Cyclic Program Synthesis

Extended Version

Shachar Itzhaky
Technion
Israel
shachari@cs.technion.ac.il

Hila Peleg
University of California, San Diego
USA
hpeleg@eng.ucsd.edu

Nadia Polikarpova
University of California, San Diego
USA
nadia.polikarpova@ucsd.edu

Reuben N. S. Rowe
Royal Holloway, University of London
United Kingdom
reuben.rowe@rhul.ac.uk

Ilya Sergey
Yale-NUS College
National University of Singapore
Singapore
ilya.sergey@yale-nus.edu.sg

Abstract

We describe the first approach to automatically synthesizing heap-manipulating programs with auxiliary recursive procedures. Such procedures occur routinely in data structure transformations (*e.g.*, flattening a tree into a list) or traversals of composite structures (*e.g.*, *n*-ary trees). Our approach, dubbed *cyclic program synthesis*, enhances deductive program synthesis with a novel application of *cyclic proofs*. Specifically, we observe that the machinery used to form cycles in cyclic proofs can be reused to systematically and efficiently abduce recursive auxiliary procedures.

We develop the theory of cyclic program synthesis by extending Synthetic Separation Logic (SSL), a logical framework for deductive synthesis of heap-manipulating programs from Separation Logic specifications. We implement our approach as a tool called Cypress, and showcase it by automatically synthesizing a number of programs manipulating linked data structures using recursive auxiliary procedures and mutual recursion, many of which were beyond the reach of existing program synthesis tools.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Keywords: Program Synthesis, Separation Logic, Cyclic Proofs

1 Introduction

Consider the task of flattening a binary tree into a linked list, which is typically solved by writing a recursive data traversal program. The promise of *program synthesis* is to automate such tedious data manipulation tasks by generating programs automatically from high-level, declarative specifications. Several recent synthesizers [1, 19, 27–29] are indeed capable of generating *recursive* programs given only the top-level description of their behavior. For example, SuSLik [29] can generate a provably correct, recursive C-like program that *deallocates* a binary tree given as input the following specification in Separation Logic (SL) [26, 34]:

$$\{\mathsf{tree}(\mathsf{x}, s)\}\ \texttt{treefree(x)}\ \{\mathsf{emp}\} \tag{1}$$

This specification says that initially the heap contains a binary tree rooted at address x with set of elements s, and that after executing `treefree` the heap must be empty. Inspired by the success with `treefree`, the programmer might try to synthesize tree flattening from the specification below:

$$\{\mathsf{r} \mapsto \mathsf{x} * \mathsf{tree}(x, s)\}\ \texttt{flatten(r)}\ \{\mathsf{r} \mapsto y * \mathsf{sll}(y, s)\} \tag{2}$$

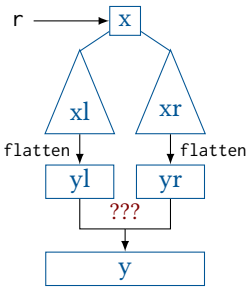
where sll describes a singly-linked list, and the location `r` initially stores the root of the input tree, and eventually the output list as computed by `flatten`. Much to the programmer's frustration, however, SuSLik fails to synthesize an implementation: it times out, without producing any useful output. In fact, given only the specification (2) or a similar one, and no additional hints, this example is out of reach for all other state-of-the-art synthesizers for recursive programs.

Challenge: recursive auxiliaries. This failure can be more readily understood when considering the expected solution. One such solution, depicted below, begins with two recursive calls which flatten the immediate subtrees of the input tree, obtaining two linked lists. Now, the synthesizer is at an impasse: how to combine the two lists to create the output list? This step requires an operationthat *appends* lists, which is in itself a recursive program (and one which cannot be implemented by another call to `flatten`, for example). This synthesis task illustrates a fundamental limitation of existing approaches to synthesis of recursion: tree flattening

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454087

cannot be accomplished without a *recursive auxiliary function*.[1] Existing synthesizers can generate `flatten` given the specification of append as a *hint* from the user [29]: using this auxiliary specification as a stepping stone, they can then generate code for both `flatten` and append as two synthesis tasks. It is, however, often non-trivial for the programmer to come up with this kind of hints.



Inferring auxiliary specifications *automatically* has been a long-standing open challenge in program synthesis. Since the hypothesis space is very large, simply allowing the synthesizer to conjecture any arbitrary auxiliary definition would be impractical, as that would make the search space explode. The state of the art is the work by Eguchi *et al.* [14], which attacks this problem in the context of functional programming by assuming a set of *predefined recursion templates* (*e.g.* a fold-right over lists). While syntactic templates help curb the search space explosion, they also limits the applicability of the technique, and in particular, preclude traversals of arbitrary user-defined data structures.

Cyclic program synthesis. We present a new synthesis technique, dubbed *cyclic program synthesis*, capable of automatically discovering recursive auxiliaries without the need for built-in templates or additional hints from the user. In particular, given the specification (2), our technique synthesizes a provably correct tree-flattening program, automatically discovering a recursive auxiliary that appends two lists.

Our technique draws inspiration from—and owes its name to—*cyclic proofs*, a powerful reasoning mechanism from the line of work on automated theorem proving [3, 9, 12, 25, 36, 37, 39]. In cyclic proofs, derivation "trees" can have *backlinks* from non-axiomatic leaves (called *buds*) to identical internal nodes (called *companions*). These backlinks enable an automation-friendly approach to proofs by induction: instead of conjecturing an induction hypothesis *a-priori*, the method exploits a similarity between a current goal and a companion goal elsewhere in the derivation, essentially turning the latter into an induction hypothesis *on demand*.

The key insight of this work is to apply the cyclic proof approach to automated deductive synthesis of recursive programs: instead of conjecturing auxiliary specifications *a priori*, one can *abduce* them from the repeated goal patterns encountered during the main program derivation. For example, in `flatten` the synthesizer starts generating the code of append *inline*, and later recognizes that the program can be completed by extracting the code into a function and adding a recursive call.

[1] The choice of auxiliary is not unique, *e.g.* one may propose an auxiliary that uses a list accumulator.

Our contributions. Realizing the idea of cyclic synthesis requires a carefully designed deductive system, in which the repeated goals observed in program derivations can be used to form backlinks. Our first contribution is the design of such a deductive system, which we dub Cyclic SSL ($SSL_{\circlearrowleft}$). This system builds on top of Synthetic Separation Logic (SSL), the deductive synthesis framework underlying SuSLik [29]. $SSL_{\circlearrowleft}$ features a new set of rules for synthesizing procedure calls, incorporating an expressive trace-based termination checking mechanism from cyclic proofs [7, 8]. This mechanism enables $SSL_{\circlearrowleft}$ to derive a wider range of recursive programs, including programs with auxiliaries and *non-structural* recursion.

Our second contribution is an implementation of $SSL_{\circlearrowleft}$ in a new synthesis tool Cypress, which extends and subsumes SuSLik. The addition of cyclic reasoning requires an efficient mechanism for detecting potential companions for the current goal during search. Cypress achieves this via a *call abduction oracle*, which matches up goals that are not syntactically equal, adjusting them accordingly. Cypress also features more efficient theory reasoning via a mechanism we dub *unification modulo theories*, as well as new *best-first* proof search, guided by the size and shape of the goal.

We evaluated Cypress on 46 synthesis benchmarks. Our evaluation shows that Cypress is able to solve a number of challenging tasks requiring nested traversals of linked structures (*e.g.*, sorting or de-duplicating a linked list) or traversals of mutually-recursive data structures (*e.g.*, *n*-ary trees). To the best of our knowledge, these programs are beyond reach of any existing approaches to automated hint-free synthesis from declarative specifications.

Paper outline. The following sections provide a brief primer on synthesis via SSL, outlining this work's innovations via a series of examples (Sec. 2); give a description of $SSL_{\circlearrowleft}$ and its meta-theory (Sec. 3); describe the synthesis algorithm (Sec. 4); and report on our evaluation (Sec. 5).

2 Cyclic Program Synthesis, by Example

This section presents the main ideas of Synthetic Separation Logic (SSL), followed by an overview of the scenarios enabled by our new rules and cyclic proofs principles. All code examples shown in this section are synthesized automatically using our novel synthesis tool Cypress.

2.1 Background: SSL and its Limitations

Specfications. Deductive synthesis based on SSL takes as input a pair of Hoare-style pre- and postconditions. For instance, recall the specification (1) for deallocating a tree:

$$\{\mathsf{tree}(x, s)\}\ \texttt{treefree(x)}\ \{\mathsf{emp}\}$$

Here the precondition $\mathsf{tree}(x, s)$ states that `treefree` may assume that it starts from a heap containing a binary tree rooted at address `x` with payload set s; the postcondition

emp states that `treefree` must guarantee that the heap is empty upon its termination.[2] Note that the tree root x also appears as a parameter to `treefree`, and hence is a *program variable*, *i.e.*, can be mentioned in the synthesized program; the payload set s, on the other hand, is a *logical variable* and must not appear in the program. In the rest of this section, 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 $\mathcal{P}$, $\mathcal{Q}$ 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 (1) the trivial pure part true is omitted from both pre- and postcondition). For the spatial part, SSL employs the standard *symbolic heap* fragment of Separation Logic [26, 34]. 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 $x \mapsto e$, which describes a single memory location with address x and payload e. For example, the formula $x \mapsto 5 * y \mapsto 10$ describes a heap with two memory locations, x and y, which store values 5 and 10, and are *distinct*, as per the semantics of the $*$ connective.

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

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

This definition consists of two guarded *clauses*: the first one describes the empty tree (and applies when the root pointer x 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. 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, correspondingly. 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 size three at address x that has been dynamically allocated by `malloc` (and hence can be de-allocated by `free`). The two disjoint heaps $\mathsf{tree}(l, s_l)$ and $\mathsf{tree}(r, s_r)$ store the two subtrees. Finally, the pure part of the second clause indicates that the payload of the whole tree consist of v and the subtree payloads, s_l and s_r.

Deductive synthesis. Given a pre-/postcondition pair $\mathcal{P}$, $\mathcal{Q}$, deductive synthesis proceeds by constructing a derivation

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

$$\textsc{Free}\ \frac{R = [\mathsf{x}, n] * *_{0 \le i \le n} (\langle \mathsf{x}, i\rangle \mapsto e_i) \quad \{\phi; P\} \rightsquigarrow \{Q\} \mid c}{\{\phi; P * R\} \rightsquigarrow \{Q\} \mid \mathtt{free(x)}; c}$$

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

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

Figure 1. Selected SSL rules (simplified).

of the SSL judgment $\{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\} \mid c$ for some program c. Intuitively, this judgment has the same meaning as the Hoare triple $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$ (different syntax is used to emphasise that operationally the program c is "the output" rather than "the input"). The derivation is constructed by applying inference rules, a subset of which is presented in Fig. 1.

Inference rules gradually simplify the initial synthesis goal $\{\mathcal{P}\} \rightsquigarrow \{\mathcal{Q}\}$, until symbolic heaps in both pre- and post-conditions are empty, at which point the *terminal* rule Emp concludes the derivation and "emits" a trivial program `skip`. The Frame rule reduces the synthesis goal to a smaller one by removing matching symbolic heaps in its pre- and post-condition. The Free rule eliminates a dynamically-allocated block of memory from the precondition by emitting a `free` statement. The rules Read and Write synthesize reads and writes of heap locations, correspondingly. The role of the Write rule is to "equalise" points-to heaplets that have the same address but different payloads, so that they can be subsequently "trimmed" by Frame. The role of the Read rule is to turn a logical variable a into a program variable y, which might enable subsequent application of Free or Write. Note that reading from the heap always creates a fresh program variable (hence the `let` syntax), and variables, unlike heap locations, are never re-assigned.

To deal with inductive predicates, SSL features rules Open and Close (elided from this overview), which unfold predicate definitions in the pre- and the postcondition, respectively. Finally, the Call rule synthesizes a recursive call if some part of the current goal's precondition matches the precondition $\mathcal{P}$ of the top-level, user-provided specification.

***Deriving* `treefree`.** Let us illustrate how all those rules work in tandem to synthesize the implementation of `treefree` shown on the right from the specification (1). We start by unfolding the definition of tree in the top-level goal $\{\mathsf{tree(x}, s)\} \rightsquigarrow \{\mathsf{emp}\}$, which generates two sub-goals (one for each clause of the predicate):

```
1  void treefree(x) {
2    if (x = 0) {
3    } else {
4      let l = *(x + 1);
5      let r = *(x + 2);
6      free(x);
7      treefree(l);
8      treefree(r);
9  }}
```

[2] This specification also implicitly guarantees that `treefree` always terminates and executes without memory errors (*e.g.*, null-pointer dereferencing).

$$\{\mathsf{x} = 0 \wedge s = \emptyset; \mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid c_1 \quad (4)$$

$$\{\mathsf{x} \neq 0; [\mathsf{x}, 3] * \mathsf{x} \mapsto v * \langle \mathsf{x}, 1\rangle \mapsto l * \langle \mathsf{x}, 2\rangle \mapsto r * \mathsf{tree}(l, s_l) * \mathsf{tree}(r, s_r)\} \rightsquigarrow \{\mathsf{emp}\} \mid c_2 \quad (5)$$

The programs c_1 and c_2 emitted by these subgoals will be conjoined via the statement `if (x = 0) { c1 } else { c2 }`. The first subgoal (4) is trivially solved by the rule Emp, resulting in a program `skip`. In the second subgoal (5), the two grayed fragments enable two subsequent applications of the rule Read, adding two reads statements, from `*(x + 1)` and `*(x + 2)`, correspondingly, creating two new program-level bindings, `l` and `r` and transforming the current goal into

$$\{\mathsf{x} \neq 0; [\mathsf{x}, 3] * \mathsf{x} \mapsto v * \langle \mathsf{x}, 1\rangle \mapsto \mathsf{l} * \langle \mathsf{x}, 2\rangle \mapsto \mathsf{r} * \mathsf{tree}(\mathsf{l}, s_l) * \mathsf{tree}(\mathsf{r}, s_r)\} \rightsquigarrow \{\mathsf{emp}\} \quad (6)$$

The Free rule now applies to the grayed fragment of the goal's precondition, emitting `free(x)`, and simplifying the synthesis goal to:

$$\{\ldots; \mathsf{tree}(\mathsf{l}, s_l) * \mathsf{tree}(\mathsf{r}, s_r)\} \rightsquigarrow \{\mathsf{emp}\} \quad (7)$$

To complete the synthesis, we notice that each of the two heaplets in the precondition matches the top-level specification (1), and hence can trigger the rule Call, synthesizing a procedure call with a suitable argument.

Limitations. The Call rule of SSL imposes significant limitations on the kinds of recursive functions it can derive. The first limitation is that it only allows using the *top-level* synthesis goal provided by the user as the specification for the callee; this precludes synthesis of recursive auxiliary functions, as required, for example, to flatten a tree into a list, as explained in the introduction.

A second, somewhat subtler limitation arises from the way SSL enforces termination of synthesized programs. To avoid generating trivial non-terminating solutions—such as `treefree(x)` immediately calling `treefree(x)` again—SSL restricts synthesized programs to be *structurally recursive*.[3] More precisely, synthesis starts by picking a *single inductive predicate* in the precondition that the program will "recurse on", and only allows a recursive call once this predicate has been unfolded at least once. This limitation makes it impossible to synthesize programs with more complex recursion patterns: even something as simple as a (helper-free) function that *deallocates two trees*, as such a function would need to traverse both of those trees recursively in a single run.

In the remainder of this section we will demonstrate how $\mathrm{SSL}_{\circlearrowleft}$ overcomes both of these limitations by harnessing the cyclic proof methodology to enhance the Call rule.

[3]Proof assistants like Coq [10] impose similar restrictions on recursion.

$$\textsc{Call}\ \frac{\{\phi_f; P\} \rightsquigarrow \{\psi_f; S\} \,|\, f(\overline{x_i}) \quad \vdash \phi \Rightarrow [\sigma]\phi_f \quad \{\phi \wedge [\sigma]\psi_f; [\sigma]S * R\} \rightsquigarrow \{Q\} \,|\, c}{\{\phi; [\sigma]P * R\} \rightsquigarrow \{Q\} \,|\, f(\sigma(\overline{x_i})); c} \qquad \textsc{Proc}\ \frac{\{\phi; P\} \rightsquigarrow \{\psi; Q\} \,|\, c}{\{\phi; P\} \rightsquigarrow \{\psi; Q\} \,|\, f(\overline{x_i})}$$

Figure 2. Rules for calls and definitions in $\mathrm{SSL}_{\circlearrowleft}$ (simplified).

2.2 Recursive Programs from Cyclic Proofs

The primary difference between SSL and $\mathrm{SSL}_{\circlearrowleft}$, is that the latter synthesizes a program by searching for a *cyclic* derivation, with recursive calls arising from *backlinks* in the proof.

New rules for function definitions and calls. To cater for synthesizing recursive calls using cyclic proofs, $\mathrm{SSL}_{\circlearrowleft}$ introduces a new, more general version of the Call rule, shown in Fig. 2. As before, this rule synthesizes a function call $f(\sigma(\overline{x_i}))$ (where σ substitutes actuals for formals) whenever some part of the current symbolic heap can be unified with f's spatial precondition P. The main difference is that the new rule no longer requires f's pre-/postcondition to be the top-level specification provided by the user. Instead—perhaps surprisingly—this rule simply requires that this pre-/post can *synthesize* the call $f(\overline{x_i})$. At a first glance, this rule is not very useful: in order to synthesize a function call, we (seemingly) need to synthesize a function call! The intention of this rule is to use its first premise as a *bud*, *i.e.*, link it back to an identical *companion* goal earlier in the derivation.

But where do we find a companion goal whose emitted program is $f(\overline{x_i})$? This is where the second new rule of $\mathrm{SSL}_{\circlearrowleft}$, Proc, comes into play (Fig. 2). The Proc rule should be understood as a way to "label" a node in a program derivation with a function definition $f(\overline{x_i})$. According to this rule, one can synthesize a definition for the function $f(\overline{x_i})$, if one can synthesize the body c for this function out of the goal with same pre- and postconditions. Note that unlike all the rules we've seen so far, Proc does not propagate the code emitted by its premise (the function body) to the conclusion; instead it abstracts this code away into an *identity call* $f(\overline{x_i})$, *i.e.*, a call whose actual parameters coincide with the formals. The most common occurrence of the rule Proc is at the root of the derivation, with the top-level goal being its conclusion. However, we will see in Sec. 2.3 how this rule also makes it possible to synthesize auxiliary recursive functions.

***Cyclic derivation of* `treefree`.** We now demonstrate the application of these new rules to build a cyclic derivaiton of `treefree`, shown in Fig. 3. For the sake of brevity, we omit the payload set variables and pure constraints related to them (*e.g.*, $s = \{v\} \cup s_l \cup s_r$) from the goals, as they don't play any role in our example. The instances of the tree predicate are annotated with *cardinality* variables α, α_l, and α_r, which play a role in the termination argument, as we will explain shortly. The proof's root goal serves as a conclusion for an application of Proc, with the "conjectured" function f being `treefree`. Most of the subsequent applications follow the sequence of rewritings via standard SSL rules outlined in

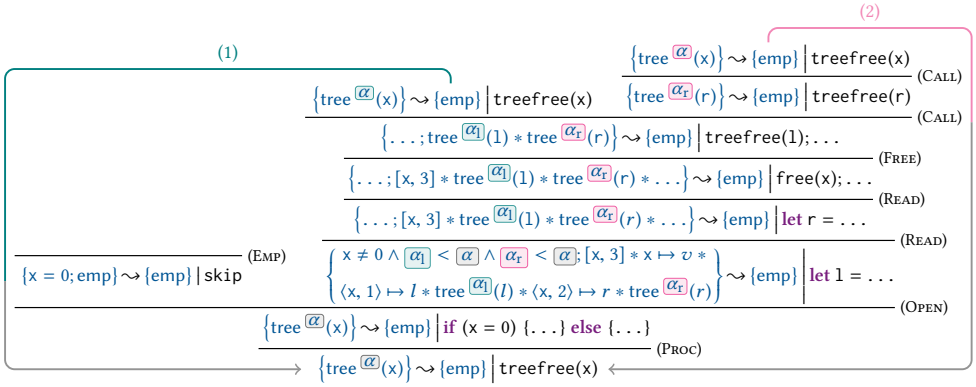


Figure 3. A derivation of `treefree(x)` with backlinks. Some pure assertions are omitted.

Sec. 2.1, with some evident parts of the goals elided. An important twist happens at the two inner derivation nodes corresponding to the premise of applying of the rule Call. The goal in both nodes matches precisely the conclusion of the top-level application of Proc. These pairs of matching nodes create *backlinks* in the program derivation, connecting the premises of two Call applications to the top-level goal.

Proving termination. Cyclic reasoning is not valid in general: cyclic proofs must satisfy a well-formedness condition in order to preclude infinite derivations. Intuitively, one can view the derivation as a directed graph with edges pointing from conclusions to premises and from buds to companions. The well-formedness condition requires that along every infinite path in this graph, some well-founded measure decreases infinitely often [8]. In $\text{SSL}_{\circlearrowleft}$, infinite paths in the derivation correspond to *potential* infinite traces of the program's execution. The well-formedness condition ensures that no such infinite executions exist since this would entail an infinitely decreasing chain in a well-founded set.

To trace well-founded measures, $\text{SSL}_{\circlearrowleft}$ annotates inductive predicates with *cardinality* variables, which can be seen as *sizes of the heap models* of the corresponding predicate. We automatically instrument all predicate definitions with cardinality information; for example, the instrumentation of the tree predicate (3) is highlighted via gray boxes below:

$$\text{tree}^{\boxed{\alpha}}(x, s) \triangleq x = 0 \Rightarrow \{s = \emptyset; \text{emp}\} \left| \; x \neq 0 \Rightarrow \left\{ \begin{array}{l} \boxed{\alpha_l < \alpha} \wedge \boxed{\alpha_r < \alpha} \wedge s = \{v\} \cup s_l \cup s_r; \\ [x, 3] * x \mapsto v * \langle x, 1\rangle \mapsto l * \langle x, 2\rangle \mapsto r \\ *\text{tree}^{\boxed{\alpha_l}}(l, s_l) * \text{tree}^{\boxed{\alpha_r}}(r, s_r) \end{array} \right\} \right.$$

Note the highlighted *cardinality constraints* we add to the second (recursive) clause, which state that cardinalities α_l and α_r of the the left and right subtrees are strictly smaller than α, that of the enclosing tree. Given this instrumentation, the cardinality constraints end up in the goal's precondition upon unfolding the predicate (via Open), as shown in Fig. 3.

To show that the derivation in Fig. 3 is well-formed, we must pick a sequence of cardinality variables to *trace* along each infinite path and prove that this sequence is strictly decreasing. All infinite paths in this derivation consist of arbitrarily alternating cycles (1) and (2). In each cycle, we will start by tracing $\boxed{\alpha}$, and then switch to either $\boxed{\alpha_l}$ or $\boxed{\alpha_r}$, depending on which cycle is being traversed. In either case, when we make the switch, the traced cardinality strictly decreases (following the cardinality constraints in the predicate definition), while elsewhere along the cycle is stays unchanged. Hence we have shown that along each infinite path the cardinality must strictly decrease infinitely often.

As prior work has shown [35], given cardinally constraints, the appropriate sequence of cardinalities to trace can be inferred *automatically* using automata-theoretic tools—an observation that is crucial for automating synthesis via $\text{SSL}_{\circlearrowleft}$. This mechanism subsumes termination measures based on maximum and/or lexicographic ordering of multiple arguments, and enable $\text{SSL}_{\circlearrowleft}$ to synthesize non-structurally recursive programs, for example, one that deallocates *two trees* as part of the same traversal (see Appendix A).

2.3 Synthesizing Auxiliary Recursive Functions

We now move on to the core of our contribution and illustrate how cyclic synthesis handles the motivating example from the introduction: the procedure that flattens a tree into a linked list, whose specification (2) we repeat for convenience:

$$\{\mathsf{r} \mapsto x * \text{tree}(x, s)\} \rightsquigarrow \{\mathsf{r} \mapsto y * \text{sll}(y, s)\}$$

We omit the definition of the singly-linked list predicate sll, since it is standard for Separation Logic [34] and analogous to the tree predicate (3). The specification enforces that the payload set of the output list be the same as that of the input tree by using the same logical variable s in the pre- and postcondition. For the sake of brevity, we omit the set-related reasoning in the derivations shown below.

Synthesized code. The program synthesized from this specification is shown in Fig. 5. We have given more descriptive names (in lieu of automatically-generated ones) to local variables and the auxiliary function. The main procedure `flatten` calls itself recursively twice to produce the lists corresponding to the left and right subtrees of the node `x`. The head of the first list, returned from the call in line 9 via `r`, is then stored in `y1`, while the head of the second list is stored in `r` (through an extra level of indirection) after the call in line

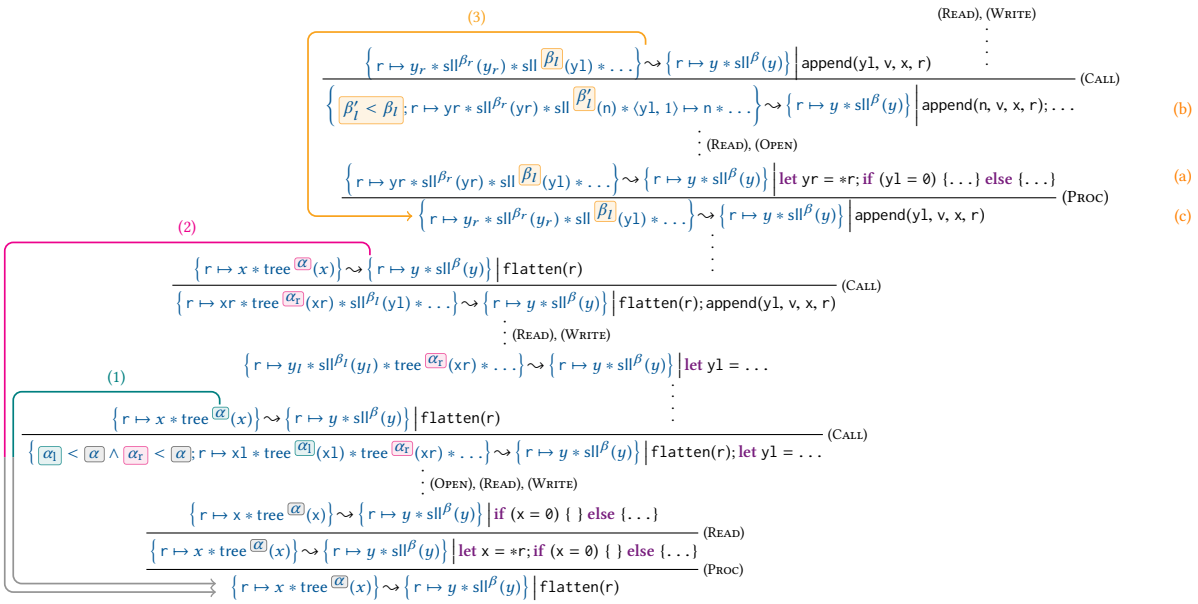


Figure 4. A derivation of `flatten` and its recursive auxiliary append.

```
1  void flatten(r) {
2    let x = *r;
3    if (x = 0) {
4    } else {
5      let v = *x;
6      let xl = *(x + 1);
7      let xr = *(x + 2);
8      *r = xl;
9      flatten(r);
10     let yl = *r;
11     *r = xr;
12     flatten(r);
13     append(yl, v, x, r);
14   }
15 }
```

```
16 void append(yl, v, x, r){
17   let yr = *r;
18   if (yl = 0) {
19     let y = malloc(2);
20     free(x);
21     *y = v;
22     *(y + 1) = yr;
23     *r = y;
24   } else {
25     let n = *(yl + 1);
26     append(n, v, x, r);
27     let y = *r;
28     *(yl + 1) = y;
29     *r = yl;
30   } }
```

Figure 5. Tree flattening program synthesized by Cypress.

12. As its last statement, `flatten` calls the recursive auxiliary function append, passing it the pointers to the two lists, `yl` and `r`, as well as the parent tree node `x` and its payload `v`.

The auxiliary procedure append concatenates the two lists, `yl` and `r`, inserting a new element with payload `v` in the middle (hence, computing an *in-order* unfolding of the tree). To this end, it traverses the first list, `yl`, recursively. Once it reaches the base case where `yl` is empty (lines 19–23), it frees the tree node `x`[4], allocates a new list node `y` with payload `v`, and prepends it to the second list, storing the result in `r`. Its inductive case (lines 25–29) calls append recursively on the tail of `yl`, after which it adjusts the tail pointer `(yl + 1)` appropriately and stores the result again in `r`.

***Cyclic derivation of* `flatten`.** To understand how the auxiliary append has been discovered, let us take a look at the SSL$_\circlearrowleft$ derivation of `flatten`, shown in Fig. 4. The first half of this derivation is uneventful and quite similar to what we have already seen for `treefree` in Fig. 3: the two applications of Call induce two backlinks (1) and (2) to the top-level goal and correspond to the recursive calls to `flatten(r)` in lines 9 and 12 of Fig. 5.

After these two calls, however, we find ourselves in the node (a) of the derivation[5], with the following synthesis goal:

$$\left\{\begin{array}{c} s = \{v\} \cup s_l \cup s_r; r \mapsto y_r * \mathsf{sll}(yl, s_l) * \mathsf{sll}(y_r, s_r) * \\ [x, 3] * x \mapsto v * \langle x, 1\rangle \mapsto \_ * \langle x, 2\rangle \mapsto \_ \end{array}\right\} \rightsquigarrow \{r \mapsto y * \mathsf{sll}(y, s)\} \quad (8)$$

Here, we are given two lists, `yl` and y_r, and we need to obtain a single list y that contains all of their elements plus `v`. How should the synthesizer proceed to solve this goal? It proceeds just like it would for any goal that contains inductive predicates in the precondition: by unfolding one of their instances—in this case $\mathsf{sll}(yl, s_l)$—via Open. The recursive premise of Open is depicted in the derivation node (b), whose precondition again features two lists: one of them, `yr`, is the same as before, and the other one, `n`, is the tail of `yl`.

Abducing the auxiliary. At this point the synthesizer realizes that the Call rule is applicable again; this time, however, the companion goal is *not* the top-level specification (2), but rather the goal (8) from node (a). Indeed, a sub-heap of the current precondition (the one containing everything but the head of `yl`) can be *unified* with the precondition of (8) by substituting `n` for `yl`. In order to become an eligible companion, however, a node must emit a procedure call, which is clearly not the case for (a). To bridge this mismatch, the synthesizer retroactively inserts an application of Proc just below (a) and creates a new node (c) as its conclusion. The

[4]Instead of passing `x` to append, it would have been more natural to deallocate it immediately in `flatten`. Although SSL$_\circlearrowleft$ is capable of deriving either program, our implementation makes the less natural choice, which we discuss further in Sec. 5.4.

[5]For now, imagine that node (c) and the adjacent application of Proc are not there. We will come back to them shortly.

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

Logical variable	ν, ω	Cardinality variable	α
$\mathcal{T}$-term	κ	$::= \nu \mid e \mid \cdots$	
Pure logic term	ϕ, ψ, χ	$::= \kappa \mid \phi = \phi \mid \phi \wedge \phi \mid \neg\phi$	
Symbolic heap	P, Q, R	$::= \mathsf{emp} \mid \langle e, \iota\rangle \mapsto e \mid [e, \iota] \mid \mathrm{p}^{\alpha}(\overline{\phi_i}) \mid P * Q$	
Heap predicate	$\mathcal{D}$	$::= \mathrm{p}^{\alpha}(\overline{x_i}) : \overline{e_j \Rightarrow \exists \overline{y}. \{\chi_j; R_j\}}$	
Assertion	$\mathcal{P}, \mathcal{Q}$	$::= \{\phi; P\}$	
Environment	Γ	$:= \forall \overline{x_i}.\ \exists \overline{y_j}.$	Context $\Sigma ::= \overline{\mathcal{D}}$

Figure 6. SSL$_{\circlearrowleft}$ programming language and assertion syntax.

new node has the same synthesis goal as (a) and an identity call to a fresh procedure—here `append(yl,v,x,r)`—as the emitted code.

This *lazy* application of Proc corresponds to abducing the recursive auxiliary on demand. The parameters of the new procedure are all the program variables of (a); its pre- and postcondition are defined by (8), the synthesis goal of (a); finally, its body is the code emitted by (a) (derived by the SSL$_{\circlearrowleft}$ sub-derivation rooted at (a)).

Termination. As before, we perform a global well-formedness check on the entire derivation. Note, however, that in this case infinite paths can either follow some combination of backlinks (1) and (2) or always follow the backlink (3); in other words, in the absence of mutual recursion, the termination arguments for the two procedures are entirely disjoint. Termination follows from the highlighted cardinality constraints: $\boxed{\alpha_l} < \boxed{\alpha}$, $\boxed{\alpha_r} < \boxed{\alpha}$, and $\boxed{\beta'_l < \beta_l}$.

2.4 More Examples

We conclude this section by outlining two examples that showcase unique capabilities of SSL$_{\circlearrowleft}$ and Cypress and go beyond the state of the art in synthesis with auxiliaries [14]. More details and synthesized code for these examples can be found in Appendix C.

Flattening a tree in-place. Our first example leverages the imperative nature of our underlying language, allowing Cypress to flatten a binary tree into a *doubly-linked* list *in-place*: $\{\mathtt{tree}(x, s)\}$ `flatten_to_dll(x)` $\{\mathtt{dll}(x, z, s)\}$. Note that here we require that the root of the input tree and the head of the output list be located at the *same* address `x`. We omit the standard definition of the `dll` predicate, but note that in-place flattening is possible because both a tree node and a doubly-linked list node are represented as a three-element record, so one can be reinterpreted as the other.

Mutual recursion. Our final example shows the ability of Cypress to synthesize manipulations with mutually-recursive data structures. An n-ary tree, *a.k.a. rose tree* [23], can be implemented by storing the children of a tree node in a linked list. Its definition in Separation Logic uses a pair of mutually recursive predicates: rtree, representing a tree, and children, representing a linked list of trees. Given these predicates and a specification $\{\mathtt{rtree}(x, s)\}$ `rtree_free(x)` $\{\mathsf{emp}\}$, Cypress is able to generate a pair of *mutually recursive* functions that deallocate a rose tree.

3 Cyclic Program Synthesis, Formally

In this section, we give a formal presentation of declarative rules of SSL$_{\circlearrowleft}$ and describe the underlying metatheory.

3.1 Programs and Assertions

Programming language. The target language of SSL$_{\circlearrowleft}$ is an imperative, C-like fragment with dynamic memory allocation, deallocation, store and load (Fig. 6, left). Values include at least booleans and integers, and a special type `loc` designates pointer variables. Pointers are isomorphic to unsigned integers, but there is only a single pointer constant, `0` (null). Expressions include at least variables, literal constants, equality check and logical connectives. Additional *theory-specific expressions* are allowed depending on the underlying theory used for checking entailment in derivations; our implementation supports linear integer arithmetic and sets. The language allows pointer arithmetic in the form $x + \iota$, but other arithmetic operations are disabled for pointers. The statement $f(\overline{e_i})$ denotes a procedure call with actual parameters $\overline{e_i}$. Procedures do not have a return value: the behavior of `return` is emulated by passing in an address of the heap location where the result should be stored. There are no variable re-assignments and no `while` loops. A program is a sequence of procedure definitions, followed by a statement.

Assertion language. The assertion language comprises pure assertions in the underlying theory, and SL assertions with inductively defined predicates (Fig. 6, right). Its semantics have been explored throughout Sec. 2. The set of pure logic terms (ϕ, ψ, χ) is a superset of program expressions e. The logic is sorted, and pure parts (ϕ in $\{\phi; P\}$) are ensured to be Boolean expressions via simple type checking. Predicate instances are annotated with cardinality variables α and β.

Assertions are interpreted in an environment Γ in which some of the variables are universally quantified and others existentially quantified, with a prefix of the form $\forall\overline{x}.\exists\overline{y}$. Program variables are *always* included in the universal prefix. Logical variables are split between universal (also called *ghost* variables) and existential. We denote $\mathrm{Vars}(\Gamma) = \{\overline{x}, \overline{y}\}$ for all quantified variables, and $\mathrm{PV}(\Gamma) = \{\overline{x}\} \cap \mathrm{PV}$, $\mathrm{GV}(\Gamma) = \{\overline{x}\} \setminus \mathrm{PV}$, $\mathrm{EV}(\Gamma) = \{\overline{y}\}$ for program variables, ghost variables and existentials of Γ, respectively. We also use $e[\Gamma]$ for the set of all

R0 Terminal rules

$$\textsc{Emp}\ \frac{\vdash \phi \Rightarrow \psi}{\Gamma;\ \{\phi;\mathsf{emp}\} \rightsquigarrow \{\psi;\mathsf{emp}\} \mid \mathtt{skip}} \qquad \textsc{Inconsistency}\ \frac{\vdash \phi \Rightarrow \bot}{\Gamma;\ \{\phi;P\} \rightsquigarrow Q \mid \mathtt{error}}$$

R1 Pointer operations

$$\textsc{Read}\ \frac{\forall y.\Gamma;\ \{\phi \wedge y = e; \langle x,\iota\rangle \mapsto e * P\} \rightsquigarrow Q \mid c \qquad x \in PV \quad y \in \mathsf{PV} \setminus \mathsf{Vars}(\Gamma)}{\Gamma;\ \{\phi;\langle x,\iota\rangle \mapsto e * P\} \rightsquigarrow Q \mid \mathbf{let}\ y = *(x+\iota); c}$$

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

$$\textsc{Alloc}\ \frac{\Gamma;\ \left\{\phi;[y,n] * \left(\langle y,i\rangle \mapsto t_i\right)_{0\le i<n} * P\right\} \rightsquigarrow \left\{\psi;[x,n] * \left(\langle x,i\rangle \mapsto e_i\right)_{0\le i<n} * Q\right\} \mid c \qquad x \in \mathsf{EV}(\Gamma)}{\Gamma;\ \{\phi;P\} \rightsquigarrow \left\{\psi;[x,n] * \left(\langle x,i\rangle \mapsto e_i\right)_{0\le i<n} * Q\right\} \mid \mathbf{let}\ y = \mathtt{malloc}(n); c}$$

$$\textsc{Free}\ \frac{\Gamma;\ \{\phi;P\} \rightsquigarrow Q \mid c \qquad x \in \mathsf{PV}}{\Gamma;\ \left\{\phi;[x,n] * \left(\langle x,i\rangle \mapsto e_i\right)_{0\le i<n} * P\right\} \rightsquigarrow Q \mid \mathtt{free}(x); c}$$

R2 Predicate unfolding

$$\textsc{Open}\ \frac{\Gamma \cup \forall\overline{\omega_{jk}};\ [\overline{t_i/\nu_i}]\left\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * P\right\} \rightsquigarrow Q \mid c_j \quad \textit{for all } j{=}1..r \qquad \mathsf{p}^{\alpha}(\overline{\nu_i}) : e_j \Rightarrow \exists\overline{\omega_{jk}}.\left\{\chi_j;R_j\right\}_{j=1..r} \in \Sigma \qquad \omega_{jk} \notin \mathsf{Vars}(\Gamma), \mathsf{GV}(t_i) = \varnothing}{\Gamma;\ \left\{\phi;\mathsf{p}^{\alpha}(\overline{t_i}) * P\right\} \rightsquigarrow Q \mid \begin{array}{l}\mathbf{if}\ ([\overline{t_i/\nu_i}]e_1)\ \{c_1\} \\ \mathbf{else\ if}\ ([\overline{t_i/\nu_i}]e_2)\ \{c_2\}\ \mathbf{else}\ \cdots\end{array}}$$

$$\textsc{Close}\ \frac{\Gamma \cup \exists\overline{\omega_{jk}};\ \mathcal{P} \rightsquigarrow [\overline{t_i/\nu_i}]\left\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * Q\right\} \mid c_j \quad \textit{for some } j{=}1..r \qquad \mathsf{p}^{\alpha}(\overline{\nu_i}) : e_j \Rightarrow \exists\overline{\omega_{jk}}.\left\{\chi_j;R_j\right\}_{j=1..r} \in \Sigma \qquad \omega_{jk} \notin \mathsf{Vars}(\Gamma)}{\Gamma;\ \mathcal{P} \rightsquigarrow \left\{\phi;\mathsf{p}^{\alpha}(\overline{t_i}) * Q\right\} \mid c}$$

R3 Procedures

$$\textsc{Proc}\ \frac{\Gamma;\ \mathcal{P} \rightsquigarrow Q \mid c \qquad \{\overline{x_i}\} = \mathsf{PV}(\Gamma)}{\Gamma;\ \mathcal{P} \rightsquigarrow Q \mid f(\overline{x_i})}$$

$$\textsc{Call}\ \frac{\begin{array}{l}\forall\overline{x_i},\overline{\nu_j}.\exists\overline{\omega_k};\ \{\phi';P\} \rightsquigarrow \{\psi';S\} \mid f(\overline{x_i}) \\ \Gamma \cup \forall\sigma(\overline{\omega_i});\ \{[\sigma]\psi' \wedge \phi; [\sigma]S * R\} \rightsquigarrow Q \mid c \qquad \mathsf{dom}\,(\sigma) = \{\overline{x_i},\overline{\nu_j},\overline{\omega_k}\} \\ \vdash \phi \Rightarrow [\sigma]\phi' \qquad \sigma(x_i) \in e[\Gamma]; \sigma(\nu_j) \in \kappa[\Gamma]\end{array}}{\Gamma;\ \{\phi;[\sigma]P * R\} \rightsquigarrow Q \mid f(\sigma(\overline{x_i})); c}$$

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

Figure 7. Synthesis rules of $\mathrm{SSL}_{\circlearrowleft}$.

expressions that can be constructed using $\mathsf{PV}(\Gamma)$, and $\kappa[\Gamma]$—all logical terms that can be constructed with $\mathsf{Vars}(\Gamma)$. When joining environments, the quantifier pattern is preserved, and we use the notation $\forall\overline{x}.\exists\overline{y} \cup \forall\overline{x'}.\exists\overline{y'} = \forall\overline{x},\overline{x'}.\exists\overline{y},\overline{y'}$. All variable sets $\overline{x}$, $\overline{x'}$, $\overline{y}$, $\overline{y'}$ must be disjoint.

A clause $e \Rightarrow \exists\overline{y}.\{\chi;R\}$ in a definition of a heap predicate $\mathsf{p}^{\alpha}(\overline{x_i})$ consists of a *selector* expression e and an assertion $\{\chi;R\}$. As explained in Sec. 2.2, the pure part χ contains a *cardinality constraint* of the form $\alpha_k < \alpha$ for each predicate instance $\mathsf{q}^{\alpha_k}(\ldots)$ inside R. These are not part of the surface syntax, but are added automatically during pre-processing.

3.2 Proof Rules

Fig. 7 lists all synthesis-related $\mathrm{SSL}_{\circlearrowleft}$ rules in a declarative manner. The rules operate on *transforming entailment* judgments $\Gamma;\ \mathcal{P} \rightsquigarrow Q \mid c$, which informally means that any state that satisfies the assertion $\mathcal{P}$ can be transformed into some state that satisfies Q by a statement c [29]. Within rules, for clarity, we follow the convention of using lower latin letters x, y for program variables, e, t for program-level terms (of the syntactic class e in Fig. 6), greek letters ν, ω for logical variables, and ϕ, ψ, χ for logical formulas. We will now describe the rules and their effects on the synthesis.

R0 ***Terminal rules.*** The rules Emp and Inconsistency form the leaves of a derivation by emitting a `skip` for a trivial goal or `error` for a vacuous goal (unsatisfiable precondition). Here $\vdash \phi \Rightarrow \psi$ denotes entailment between pure formulas in the underlying theory; our implementation uses an SMT solver to discharge these premises.

R1 ***Rules for atomic operations.*** Each of the pointer operations of the target language has a corresponding rule that describes when to emit it: Read and Write for memory access, Alloc and Free for dynamic memory management. These rules are more restricted versions of the corresponding symbolic execution rules of standard Separation Logic [5]; the additional restrictions are required to guide rule applications in the context of synthesis, where the program is not available. For example, the Write rule uses the heaplet $\langle x,\iota\rangle \mapsto e$ in the goal's *postcondition* to determine what should be written into the address x.

R2 ***Rules for inductive predicates.*** The two rules Open and Close unfold definitions of inductive predicates in the pre- and the postcondition, respectively. These rules have dual effects on the program in the sense that Open *eliminates* a predicate instance and hence performs a case split on its clauses, while Close *introduces* an instance and hence picks a single clause non-deterministically. As a result, Close does not emit any code, while Open emits a conditional statement with one branch per clause, using clause selectors as guards.

R3 ***Rules for procedures.*** These rules are the main innovation of $\mathrm{SSL}_{\circlearrowleft}$ as compared with SSL. The rules Proc and Call were discussed in detail in Sec. 2.2. The rule Proc delineates procedure boundaries by emitting a call to a procedure whose body is emitted by its single premise.

Procedure calls are generated via the rule Call. It bundles together Separation Logic-style framing, with R as the frame, and substitution of actual into formal parameters via σ, which is also applied to the postcondition $\{\psi';S\}$. Existential variables in the procedure's environment are remapped to fresh ghost variables in the second premise of the rule. Formal parameters x_i are mapped to program expressions $e[\Gamma]$ using program variables of Γ, and ghosts ν_j are mapped to logical terms using any variables of Γ.

CallSetup handles the situations when some code has to be emitted before the call can be made. To see why this rule

is required, consider the recursive call `flatten(r)` in line 9 of Fig. 5. To enable this call we first need to write the root of the left subtree `x1` into the return location `r`; but to trigger this write, we need to have $\texttt{r} \mapsto \texttt{x1}$ in the goal's postcondition! CallSetup enables that: it performs *sequential decomposition* of the synthesis goal with the call's precondition $\{\psi; S\}$ as the intermediate state. As a result, $\texttt{r} \mapsto \texttt{x1}$ ends up in the postcondition of its first subgoal, triggering the synthesis of the write in line 8. Although the declarative presentation in Fig. 7 makes it look like the assertion $\{\psi; S\}$ is chosen non-deterministically, in Sec. 4 we explain how Cypress implements this rule efficiently via a *call abduction oracle*.

Logical rules. The remainder of $\text{SSL}_\circlearrowleft$ rules are *logical rules*, which do not emit new code, but instead transform the goal in a way that would ultimately allow the application of other (operational) rules. Our logical rules are mostly standard for SL-based theorem provers [5], so we relegate them to Appendix B to save space. They include the Frame rule we have seen in Sec. 2, as well as rules for pure reasoning, *e.g.* eliminating existentials. In Sec. 4 we discuss how Cypress implements such pure reasoning efficiently.

3.3 Cyclic Program Derivations

An $\text{SSL}_\circlearrowleft$ derivation is a tree consisting of goals $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$, and constructed using the inference rules in Fig. 7. In contrast to the standard notion of a proof as a finite derivation tree, $\text{SSL}_\circlearrowleft$ derivations are permitted to be regular, non-well-founded (*i.e.*, infinitely tall) trees. Regularity ensures that an $\text{SSL}_\circlearrowleft$ derivation tree always has a finite representation as a (possibly) cyclic graph. Concretely, we represent $\text{SSL}_\circlearrowleft$ derivations as finite trees, along with a set of *backlinks* connecting each non-terminal leaf node to a syntactically identical ancestor node. In other words, we allow goals from the middle of the proof to be used again as premises higher up the derivation tree. Formally, each backlink denotes the infinite unfolding of the path connecting the leaf—called *bud*—with its associated internal node—called *companion*.

We call $\text{SSL}_\circlearrowleft$ derivations *pre-proofs*, since they do not necessarily derive *terminating* programs. To ensure termination, we require that pre-proofs satisfy an additional global property, defined in terms of traces of cardinality variables.

Definition 3.1 (Trace pairs)**.** Let $\mathcal{G}$ and $\mathcal{G}'$ be, respectively, the conclusion and a premise of an inference rule r, and let α and β be cardinality variables occurring universally in $\mathcal{G}$ and $\mathcal{G}'$, respectively. We say that (α, β) is a *trace pair* for $(\mathcal{G}, \mathcal{G}')$ when: either r is Call with substitution σ, $\mathcal{G}'$ is the left-hand premise, and $\alpha = \sigma(\beta)$; or $\vdash \phi \Rightarrow \beta \leq \alpha$ holds, where ϕ is the pure precondition of $\mathcal{G}$. If $\vdash \phi \Rightarrow \beta < \alpha$ also holds, we say that the trace pair is *progressing*.

A *path* in a pre-proof P is a sequence $\mathcal{G}_i$ $(i \geq 0)$ of nodes in P such that each node $\mathcal{G}_i$ is the parent of $\mathcal{G}_{i+1}$ in the infinite derivation corresponding to P.

Definition 3.2 (Traces)**.** A *trace* is an infinite sequence α_i $(i \geq 0)$ of cardinality variables. We say that a trace follows a path $\mathcal{G}_i$ $(i \geq 0)$ in a pre-proof P when (α_i, α_{i+1}) is a trace pair for $(\mathcal{G}_i, \mathcal{G}_{i+1})$ for each $i \geq 0$. When (α_i, α_{i+1}) is progressing, we say that the trace *progresses at* i. A trace is called infinitely progressing if it progresses at infinitely many points.

Definition 3.3 (Proofs)**.** A pre-proof P is said to satisfy the global trace condition when every infinite path in P is followed by an infinitely progressing trace.

The global trace condition is an ω-regular property, so it is decidable when a pre-proof is a proof, *cf.* [9, Prop. 7.4]. We write $\vdash \mathcal{G}$ when there is a proof deriving $\mathcal{G}$.

3.4 Soundness

$\text{SSL}_\circlearrowleft$ inherits the memory model and operational semantics from traditional SL [5]. In the interest of space we only state the soundness theorem here. Its proof and the rest of the metatheory are relegated to Appendix B.

Theorem 3.4 (Soundness)**.** *If* $\vdash \Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$*, then for any heap/stack pair* $\langle h, s\rangle$ *that satisfies* $\mathcal{P}$*, there exists a heap/stack pair* $\langle h', s'\rangle$ *that satisfies* $\mathcal{Q}$*, such that executing* c *from state* $\langle h, s\rangle$ *terminates in state* $\langle h', s'\rangle$*.*

4 Cyclic Program Synthesis, Pragmatically

We implemented cyclic program synthesis in a new synthesizer called Cypress [17]. Cypress takes as input a synthesis goal $\{\mathcal{P}\}\ f(\overline{x_i})\ \{\mathcal{Q}\}$ together with the definitions of all inductive predicates it mentions, and performs backtracking *proof search* for an $\text{SSL}_\circlearrowleft$ derivation of the judgment $\mathcal{P} \rightsquigarrow \mathcal{Q} \mid f(\overline{x_i})$. Once a proof has been constructed, *extracting* the synthesized program is straightforward: we simply consider each application of the Proc rule in isolation; each such application gives rise to a procedure, whose signature and body are the code emitted by the conclusion and the premise of Proc, respectively. For example, the two applications of Proc in Fig. 4 give rise to two procedures, `flatten` and `append`.

In the rest of this section we outline the mechanisms that make our proof search tractable. Since Cypress builds upon SuSLik—the original implementation of SSL—we only focus on the new mechanisms, which make Cypress more general and/or more efficient than its predecessor. Cypress also leverages existing proof search features of SuSLik, such as early failure rules, phased proof search, and branch abduction (which enables it to synthesize conditionals beyond those in predicate selectors); we refer the reader to [29] for a detailed account of these features.

Best-first search. One difference in the overall search algorithm is that Cypress uses memoizing *best-first search* (inspired by [19]), instead of SuSLik's naïve depth-first search. The search is guided by a cost function that assigns a cost to each heaplet in the goal's pre- and postcondition, with predicate instances growing more expensive as they get unfolded

$$\textsc{Unify}\ \frac{\Gamma;\ \{\phi; P * [\sigma_1]R\} \rightsquigarrow \{\psi \wedge \psi'; Q * [\sigma_1]R\} \mid c \qquad \psi' = \bigwedge_{v \in V} (\sigma_1(v) = \sigma_2(v)) \qquad \mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) = V}{\Gamma;\ \{\phi; P * [\sigma_1]R\} \rightsquigarrow \{\psi; Q * [\sigma_2]R\} \mid c}$$

$$\textsc{Solve-}\exists\ \frac{\Gamma;\ \{\phi; P\} \rightsquigarrow [\sigma]\{\psi; Q\} \mid c \qquad \vdash \phi \Rightarrow [\sigma]\psi \qquad \mathsf{dom}(\sigma) = \{\overline{\omega}\}; \mathsf{EV}(\Gamma) = \varnothing}{\Gamma \cup \exists\overline{\omega};\ \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c}$$

Figure 8. Two algorithmic rules for pure reasoning.

or go through a call. This cost function prevents the search from getting stuck in a branch that performs infinitely many unfoldings or calls, and encourages it to focus on smaller (and therefore, hopefully, easier to solve) goals.

4.1 Synthesizing Calls

To understand how Cypress applies the rules Call and CallSetup in practice, let us revisit the tree flattening program from Fig. 5. After the initial sequence of Read and Open applications, which generate lines 1–7, we find ourselves with a pre-heap $\mathtt{r} \mapsto \mathtt{x} * \mathtt{tree(xl)} * \mathtt{tree(xr)} * \ldots$ (where ellipsis stands for the node at x and its payload, and we omit cardinality and set parameters for brevity). To decide whether a call can be synthesized from this goal, Cypress considers all *candidate companion* goals, *i.e.* all ancestor goals separated from the current goal by at least one application of Open. Open is the only rule where cardinality traces can progress, hence any well-formed cycle in the derivation must include an application of Open. In our case, the top-level goal with the pre-heap $\mathtt{r} \mapsto \mathtt{x} * \mathtt{tree(x)}$ is such a candidate.

Call abduction oracle. Given a candidate companion, synthesizing a call involves guessing (1) the *substitution* σ of formals into actuals (*e.g.*, $[\mathtt{x} \mapsto \mathtt{xl}]$); (2) the *frame* R, *i.e.*, the part of the precondition untouched by the call (*e.g.*, $\mathtt{tree(xr)} * \ldots$); and (3) the *setup statement* c required to satisfy the companion's precondition (here $*\mathtt{r} = \mathtt{xl}$). Cypress finds all these three components at once, using a mechanism we dub *call abduction oracle*. This oracle is a separate synthesis problem that attempts to "bridge the gap" between the current goal's precondition and that of the companion. In our example, the oracle attempts to derive:

$$\{\mathtt{r} \mapsto \mathtt{x} * \mathtt{tree(xl)} * \mathtt{tree(xr)} * \ldots\} \rightsquigarrow \{r' \mapsto x' * \mathtt{tree}(x')\} \mid c$$

Note that all variables in the companion goal are replaced with fresh existentials. The call abduction oracle only uses the subset of $\mathrm{SSL}_{\heartsuit}$ rules triggered by the post-condition (*e.g.* Write and Alloc), since the rest of rules (*e.g.* Read, Open, Call) could already fire before the oracle was invoked. It also uses a modified version of the Emp rule, which allows the pre-heap to remain non-empty. Upon successful completion, the remaining pre-heap becomes the frame R, the code emitted during this derivation becomes the setup statement c, and the σ comprises all existential substitutions from this derivation.

$$\textsc{Unify}\ \frac{\textsc{Frame}\ \dfrac{\textsc{Solve-}\exists\ \dfrac{\textsc{Emp}\ \dfrac{}{\{\mathsf{emp}\} \rightsquigarrow \{s \cup \{a\} = \{a\} \cup s; \mathsf{emp}\}}}{\{\mathsf{emp}\} \rightsquigarrow \{s \cup \{a\} = \{a\} \cup w; \mathsf{emp}\}}}{\{\mathsf{sll(x,}\ s \cup \{a\})\} \rightsquigarrow \{s \cup \{a\} = \{a\} \cup w; \mathsf{sll(x,}\ s \cup \{a\})\}}}{\{\mathsf{sll(x,}\ s \cup \{a\})\} \rightsquigarrow \{\mathsf{sll(x,}\ \{a\} \cup w)\}}$$

Figure 9. An example derivation with rules from Fig. 8.

Termination checking. Whenever the call abduction oracle succeeds, Cypress adds appropriate applications of Call and CallSetup, inserts an application of Proc below the companion candidate (using all its program variables as formals and a fresh procedure name), and forms a backlink from the first premise of Call to the conclusion of Proc. Every time a backlink is formed, Cypress checks whether the pre-proof constructed so far satisfies the trace condition from Sec. 3.3. To this end, it builds a graph of the current pre-proof with edges labeled with all available trace pairs, according to Def. 3.1. It then invokes Cyclist [35], an off-the-shelf cyclic theorem prover, which uses an automata-theoretic algorithm to check whether every infinite path in the graph contains an infinitely progressing trace.

4.2 Pure Reasoning

Until now we have focused on synthesis rules, which directly emit code; the success of synthesis, however, also crucially relies on logical rules, which transform the goal into a form where synthesis rules can fire. Logical reasoning in $\mathrm{SSL}_{\heartsuit}$ is far from straightforward because it needs to support arbitrary SMT-decidable theories in its pure formulas.

To illustrate the challenges of pure reasoning, consider the following (simplified but representative) synthesis goal:

$$\forall \mathtt{x}, s, a. \exists w. \{\mathsf{sll(x}, s \cup \{a\})\} \rightsquigarrow \{\mathsf{sll(x}, \{a\} \cup w)\} \qquad (9)$$

The simplest—and hence most desirable—solution to this goal is `skip`, which informally can be obtained by simply framing away the sll heaplets; but formally, before Frame can apply, we need to transform the post-heap to be syntactically identical to the pre-heap, which requires: (a) exploiting commutativity of set union, and (b) instantiating the existential w with s. Fig. 8 shows two novel logical rules implemented in Cypress, which perform such transformations efficiently; Fig. 9 demonstrates how these rules work together to solve the synthesis goal (9) (we omit the emitted program `skip` and highlight new parts in each sub-goal for readability).

Unification modulo theories. The traditional approach to exploiting equational theories—such as commutativity of union in our example—is to eagerly normalize the goal after every proof step, by computing all implied equalities between its subterms [4]. For example, if a specification mentions both $s \cup \{a\}$ and $\{a\} \cup s$, normalization would replace one with the other. The normalization approach has two downsides: first, it is rather inefficient if you assume only blackbox access to the SMT solver; second, it does not help with the goal like (9), because here $s \cup \{a\} = \{a\} \cup$

w is not a logical necessity, but rather a possibility, which happens to lead to the shortest solution. To circumvent these limitations, Cypress implements a new, lazy approach to equational reasoning via the rule Unify in Fig. 8, which we dub *unification modulo theories*. Unify looks for a pair of heaplets $[\sigma_1]R$ and $[\sigma_2]R$ in the pre- and postcondition that only differ in pure subterms, and speculatively unifies them, adding equalities between mismatched subterms, $\sigma_1(v) = \sigma_2(v)$, as proofs obligation to the pure postcondition. In Fig. 9, Unify is used to unify the pre- and post-heap of the goal (9), producing the proof obligation $s \cup \{a\} = \{a\} \cup w$. Even in the absence of existentials, this approach is more efficient than eager normalization, because the SMT solver only needs to check equalities between terms that appear in matching positions inside a heaplet.

Pure synthesis. The other major challenge is to find appropriate instantiations for existential variables. To this end, Cypress includes the rule Solve-∃ in Fig. 8, which picks a substitution σ from existentials to universals that validates the pure specification. To find such a substitution, Cypress needs to solve the constraint $\exists\sigma.\forall\overline{x}.\phi \Rightarrow [\sigma]\psi$, which is itself a synthesis problem in the pure subset of our logic. Although such pure synthesis is generally a challenging task, it has been the subject of much prior work [2, 21, 32]. Cypress outsources pure synthesis queries to the CVC4 synthesizer [33].

5 Evaluation

We evaluated Cypress empirically along three axes: (1) *generality:* its ability to synthesize programs with complex recursion; (2) *efficiency:* the time it takes to synthesize programs; and (3) *utility:* the size of the input specification compared to the size of the generated programs, and the quality of generated programs.

5.1 Benchmarks and Setup

For our empirical evaluation we have assembled a suite of 46 synthesis benchmarks for pointer-manipulating programs. Each benchmark is defined by a top-level synthesis goal expressed as separation-logic specification (and optionally, specifications of library functions the code is allowed to invoke). We collected these benchmarks from three sources:

1. State of the art in synthesis *with recursive auxiliaries* [14]. We include eight out of their nine benchmarks. The remaining one, `merge_sort`, has a specification identical to another one (`sort`), but a different program template, which forces their tool to synthesize merge sort instead of insertion sort; Cypress does not use program templates, so the difference between these two benchmarks does not make sense in our setting. The tool [14] synthesizes functional programs from refinement types, so we manually translated these benchmarks into Separation Logic.
2. State of the art in synthesis of *heap-manipulating programs*: SuSLik [29] and ImpSynt [31]. We include all 22 benchmarks from SuSLik, and the 11 recursive benchmarks from ImpSynt (these two sets overlap); the excluded five ImpSynt benchmarks are iterative versions of their recursive benchmarks, and similarly require sketches.
3. We supplement this set with 14 *new benchmarks*, half of which, to the best of our knowledge, cannot be solved by any existing synthesis tools. These benchmarks involve the interplay between auxiliary functions and heap manipulation (*e.g.*, an in-place tree flattening) or operate on cyclic or mutually recursive structures (*e.g.*, rose trees).

Out of the 46 benchmarks, 19 exercise *complex recursion*, *i.e.*, they either require a non-trivial termination measure or a recursive auxiliary procedure (not given as a library function). We refer to this benchmark set as `complex` and use it as a primary focus of our empirical evaluation. All complex benchmarks are by construction out of reach for SuSLik. The remaining 27 benchmarks only exercise *simple (structural) recursion*. We refer to this benchmark set as `simple`; although not the focus of our evaluation, we use these benchmarks to demonstrate the versatility of Cypress.

Experiment setup. For our main experiment, we ran Cypress on the `complex` benchmark set, and measured synthesis time and size of the generated code. For our second experiment, we ran Cypress on the `simple` benchmark suite and compared the synthesis times with SuSLik. The purpose of this experiment is to confirm that searching a larger program space does not lead to significant degradation in performance. All experiments were conducted on a commodity laptop (2.7 GHz Intel Core i7 Lenovo Thinkpad with 16GB RAM), and Cypress was run as a single-threaded process. Timeout for all experiments was set to one hour.

5.2 Results

Experiment results on the `complex` benchmarks are shown in Tab. 1, and on the `simple` benchmarks in Tab. 2. All specifications and generated code can be found in Appendix C.

5.2.1 Generality. The results in Tab. 1 confirm that Cypress is able to synthesize a variety of functions with complex recursion. Benchmarks 10–13 were discussed in Sec. 2. Out of all benchmarks in Tab. 1, only 10 and 17 can be solved without auxiliaries, but they require a complex termination metric; interestingly, for 17—sorted list merge—Cypress generates an auxiliary anyway (see discussion in Sec. 5.3) All the other benchmarks require one or even two auxiliaries; for example, benchmark 14 flattens a rose tree into a list, and needs one auxiliary for flattening the list of children and another one for appending two lists (a flattened tree and a flattened list of children).

Finally, four of the synthesized programs feature *mutual recursion*. Two of them operate on rose trees, where mutual recursion is expected, while the other two—flattening a tree

Table 1. Benchmarks with complex recursion; *all of these* are out of reach for SuSLik. We report the number of *Proc*edures generated, total number *Stmt* of statement in those procedures, the ratio *Code/Spec* of code to specification (in AST nodes), and the synthesis *Time* (in seconds).

Group	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Singly Linked List	1	deallocate two	2	9	6.2x	0.3
	2	append three	2	14	2.3x	1.2
	3	non-destructive append	2	21	3.0x	5.2
	4	union	2	24	5.9x	9.6
	5	intersection[1]	3	33	7.3x	95.6
	6	difference[1]	2	22	5.5x	8.1
	7	deduplicate[1]	2	23	7.8x	6.2
List of Lists	8	deallocate	2	11	10.7x	0.3
	9	flatten[1]	2	19	4.8x	0.8
Binary Tree	10	deallocate two	1	16	11.8x	0.3
	11	flatten	2	24	7.4x	1.5
	12	flatten to dll in place	2†	15	9.6x	2.7
Rose Tree	13	deallocate	2†	9	12.0x	0.3
	14	flatten	3†	25	8.0x	12.6
Sorted list	15	reverse[1]	2	11	3.3x	1.1
	16	sort[1]	2	12	3.6x	1.9
	17	merge[2]	2	23	2.2x	33.6
BST	18	from list[1]	2	27	5.0x	11.5
	19	to sorted list[1]	2†	35	6.4x	10.2

[1] From [14] [2] From [31] † Mutually-recursive

into doubly-linked list (12) and flattening a binary search tree (BST) into a sorted list (19)—came as a surprise.

Comparison with other tools. Cypress was able to synthesize *all eight* selected benchmarks from [14]. Unlike their tool, Cypress does not use function templates as hints, and also targets pointer-manipulating programs instead of functional programs, which is arguably a harder problem. To the best of our knowledge, Cypress generates code with similar recursive structure to their tool: for example, list reversal, deduplication, and sorting all have the structure of two nested right-folds (and hence quadratic complexity); one exception is list intersection, discussed below, where Cypress generates an overly complex solution.

Cypress can also handle *all 22* original SuSLik benchmarks (which are all part of the `simple` set). Finally, Cypress was also able to solve *all 11* recursive benchmarks from ImpSynt [31] (one of which is in the `complex` set and the rest are in the `simple` set). Unlike ImpSynt, we do not require the user to provide program sketches, and additionally, our synthesis times on a commodity laptop are at least an order of magnitude faster than the run times reported in [31] on a server with 10 cores and 96GB of RAM.

On the other hand, SuSLik cannot handle any of the benchmarks in the `complex` set—because of complex recursion—and also fails on five benchmarks in the `simple` set. All five failures are due to restrictions on predicate unfolding, which SuSLik had to impose due to its ad-hoc termination checking mechanism, which in Cypress has been replaced with the more powerful cardinality-based mechanism from Sec. 3.3.

Table 2. Comparison with SuSLik on benchmarks with simple recursion. We report the number of statements *Stmt* and synthesis *Time* in seconds for both Cypress and SuSLik. Each benchmark generates a single procedure.

Group	*Id*	*Description*	*Stmt*	*Code/Spec*	*Time* Cypress	*Time* SuSLik
Integers	20	swap two[3]	4	1.0x	0.2	< 0.1
	21	min of two[3,4]	3	1.1x	1.5	0.4
Singly Linked List	22	length[2,3]	6	1.2x	1.1	1.1
	23	max[2,3]	7	1.9x	0.7	0.7
	24	min[2,3]	7	1.9x	0.6	0.7
	25	singleton[3,4]	4	0.9x	0.3	< 0.1
	26	dispose[3]	4	5.5x	0.2	< 0.1
	27	initialize[3]	4	1.6x	0.6	0.1
	28	copy[3,5]	11	2.7x	0.8	0.3
	29	append[3,5]	6	1.1x	0.5	0.4
	30	delete[3,5]	12	2.6x	1.6	0.4
Sorted list	31	prepend[2,3]	4	0.5x	0.3	0.2
	32	insert[2,3]	25	2.6x	4.4	5.2
	33	insertion sort[2,3]	7	1.0x	1.2	1.4
Tree	34	size[3]	9	2.5x	0.7	0.3
	35	dispose[3]	6	8.0x	0.2	< 0.1
	36	copy[3]	16	3.8x	2.8	0.7
	37	flatten w/append[3]	19	5.4x	0.4	0.7
	38	flatten w/acc[3]	12	2.1x	0.7	0.7
BST	39	insert[2,3]	19	1.9x	9.8	36.9
	40	rotate left[2,3]	5	0.2x	6.2	23.9
	41	rotate right[2,3]	5	0.2x	4.8	9.1
	42	delete root[2]	29	1.7x	1304.3	-
Doubly Linked List	43	copy	22	4.3x	7.3	-
	44	append[5]	10	1.6x	2.3	-
	45	delete[5]	19	3.7x	4.7	-
	46	single to double	21	5.5x	1.3	-

[2]From [31] [3]From SuSLik [29] [4]From [22] [5]From [30]

Although in principle ImpSynt is capable of solving `complex` benchmarks that require recursive auxiliaries using nested loops instead, none of the results reported in [31] contain nested loops, and in any case, ImpSynt relies on program sketches. To our knowledge, no existing synthesizer (for either functional or heap-manipulating programs) is able to generate mutually-recursive programs.

5.2.2 Efficiency. Our experiments show that Cypress is efficient in synthesizing a variety of programs: all 19 `complex` benchmarks were synthesized within two minutes, and all but two of them take less than fifteen seconds. Our comparison of Cypress with SuSLik on the `simple` benchmarks demonstrates that despite searching a larger space of programs, Cypress remains efficient. It is slightly slower on easy benchmarks: of the 18 benchmarks that take less than five seconds for SuSLik, the average time is 0.5 seconds for SuSLik compared to 0.8 seconds for Cypress; the remaining four hard benchmarks is where Cypress's new search strategy pays off: the average time for those four benchmarks is 18.8 seconds for SuSLik and only 6.3 seconds for Cypress.

5.2.3 Utility. To quantify synthesis utility, we measured the ratio between the size of synthesized code and the input specification (taken as AST sizes). For all benchmarks from

the `complex` set, the generated code is larger than the specification by at least 2.2x (sorted list merge) and at most 12x (deallocate rose tree). This confirms our intuition that for simple, boilerplate data structure manipulations, like deallocation and copying, synthesis offers a good trade-off, since their specifications are very simple, while the code can be quite tricky. We only include pre- and postconditions in the specification size and omit predicate definitions, since those are reused between benchmarks. Anecdotally, the new benchmarks created for this paper were quick and easy to write: most predicate definitions were either reused from SuSLik or were standard SL predicates. In both cases, the pre- and postconditions were very straightforward. Note that the code-to-spec ratios for the `simple` benchmarks are lower (between 0.2x and 8.0x), which serves as evidence that deductive synthesis for heap-manipulating programs delivers larger pay-off for complex traversals.

5.3 Notable Benchmarks

Merge. One benchmark where Cypress surprised us is merging two sorted lists (benchmark 17). We initially thought that this problem required an extension to the tool, because the traditional recursive implementation—without auxiliaries—has to unfold both input lists in order to compare their heads, but then fold one of the lists back again to pass it to the recursive call; this "folding back" is something that Cypress does not explicitly support. To our surprise, Cypress was instead able to solve this problem by inventing an auxiliary that merges two lists *one of which is non-empty*. This implementation is even slightly more efficient, since it eliminates a redundant emptiness check for one of the lists.

Sort. An important difference between the functional setting of [14] and our setting is that SL specifications give the the user more fine-grained control over the relationship between input and output data structures in memory. For sorting (benchmark 16), we took the liberty of requiring the sort to be *in-place*, by using the following specification:

$$\{\mathsf{sll}(\mathsf{x}, n, l, h)\} \rightsquigarrow \{\mathsf{srtl}(\mathsf{x}, n, l, h)\}$$

Here sll is a list rooted at x with length n and lower and upper bounds on the elements l and h, and srtl is a sorted list with the same parameters (notably, rooted at the same address x). Such in-place sorting is not expressible using refinement types from [14]. Given this specification—which contains no insight as to which algorithm to use to sort the list—Cypress synthesizes a rather peculiar version of in-place insertion sort: while traditionally insertion sort on linked lists performs insertion by switching `next` pointers, Cypress chose instead to do it by swapping elements in out-of-order list cells (similarly to a typical insertion sort on an array).

Intersection. Perhaps the most curious case is benchmark 5, which computes the intersection of two sets represented as linked lists with unique elements (denoted by the inductive predicate ul). The simplest specification for intersection is:

$$\{r \mapsto \mathsf{x} * \mathsf{ul}(\mathsf{x}, s_1) * \mathsf{ul}(\mathsf{y}, s_2)\} \rightsquigarrow \{r \mapsto z * \mathsf{ul}(z, s_1 \cap s_2)\}$$

With this specification, however, Cypress fails to find a solution due to a limitation on the class of auxiliaries it can generate. Unlike the original functional setting, there is no simple, fold-like solution for this specification, because it is *destructive*: once it computes the intersection z' of y and the tail of x , the list y is lost, and we cannot decide whether the head of x should be added to z'. A more sophisticated solution would require an auxiliary that tests membership, which is beyond the capabilities of cyclic synthesis.

To circumvent this issue, we added $*\mathsf{ul}(\mathsf{y}, s_2)$ to the postcondition to preserve the input list y. Even with this change, however, Cypress fails to generate the simple solution with a single auxiliary (one that searches for the head of x in y, and if found, prepends it to z'), because doing so requires *weakening* of the auxiliary's pure precondition, which cyclic synthesis currently does not support. Surprisingly, Cypress finds another solution, which uses a second auxiliary to *append* the head of x to z' instead of *prepending* it. This solution is so unusual that we worried we found a soundness issue, until we checked it with an external program verifier: the solution is indeed correct (albeit inefficient), and both of the inferred auxiliary specifications are inductive.

5.4 Limitations

When does Cypress fail? The main limitation of $\mathrm{SSL}_{\circlearrowleft}$ is that it can only derive a certain class of auxiliary functions. Intuitively, it can only extract auxiliary specifications from the goals in the main derivation, and cannot invent them "out of thin air", so, for example, the auxiliary cannot take extra parameters or return a new data structure, which the main specification does not mention. For this reason, Cypress cannot synthesize linear-time implementations of list reversal or tree flattening, which would require conjecturing an extra accumulator parameter. As explained in Sec. 5.3, $\mathrm{SSL}_{\circlearrowleft}$ also lacks support for generalizing the pure part of the auxiliary specification, which prevents it from deriving simpler version of set intersection. It is difficult to precisely characterize the class of synthesis problems that cyclic synthesis can and cannot solve, because, as illustrated above, it often finds alternative solutions that circumvent the limitations. Such characterization is an interesting direction for future work; another important and challenging direction is extending $\mathrm{SSL}_{\circlearrowleft}$ with additional rules to support a wider class of programs, while continuing to strike a careful balance between expressiveness and tractability of proof search.

Quality of solutions. A Cypress solution might have suboptimal performance *even if* a more efficient program is derivable via $\mathrm{SSL}_{\circlearrowleft}$. The core issue is that $\mathrm{SSL}_{\circlearrowleft}$ has no means of analyzing program efficiency, and hence no reason to prefer a more efficient solution. A promising approach for producing efficient heap-manipulating would be to adopt a

flavor of Separation Logic with *time credits* [16], combining it with techniques for resource-aware synthesis for pure functional programs [20].

We also noticed that the division of code between the procedures in Cypress solutions is sometimes unnatural: for example, the tree flattening program in Fig. 5 deallocates tree nodes inside the auxiliary append function instead of the main function; in terms of abstraction boundaries, it would be better if append were only concerned with appending lists and had no knowledge of tree nodes. The division of code depends on the order in which Cypress tries applying synthesis rules (in this case, the relative order of Call and Free); currently the order is optimized for efficiency of proof search rather than for optimal abstraction boundaries. We leave the investigation of improving program quality to future work.

Loop support. $\text{SSL}_{\circlearrowleft}$ currently has no support for loops, which are often a more natural and efficient alternative to recursion in imperative programs. There are several existing techniques for deductive *verification* of loops using cyclic proofs [7, 43]; hence we believe the general cyclic synthesis technique could be extended to also handle them. To keep proof search tractable, however, we might not want to support both recursion and loops in our target language; a better idea might be to synthesize *tail-recursive* programs and then translate them into loops using standard techniques. The challenge, however, is that most tail-recursive programs require invented accumulator parameters (equivalently: most loops require temporary variables). Hence adding support for loops goes hand-in-hand with the extending the logic to support a wider class of auxiliaries.

6 Related Work

Deductive program synthesis. Our work on $\text{SSL}_{\circlearrowleft}$ extends a line of research on synthesizing programs from logical specifications [22, 31, 38, 40, 44], and, in particular, on *deductive synthesis* [13, 19, 24, 28], which implements a search in the space of *proofs* of program correctness (rather than in the space of programs). Our work is the first to combine deductive synthesis with cyclic proof to generate provably correct and terminating programs with complex recursion patterns. Our technical contributions build primarily on the work by Polikarpova and Sergey [29], extending their logic SSL and the tool SuSLik with ideas from cyclic proofs. The best-first search algorithm of Cypress is inspired by the Leon system for deductive program verification, synthesis, and repair [18, 19], but tailored to Separation Logic.

Our work advances the state of the art for deductive synthesis tools with regard to establishing termination of synthesized programs. Specifically, it enables automated derivation of a termination argument along with the program being synthesized without being subject to restrictions of previous approaches: (1) requiring the user to provide an explicit termination measure [28]; (2) restricting the recursion to syntactic structural one [27, 29]; and (3) determining the termination measure by instantiating one of the pre-defined recursion/looping schemas [19, 31].

Synthesis of auxiliary functions. Eguchi et al. [14] implement a technique for automatically synthesizing implementations of *pure functional programs* with recursive helper functions from refinement types [45]. Their approach infers specifications for recursive helper functions by trying a number of *predefined templates* for the "main" function, which exercise different flavours of recursion (structural folds or divide-and-conquer) on the input data structures. Our work is complementary in that it considers imperative heap-manipulating programs, and also takes a fundamentally different *proof-driven* (rather than *template-driven*) approach to identify recursion patterns. By doing so, our technique removes the need to conjecture recursion principles upfront, yet it is capable of discovering many of those automatically, including non-trivial ones, such as traversals of a rose tree.

Several other techniques cannot synthesize recursive auxiliaries directly, but can synthesize nested folds [15] or nested loops [31, 38, 40]. Out of these techniques, λ^2 [15] is restricted to a set of predefined templates, and also provides no correctness guarantees beyond a finite set of input-output examples. Sketching-based approaches [31, 38, 40] are very flexible, but require the programmer to provide an extensive program sketch in order to make synthesis with nested loops tractable.

Cyclic proofs. The techniques and metatheory of cyclic proofs originates in the logic and proof theory community. Most related to our current work is the application of cyclic proof to reasoning about program correctness [7, 35, 43], as well as to proving pure entailments of Separation Logic with inductively defined predicates [6, 41, 42]. In particular, our use of cardinalities with Separation Logic coincides with the approach used by Rowe and Brotherston [35].

7 Conclusion

We have demonstrated that cyclic proofs, already known as an automation-friendly method for reasoning about program termination, can also be an effective tool for automatically synthesizing programs with recursive auxiliary functions. By investigating this synergy between verification and synthesis, we have once again witnessed that the ideas developed by the logic community for scalable reasoning about heap-manipulating procedures can be instrumental in practical synthesis of correct-by-construction programs.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Kwangkeun Yi, for their valuable feedback on earlier drafts of this paper. This work was supported by the National Science Foundation under Grant No. 1911149, by Singapore MoE Tier 1 Grant No. IG18-SG102, by the Israeli Science Foundation (ISF) Grants No. 243/19 and 2740/19, and by the United States-Israel Binational Science Foundation (BSF) Grant No. 2018675.

References

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

[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. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.

[3] David Baelde, Amina Doumane, and Alexis Saurin. 2016. Infinitary Proof Theory: the Multiplicative Additive Case. In *CSL (LIPIcs, Vol. 62)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:17.

[4] Josh Berdine and Nikolaj Bjørner. 2014. Computing All Implied Equalities via SMT-Based Partition Refinement. In *IJCAR*. 168–183.

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

[6] James Brotherston. 2007. Formalised Inductive Reasoning in the Logic of Bunched Implications. In *SAS (LNCS, Vol. 4634)*. Springer, 87–103.

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

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

[9] James Brotherston and Alex Simpson. 2011. Sequent Calculi for Induction and Infinite Descent. *J. Log. Comput.* 21, 6 (2011), 1177–1216.

[10] Coq Development Team. 2020. *The Coq Proof Assistant Reference Manual - Version 8.11*. Available at http://coq.inria.fr/.

[11] Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. 2020. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP (LNCS, Vol. 12075)*. Springer, 141–168.

[12] Anupam Das and Damien Pous. 2018. Non-Wellfounded Proof Theory For (Kleene+Action)(Algebras+Lattices). In *CSL (LIPIcs, Vol. 119)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:18.

[13] 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.

[14] Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *APLAS (LNCS, Vol. 11275)*. Springer, 223–241.

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

[16] Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *ESOP (LNCS, Vol. 10801)*. Springer, 533–560.

[17] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. *Cypress (PLDI 2021 Artifact): Code and Benchmarks*. https://doi.org/10.5281/zenodo.4639933

[18] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *CAV (LNCS, Vol. 9207)*. Springer, 217–233.

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

[20] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *PLDI*. ACM, 253–268.

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

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

[23] Grant Malcolm. 1990. Data Structures and Program Transformation. *Sci. Comput. Program.* 14, 2-3 (1990), 255–279.

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

[25] Damian Niwiński and Igor Walukiewicz. 1996. Games for the μ-Calculus. *Theor. Comput. Sci.* 163, 1&2 (1996), 99–116.

[26] 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.

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

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

[29] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30.

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

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

[32] 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.

[33] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2019. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.* 55, 2 (2019), 73–102.

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

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

[36] Luigi Santocanale. 2002. A Calculus of Circular Proofs and Its Categorical Semantics. In *FoSSaCS (LNCS, Vol. 2303)*. Springer, 357–371.

[37] Alex Simpson. 2017. Cyclic Arithmetic Is Equivalent to Peano Arithmetic. In *FoSSaCS (LNCS, Vol. 10203)*. Springer, 283–300.

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

[39] Christoph Sprenger and Mads Dam. 2003. On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus. In *FoSSaCS (LNCS, Vol. 2620)*. Springer, 425–440.

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

[41] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. In *FM (LNCS, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer, 659–676.

[42] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2018. Automated lemma synthesis in symbolic-heap separation logic. *Proc. ACM Program. Lang.* 2, POPL (2018), 9:1–9:29.

[43] Gadi Tellez and James Brotherston. 2020. Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof. *J. Autom. Reasoning* 64, 3 (2020), 555–578.

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

[45] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282.

```
1  void treefree2 (x, y) {
2    if (x = 0) {
3      if (y ≠ 0) {
4        let ly = *(y + 1);
5        let ry = *(y + 2);
6        free(y);
7        treefree2(ly, ry);
8      }
9    } else {
10     let lx = *(x + 1);
11     let rx = *(x + 2);
12     free(x);
13     treefree2(lx, rx);
14     if (y ≠ 0) {
15       let ly = *(y + 1);
16       let ry = *(y + 2);
17       free(y);
18       treefree2(ly, ry);
19   }}}
```

Figure 10. Synthesis of double tree deallocation via Cypress.

A Further Motivating Examples

A.1 Deallocating Two Trees

Recall that SSL only allowed the synthesized program to "recurse on" a single predicate from the precondition. This restriction made it impossible to synthesize a simple extension of our tree deallocation example: a program that deallocates two trees:

$$\{\mathsf{tree}(x) * \mathsf{tree}(y)\}\ \texttt{treefree2(x, y)}\ \{\mathsf{emp}\} \tag{10}$$

Of course, one way to implement `treefree2` is to invoke `treefree` twice as an auxiliary. Curiously, this problem can also be solved without auxiliaries, *and yet* vanilla SSL is unable to derive this solution, because the required termination argument goes beyond its structural recursion restriction.

The code synthesized via $\mathrm{SSL}_{\circlearrowleft}$ from the specification (10) is given in Fig. 10. The program features three recursive calls, two of which (lines 7 and 18) deallocate sub-trees of `y`, and the other one (line 13) deallocates sub-trees of `x`. Hence, we cannot pick a single predicate, either tree(x) or tree(x) that decreases at all three call sites: if we pick the former, we cannot show that `ly` is smaller than `x` in lines 7 and 18, and if we pick the latter, we cannot show that `rx` is smaller than `y` in line 13. Instead, a suitable termination measure for this program is the *maximum* of the two input tree sises, which can easily be seen to strictly decrease at each recursive call. The benefit of the cyclic approach to termination checking adopted by $\mathrm{SSL}_{\circlearrowleft}$, is that we need not explicitly infer such complex termination measures. Instead, the trace-based well-formedness condition outlined in Sec. 2.2 subsumes "max"-based measures, as well as lexicographic measures, and combinations of the two.

The derivation of `treefree2` with heap cardinalities and backlinks between important proof nodes is provided in Fig. 11. To verify the global trace condition, we again consider an arbitrary infinite path in the proof, which consists of an infinite sequence of the cycles (1), (2), and (3). Each cycle starts at either $\boxed{\alpha}$ (cycle (2)) or $\boxed{\beta}$ (cycles (1) and (3)), as appropriate. This time, notice that each path to a bud has a choice of which cardinality variable to end up at. To produce a trace, we must choose based on which cycle will be traversed *next*. Crucially, though, along each such segment the trace strictly decreases, as witnessed by the inequalities in the proof.

A.2 Deallocating a Rose Tree

A rose tree can defined in SL via a pair of mutually-recursive predicates, rtree and children are defined as follows:

$$\begin{aligned}
\mathsf{rtree}^{\alpha}(x) \triangleq\ & x = 0 \wedge \{\alpha = 0; \mathsf{emp}\} \\
& \mid\ x \neq 0 \wedge \left\{ \begin{array}{l} \beta < \alpha; [x, 2] * x \mapsto v * \langle x, 1\rangle \mapsto c * \\ \mathsf{children}^{\beta}(c) \end{array} \right\} \\
\mathsf{children}^{\beta}(x) \triangleq\ & x = 0 \wedge \{\beta = 0; \mathsf{emp}\} \\
& \mid\ x \neq 0 \wedge \left\{ \begin{array}{l} \alpha' < \beta \wedge \beta' < \beta; [x, 2] * x \mapsto r * \\ \mathsf{rtree}^{\alpha'}(r) * \langle x, 1\rangle \mapsto c * \mathsf{children}^{\beta'}(c) \end{array} \right\}
\end{aligned} \tag{11}$$

The code of the function `rtfree(x)`, synthesized from the goal $\{\mathtt{x}\}; \{\mathsf{rtree}^{\alpha}(\mathtt{x})\} \rightsquigarrow \{\mathsf{emp}\}$, and its recursive auxiliary helper `rtfree_helper(x)`, are given in Fig. 13, and Fig. 12 shows the key features of its $\mathrm{SSL}_{\circlearrowleft}$ derivation. The Proc rule is applied twice. Firstly to the root, conjecturing the `rtfree(x)` function, and then secondly to the premise of the Open rule instance explicitly shown, where it conjectures the auxiliary `rtfree_helper(x)` function. The Open rule unfolds the occurrence of the predicate rtree(x), which results in the generation of the **`if-else`** statement at line 2, as well as the constraint $\boxed{\beta < \alpha}$. The derivation then proceeds to apply the rule Open once again, this time to the occurrence of children(c), resulting in the **`if-else`** statement at line 10. At this point, the goal's precondition features two heaplets, constrained by $\mathsf{rtree}^{\alpha'}(\mathtt{r})$ and by $\mathsf{children}^{\beta'}(c')$, so that $\boxed{\alpha'} < \boxed{\beta}$ and $\boxed{\beta'} < \boxed{\beta}$. The heaplet $\mathsf{rtree}^{\alpha'}(\mathtt{r})$ triggers an application of Call, which links back, via (1), to the top-level conclusion of Proc, and generating a recursive call to the main function `rtfree(x)`. Finally, the symbolic heap $\mathsf{children}^{\beta'}(c')$, together with some other heaplets, is then used by another application of Call, whose premise allows for a backlink, via (2), to the conclusion of the inner application of Proc (for `rtfree_helper(x)`), determining the body of the auxiliary helper with the following specification:

$$\left\{[\mathtt{x}, 2] * \mathtt{x} \mapsto v * \langle \mathtt{x}, 1\rangle \mapsto c * \mathsf{children}^{\beta}(c)\right\}\ \texttt{rtfree\_helper(x)}\ \{\mathsf{emp}\} \tag{12}$$

Extracting the two mutually-recursive procedures form the derivation in Fig. 12 is straightforward and is performed locally, just by considering each application of the Proc rule in isolation. The top-level application generates a procedure `rtfree(x)` with the body emitted by the rule's premise; importantly, this body contains a *call* to `rtfree_helper` but not *its body*, which is abstracted away by the second Proc rule. The second application of Proc generates a procedure `rtfree_helper(x)` in a similar manner. Termination of both mutually-recursive procedures follows from tracing ($\boxed{\alpha}$ to) $\boxed{\beta}$ to either $\boxed{\alpha'}$ or $\boxed{\beta'}$, depending on which of cycles (1) and (2) is being traversed. Strict decreases $\boxed{\alpha'} < \boxed{\beta}$ and $\boxed{\beta'} < \boxed{\beta}$ along each cycle are guaranteed by constraints in the proof.

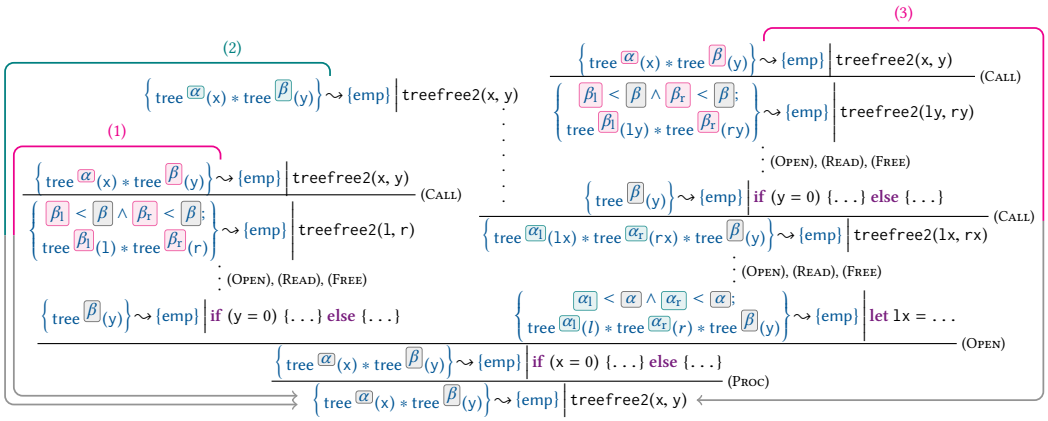


Figure 11. Selected parts of the `treefree2` derivation with backlinks and cardinality constraints.

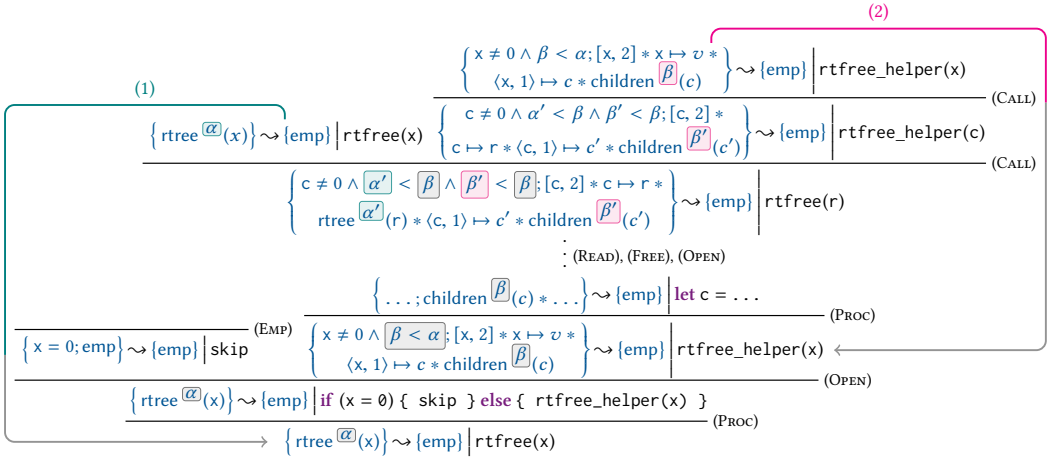


Figure 12. A derivation of the `rtfree` with backlinks and cardinalities.

$$\textsc{NullNotLVal}\ \frac{\Gamma;\ \{\phi \wedge x \neq 0; P\} \rightsquigarrow Q \mid c}{\Gamma;\ \{\phi; (x + \_ \mapsto \_) * \_\} \rightsquigarrow Q \mid c} \qquad \textsc{StarPartial}\ \frac{\Gamma;\ \{\phi \wedge x \neq y; P\} \rightsquigarrow Q \mid c}{\Gamma;\ \{\phi; (x + \_ \mapsto \_) * (y + \_ \mapsto \_) * \_\} \rightsquigarrow Q \mid c}$$

$$\textsc{EqNorm}\ \frac{\vdash [\chi_1/\nu]\phi \Rightarrow \chi_1 = \chi_2 \qquad \Gamma;\ [\chi_2/\nu]\{\phi; P\} \rightsquigarrow [\chi_2/\nu]Q \mid c}{\Gamma;\ [\chi_1/\nu]\{\phi; P\} \rightsquigarrow [\chi_1/\nu]Q \mid c}$$

$$\exists\textsc{-Elim}\ \frac{\Gamma;\ \mathcal{P} \rightsquigarrow [e/\omega]Q \mid c}{\Gamma \cup \exists\omega;\ \mathcal{P} \rightsquigarrow Q \mid c} \qquad \textsc{Frame}\ \frac{\Gamma;\ \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c}{\Gamma;\ \{\phi; P * R\} \rightsquigarrow \{\psi; Q * R\} \mid c}$$

Figure 14. Logical rules (rules that do not emit code).

```
1   void rtfree (loc x) {
2     if (x = 0) {
3     } else {
4       rtfree_helper(x);
5     }}
6
7   void rtfree_helper (loc x) {
8     let c = *(x + 1);
9     free(x);
10    if (c = 0) {
11    } else {
12      let r = *c;
13      rtfree(r);
14      rtfree_helper(c);
15    }}
```

Figure 13. `rtfree` and its helper.

B Rules and Metatheory

B.1 Logical Rules

Logical rules of SSL$_\circlearrowleft$ are shown in Fig. 14. The rule ∃-Elim eliminates an existential variable, replacing it with an arbitrary term; note that this is always sound, but choosing a wrong term will eventually result in a stuck goal. The rule EqNorm normalizes the synthesis goal based on equalities implied by the pure precondition: *i.e.* it replaces a term χ_1 with χ_2 if the two are semantically equivalent in the underlying theory. Both rules are presented here in their declarative, highly non-deterministic form; in Sec. 4 we discuss how Cypress implements them efficiently.

B.2 Soundness

Programs manipulate values from a set Val, which contains a (strict) subset Loc ⊂ Val of memory addresses. Program statements c are executed in the context of a *stack* s and a heap h, which are (partial) mappings s : PV ⇀ Val and h : Loc ⇀ Val, respectively. When evaluating expressions,

variables occurring in them must be in dom (s). Addresses being dereferenced must be in dom (h). Any other accesses, including dereferences of the special *null* location $0 \in \mathsf{Loc}$, result in an error state. The execution of a program statement produces a new stack and heap: dynamic memory allocation and de-allocation change dom (h), and `let`-bindings can change dom (s). Execution of procedure calls is straightforward, see for example [35]. The operational semantics of SL is usually presented in a small-step style. However, here we will write the big-step style statement $(c, \langle h, s\rangle) \Downarrow \langle h', s'\rangle$ to mean that, when executed in state $\langle h, s\rangle$, the program statement c terminates in state $\langle h', s'\rangle$.

The interpretation of assertions, over models $\langle h, s\rangle$, is also standard. In particular, emp denotes models in which dom $(h) = \emptyset$, $\langle e, \iota\rangle \mapsto e'$ denotes models in which $s(e) + \iota \in$ dom (h) with $h(s(e) + \iota) = s(e')$, and $P * Q$ denotes models in which $h = h_1 \cup h_2$ for heaps h_1 and h_2 with disjoint domains, and $\langle h_1, s\rangle$ and $\langle h_2, s\rangle$ are models of P and Q, respectively. The assertion $[e, \iota]$ tracks block allocation of memory; its semantics are given in [11]. We write $\langle h, s\rangle \models \phi; P$ to mean that the state $\langle h, s\rangle$ is a model of the spatial formula P and that s satisfies the pure assertion ϕ.

Inductive heap predicates are interpreted using a least fixed point semantics, *i.e.*, as the least set of models satisfying all of their clauses. The definition is via a standard construction, and we refer the reader to [7, 35] for details. In particular, cardinality variables are interpreted by the (well-founded) approximations of the fixed point semantics. Abstractly, this can be seen as coinciding with the sizes of the satisfying models heaps'.

The soundness of $\mathrm{SSL}_{\circlearrowleft}$ relies on the local soundness of the inference rules, taking into account the role of cardinality variables in traces.

Proposition B.1 (Trace-aware Local Soundness)**.** *Suppose* $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$ *is the conclusion of an inference rule, and let* $\langle h, s\rangle$ *be a model such that* $\langle h, s\rangle \models \mathcal{P}$ *and* $(c, \langle h, s\rangle) \Downarrow \langle h'', s''\rangle$ *implies* $\langle h'', s''\rangle \not\models \mathcal{Q}$ *for all* $\langle h'', s''\rangle$*. Then there is a premise* $\Gamma'; \mathcal{P}' \rightsquigarrow \mathcal{Q}' \mid c'$ *and a model* $\langle h', s'\rangle$ *such that* $\langle h', s'\rangle \models \mathcal{P}'$ *and* $(c', \langle h', s'\rangle) \Downarrow \langle h'', s''\rangle$ *implies* $\langle h'', s''\rangle \not\models \mathcal{Q}'$ *for all* $\langle h'', s''\rangle$*. Moreover, if* (α, β) *is a trace pair for this conclusion-premise pair then* $s(\beta) \leq s(\alpha)$ *(and* $s(\beta) < s(\alpha)$ *if it is progressing).*

Soundness then follows from the global trace condition, which ensures that any infinite execution of a derived program would entail an infinitely descreasing chain of cardinalities.

Theorem B.2 (Soundness)**.** *If* $\vdash \Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$ *and* $\langle h, s\rangle \models \mathcal{P}$*, then there is some* $\langle h', s'\rangle$ *such that* $(c, \langle h, s\rangle) \Downarrow \langle h', s'\rangle$ *and* $\langle h', s'\rangle \models \mathcal{Q}$*.*

Proof. By contradiction. Suppose that a proof P derives $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} \mid c$ and that $\langle h, s\rangle \models \mathcal{P}$, but that $(c, \langle h, s\rangle) \Downarrow \langle h', s'\rangle$ implies $\langle h', s'\rangle \models \mathcal{Q}$ for all $\langle h', s'\rangle$. Then by Proposition B.1 we can infer an infinite sequence $\langle h_i, s_i\rangle$ $(i \geq 0)$ of models that, pointwise, invalidates each of the goals along an infinite path in P. Since P is a proof, there is an infinitely progressing trace following this path, which by the latter part of Proposition B.1 we can map to a descending chain of fixed point approximations. Moreover, since the trace is infinitely progressing, this must be an infinitely descending chain. However, this is impossible since the fixed point approximations are well-founded. □

C Specifications and Synthesized Code

Below we list specifications and synthesized code for all benchmarks in Tab. 1 and Tab. 2. Predicate definitions, which are shared between benchmarks, are given at the end.

C.1 Complex benchmarks

1. List: deallocate two.

```
{ sll(x, s1) * sll(y, s2)}
void listfree2(loc x, loc y)
{ emp }

void listfree2 (loc x, loc y) {
  if (x = 0) {
    listfree200(y);
  } else {
    let n = *x;
    listfree2(n, y);
    free(x);
  }
}

void listfree200 (loc y) {
  if (y = 0) {
  } else {
    let n = *y;
    listfree200(n);
    free(y);
  }
}
```

2. List: append three.

```
{r ↦ a * sll(x, s1) * sll(y, s2) * sll(z, s3)}
void append3(loc x, loc y, loc z, loc r)
{s = s1 ∪ s2 ∪ s3; r ↦ v * sll(v, s) }

void append3 (loc x, loc y, loc z, loc r) {
  if (x = 0) {
    append300(y, z, r);
  } else {
    let n = *(x + 1);
    append3(n, y, z, r);
    let v = *r;
    *(x + 1) = v;
    *r = x;
  }
}

void append300 (loc y, loc z, loc r) {
  if (y = 0) {
    *r = z;
```

```
  } else {
    let n = *(y + 1);
    append300(n, z, r);
    let v = *r;
    *(y + 1) = v;
    *r = y;
  }
}
```

3. List: non-destructive append.

```
{ r ↦ x2 * sll(x1, s1) * sll(x2, s2) }
void sll_append_copy(loc x1, loc r)
{ s = s1 ∪ s2 ; r ↦ y * sll(x1, s1) * sll(x2, s2) *
      sll(y, s) }

void sll_append_copy (loc x1, loc r) {
  let x = *r;
  if (x1 = 0) {
    sll_append_copy00(x, r);
  } else {
    let v = *x1;
    let n = *(x1 + 1);
    sll_append_copy(n, r);
    let y1 = *r;
    let y = malloc(2);
    *r = y;
    *(y + 1) = y1;
    *y = v;
  }
}

void sll_append_copy00 (loc x22, loc r) {
  if (x22 = 0) {
  } else {
    let v = *x22;
    let n = *(x22 + 1);
    *r = n;
    sll_append_copy00(n, r);
    let y1 = *r;
    let y = malloc(2);
    *r = y;
    *(y + 1) = y1;
    *y = v;
  }
}
```

4. List: set union.

```
{ r ↦ x * ulist(x, s1) * ulist(y, s2) }
void union (loc r, loc y)
{ r ↦ z * ulist(z, s1 ∪ s2) }

void union (loc r, loc y) {
  let x = *r;
  if (x = 0) {
    *r = y;
  } else {
    let v = *x;
    let n = *(x + 1);
    *r = n;
    union(r, y);
    union119(n, v, x, r);
  }
}

void union119 (loc nxtx22, int vx22, loc x2, loc r)
    {
  let z1 = *r;
  if (z1 = 0) {
    *(x2 + 1) = 0;
    *r = x2;
  } else {
    let v = *z1;
    if (vx22 ≤ v ∧ v ≤ vx22) {
      free(x2);
    } else {
      let n = *(z1 + 1);
      *r = n;
      *(z1 + 1) = nxtx22;
      *z1 = vx22;
      union119(nxtx22, vx22, z1, r);
      let z = *r;
      *(x2 + 1) = z;
      *r = x2;
      *x2 = v;
    }
  }
}
```

5. List: set intersection.

```
{ r ↦ x * ulist(x, s1) * ulist(y, s2) }
void union (loc r, loc y)
{ r ↦ z * ulist(z, s1 ∩ s2) * ulist(y, s2) }

void intersect (loc r, loc y) {
  let x = *r;
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    *r = n;
    intersect(r, y);
    intersect1(n, v, x, r, y);
  }
}

void intersect1 (loc nx, int vx, loc x, loc r,
     loc y) {
  let z1 = *r;
  if (z1 = 0) {
    intersect2(vx, x, r, y);
  } else {
    let v = *z1;
    let n = *(z1 + 1);
    *r = n;
    *(z1 + 1) = nx;
    *z1 = vx;
    intersect1(nx, vx, z1, r, y);
    let z = *r;
    *(x + 1) = z;
    *r = x;
    *x = v;
  }
}

void intersect2 (int vx, loc x, loc r, loc y) {
  if (y = 0) {
    free(x);
  } else {
```

```
      let v = *y;
      let n = *(y + 1);
      if (vx ≤ v ∧ v ≤ vx) {
        intersect2(vx, x, r, n);
        let z2 = *r;
        let z = malloc(2);
        *r = z;
        *(z + 1) = z2;
        *y = vx;
        *z = vx;
      } else {
        intersect2(vx, x, r, n);
      }
    }
  }
```

6. List: set difference.

```
{ r ↦ x ∗ ulist(x, s1) ∗ ulist(y, s2) }
void diff (loc r, loc y)
{ r ↦ z ∗ ulist(z, s1 \ s2) }

void diff (loc r, loc y) {
  if (y = 0) {
  } else {
    let v = *y;
    let n = *(y + 1);
    diff(r, n);
    diff119(n, v, r, y);
  }
}

void diff119 (loc nxty2, int vy2, loc r, loc y) {
  let z1 = *r;
  if (z1 = 0) {
    free(y);
  } else {
    let v = *z1;
    if (vy2 ≤ v ∧ v ≤ vy2) {
      let n = *(z1 + 1);
      *r = n;
      diff119(n, v, r, z1);
      free(y);
    } else {
      let n = *(z1 + 1);
      *r = n;
      *(z1 + 1) = nxty2;
      *z1 = vy2;
      diff119(nxty2, vy2, r, z1);
      let z = *r;
      *(y + 1) = z;
      *r = y;
      *y = v;
    }
  }
}
```

7. List: deduplicate.

```
{ r ↦ x ∗ sll(x, s) }
void unique (loc r)
{ r ↦ y ∗ ulist(y, s) }

void unique (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    *r = n;
    unique(r);
    unique114(n, v, x, r);
  }
}

void unique114 (loc nxtx22, int vx22, loc x2, loc
    r) {
  let y1 = *r;
  if (y1 = 0) {
    *(x2 + 1) = 0;
    *r = x2;
  } else {
    let v = *y1;
    if (vx22 ≤ v ∧ v ≤ vx22) {
      free(x2);
    } else {
      let n = *(y1 + 1);
      *r = n;
      *(y1 + 1) = nxtx22;
      *y1 = vx22;
      unique114(nxtx22, vx22, y1, r);
      let y = *r;
      *(x2 + 1) = y;
      *r = x2;
      *x2 = v;
    }
  }
}
```

8. List of lists: deallocate.

```
{ multilist(x, len, s) }
void multilist_free(loc x)
{ emp }

void multilist_free (loc x) {
  if (x = 0) {
  } else {
    let h = *x;
    let t = *(x + 1);
    multilist_free(t);
    multilist_free112(h, x);
  }
}

void multilist_free112 (loc hx2, loc x) {
  if (hx2 = 0) {
    free(x);
  } else {
    let n = *(hx2 + 1);
    *hx2 = n;
    multilist_free112(n, hx2);
    free(x);
  }
}
```

9. List of lists: flatten.

```
{ r ↦ x ∗ multilist(x, len, s) }
void multilist_flatten(loc r)
{ r ↦ y ∗ sll(y, len, s) }
```

```
void multilist_flatten (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    multilist_flatten10(x, r);
  }
}

void multilist_flatten10 (loc x2, loc r) {
  let h = *x2;
  let t = *(x2 + 1);
  if (h = 0) {
    *r = t;
    multilist_flatten(r);
    free(x2);
  } else {
    let v = *h;
    let n = *(h + 1);
    *h = n;
    *(h + 1) = t;
    *r = h;
    multilist_flatten10(h, r);
    let y = *r;
    *(x2 + 1) = y;
    *r = x2;
    *x2 = v;
  }
}
```

10. Binary tree: deallocate two.

```
{tree(x, s1) ∗ tree(y, s2)}
  void treefree2(loc x, loc y)
{emp }

void treefree2 (loc x, loc y) {
  if (x = 0) {
    if (y = 0) {
    } else {
      let l = *(y + 1);
      let r = *(y + 2);
      treefree2(l, r);
      free(y);
    }
  } else {
    let lx = *(x + 1);
    let rx = *(x + 2);
    treefree2(lx, rx);
    if (y = 0) {
      free(x);
    } else {
      let l = *(y + 1);
      let r = *(y + 2);
      treefree2(l, r);
      free(y);
      free(x);
    }
  }
}
```

11. Binary tree: flatten.

```
{ z ↦ x ∗ tree(x, s) }
void flatten(loc z)
{ z ↦ y ∗ sll(y, s) }

void flatten (loc z) {
  let x = *z;
  if (x = 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    *z = l;
    flatten(z);
    let y = *z;
    *z = r;
    flatten(z);
    flatten126(y, v, x, z);
  }
}

void flatten126 (loc y12, int vx22, loc x2, loc z) {
  let y2 = *z;
  if (y12 = 0) {
    let y = malloc(2);
    free(x2);
    *z = y;
    *(y + 1) = y2;
    *y = vx22;
  } else {
    let n = *(y12 + 1);
    flatten126(n, vx22, x2, z);
    let y = *z;
    *(y12 + 1) = y;
    *z = y12;
  }
}
```

Alternative solution (with different cost function):

```
void flatten (loc z) {
  let x = *z;
  if (x = 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    *z = l;
    flatten(z);
    flatten115(r, v, x, z);
  }
}

void flatten115 (loc rx22, int vx22, loc x2, loc z)
    {
  let y1 = *z;
  if (y1 = 0) {
    *z = rx22;
    flatten(z);
    let y2 = *z;
    let y = malloc(2);
    free(x2);
    *z = y;
    *(y + 1) = y2;
    *y = vx22;
  } else {
    let n = *(y1 + 1);
    *z = n;
    flatten115(rx22, vx22, x2, z);
    let y = *z;
```

```
    *(y1 + 1) = y;
    *z = y1;
  }
}
```

12. Binary tree: flatten to dll in place.

```
{ tree(x, s) }
void flatten(loc x)
{ dll(x, y, s) }

void flatten (loc x) {
  if (x = 0) {
  } else {
    let l = *(x + 1);
    let r = *(x + 2);
    flatten(l);
    flatten111(r, l, x);
  }
}

void flatten111 (loc rx2, loc lx2, loc x) {
  if (lx2 = 0) {
    flatten(rx2);
    if (rx2 = 0) {
    } else {
      *(rx2 + 2) = x;
      *(x + 1) = rx2;
    }
  } else {
    let v = *lx2;
    let w = *(lx2 + 1);
    *(lx2 + 2) = rx2;
    flatten111(rx2, w, lx2);
    *(lx2 + 2) = x;
  }
}
```

13. Rose tree: deallocate.

```
{ rose_tree(x, s) }
void rose_tree_free(loc x)
{ emp }

void rose_tree_free (loc x) {
  if (x = 0) {
  } else {
    rose_tree_free10(x);
  }
}

void rose_tree_free10 (loc x) {
  let b = *(x + 1);
  if (b = 0) {
    free(x);
  } else {
    let r = *b;
    rose_tree_free(r);
    rose_tree_free10(b);
    free(x);
  }
}
```

14. Rose tree: flatten.

```
{ r ↦ x * rose_tree(x, s) }
void rose_tree_flatten(loc r)
{ r ↦ y * sll(y, s) }

void rose_tree_flatten (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    rose_tree_flatten10(x, r);
  }
}

void rose_tree_flatten10 (loc x2, loc r) {
  let v = *x2;
  let b = *(x2 + 1);
  if (b = 0) {
  } else {
    let rb = *b;
    *r = rb;
    rose_tree_flatten(r);
    let y = *r;
    *r = b;
    *b = v;
    rose_tree_flatten10(b, r);
    rose_tree_flatten14136(y, v, x2, r);
  }
}

void rose_tree_flatten14136 (loc y12, int vx22, loc
    x2, loc r) {
  if (y12 = 0) {
    free(x2);
  } else {
    let v = *y12;
    let n = *(y12 + 1);
    *(y12 + 1) = x2;
    *y12 = vx22;
    rose_tree_flatten14136(n, vx22, y12, r);
    let y = *r;
    *(x2 + 1) = y;
    *r = x2;
    *x2 = v;
  }
}
```

15. Sorted list: reverse.

```
{ 0 ≤ n ; srtl(x, n, lo, hi) }
void reverse (loc x)
{ descl(x, n, lo, hi) }

void reverse (loc x) {
  if (x = 0) {
  } else {
    let l = *x;
    let n = *(x + 1);
    reverse(n);
    reverse114(n, l, x);
  }
}

void reverse114 (loc nxtx2, int lo2, loc x) {
  if (nxtx2 = 0) {
  } else {
    let h = *nxtx2;
    let n = *(nxtx2 + 1);
    *nxtx2 = lo2;
```

```
    reverse114(n, lo2, nxtx2);
    *x = h;
  }
}
```

16. Sorted list: sort.

```
{ 0 ≤ n ; sll_bounds(x, n, lo, hi) }
void sort (loc x)
{ srtl(x, n, lo, hi) }

void sort (loc x) {
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    sort(n);
    sort115(n, v, x);
  }
}

void sort115 (loc nxtx2, int vx2, loc x) {
  if (nxtx2 = 0) {
  } else {
    let l = *nxtx2;
    if (vx2 ≤ l) {
    } else {
      let n = *(nxtx2 + 1);
      *nxtx2 = vx2;
      sort115(n, vx2, nxtx2);
      *x = l;
    }
  }
}
```

17.Sorted list: merge.

```
{ 0 ≤ nx ∧ 0 ≤ ny ;
  r ↦ y ∗
  srtl(x, nx, lox, hix) ∗ srtl(y, ny, loy, hiy) }
void srtl_merge (loc x, loc r)
{ n = nx + ny ∧  lo = (lox ≤ loy ? lox : loy) ∧
  hi = (hix ≤ hiy ? hiy : hix) ;
  r ↦ z ∗ srtl(z, n, lo, hi) }

void srtl_merge (loc y, loc r) {
  let x = *r;
  if (x = 0) {
    *r = y;
  } else {
    srtl_merge10(x, y, r);
  }
}

void srtl_merge10 (loc x2, loc y, loc r) {
  let vx = *x2;
  let nx = *(x2 + 1);
  if (y = 0) {
  } else {
    let v = *y;
    if (vx ≤ v) {
      *r = y;
      srtl_merge10(y, nx, r);
      let z = *r;
      *(x2 + 1) = z;
      *r = x2;
    } else {
      let n = *(y + 1);
      *(y + 1) = nx;
      *r = y;
      *y = vx;
      srtl_merge10(y, n, r);
      let z = *r;
      *(x2 + 1) = z;
      *r = x2;
      *x2 = v;
    }
  }
}
```

18. BST: from list.

```
{ 0 ≤ n ; r ↦ 0 ∗ sll_bounds(x, n, lo, hi) }
void toBST (loc x, loc r)
{ r ↦ y ∗ bst(y, n, lo, hi) }

void toBST (loc x, loc r) {
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    toBST(n, r);
    toBST119(v, x, r);
  }
}

void toBST119 (int vx2, loc x, loc r) {
  let y1 = *r;
  if (y1 = 0) {
    let y = malloc(3);
    free(x);
    *r = y;
    *(y + 1) = 0;
    *(y + 2) = 0;
    *y = vx2;
  } else {
    let v = *y1;
    if (vx2 ≤ v) {
      let l = *(y1 + 1);
      *r = l;
      toBST119(vx2, x, r);
      let y = *r;
      *(y1 + 1) = y;
      *r = y1;
    } else {
      let ry = *(y1 + 2);
      *r = ry;
      toBST119(vx2, x, r);
      let y = *r;
      *(y1 + 2) = y;
      *r = y1;
    }
  }
}
```

19. BST: to sorted list.

```
{ 0 ≤ n ; r ↦ 0 ∗ bst(x, n, lo, hi) }
void flatten(loc x, loc r)
{ r ↦ y ∗ srtl(y, n, lo, hi) }

void flatten (loc x, loc r) {
```

```
  if (x = 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let rx = *(x + 2);
    flatten(l, r);
    flatten120(rx, v, x, r);
  }
}

void flatten120 (loc rx2, int vx2, loc x, loc r) {
  let y1 = *r;
  if (y1 = 0) {
    flatten(rx2, r);
    let y2 = *r;
    if (y2 = 0) {
      let y = malloc(2);
      free(x);
      *r = y;
      *(y + 1) = 0;
      *y = vx2;
    } else {
      let v = *y2;
      let nx = *(y2 + 1);
      let n = malloc(2);
      free(x);
      *(y2 + 1) = n;
      *(n + 1) = nx;
      *y2 = vx2;
      *n = v;
    }
  } else {
    let vy = *y1;
    let nx = *(y1 + 1);
    *r = nx;
    flatten120(rx2, vx2, x, r);
    let y = *r;
    let v = *y;
    let n = *(y + 1);
    *(y + 1) = y1;
    *(y1 + 1) = n;
    *y = vy;
    *y1 = v;
  }
}
```

C.2 Simple benchmarks

20. Integers: swap two.

```
{x ↦ a ∗ y ↦ b}
void swap (loc x, loc y)
{x ↦ b ∗ y ↦ a}

void swap (loc x, loc y) {
  let a = *x;
  let b = *y;
  *x = b;
  *y = a;
}
```

21. Integers: min of two.

```
{r ↦ 0}
void min2 (loc r, int x, int y)
{m ≤ x ∧ m ≤ y ; r ↦ m}

void min2 (loc r, int x, int y) {
  if (x ≤ y) {
    *r = x;
  } else {
    *r = y;
  }
}
```

22. Linked list: length.

```
{0 ≤ n ; ret ↦ a ∗ sll_bounds(x, n, lo, hi)}
void sll_len (loc x, loc ret)
{ret ↦ n ∗ sll_bounds(x, n, lo, hi)}

void sll_len (loc x, loc ret) {
  if (x = 0) {
    *ret = 0;
  } else {
    let n = *(x + 1);
    sll_len(n, ret);
    let l = *ret;
    *ret = 1 + l;
  }
}
```

23. Linked list: max.

```
{ret ↦ a ∗ sll_bounds(x, n, lo, hi)}
void sll_max (loc x, loc ret)
{ret ↦ hi ∗ sll_bounds(x, n, lo, hi)}

void sll_max (loc x, loc ret) {
  if (x = 0) {
    *ret = 0;
  } else {
    let v = *x;
    let n = *(x + 1);
    sll_max(n, ret);
    let h = *ret;
    *ret = h ≤ v ? v : h;
  }
}
```

24. Linked list: min.

```
{ret ↦ a ∗ sll_bounds(x, n, lo, hi)}
void sll_min (loc x, loc ret)
{ret ↦ lo ∗ sll_bounds(x, n, lo, hi)}

void sll_min (loc x, loc ret) {
  if (x = 0) {
    *ret = 7;
  } else {
    let v = *x;
    let n = *(x + 1);
    sll_min(n, ret);
    let l = *ret;
    *ret = v ≤ l ? v : l;
  }
}
```

25. Linked list: singleton.

```
{ret ↦ a}
void sll_singleton (int x, loc ret)
{s = {x} ; ret ↦ y ∗ sll(y, s)}

void sll_singleton (int x, loc ret) {
  let y = malloc(2);
  *ret = y;
  *(y + 1) = 0;
  *y = x;
}
```

26. Linked list: dispose.

```
{sll(x, s)}
void sll_free (loc x)
{emp}

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

27. Linked list: initialize.

```
{sll(x, s)}
void sll_init (loc x, int v)
{s1 ⊆ {v} ; sll(x, s1)}

void sll_init (loc x, int v) {
  if (x = 0) {
  } else {
    let n = *(x + 1);
    sll_init(n, v);
    *x = v;
  }
}
```

28. Linked list: copy.

```
{r ↦ x ∗ sll(x, s)}
void sll_copy (loc r)
{r ↦ y ∗ sll(x, s) ∗ sll(y, s)}

void sll_copy (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    *r = n;
    sll_copy(r);
    let y1 = *r;
    let y = malloc(2);
    *r = y;
    *(y + 1) = y1;
    *y = v;
  }
}
```

29. Linked list: append.

```
{ret ↦ x2 ∗ sll(x1, s1) ∗ sll(x2, s2)}
void sll_append (loc x1, loc ret)
{s = s1 ∪ s2 ; ret ↦ y ∗ sll(y, s)}

void sll_append (loc x1, loc ret) {
  if (x1 = 0) {
  } else {
    let n = *(x1 + 1);
    sll_append(n, ret);
    let y = *ret;
    *(x1 + 1) = y;
    *ret = x1;
  }
}
```

30. Linked list: delete.

```
{ret ↦ a ∗ sll(x, s)}
void sll_delete_all (loc x, loc ret)
{s1 = s \ {a} ; ret ↦ y ∗ sll(y, s1)}

void sll_delete_all (loc x, loc ret) {
  let a = *ret;
  if (x = 0) {
    *ret = 0;
  } else {
    let v = *x;
    let n = *(x + 1);
    if (a ≤ v ∧ v ≤ a) {
      sll_delete_all(n, ret);
      free(x);
    } else {
      sll_delete_all(n, ret);
      let y = *ret;
      *(x + 1) = y;
      *ret = x;
    }
  }
}
```

31. Sorted list: prepend.

```
{0 ≤ k ∧ 0 ≤ n ∧ k ≤ 7 ∧ k ≤ lo ;
  r ↦ a ∗ srtl(x, n, lo, hi)}
void srtl_prepend (loc x, int k, loc r)
{n1 = n + 1 ; r ↦ y ∗ srtl(y, n1, k, hi1)}

void srtl_prepend (loc x, int k, loc r) {
  let y = malloc(2);
  *r = y;
  *(y + 1) = x;
  *y = k;
}
```

32. Sorted list: insert.

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

void srtl_insert (loc x, loc r) {
  let k = *r;
  if (x = 0) {
    let y = malloc(2);
    *r = y;
    *(y + 1) = 0;
    *y = k;
```

```
  } else {
    let v = *x;
    let nx = *(x + 1);
    if (k ≤ v ∧ v ≤ k) {
      let n = malloc(2);
      *(x + 1) = n;
      *r = x;
      *(n + 1) = nx;
      *n = k;
    } else {
      if (k ≤ v) {
        let n = malloc(2);
        *(x + 1) = n;
        *r = x;
        *(n + 1) = nx;
        *n = v;
        *x = k;
      } else {
        srtl_insert(nx, r);
        let y = *r;
        *(x + 1) = y;
        *r = x;
      }
    }
  }
}
```

33. Sorted list: insertion sort.

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

// Synthesis goal:
{ 0 ≤ n ; r ↦ 0 ∗ sll(x, n, lo, hi) }
void insertion_sort (loc x, loc r)
{ r ↦ y ∗ sll(x, n, lo, hi) ∗ srtl(y, n, lo, hi) }

void insertion_sort (loc x, loc r) {
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    insertion_sort(n, r);
    let y = *r;
    *r = v;
    srtl_insert(y, r);
  }
}
```

34. Tree: size.

```
{0 ≤ n ; r ↦ 0 ∗ treeN(x, n)}
void tree_size (loc x, loc r)
{r ↦ n ∗ treeN(x, n)}

void tree_size (loc x, loc r) {
  if (x = 0) {
  } else {
    let l = *(x + 1);
    let rx = *(x + 2);
    tree_size(l, r);
    let n1 = *r;
    *r = 0;
    tree_size(rx, r);
    let n = *r;
    *r = 1 + n1 + n;
  }
}
```

35. Tree: dispose.

```
{tree(x, s)}
void tree_free (loc x)
{emp}

void tree_free (loc x) {
  if (x = 0) {
  } else {
    let l = *(x + 1);
    let r = *(x + 2);
    tree_free(l);
    tree_free(r);
    free(x);
  }
}
```

36. Tree: copy.

```
{r ↦ x ∗ tree(x, s)}
void tree_copy (loc r)
{r ↦ y ∗ tree(x, s) ∗ tree(y, s)}

void tree_copy (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let rx = *(x + 2);
    *r = l;
    tree_copy(r);
    let y1 = *r;
    *r = rx;
    tree_copy(r);
    let y2 = *r;
    let y = malloc(3);
    *r = y;
    *(y + 1) = y1;
    *(y + 2) = y2;
    *y = v;
  }
}
```

37. Tree: flatten w/append.

```
// Library component:
{ sll(x1, s1) ∗ sll(x2, s2) ∗ ret ↦ x2 }
void sll_append (loc x1, loc ret)
{ s = s1 ∪ s2 ; sll(y, s) ∗ ret ↦ y }

// Synthesis spec:
{z ↦ x ∗ tree(x, s)}
void tree_flatten (loc z)
{z ↦ y ∗ sll(y, s)}

void tree_flatten (loc z) {
  let x = *z;
  if (x = 0) {
  } else {
```

```
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    *z = l;
    tree_flatten(z);
    let y1 = *z;
    *z = r;
    tree_flatten(z);
    let y2 = *z;
    *x = y2;
    sll_append(y1, x);
    let y3 = *x;
    let y = malloc(2);
    free(x);
    *z = y;
    *(y + 1) = y3;
    *y = v;
  }
}
```

38. Tree: flatten w/acc.

```
{z ↦ y ∗ sll(y, acc) ∗ tree(x, s)}
void tree_flatten (loc x, loc z)
{s1 = s ∪ acc ; z ↦ t ∗ sll(t, s1)}

void tree_flatten (loc x, loc z) {
  if (x = 0) {
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    tree_flatten(l, z);
    tree_flatten(r, z);
    let t2 = *z;
    let t = malloc(2);
    free(x);
    *z = t;
    *(t + 1) = t2;
    *t = v;
  }
}
```

39. BST: insert.

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

void bst_insert (loc x, loc ret) {
  let k = *ret;
  if (x = 0) {
    let y = malloc(3);
    *ret = y;
    *(y + 1) = 0;
    *(y + 2) = 0;
    *y = k;
  } else {
    let v = *x;
    let l = *(x + 1);
    let r = *(x + 2);
    if (k ≤ v) {
      bst_insert(l, ret);
      let y = *ret;
      *(x + 1) = y;
      *ret = x;
    } else {
      bst_insert(r, ret);
      let y = *ret;
      *(x + 2) = y;
      *ret = x;
    }
  }
}
```

40. BST: rotate left.

```
{ not (r = 0) ∧ 0 ≤ sz1 ∧  0 ≤ sz2  ∧
  0 ≤ v ∧ v ≤ 7 ∧ hi1 ≤ v ∧ v ≤ lo2  ;
  ret ↦ unused ∗
  [x, 3] ∗ x ↦ v ∗ (x + 1) ↦ l ∗ (x + 2) ↦ r ∗
  bst(l, sz1, lo1, hi1) ∗ bst(r, sz2, lo2, hi2) }
void bst_left_rotate (loc x, loc ret)
{ sz3 + sz4 = sz1 + sz2 ∧  0 ≤ sz3 ∧  0 ≤ sz4  ∧
  0 ≤ v3 ∧ v3 ≤ 7 ∧ hi3 ≤ v3 ∧ v3 ≤ lo4 ;
  ret ↦ y ∗
  [y, 3] ∗ y ↦ v3 ∗ (y + 1) ↦ x ∗ (y + 2) ↦ r3 ∗
  bst(x, sz3, lo3, hi3) ∗ bst(r3, sz4, lo4, hi4) }

void bst_left_rotate (loc x, loc ret) {
  let r = *(x + 2);
  let l = *(r + 1);
  *(r + 1) = x;
  *ret = r;
  *(x + 2) = l;
}
```

41. BST: rotate right.

```
{ not (l = 0) ∧ 0 ≤ sz1 ∧  0 ≤ sz2  ∧
  0 ≤ v ∧ v ≤ 7 ∧ hi1 ≤ v ∧ v ≤ lo2 ;
  ret ↦ unused ∗
  [x, 3] ∗ x ↦ v ∗ (x + 1) ↦ l ∗ (x + 2) ↦ r ∗
  bst(l, sz1, lo1, hi1) ∗ bst(r, sz2, lo2, hi2) }
void bst_right_rotate (loc x, loc ret)
{ sz3 + sz4 = sz1 + sz2 ∧ 0 ≤ sz3 ∧  0 ≤ sz4  ∧
  0 ≤ v3 ∧ v3 ≤ 7 ∧ hi3 ≤ v3 ∧ v3 ≤ lo4 ;
  ret ↦ y ∗
  [y, 3] ∗ y ↦ v3 ∗ (y + 1) ↦ l3 ∗ (y + 2) ↦ x ∗
  bst(l3, sz3, lo3, hi3) ∗ bst(x, sz4, lo4, hi4) }

void bst_right_rotate (loc x, loc ret) {
  let l = *(x + 1);
  let r = *(l + 2);
  *(l + 2) = x;
  *ret = l;
  *(x + 1) = r;
}
```

42. BST: delete root.

```
{ 0 ≤ sz1 ∧  0 ≤ sz2  ∧
  0 ≤ v ∧ v ≤ 7 ∧ hi1 ≤ v ∧ v ≤ lo2 ;
  ret ↦ 0 ∗
  [x, 3] ∗ x ↦ v ∗ (x + 1) ↦ l ∗ (x + 2) ↦ r ∗
  bst(l, sz1, lo1, hi1) ∗ bst(r, sz2, lo2, hi2) }
void bst_delete_root (loc x, loc ret)
{ n1 = sz1 + sz2 ∧
  lo = (l = 0 ? (r = 0 ? 7 : lo2) : lo1) ∧
  hi = (r = 0 ? (l = 0 ? 0 : hi1) : hi2) ;
  ret ↦ y ∗ bst(y, n1, lo, hi)  }
```

```
void bst_delete_root (loc x, loc ret) {
  let v2 = *x;
  let l2 = *(x + 1);
  let r2 = *(x + 2);
  if (l2 = 0) {
    if (r2 = 0) {
      free(x);
    } else {
      free(x);
      *ret = r2;
    }
  } else {
    let vl = *l2;
    let ll = *(l2 + 1);
    let rl = *(l2 + 2);
    if (r2 = 0) {
      free(x);
      *ret = l2;
    } else {
      let v = *r2;
      let l = *(r2 + 1);
      let r = *(r2 + 2);
      *(r2 + 1) = rl;
      *(r2 + 2) = l;
      *r2 = v2;
      bst_delete_root(r2, ret);
      let y = *ret;
      *(l2 + 1) = x;
      *(l2 + 2) = r;
      *(x + 1) = ll;
      *(x + 2) = y;
      *ret = l2;
      *l2 = v;
      *x = vl;
    }
  }
}
```

43. Doubly-linked list: copy.

```
{ r ↦ x * dll(x, a, s)}
void dll_copy(loc r)
{ r ↦ y * dll(x, a, s) * dll(y, b, s) }

void dll_copy (loc r) {
  let x = *r;
  if (x = 0) {
  } else {
    let vx = *x;
    let w = *(x + 1);
    *r = w;
    dll_copy(r);
    let y1 = *r;
    if (y1 = 0) {
      let y = malloc(3);
      *(x + 1) = 0;
      *r = y;
      *(y + 1) = 0;
      *y = vx;
      *(y + 2) = 0;
    } else {
      let v = *y1;
      let y = malloc(3);
      *(y1 + 2) = y;
      *r = y;
      *(y + 1) = y1;
      *y1 = vx;
      *y = v;
      *(y + 2) = 0;
    }
  }
}
```

44. Doubly-linked list: append.

```
{ dll(x1, a, s1) * dll(x2, b, s2) * ret ↦ x2 }
void dll_append (loc x1, loc ret)
{ s = s1 ∪ s2 ; dll(y, c, s) * ret ↦ y }

void dll_append (loc x1, loc ret) {
  if (x1 = 0) {
  } else {
    let w = *(x1 + 1);
    dll_append(w, ret);
    let y = *ret;
    if (y = 0) {
      *(x1 + 1) = 0;
      *ret = x1;
    } else {
      *(y + 2) = x1;
      *(x1 + 1) = y;
      *ret = x1;
    }
  }
}
```

45. Doubly-linked list: delete.

```
{ dll(x, b, s) * ret ↦ a }
void dll_delete_all (loc x, loc ret)
{ s1 = s \ {a} ; dll(y, c, s1) * ret ↦ y }

void dll_delete_all (loc x, loc ret) {
  let a = *ret;
  if (x = 0) {
    *ret = 0;
  } else {
    let vx = *x;
    let w = *(x + 1);
    if (a ≤ vx ∧ vx ≤ a) {
      dll_delete_all(w, ret);
      free(x);
    } else {
      dll_delete_all(w, ret);
      let y = *ret;
      if (y = 0) {
        *(x + 1) = 0;
        *ret = x;
      } else {
        let v = *y;
        if (vx ≤ v ∧ v ≤ vx) {
          free(x);
        } else {
          *(y + 2) = x;
          *(x + 1) = y;
          *ret = x;
        }
      }
    }
  }
```

```
}
```

46. Doubly-linked list: single to double.

```
{ f ↦ x * sll(x, s)}
void sll_to_dll(loc f)
{ f ↦ i * dll(i, 0, s)}

void sll_to_dll (loc f) {
  let x = *f;
  if (x = 0) {
  } else {
    let v = *x;
    let n = *(x + 1);
    *f = n;
    sll_to_dll(f);
    let i1 = *f;
    if (i1 = 0) {
      let i = malloc(3);
      free(x);
      *f = i;
      *(i + 1) = 0;
      *(i + 2) = 0;
      *i = v;
    } else {
      let i = malloc(3);
      free(x);
      *(i1 + 2) = i;
      *f = i;
      *(i + 1) = i1;
      *(i + 2) = 0;
      *i = v;
    }
  }
}
```

C.3 Predicate definitions

```
// Singly-linked list with payload set
predicate sll(loc x, set s) {
  |  x = 0        ⇒ { s = {} ; emp }
  |  not (x = 0)  ⇒ { s = {v} ∪ s1 ;
      [x, 2] * x ↦ v * (x + 1) ↦ nxt *
      sll(nxt, s1) }
}

// List with unique elements
predicate ulist(loc x, set s) {
  |  x = 0        ⇒ { s = {} ; emp }
  |  not (x = 0)  ⇒ { s = {v} ∪ s1 ∧ not (v in s1);
      [x, 2] * x ↦ v * (x + 1) ↦ nxt *
      ulist(nxt, s1) }
}

// List of lists
predicate multilist(loc x, int size, set s) {
|  x = 0        ⇒ { size = 0 ∧ s = {} ; emp }
|  not (x = 0)  ⇒ { size = len1 + size2 ∧
      s = s1 ∪ s2 ;
      [x, 2] * x ↦ h * (x + 1) ↦ t *
      sll(h, len1, s1) * multilist(t, size2, s2) }
}

// Doubly-linked list
predicate dll(loc x, loc z, set s) {
|  x = 0        ⇒ { s = {} ; emp }
|  not (x = 0) ⇒ { s = {v} ∪ s1 ;
      [x, 3] * x ↦ v * (x + 1) ↦ w *
      (x + 2) ↦ z * dll(w, x, s1) }
}

// Binary tree
predicate tree(loc x, set s) {
  |  x = 0        ⇒ {s = {} ; emp}
  |  not (x = 0)  ⇒ {s = {v} ∪ s1 ∪ s2  ;
      [x, 3] * x ↦ v * (x + 1) ↦ l *
      (x + 2) ↦ r * tree(l, s1) * tree(r, s2)}
}

// Binary tree parameterized by size
predicate treeN(loc x, int n) {
|  x = 0        ⇒ { n = 0 ; emp }
|  not (x = 0)  ⇒ { n = 1 + n1 + n2  ∧
    0 ≤ n1  ∧  0 ≤ n2 ;
    [x, 3] * x ↦ v * (x + 1) ↦ l *
    (x + 2) ↦ r * treeN(l, n1) * treeN(r, n2)}
}

// Rose tree
predicate rose_tree(loc x, set s) {
|  x = 0        ⇒ { s = {} ; emp }
|  not (x = 0)  ⇒ { s = {v} ∪ s1 ;
      [x, 2] * x ↦ v * (x + 1) ↦ b *
      buds(b, s1)}
}

// Children of the rose tree
predicate buds(loc x, set s) {
|  x = 0        ⇒ { s = {}; emp }
|  not (x = 0)  ⇒ { s = s1 ∪ s2 ;
      [x, 2] * x ↦ r * (x + 1) ↦ nxt *
      rose_tree(r, s1) * buds(nxt, s2) }
}

// Sorted list (acceding order)
// The exact bounds 0 and 7 are inconsequential,
// but some bounds are needed to specify a sorted
    list
predicate srtl(loc x, int len, int lo, int hi) {
|  x = 0        ⇒ { len = 0 ∧ lo = 7 ∧ hi = 0
      ; emp }
|  not (x = 0) ⇒ { len = 1 + len1 ∧ 0 ≤ len1 ∧
      hi = (hi1 ≤ lo ? lo : hi1) ∧
      lo ≤ lo1 ∧ 0 ≤ lo ∧ lo ≤ 7 ;
      [x, 2] * x ↦ lo * (x + 1) ↦ nxt *
      srtl(nxt, len1, lo1, hi1) }
}

// Sorted list (descending order)
predicate descl(loc x, int len, int lo, int hi) {
|  x = 0        ⇒ { len = 0 ∧ lo = 7 ∧ hi = 0
      ; emp }
|  not (x = 0) ⇒ { len = 1 + len1 ∧ 0 ≤ len1 ∧
      lo = (hi ≤ lo1 ? hi : lo1) ∧
      hi ≥ hi1 ∧ 0 ≤ hi ∧ hi ≤ 7 ;
      [x, 2] * x ↦ hi * (x + 1) ↦ nxt *
      descl(nxt, len1, lo1, hi1) }
}
```

```
// Linked list parametrized by bounds
predicate sll_bounds(loc x,int len,int lo,int hi) {
|  x = 0       ⇒ { len = 0 ∧ lo = 7 ∧ hi = 0
      ; emp }
|  not (x = 0) ⇒ { len = 1 + len1 ∧ 0 ≤ len1 ∧
      lo = (v ≤ lo1 ? v : lo1) ∧
      hi = (hi1 ≤ v ? v : hi1)
      ∧ 0 ≤ v ∧ v ≤ 7;
      [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ nxt ∗
      sll_bounds(nxt, len1, lo1, hi1) }
}
```

```
// Binary search tree
predicate bst(loc x, int sz, int lo, int hi) {
|  x = 0       ⇒ { sz = 0 ∧ lo = 7 ∧ hi = 0
      ; emp }
|  not (x = 0) ⇒ { sz = 1 + sz1 + sz2 ∧
      0 ≤ sz1 ∧ 0 ≤ sz2 ∧
      lo = (v ≤ lo1 ? v : lo1) ∧
      hi = (hi2 ≤ v ? v : hi2) ∧
      0 ≤ v ∧ v ≤ 7 ∧ hi1 ≤ v ∧ v ≤ lo2 ;
      [x, 3] ∗ x ↦ v ∗ (x + 1) ↦ l ∗
      (x + 2) ↦ r ∗
      bst(l, sz1, lo1, hi1) ∗
      bst(r, sz2, lo2, hi2)  }
}
```