style of Smallfoot [7], using a combination of symbolic execution rules and logical rules, with the addition of cyclic proofs to handle recursion [76].

Relative *soundness* means that any solution SuSLik finds can be verified: $\forall P, Q, c.\ P \rightsquigarrow Q \mid c \Rightarrow \{P\}\ c\ \{Q\}$. Relative *completeness* means that whenever there exists a verifiable program, SuSLik can find one: $\forall P, Q. (\exists c. \{P\}\ c\ \{Q\}) \Rightarrow (\exists c'.\ P \rightsquigarrow Q \mid c')$. Proving relative soundness is rather straightforward, because SSL rules are essentially more restrictive versions of verification rules, hence an SSL derivation can be rewritten by translating every $P \rightsquigarrow Q \mid c$ into $\{P\}\ c\ \{Q\}$.[5] Completeness on the other hand is quite tricky, exactly because SSL rules impose more restrictions on the pre- and postconditions, in order to avoid blind enumeration of programs and instead guide synthesis by the spec. In the rest of this section we look into two major sources of relative incompleteness of SSL: recursive auxiliaries and pure reasoning.

4.1 Recursive Auxiliaries

A common assumption and source of incompleteness in recursive program synthesis [43, 67, 69] is that (1) synthesis is performed one function f at a time: if auxiliaries are required, their specifications are supplied explicitly; and (2) the specification Φ of f is *inductive*: one can prove that Φ holds of f's body assuming it holds of each recursive call. This restriction hampers the usability of synthesizers, because the user must guess all required auxiliaries and possibly generalize Φ to make it inductive, which in most cases requires knowing the implementation

[5] In our recent work we have developed an automatic translation from SSL derivations into three Coq-based verification logics [93].

```
1 intersect (loc r, y)
2 {
3   let x = *r;
4   if (x == 0) {
5   } else {
6     let v = *x;
7     let n = *(x + 1);
8     *r = n;
9     intersect(r, y);
10    insert(v, x, r, y);
11  }
12 }
```

```
13 insert(int v, loc x, r, y) {
14   let z = *r;
15   if (y == 0) { free(x); }
16   else {
17     let vy = *y;
18     let n = *(y + 1);
19     if (v == vy) {
20       *(x + 1) = z;
21       *r = x;
22     } else {
23       insert(v, x, r, n);
24 }}}
```

Fig. 6: Intersection of lists with unique elements. This implementation *cannot be synthesized* from (7), but a slight modification of it can, as explained in the text.

of f. As we have shown in Sec. 2, SuSLik mitigates these limitations to some extent, as it is able to discover auxiliary functions, such as `helper` in Fig. 1, automatically. To make the search tractable, however, cyclic synthesis restricts the space of auxiliary specifications considered by SuSLik to synthesis goals observed earlier in the derivation. Although this restriction is easy to state, we still do not have a formal characterization (or even a firm intuitive understanding) of the class of auxiliaries that SSL fundamentally can and cannot derive. Below we illustrate the intricacies on a series of examples.

Generalizing Pure Specs. One reason SuSLik might fail to abduce an auxiliary is that the pure part of the companion's goal might be too specific for the recursive call. Let us illustrate this phenomenon using the list intersection problem (benchmark 16 in Tab. 1) with the following specification, where ulist denotes a singly-linked list with unique elements:

$$\{\mathtt{r} \mapsto x * \mathsf{ulist}(x, S_x) * \mathsf{ulist}(\mathtt{y}, S_y)\} \rightsquigarrow \{\mathtt{r} \mapsto z * \mathsf{ulist}(z, S_x \cap S_y) * \mathsf{ulist}(\mathtt{y}, S_y)\} \quad (7)$$

Given this specification, we expected SuSLik to generate the program shown in Fig. 6. To compute the intersection of two input lists rooted in `x` and `y`, this program first computes the intersection of `y` and the tail of `x` (line 9). The auxiliary `insert` then traverses `y` to check if it contains `v` (the head of `x`), and if so, inserts it into the intermediate result `z` (line 23), and otherwise, de-allocates the node `x` (line 15). This program, however, cannot be derived by SSL; to see why let us take a closer look at the synthesis goal after line 9, which serves as the spec for `insert`:

$$\begin{aligned}\{S_x = \{\mathtt{v}\} \cup S_1 \wedge \mathtt{v} \notin S_1 \wedge S_z = S_1 \cap S_y; \mathtt{r} \mapsto z * \mathsf{ulist}(z, S_z) * \mathsf{ulist}(\mathtt{y}, S_y) * \\ \mathtt{x} \mapsto \mathtt{v} * \ldots\} \rightsquigarrow \{S'_z = S_x \cap S_y; \mathtt{r} \mapsto z' * \mathsf{ulist}(z', S'_z) * \mathsf{ulist}(\mathtt{y}, S_y)\}\end{aligned} \quad (8)$$

The issue here is that the pure spec is too specific: the precondition $S_z = S_1 \cap S_y$ and the postcondition $S'_z = S_x \cap S_y$ define the behavior of this function in terms

of the elements of input lists `x` and `y`, but the recursive call in line 23 replaces `y` with its tail `n` so these specifications do not hold anymore. The solution is to generalize the pure part of spec (8), so that it does not refer to S_x:

$$\{\mathtt{v} \notin S_z; \mathtt{r} \mapsto z * \mathsf{ulist}(z, S_z) * \mathsf{ulist}(\mathtt{y}, S_y) * \mathtt{x} \mapsto \mathtt{v} * \ldots\} \\ \rightsquigarrow \{S'_z = S_z \cup (\{\mathtt{v}\} \cap S_y); \mathtt{r} \mapsto z' * \mathsf{ulist}(z', S'_z) * \mathsf{ulist}(\mathtt{y}, S_y)\} \quad (9)$$

Alas, such a transformation of the pure spec is beyond SuSLik's capabilities.

To our surprise, SuSLik was nevertheless able to generate a solution to this problem by finding an alternative implementation for `insert`, shown on the right. This implementation *appends* `v` to `z` instead of *prepending* it; more specifically, `insert` starts by traversing `z`, and once it reaches the base case, it calls another auxiliary, `intersectOne` (omitted for brevity), which traverses `y` and returns a list whose elements are $\{\mathtt{v}\} \cap S_y$ (*i.e.*, a list with at most one element), which is then appended to the intersection. At a first glance it is unclear how this superfluous traversal of `z` can possibly help with generalizing the spec (8); the key to this mystery lies in the recursive call in line 22: note that as the second parameter, instead of the input list `x`, it actually uses `z` after replacing its head element with `v`! This substitution makes the overly restrictive spec of (8) actually hold.

```
13 insert(int v, loc x, r, y) {
14   let z = *r;
15   if (z == 0) {
16     intersectOne(v, x, r, y)
17   } else {
18     let vz = *z;
19     let n = *(z + 1);
20     *r = n;
21     *z = v;
22     insert(v, z, r, y);
23     ...
24 }}
```

Of course this implementation is overly convoluted and inefficient, so in the future we plan equip SuSLik with the capability to generalize pure specs. To this end, we plan to combine deductive synthesis with invariant inference techniques via bi-abduction [86]. For instance, whenever the Call rule identifies a companion goal, we can replace its pure pre- and post-condition ϕ and ψ with unknown predicates U_ϕ and U_ψ. During synthesis, we would maintain a set of Constrained Horn Clauses over these unknown predicates (starting with: $\phi \Rightarrow U_\phi$ and $U_\psi \Rightarrow \psi$); these constraints can be solved incrementally, like in our prior work [69], pruning the current derivation whenever the constraints have no solution. If synthesis succeeds, the assignment to U_ϕ and U_ψ corresponds to the inductive generalization of the original auxiliary spec. Since only the pure part of the spec is generalized, the spatial part can still be used to guide synthesis.

Accumulator Parameters. It is common practice to introduce an auxiliary recursive function to thread through additional data in the form of "accumulator" inputs or outputs. Cyclic program synthesis has trouble conjuring up arbitrary accumulators, since it constructs auxiliary specifications from the original specification via unfolding and making recursive calls.

Consider linked list reversal (23 in Tab. 1): SuSLik generates an inefficient, quadratic version of this program, which reverses the tail of the list and then appends its head to the result (hence discovering "append element" as the auxiliary). The canonical linear-time version of reversal requires an auxiliary with *two*

```
1 flatten (loc x) {
2   if (x == 0) {
3   } else {
4     let l = *(x + 1);
5     let r = *(x + 2);
6     flatten(l);
7     helper(r, l, x);
8   }
9 }
10
11 helper (loc r, loc l, loc x) {
12   if (r == 0) {
13     if (l == 0) {} else {
14       *(l + 2) = x;
15     }
16   } else {
17     let rl = *(r + 1);
18     let rr = *(r + 2);
19     *(r + 2) = rl;
20     *(r + 1) = l;
21     helper(rl, l, r);
22     *(x + 2) = rr;
23     *(x + 1) = r;
24     helper(rr, r, x);
25   }
26 }
```

Fig. 7: Flattening a tree into a DLL in linear time.

list arguments—the already reversed portion and the portion yet to be reversed—and hence is outside of SuSLik's search space: cyclic synthesis cannot encounter a precondition with two lists, as it starts with a single list predicate in the precondition, and neither unfolding nor making a call can duplicate it.

There are examples, however, where SuSLik surprized us by inventing the necessary accumulator parameters. Consider again our running example, flattening a tree into a doubly-linked list. Recall that given the spec (1), SuSLik synthesizes an inefficient implementation with quadratic complexity. A canonical linear-time solution requires an auxiliary that takes as input a tree and a list accumulator, and simply prepends every traversed tree element to this list; because of the accumulator parameter, discovering this auxiliary seems to be outside of scope of cyclic synthesis. To our surprise, SuSLik is actually able to synthesize a linear-time version of `flatten`, shown in Fig. 7 (and encoded as benchmark 41 in Tab. 1), given the following specification:

$$\{\mathsf{tree}(\mathtt{x}, S)\}\ \texttt{flatten (loc x)}\ \{\mathsf{dll}(\mathtt{x}, 0, S)\} \tag{10}$$

Compared with (1), the existential back-pointer y of the output list is replaced with the null-pointer 0, precluding SuSLik from traversing the output of the recursive call (*cf.* Sec. 2), which in this case comes in handy, since it enforces that every tree element is traversed only once.

The new solution starts the same way as the old one, by flattening the left sub-tree `l`, which leads to the following synthesis goal after line 6:

$$\{\mathsf{dll}(\mathtt{l}, 0, S_l) * \mathsf{tree}(\mathtt{r}, S_r) * [\mathtt{x}, 3] * \mathtt{x} \mapsto v * \ldots\} \rightsquigarrow \{\mathsf{dll}(\mathtt{x}, 0, \{v\} \cup S_l \cup S_r\}) \tag{11}$$

As you can see, the precondition now contains a tree and a list! Since it cannot recurse on the list $\mathsf{dll}(\mathtt{l}, 0, S_l)$, the synthesizer instead proceeds to unfold the tree $\mathsf{tree}(\mathtt{r}, S_r)$ and then use (11) as a companion for two recursive calls on `r`'s sub-trees, turning (11) into a specification for `helper` in Fig. 7.

4.2 Pure Reasoning

To enable synthesis of the wide range of programs demonstrated in Sec. 2, SuSLik must support a sufficiently rich logic of pure formulas. Our implementation

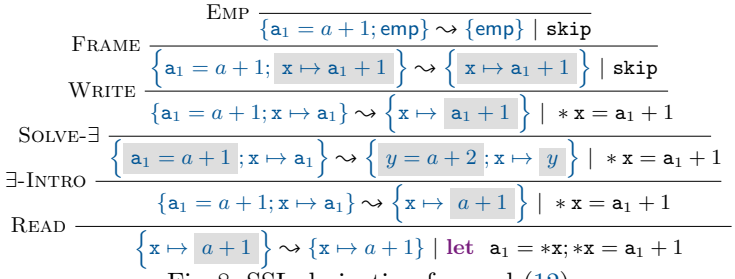


Fig. 8: SSL derivation for goal (12).

currently supports linear integer arithmetic and sets, but the general idea is to make SuSLik parametric *wrt.* the pure logic (as long as it can be translated into an SMT-decidable theory), and outsource all pure reasoning to an SMT solver.

In the context of synthesis, however, outsourcing pure reasoning is trickier than it might seem (or at least trickier than in the context of verification). Consider the following seemingly trivial goal:

$$\{\mathtt{x} \mapsto a + 10\} \rightsquigarrow \{\mathtt{x} \mapsto a + 11\} \tag{12}$$

This goal can be solved by incrementing the value stored in `x`, *i.e.*, by the program `let a1 = *x; *x = a1 + 1`. *Verifying* this program is completely straightforward: a typical SL verifier would use symbolic execution to obtain the final symbolic state $\{\mathtt{x} \mapsto a + 10 + 1\}$, reducing verification to a trivial SMT query $\exists a.a + 10 + 1 \neq a + 11$. *Synthesizing* this program, on the other hand, requires guessing the program expression `a1 + 1`, which does not occur anywhere in the specification.

To avoid blind enumeration of program expressions, SuSLik attempts to reduce the goal (12) to a *syntax-guided synthesis* (SyGuS) query [2]:

$$\exists f. \forall x, a, a_1. a_1 = a + 10 \implies f(x, a_1) = a + 11$$

Queries like this can be outsourced to numerous existing SyGuS solvers [3, 32, 46, 77]; SuSLik uses CVC4 [74] for this purpose. Because SyGuS queries are expensive, the challenge is to design SSL rules to issue these queries sparingly.

Fig. 8 shows how two pure reasoning rules, ∃-Intro and Solve-∃, work together to solve the goal (12). ∃-Intro is triggered by the postcondition heaplet $\mathtt{x} \mapsto a + 1$, whose right-hand side is a ghost expression, which blocks the application of Write. ∃-Intro replaces the ghost expression with a *program-level* existential variable y (*i.e.* an existential which can only be instantiated with program expressions). Now Solve-∃ takes over: this rule constructs a SyGuS query using all existentials in the current goal as unknown terms and the pure pre- and and post-condition as the SyGuS specification. In this case, the SyGuS query succeeds, replacing the existential y with the program term $\mathtt{a1} + 1$. From here on, the regular Write rule finishes the job.

Note that although the goal (12) is artificially simplified, it is extracted from a real problem: benchmark 32 in Tab. 1, length of a list of lists. In fact the versions of SuSLik reported in our previous work were incapable of solving this benchmark because they were lacking the ∃-Intro rule, which we only introduced recently. Although the current combination of pure reasoning rules works

well for all our benchmarks, it is still incomplete (even modulo the completeness of the pure synthesizer), because, for efficiency reasons, SOLVE-∃ only returns a single solution to the SyGuS problem, even if the pure specification allows for many. This might be insufficient when SOLVE-∃ is called before the complete pure postcondition is known (for example, to synthesize actual arguments for a call). Developing an approach to outsourcing pure reasoning that is both complete and efficient is an open challenge for future work.

5 Quality of Synthesized Programs

Should we hope that the output of deductive synthesis will be directly integrated into high-assurance software, we need to make sure that the code it generates is not only correct, but also efficient, concise, readable, and maintainable. The current implementation of SUSLIK does not take any of these considerations into account during synthesis; in this section we discuss two of these challenges, and outline some directions towards addressing them.

5.1 Performance

We have already mentioned examples of SUSLIK solutions with sub-optimal asymptotic complexity in Sec. 4: for example, SUSLIK generates quadratic programs for linked list reversal and tree flattening instead of optimal linear-time versions. Although a linear-time solution to tree flattening from Fig. 7 is actually within SUSLIK's search space (even with the more general spec (1)), SUSLIK opts for the sub-optimal one simply because it has no ability to reason about performance and hence has no reason to prefer one over the other.

To enable SUSLIK to pick the more efficient of the two implementations, we can integrate SSL with a resource logic, such as [56], following the recipe from our prior work on resource-guided synthesis [44]. One option is to annotate each points-to heaplet $x \mapsto^p e$ with non-negative *potential* p, which can be used to pay for execution of statements, according to a user-defined cost model. Predicate definitions can describe how potential is allocated inside the data structure; for example, we can define a tree with p units of potential per node as follows:

$$\begin{aligned}\mathsf{tree}(x, S, p) \triangleq\ & x = 0 \Rightarrow \{S = \emptyset; \mathsf{emp}\} \\ \mid\ & x \neq 0 \Rightarrow \{S = \{v\} \cup S_l \cup S_r; \\ & \quad [x, 3] * x \mapsto^p v * \langle x, 1\rangle \mapsto l * \langle x, 2\rangle \mapsto r * \mathsf{tree}(l, S_l, p) * \mathsf{tree}(r, S_r, p)\}\end{aligned}$$

We can now annotate the specification (1) with potentials as follows:

$$\{\mathsf{tree}(\mathtt{x}, S, 2)\}\ \texttt{flatten (loc x)}\ \{\mathsf{dll}(\mathtt{x}, y, S, 0)\} \tag{13}$$

If we define the cost of a procedure call to be 1, and the cost of other statements to be 0, this specification guarantees that `flatten` only makes a number of recursive calls that is linear in the size of the tree (namely, two calls per tree element). With this specification, the inefficient solution in Fig. 1 does not verify: since `helper` traverses the list `r`, it must assign some positive potential to every element of this list in order to pay for the call in line 24, but the specification

(13) assigns no potential to the output list. On the other hand, the efficient solution in Fig. 7 verifies: after the recursive call to `flatten` in line 6 we obtain $\{\mathsf{dll}(\mathtt{l}, y, S_l, 0) * \mathsf{tree}(\mathtt{r}, S_r, 2) * \ldots\}$; `helper` verifies against this specification since it only traverses the tree `r` and hence can use the two units of potential stored in its root to pay for the two calls in lines 21 and 24. In fact, the user need not guess the precise amount of potential $p = 2$ in the spec (13): any constant $p \geq 2$ would work to reject the quadratic solution and admit the linear one.

5.2 Readability

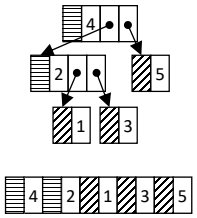
Although readability is hard to quantify, we have noticed several patterns in SuSLik-generated code that are obviously unnatural to a human programmer, and hence need to be addressed. Perhaps the most interesting problem arises due to inference of recursive auxiliaries: because SuSLik has no notion of abstraction boundaries, the allocation of work between the different procedures is often suboptimal. One example is benchmark 39 in Tab. 1, which flattens a binary tree into a *singly-linked* list. This example is discussed in detail in our prior work [34]; the solution is similar to `flatten` from Fig. 1, except that this transformation cannot be performed in-place: instead, the original tree nodes have to be deallocated, and new list nodes have to be allocated. Importantly, in SuSLik's solution, tree nodes are deallocated inside the `helper` function, whose main purpose is to append two lists. A better design would be to perform deallocation in the main function, so that `helper` has no knowledge of tree nodes whatsoever. To address this issue in the future we might consider different quality metrics when abducing specs for auxiliaries, such as encouraging all heaplets generated by unfolding the same predicate to be processed by the same procedure.

6 Applications

6.1 Program Repair

In our statement of the synthesis problem, complete programs are generated *from scratch* from Hoare-style specifications. But what if the program is already written previously but is buggy—would it be possible to automatically find a fix for it *if* we know what its specification is? This line of research, employing deductive synthesis for automated program repair [30], known as *deductive program repair*, has been explored in the past for functional programs [42] and simple memory safety properties [90], and only recently has been extended to heap-manipulating programs using the approach pioneered by SuSLik [63].

The SL-based deductive repair relies on existing automated deductive verifiers [17] to identify a buggy code fragment (which breaks the verification), followed by the discovery of the correct specification, which is used for the subsequent synthesis of the patch. The main shortcoming of the existing SL-based repair tools is the need to provide the top-level specs for the procedures in order to enable their verification (and potential bug discovery) in the first place. As



$$
\begin{aligned}
\mathsf{ptree}(x, n, S) &\triangleq && \{x \mapsto \mathit{tag} * \mathsf{ptree}'(x, \mathit{tag}, n, S)\} \\
\mathsf{ptree}'(x, \mathit{tag}, n, S) &\triangleq \mathit{tag} = 1 \Rightarrow && \{n = 1 \wedge S = \{v\}; \langle x, 1\rangle \mapsto v\} \\
&\mid \mathit{tag} = 0 \Rightarrow && \{n = 1 + n_l + n_r \wedge \\
& && S = \{v\} \cup S_l \cup S_r; \\
& && \langle x, 1\rangle \mapsto v * \mathsf{ptree}(x + 2, n_l, S_l) \\
& && * \mathsf{ptree}(x + 2 \cdot (1 + n_l), n_r, S_r)\}
\end{aligned}
$$

Fig. 9: (Left) Pointer-based and packed representations of the same binary tree. (Right) An SL predicate for packed trees.

a way to improve the utility of those tools, a promising direction is to employ existing static analyzers, such as Infer [12], to derive those specifications by *abducing* them from the usages of the corresponding functions [13].

6.2 Data Migration and Serialization

The pay-off of deductive synthesis is especially high for programs like tree flattening, which change the internal representation of a data structure without changing its payload; these programs usually have a simple specification, while their implementations can get much more intricate. One example where such programs can be useful is migration of persistent data: thanks to recent advancements in *non-volatile memory* (NVM) [40,45,84], large amounts of data are now persistently stored in memory, in arbitrary programmer-defined data structures. If the programmer decides to change the data structure, data has to be migrated between the old and the new representations, and writing those migration functions by hand can be tedious. In addition, reallocating large data structures is often prohibitively expensive, so the migration needs to be performed in-place, without reallocation. As we have demonstrated in our running example, this is something that can be easily specified and synthesized in SuSLik.

Another real-world application of this kind of programs is data serialization and de-serialization, where data is transformed back and forth between a standard pointer-based representation and an array so that it can be written to disk or sent over the network [16,91]. For example, Fig. 9 shows a pointer-based full binary tree and its serialized (or *packed*) representation, where the nodes are laid out sequentially in pre-order [92]. The right-hand-side of the figure shows an SL predicate ptree that describes packed trees: every node x starts with a *tag* that indicates whether it is a leaf; if x is not a leaf, its left child starts at the address $x + 2$ and its right child at $x + 2 \cdot (1 + n_l)$, where n_l is the size of the left child, which is typically unknown at the level of the program.

Imagine a programmer wants to synthesize functions that translate between these two representations, *i.e.*, `pack` and `unpack` the tree. The most natural specification for `unpack` would be:

$$
\{\mathtt{r} \mapsto x * \mathsf{packed}(x, sz, S)\}\texttt{unpack\_simple(loc r)}\left\{\begin{array}{c}\mathtt{r} \mapsto y * \mathsf{packed}(x, sz, S) \\ * \mathsf{tree}(y, sz, S)\end{array}\right\} \quad (14)
$$

This specification, however, cannot be implemented in SSL: when x is an internal node, we do not know the address of its right subtree, so we have nothing to pass into the second recursive call. Instead `unpack` must traverse the packed tree and discover the address of the right subtree by moving past the end of the left subtree; this can be implemented by returning the address past the end of the `ptree` together with the root of the newly built `tree`, as a record:

$$\{\mathtt{r} \mapsto x * \langle \mathtt{r}, 1\rangle \mapsto \_ * \ldots\}\ \texttt{unpack(loc r)}\ \{\mathtt{r} \mapsto x + 2 \cdot sz * \langle \mathtt{r}, 1\rangle \mapsto y * \ldots\} \quad (15)$$

With this specification, SuSLik is able to synthesize `unpack` in 20 seconds (benchmark 54 in Tab. 1); as for `pack` (benchmark 53), it is within the search space (which we confirmed in interactive mode) but automatic search currently times out after 30 minutes. In the future, it would be great if SuSLik could automatically discover an auxiliary with specification (15), given only (14) as inputs; this is similar to the problem of discovering accumulator parameters, which we discussed in Sec. 4, and is outside of capabilities of cyclic synthesis at the moment.

6.3 Fine-Grained Concurrency

Finally, we envision that deductive logic-based synthesis will make it possible to tackle the challenge of synthesizing provably correct concurrent libraries. The most efficient shared-memory concurrent programs implement custom synchronization patterns via fine-grained primitives, such as *compare-and-set* (CAS). Due to sophisticated interference scenarios between threads, reasoning about such programs is particularly challenging and error-prone, and is the reason for the existence of many extensions of *Concurrent* Separation Logic (CSL) [10,65] for verification of fine-grained concurrency [22,23,36,38,61,85,87–89].

For instance, *Fine-Grained* Concurrent Separation Logic (FCSL) [61,80,81], takes a very specific approach to fine-grained concurrency verification, following the tradition of logics such as LRG [25] and CAP [22] and building on the idea of splitting the specification of a concurrent library to a *resource protocol* and Hoare-style pre/postconditions. State-of-the art automated tools for fine-grained concurrency verification require one to describe *both* the protocol *and* Hoare-style pre/postconditions for the methods to be verified [21,94]. We believe, it should be possible to take those two components and instead synthesize the concurrent method implementations. The resource protocol will provide an extended set of language primitives to compose programs from. Those data structure-specific primitives can be easily specified in FCSL and contribute *derived* inference rules describing when these primitives can be used safely.

Acknowledgements. We thank Andreea Costea and Yutaka Nagashima for their feedback on the drafts of this paper. This research was supported by the National Science Foundation under Grant No. 1911149, by the Israeli Science Foundation (ISF) Grants No. 243/19 and 2740/19, by the United States-Israel Binational Science Foundation (BSF) Grant No. 2018675, by Singapore MoE Tier 1 Grant No. IG18-SG102, and by the Grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS).

References

1. Matteo Acclavio and Lutz Straßburger. From syntactic proofs to combinatorial proofs. In *IJCAR*, volume 10900 of *LNCS*, pages 481–497. Springer, 2018.
2. 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. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013.
3. Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS (Part I)*, volume 10205 of *LNCS*, pages 319–336. Springer, 2017.
4. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
5. Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.
6. Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA):227:1–227:29, 2020.
7. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
8. Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. Tactic Learning and Proving for the Coq Proof Assistant. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 138–150. EasyChair, 2020.
9. Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A Learning-Based Fact Selector for Isabelle/HOL. *J. Autom. Reason.*, 57(3):219–244, 2016.
10. Stephen Brookes and Peter W. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3):47–65, 2016.
11. James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, pages 101–112. ACM, 2008.
12. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.
13. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.
14. Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *SOSP*, page 243–258. ACM, 2019.
15. Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.
16. Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI*, pages 1–12. ACM, 1999.
17. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

18. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.
19. Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP*, volume 12075 of *LNCS*, pages 141–168. Springer, 2020.
20. Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the tls 1.3 record layer. In *S&P*, pages 463–482. IEEE Computer Society, 2017.
21. Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP*, volume 10201 of *LNCS*, pages 420–447. Springer, 2017.
22. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
23. Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*, volume 5502 of *LNCS*, pages 363–377. Springer, 2009.
24. Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. Automated synthesis of functional programs with auxiliary functions. In *APLAS*, volume 11275 of *LNCS*, pages 223–241. Springer, 2018.
25. Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009.
26. Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, page 287–305. ACM, 2017.
27. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, pages 802–815. ACM, 2016.
28. Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to Reason with HOL4 Tactics. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.
29. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
30. Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, 2019.
31. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, pages 653–669. USENIX Association, 2016.
32. Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.
33. Dominic J. D. Hughes. Unification nets: Canonical proof net quantifiers. In *LICS*, pages 540–549. ACM, 2018.
34. Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic Program Synthesis. In *PLDI*. ACM, 2021.
35. Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. SuSLik (CAV 2021 Artifact): Code and Benchmarks, May 2021. Available at https://doi.org/10.5281/zenodo.4850342.
36. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282. ACM, 2011.

37. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55, 2011.
38. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.
39. Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *ICLR*. OpenReview.net, 2018.
40. Takayuki Kawahara, Kenchi Ito, Riichiro Takemura, and Hideo Ohno. Spin-transfer torque RAM technology: Review and prospect. *Microelectron. Reliab.*, 52(4):613–627, 2012.
41. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *SOSP*, page 207–220. ACM, 2009.
42. Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive Program Repair. In *CAV*, volume 9207 of *LNCS*, pages 217–233. Springer, 2015.
43. Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426. ACM, 2013.
44. Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *PLDI*, pages 253–268. ACM, 2019.
45. Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pages 2–13. ACM, 2009.
46. Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*. ACM, 2018.
47. K. Rustan M. Leino and Aleksandar Milicevic. Program extrapolation with jennisys. In *OOPSLA*, pages 411–430. ACM, 2012.
48. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM, 2006.
49. Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.
50. Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012.
51. Grant Malcolm. Data structures and program transformation. *Sci. Comput. Program.*, 14(2-3):255–279, 1990.
52. Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
53. Jean marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.
54. Alberto Martelli and Ugo Montanari. Additive AND/OR graphs. In *IJCAI*, pages 1–11. William Kaufmann, 1973.
55. Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.
56. Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in iris. In *ESOP*, volume 11423 of *LNCS*, pages 3–29. Springer, 2019.
57. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

58. Yutaka Nagashima. LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL. In *APLAS*, volume 11893 of *LNCS*, pages 266–287. Springer, 2019.
59. Yutaka Nagashima. Smart Induction for Isabelle/HOL (Tool Paper). In *FMCAD*, pages 245–254. IEEE, 2020.
60. Yutaka Nagashima and Yilun He. PaMpeR: proof method recommendation system for Isabelle/HOL. In *ASE*, pages 362–372. ACM, 2018.
61. Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.
62. Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.
63. Thanh-Toan Nguyen, Quang-Trung Ta, Ilya Sergey, and Wei-Ngan Chin. Automated repair of heap-manipulating programs using deductive synthesis. In *VMCAI*, volume 12597 of *LNCS*, pages 376–400. Springer, 2021.
64. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333. ACM, 2006.
65. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
66. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
67. Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630. ACM, 2015.
68. Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper - complete heap verification with mixed specifications. In *TACAS*, volume 8413 of *LNCS*, pages 124–139. Springer, 2014.
69. Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016.
70. Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019.
71. Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *S&P*, pages 983–1002. IEEE Computer Society, 2020.
72. Xiaokang Qiu and Armando Solar-Lezama. Natural synthesis of provably-correct data-structure manipulations. *PACMPL*, 1(OOPSLA):65:1–65:28, 2017.
73. Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *USENIX Security Symposium*, page 1465–1482. USENIX Association, 2019.
74. Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.*, 55(2):73–102, 2019.
75. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
76. Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*, pages 53–65. ACM, 2017.
77. Shambwaditya Saha, Pranav Garg, and P. Madhusudan. Alchemist: Learning Guarded Affine Functions. In *CAV*, volume 9206 of *LNCS*, pages 440–446. Springer.

78. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, page 1–10. ACM, 2020.
79. Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *ICFP*, pages 243–255. ACM, 2015.
80. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.
81. Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110. ACM, 2016.
82. Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.
83. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.
84. Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453:80–83, 2008.
85. Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.
86. Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. Bi-Abduction with Pure Properties for Specification Inference. In *APLAS*, volume 8301 of *LNCS*, pages 107–123. Springer, 2013.
87. Aaron Turon. *Understanding and expressing scalable concurrency*. PhD thesis, Northeastern University, 2013.
88. Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356. ACM, 2013.
89. Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.
90. Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *ICSE*, pages 151–162. ACM, 2018.
91. Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. LoCal: a language for programs operating on serialized data. In *PLDI*, pages 48–62. ACM, 2019.
92. Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan Newton. Compiling tree transforms to operate on packed representations. In *ECOOP*, volume 74 of *LIPIcs*, pages 26:1–26:29. Schloss Dagstuhl, 2017.
93. Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the Synthesis of Heap-Manipulating Programs, 2021. Conditionally accepted at ICFP'21.
94. Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. Starling: Lightweight Concurrency Verification with Views. In *CAV, Part I*, volume 10426 of *LNCS*, pages 544–569. Springer, 2017.
95. Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *ICML*, volume 97 of *PMLR*, pages 6984–6994, 09–15 Jun 2019.
96. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *CCS*, page 1789–1806. ACM, 2017.