Inductive Synthesis of Inductive Heap Predicates

ZIYI YANG, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

We present an approach to automatically synthesise recursive predicates in Separation Logic (SL) from concrete data structure instances using Inductive Logic Programming (ILP) techniques. The main challenges to make such synthesis effective are (1) making it work without negative examples that are required in ILP but are difficult to construct for heap-based structures in an automated fashion, and (2) to be capable of summarising not just the *shape* of a heap (*e.g.*, it is a *linked* list), but also the *properties* of the data it stores (*e.g.*, it is a *sorted* linked list). We tackle these challenges with a new predicate learning algorithm. The key contributions of our work are (a) the formulation of ILP-based learning only using positive examples and (b) an algorithm that synthesises property-rich SL predicates from concrete *memory graphs* based on the positive-only learning.

We show that our framework can efficiently and correctly synthesise SL predicates for structures that were beyond the reach of the state-of-the-art tools, including those featuring non-trivial payload constraints (*e.g.*, binary search trees) and nested recursion (*e.g.*, *n*-ary trees). We further extend the usability of our approach by a memory graph generator that produces positive heap examples from programs. Finally, we show how our approach facilitates deductive verification and synthesis of correct-by-construction code.

CCS Concepts: • **Theory of computation → Program specifications**; • **Computing methodologies → Logic programming and answer set programming**.

Additional Key Words and Phrases: inductive program synthesis, answer set programming, Separation Logic

ACM Reference Format:
Ziyi Yang and Ilya Sergey. 2025. Inductive Synthesis of Inductive Heap Predicates. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 86 (April 2025), 27 pages. https://doi.org/10.1145/3720420

1 Introduction

Separation Logic (SL) is a popular Hoare-style formalism for specifying and verifying imperative programs that manipulate mutable pointer-based data structures [50]. SL has been successfully applied to a wide range of applications, including program verification [3, 30], static analysis [11], bug detection [37], invariant inference [20, 38], program synthesis [56, 67], and repair [49, 65]. The key to the practical success of SL is its ability to enable *compositional* reasoning about programs in the presence of potential pointer aliasing by exploiting the *locality* of common heap-manipulating operations. To enable expressive specifications, Separation Logic offers a powerful mechanism to declaratively describe the shape and data properties of linked heap-based structures, such as lists and trees: *inductive heap predicates*. Unsurprisingly, defining precise and useful inductive predicates for non-trivial data structures in general requires a good grasp of the structure's *internal invariants*. Most existing SL-based reasoning frameworks require defining predicates *manually*; a few come with a set of pre-defined predicates for the most commonly used data structures [12, 38],—thus, limiting the ability of those approaches to verify or generally utilise *data-specific* program properties, such as, *e.g.*, correctness of searching an element in a binary search tree.

Authors' Contact Information: Ziyi Yang, National University of Singapore, Singapore, yangziyi@u.nus.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2025/4-ART86
https://doi.org/10.1145/3720420

The aim of this work is to offer a methodology for *automatically synthesising* inductive predicates for linked structures, where not only the *shape* but also the *properties* of the stored data (*e.g.*, a binary tree being *balanced*) would be captured. To achieve this goal, we develop an approach for inferring inductive SL predicates by synthesising them from *memory graphs*, *i.e.*, concrete examples of data structure memory layouts, as produced by programs that generate them. Our work is closely related to two research themes: (1) synthesising formal representations of data structures [8, 28, 43, 71], and (2) using machine learning to infer data structure invariants [10, 42, 64]. Existing approachers either impose specific restrictions on the inputs of the synthesiser by, *e.g.*, requiring functions constructing the data structure [43, 71] or a large number of both positive *and* negative examples [42, 64]; or produce weaker specification, *e.g.*, only inferring the structure shape, but not its properties [8, 28].

To deliver an effective solution to this problem, our key idea is to consider inductive heap predicates as *logic programs* in a Prolog-style language, and concrete memory graphs as *logic facts*. This perspective allows us to cast predicate synthesis as a classic instance of Inductive Logic Programming (ILP)—synthesising a logic program by generalising concrete examples and facts about concrete data instances that are also defined as logic programs [16, 45]. That said, to harness the power of ILP for synthesising SL predicates, we have to overcome the following two challenges:

C1 For *effectively* learning logic programs, ILP requires both *positive* and *negative* examples; the representative (*i.e.*, non-trivial) negative examples are essential to ILP (*cf.* Sec. 2.2) but difficult to acquire without a human in the loop (*cf.* Sec. 5 for a discussion).

C2 Synthesis of predicates in Separation Logic with arbitrary data constraints features a large search space, making it difficult to be *efficient*.

To address the challenge **C1**, we propose a novel *positive-only learning* approach to infer the *most specific* logic predicate from a set of positive examples by incorporating as many of the available (yet non-redundant) pre-defined constraints as possible. This is achieved by (1) eliminating logically *redundant* restrictions featured in generated predicate candidates (as, *e.g.*, the last one in the series `A < B`, `B < C`, `A < C`) and (2) by introducing the notion of *specificity* that selects a locally-optimal inductive SL predicate from a set of candidates with no redundancies.

Having phrased the inductive heap predicate synthesis as a search for a local optimum, we inevitably face its large computational complexity, which brings us to the challenge **C2**: *efficiency* of the search. We address this challenge by exploiting the nature of our target domain, *i.e.*, Separation Logic. The key insight that allows us to prune many non-viable candidates is to perform early detection of *invalid* combinations of heap constraints, *e.g.*, those implying that the same symbolic heap location can be `null` and not `null` at the same time. Combined, our solutions to the challenges **C1** and **C2** deliver an approach for effective and efficient inductive synthesis of inductive heap predicates from concrete memory graphs.

In summary, this work makes the following contributions:

- The first inductive synthesis approach of inductive heap predicates with arbitrary data constraints, requiring only positive examples of memory graphs, achieved by (1) positive-only learning for ILP, (2) exploiting the domain-specific properties of SL.
- Sippy—an automated tool for synthesising SL predicates from memory graphs, showcasing the *effectiveness* of our approach for synthesising non-trivial predicates in a series of benchmarks.
- Grippy—a memory graph generator that can automatically produce positive examples for the predicate synthesis via Sippy, given data-manipulating programs, which it uses as test oracles.
- Demonstration of utility of Sippy by (a) learning the predicates for verification from real-world heap-manipulating programs by obtaining memory graphs automatically via Grippy, and (b) using the synthesised predicates for automated deductive synthesis.

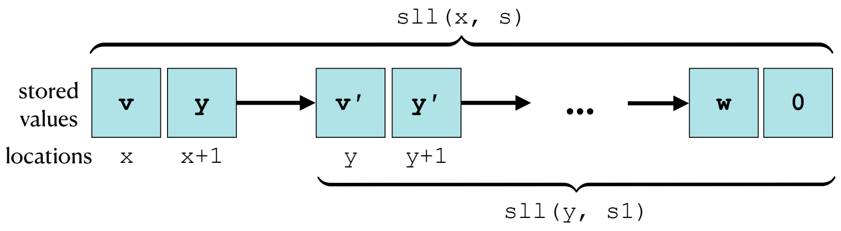


Fig. 1. Memory layout of a singly linked list.

2 Overview and Key Ideas

In this section, we provide a brief outline of the basics of Separation Logic (SL) and its inductive predicates. Next, we explain how to use the existing ILP system Popper [18] to synthesise such predicates from *both positive and negative* examples, and how we handle with learning from positive examples only. We conclude with the high-level workflow of our SL predicate synthesiser Sippy.

2.1 Inductive Predicates in Separation Logic

Consider a schematic memory layout depicted in Fig. 1 corresponding to a singly linked list (SLL). The list has a recurring structure with each of its elements represented by a consecutive pair of memory locations (the "head" one referred to by a pointer variable x), the first one storing its data value (or *payload*) v and the second containing the address y of the tail of the list. Knowing these shape constraints, the entire list can be traversed recursively by starting from the head and following the tail pointers.

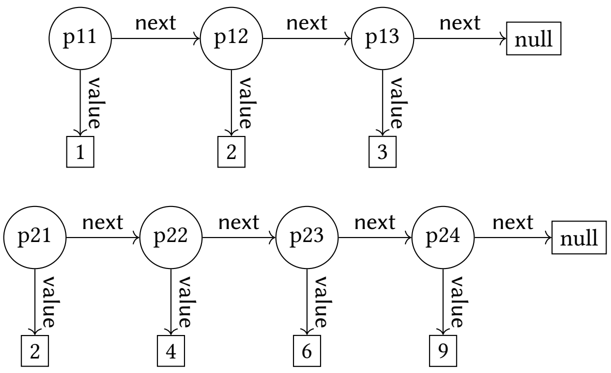
The idea of defining the repetitive shape of a heap-based linked structure, such as SLL, is precisely captured by Separation Logic and its inductive (*i.e.*, well-founded recursive) predicates. One encoding of an SLL heap shape via the SL predicate sll is given below:

```
pred sll(loc x, set s) where
  | x = 0 ⇒ { s = {}; emp }
  | x ≠ 0 ⇒ { s = {v} ∪ s1; x ↦ v * (x+1) ↦ y * sll(y, s1) }
```

The predicate sll is parameterised by a location (*i.e.*, pointer variable) x and a payload set of the data structure s; it holds true for any *heap fragment* that follows the shape of a linked list (and contains no extra heap space). What exactly that shape is, is defined by the two *clauses* (*a.k.a.* constructors) of the predicate. The first one handles the case when x is a null-pointer, constraining the payload set s of the list to be empty ({}); the same holds for the list-carrying heap—which is denoted by a standard SL assertion emp.[1] The second clause describes a more interesting case, in which x is not null, and so the payload can be split to an element v and the residual payload s1 (for simplicity of this example, we assume that all elements of the list are unique). Furthermore, the heap carrying the list is specified to have two consecutive locations, x and x + 1, storing v and some (existentially quantified) pointer value y, as denoted by the SL *points-to* notation ↦. Finally, the rest of the SLL-carrying heap is the recursive occurrence of the same predicate sll(y, s1) (with different arguments), thus replicating the recursive structure of the layout from Fig. 1.

The logical connective * appearing in the second clause of the sll predicate is known as the *separating conjunction* (sometimes pronounced "and separately") and is the main enabling feature of Separation Logic [50]. It implicitly constrains the symbolic heaps it connects in a spatial assertion to have *disjoint* domains. Specifically, in this example it implies that the heap fragment captured

[1]For simplicity, our examples use mathematical sets to encode the data payload, assuming uniqueness of the elements, instead of, *e.g.*, an algebraic list. This is not a conceptual or practical limitation of our approach, as we will show in Sec. 6.1.



```
pos(srtl(p11,[1,2,3])). pos(srtl(p21,[2,4,6,9])).

% Encoding of the first memory graph from Fig.2
next(p11, p12). next(p12, p13). next(p13, null).
value(p11, 1).  value(p12, 2).  value(p13, 3).

% Encoding of the second memory graph from Fig.2
next(p21, p22). next(p22, p23). next(p23, p24). next(p24, null).
value(p21, 2).  value(p22, 4).  value(p23, 6).  value(p24, 9).
```

Fig. 2. Positive examples of sorted list heap graphs, with the corresponding logic encoding.

by `sll(y, s1)` does not contain memory locations referred to by either `x` or `x+1`. Such disjointness constraint is what makes it possible to avoid extensive reasoning about aliasing when using SL specifications, making them *modular*, *i.e.*, holding true in the context of any heap that is larger than what is affected by the specified program.

2.2 From Memory Graphs to Heap Predicates

Our goal is to synthesise inductive SL predicates from examples of concrete memory graphs. To do so, we phrase both SL predicates and the memory graphs that satisfy them in terms of Logic Programming. For example, the Prolog predicate below defines a sorted singly linked list:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X,Y), value(X,V), srtl(Y,SY), min_set(V,S), insert(SY,V,S).
```

The predicate above defines a sorted singly linked list by enhancing the ordinary singly linked list predicate with the constraint `min_set(V, S)` that states that the value `V` is equal to the smallest value in the set `S`. The `insert(S1, V, S)` and `empty(S)` (*i.e.*, `s == {v} ++ s1` and `s == {}` in the SLL example) are defined using Prolog built-in predicates that correspond to ordinary functions in set theory. Other Prolog-style predicates used in the synthesised SL solutions are data-structure specific and are extracted from the user-provided memory graphs. We leave till later (Sec. 4.2) the issue of ensuring that a Prolog predicate is also a *valid* SL predicate in the sense that it does not use SL connectives in a contradictory way, allowing one to derive falsehood from its definition.

The top of Fig. 2 shows two memory graphs of sorted lists that can be used to synthesise `srtl()`. For a more natural representation in terms of Logic Programming, we use Java-style naming of structure components, *i.e.*, fields such as value and next instead of C-style integer pointer offsets; these fields provide the data-specific predicates (*i.e.*, `next()` and `value()`) to the synthesiser. In the bottom of Fig. 2, we provide the corresponding logic representations of the inputs to the synthesiser, consisting of positive examples (*i.e.*, instances of the sought predicate that are expected to be true), and the background knowledge (*i.e.*, encoding of the corresponding memory graphs) that should be used to derive those examples using predicate candidates. Given all this information, a synthesiser should be able to generate the predicate `srtl()` that satisfies the positive examples. That is, using the traditional program synthesis from input-output pairs as an analogy [27], the facts in background knowledge (*e.g.*, `next(p11, p12)`) are inputs, the examples (*e.g.*, `pos(srtl(p11, [1,2,3]))`) are the outputs, and the solution is the program (*i.e.*, the predicate) to be synthesised.

2.3 Predicate Synthesis via Answer Set Programming

We observe that the synthesis of SL predicates can be regarded as a Logic Programming synthesis task, studied extensively in the field of Inductive Logic Programming (ILP) [17, 45]. In ILP, the synthesis of definition is done by generating hypotheses (*i.e.*, predicates) and testing them against the provided examples. Efficient generation of hypotheses in ILP is typically implemented using Answer Set Programming (ASP) [23, 34], a constraint solving-based search-and-optimisation methodology that allows for effectively pruning the search space of candidate definitions and is used in many state-of-the-art ILP systems: ASPAL [13], Popper [18], and ASPSynth [5].

To see how ASP can be used for synthesising logic predicates, we first provide a brief introduction to its principles using very basic examples. Considered a declarative logic programming paradigm, ASP can be regarded as a syntactic extension of Datalog, but with a different semantics called *stable model semantics* [26]. The output of an ASP program is a set of *models* (*i.e.*, so-called answer set) that satisfy the program constraints. A (normal) ASP program consists of a set of *clauses* that are composed of a head (on the left of ⟵) and a body (on the right of ⟵) as:

$$a \;\longleftarrow\; b_1, \ldots\; b_m,\; \neg\, c_1,\; \ldots,\; \neg\, c_n.$$

which can be read as "if $b_1, \ldots\; b_m$ are true and $c_1,\; \ldots,\; c_n$ are false, then a is true". The statements $a,\; b_i,\; c_i$ are called *literals* and are declared in the format of `pred_name(X1, ..., Xn)` (*i.e.*, a predicate with arity n). A clause is called an *integrity constraint* when its head (*i.e.*, the statement on the left-hand side of ⟵) is empty, which means it is inconsistent if the body is true; a clause is called a *fact* when its body is empty, which means the head is always true.

Instead of describing the formal definition of a stable model, we show simple examples of ASP program and the corresponding answer sets below:

No.	ASP Program	Answer Sets
1	`a :- b. b.`	`{a,b}`
2	`a :- not b. b.`	`{b}`
3	`a :- not b. b :- not a.`	`{a},{b}`
4	`a :- not b. b :- not a. :- a.`	`{b}`

The arity of the literals `a` and `b` is 0, and `:-`, `not` in the programs mean ⟵ and ¬ correspondingly. The programs in the table above and their answer sets should be interpreted as follows.

- Program 1 is a simple program with a rule (general clause) `a :- b` and the fact `b` postulating that `b` is true. The answer set is `{a,b}` meaning that `a` and `b` can be true together, given the constraints.
- Program 2 is similar to Program 1, but with the rule `a :- not b` instead, which means `a` is true when `b` is false. The answer set is `{b}`, no clause is making `a` true.
- Program 3 is a program with two rules. The answer set is `{a},{b}` because `a` is true when `b` is false, and `b` is true when `a` is false, so the answer set is the combination of the two cases.
- Program 4 is extended from Program 3 with another clause, that is an integrity constraint `:- a` forcing `a` to be false. The answer set is `{b}` because `b` is true when `a` is false, and the program is consistent only in this case (in contrast with Program 3).

As Program 4 demonstrates, the integrity constraint can be used to prune the answer sets—a very useful feature for synthesis tasks (more discussion on ASP versus SMT is in Sec. 7).

Each program above can be regarded as an *enumeration in the powerset* of the set with two elements, `a` and `b`, returning *all sets* that satisfy the relations (*i.e.*, the ASP clauses) between the elements. An ILP system, such as Popper [18], relies on an ASP solver to encode the enumerative search among all possible combinations of literals to synthesise logic predicates.

As a concrete example of ILP via ASP, consider synthesising the definition of a predicate `plus_two(A, B)` using six literals: `succ(A, A)`, `succ(A, C)`, `succ(B, A)`, `succ(B, B)`, `succ(B, C)`, and `succ(C, B)` to build the body of the predicate, with examples `plus_two(1,3)` and `plus_two(2,4)`. An ASP-based synthesiser would try to find a definition of `plus_two(A, B)` as a suitable subset of all their $2^6 = 64$ possible combinations. While doing so, it would also make use of the natural restrictions that can be encoded as integrity constraints, such as (1) no free variable is allowed in the body (hence `{succ(A, B), succ(C, B)}` is not a valid answer set because `C` is free), and (2) all input variables `A` and `B` should appear in the body (hence `{succ(B, C)}` is not a valid synthesis candidate). As we will show, such constraints are also useful for encoding the domain-specific knowledge about validity of SL predicates, and can be efficiently solved by ASP solvers.

Moreover, the incrementality of ASP solvers make it possible to constrain the search space continuously [25]. For instance, assume the following hypothesis is obtained during the search:

```
plus_two(A, B) :- succ(A, A), succ(B, B).
```

After testing it by Prolog against the examples, Popper finds that none of the provided positive examples can be derived using this solution. As the result, other hypotheses that are more *specialised* (*i.e.*, more constrained in the bodies) than it, such as the definition of `plus_two()` below.

```
plus_two(A, B) :- succ(A, A), succ(B, A), succ(B, B).
```

will also entail no positive examples. To this end, with the help of ASP, a classic ILP performs search for a candidate hypothesis that passed all tests and has the smallest size (*i.e.*, number of literals in the predicate). Such *optimal* hypothesis for our example is the synthesised solution:

```
plus_two(A, B) :- succ(A, C), succ(C, B).
```

2.4 Synthesis without Negative Examples

The classic ILP comes with an important limitation: in general, it requires both *positive* and *negative* examples to learn a predicate. As we explain below, the need for the latter kind of examples makes it challenging to employ ILP *as-is* as a pragmatic approach for synthesising SL predicates.

Why Negative Examples are Necessary in ILP. Let us get back to our examples with synthesising a sorted singly linked list predicate from positive examples of memory graphs in Fig. 2. With the conventional ILP, the learned hypothesis by Popper is as follows, and it is not what we need:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X, Y), value(X, V), insert(SY, V, S), srtl(Y, SY).
```

The learned hypothesis does not define a sorted list, but an ordinary (unsorted) singly linked list. The reason is: in the absence of the negative example, this is a consistent hypothesis that is smaller in size than the correct definition of `srtl`. So if we want to learn the correct predicate, we need to provide negative examples that are inconsistent with the incorrect hypothesis, such as

```
neg(srtl(p11, [1,2,3])).
% Encoding of a negative example
next(n1, n2). next(n2, n3). next(n3, null).
value(n1, 2). value(n2, 1). value(n3, 3).
```

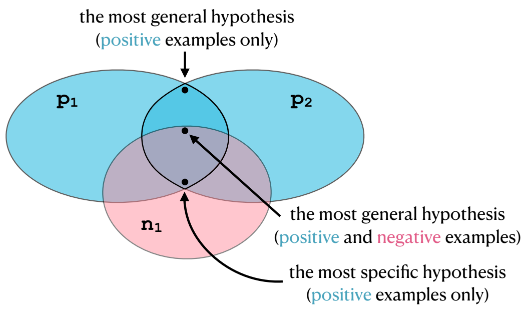

the most general hypothesis
(positive examples only)
p1
p2
n1
the most general hypothesis
(positive and negative examples)
the most specific hypothesis
(positive examples only)

Fig. 3. The effect of positive and negative examples on search.

which is a singly linked but not sorted list. To summarise, when performing synthesis via classic ILP, negative examples are necessary to avoid the predicate being *too general*.

Challenges in Obtaining Negative Examples. What makes things worse is that ILP systems rely on *representative* negative examples to correctly prune the generality, which are hard to obtain automatically. The difference between positive and negative examples is that, a good set of positive examples (**Pos**) only needs to guarantee that all instances follow the predicate ($\mathcal{P}$), while a good set of negative examples (**Neg**) need to be much more elaborated, so it could cover any possible way in which the predicate can be wrong. This difference is expressed by the following quantifications:

$$\forall \mathbf{e}^{+} \in \mathbf{Pos}, \mathcal{P}(\mathbf{e}^{+}) \quad \text{vs.} \quad \forall \mathcal{P}' \subset \mathcal{P}, \exists \mathbf{e}^{-} \in \mathbf{Neg}, \mathcal{P}'(\mathbf{e}^{-}) \land \neg\mathcal{P}(\mathbf{e}^{-})$$

That is, unlike a good positive example set, which is only quantified over the examples, a good negative example set is quantified over the *predicates* and the examples, which makes it much harder to achieve. As a concrete example of this phenomenon, imagine learning a predicate for balanced binary trees. A good set of negative examples would contain instances where (a) the height of the left subtree is too large, (b) the height of the right subtree is too large, (c) the imbalance manifests recursively in both left and right subtrees. Without all these rather specific negative instances, it is possible to learn a predicate with a constraint on the subtrees `height(Left) <= height(Right) + 1`, which is not wrong but is imprecise. This is not just an issue for SL predicates domain, but also for general logical learning, witnesses by the fact that in existing ILP benchmarks [18, 62] and specification mining framework [44] high-quality negative examples are often crafted manually.

This state of affairs brings us to the two key novel ideas of this work that enable efficient synthesis of SL predicates only from positive examples.

2.4.1 Key Idea 1: Learning with Specificity. As discussed above, without negative examples a solution delivered by Popper, while valid, may not be specific enough. To provide more intuition on the space of possible design choices in finding the best solution, together with the reason why positive-only learning is possible, let us consider the illustration in Fig. 3. The "up"/"down" in this figure (informally) means "more general/specific", where the top/bottom are constant True/False (*i.e.*, the lattice is defined by subsumption [46]). Providing two positive examples, `p1` and `p2`, restricts the search space for the solution hypothesis to the intersection of their own spaces, with the most general one chosen as the solution. Adding a negative example `n1` provides more restrictions, thus allowing for more specific most-general solution. From this diagram, one can see that, even without a negative example, we can have a *tighter* [4] solution (*i.e.*, with stronger restrictions given the same number of clauses) if we consider not the most general, but the most specific candidate in the intersection of the search spaces defined by `p1` and `p2` (generally, positive examples only).

Therefore, the basic idea of our positive-only learning is: to learn *the most specific* predicate admitting all provided examples. The only problem is: what is the definition of "specificity"? As the opposite of "generality" (the program with the smallest number of constraints), it is not practical to take the *largest* hypothesis as the most specific, as it would lead to *redundant* constraints. As an example, consider the following valid formulation of the sorted linked list predicate that requires, in its second clause, that `T = SY ∪ {V}` and `S = T ∪ {V}`:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X, Y), value(X, V), srtl(Y, SY), insert(SY, V, T), insert(T, V, S).
```

Clearly, the last conjunction `insert(T, V, S)` is redundant and can be removed because of the properties of the `insert(...)` predicate.

To eliminate such candidates with redundancies, our approach for positive-only learning encodes intrinsic logical properties of customised predicates to *minimise* the generated SL predicates. Our tool comes with a pre-defined collection of properties of predicates for common arithmetic (*e.g.*, calculation, comparison) and set operations (*e.g.*, insertion, union) that are included into the synthesis automatically. More customised predicates can be added by the user (with additional clause minimisation rules). After performing the minimisation hinted above (detailed in Sec. 3.1), we define the *specificity* of a predicate candidate based on its size *w.r.t.* other available candidates (*cf.* Sec. 4.3). The solution that is locally-optimal (*i.e.*, the strongest in the search space) will be adopted as the most specific predicate that is implied by all the positive examples.

2.4.2 Key Idea 2: Separation Logic-Based Pruning. The domain of our synthesis, *i.e.*, Separation Logic, provides effective ways to prune the search space and accelerate the synthesis process. Postponing the detailed explanation of the optimisations until Sec. 4, as an example, consider an important property the separating conjunction stating that the fact $x \mapsto a * y \mapsto b$ implies $x \neq y$ because of the disjointness assumption enforced by $*$. This property can be encoded as a pruning strategy via ASP integrity constraints (Sec. 2.3) that are generated by our tool for each synthesis task. Even for a *doubly linked list*, one of the simplest predicates in our benchmark (*cf.* Sec. 6.1), without such optimisations, the synthesis time is increased from 3 to 339 seconds; the synthesis of more complex predicates does not terminate in 20 minutes without SL-specific pruning.

2.5 Automatically Generating Positive Examples

So far, we assumed that the positive examples are provided by the user, in the format of memory graphs. In practice, one may expect that such examples can be obtained in a more automated way, *e.g.*, from the available programs that manipulate with the respective data structures. For instance, an existing work on shape analysis [38] uses a debugger for extracting memory graphs from a program's execution, with the assumption that (1) the user indicates the line of code to extract the memory graph, and (2) an input for the program is provided. Unfortunately, this rules out a large set of programs that *expect* a data structure rather than generate one: without a suitable input we simply cannot run them to obtain a graph, and constructing such input is a task not much easier than encoding a memory graph manually.

To address this issue, we implemented Grippy—a tool that can automatically generates positive example in the form of arbitrary valid memory graphs of a data structure from the program that manipulates with the structure, without requiring the user to provide any input. The only assumption we make is that the program in question is instrumented with test assertions, which can be used to filter out the invalid memory graphs from the randomly generated ones. We further show in Sec. 6.2.1 that Grippy can effectively generate input graphs for synthesising SL predicates from real heap-manipulating programs, reducing the specification burden for proving their correctness.

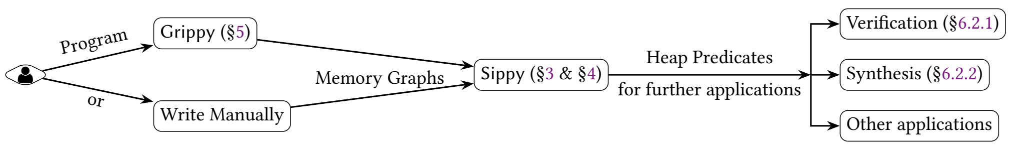


Fig. 4. The Workflow of Sippy

2.6 Putting It All Together

We implemented our algorithm for inductive synthesis of SL predicates as the core of our tool called Sippy. Fig. 4 shows the high-level workflow of Sippy. Starting from the left, the user can provide positive examples (*i.e.*, structure-specific heap graphs) for the synthesiser by either using our graph generator Grippy on programs that expect the data structure instance (*cf.* Sec. 5), or by manually writing them (*e.g.*, in the style of Story-Board Programming [61]). Given the graph examples, Sippy synthesises an inductive predicate definition, which can be further used for program verification in SL-based verifiers (Sec. 6.2.1), for program synthesis (Sec. 6.2.2), or by any other SL-based tool.

3 Positive-Only Predicate Learning

In this section, we describe our approach for learning predicates from positive-only examples not specific to SL predicates. To make the learning effective, we introduce the idea of predicate clause *minimisation*, then explain how our specificity-based positive-only learning works by showing its difference from the standard ILP systems [14, 18].

3.1 Normalising the Positive-Only Learning

As discussed in Sec. 2.4.1, without negative examples, adopting the smallest-size hypothesis, as done in many program synthesisers [32, 39], can lead to finding solutions that are too imprecise. What about selecting the candidate with the largest size instead? As shown by the sorted list (`srtl(X, S)`) example, more literals in the body do not mean the predicate is more specific if the body contains *redundant* constraints among variables. Therefore, we first define a normalisation procedure to remove the redundancy in the predicate clauses with the help of the logical *entailment* (denoted as $\models$ for Prolog clauses known as *definite clauses* in first-order logic).

Definition 3.1 (Clause Minimisation). *Let* **numlit**(h) *denotes the number of literals in the body of a clause* h*, and* $A \iff B$ *denotes* $A \models B$ *and* $B \models A$*. A predicate clause* h *is eliminated* iff $\exists h_0,\ h_0 = \arg\min_{X \in S} \mathbf{numlit}(X) \land \mathbf{numlit}(h_0) < \mathbf{numlit}(h)$*, where* $S = \{X \mid X \iff h\}$*.*

That is, if for any clause h there is a shorter clause h_0 equivalent (defined by entailments) to h, h is eliminated in the synthesis. Such minimisation is naturally encoded by ASP (see the appendix of this paper's extended version [69]). Our experiments show that enumerable (also decidable) entailment rules are effective for the synthesis of SL predicates within finite clause length (Sec. 6.1). Note that the minimisation is essentially a pruning for unnecessary clauses: we do not handle predicate-level redundancy (*a.k.a.* Horn literal minimisation [66]) as it is NP-hard in general [29].

3.2 An Algorithm for Positive-Only Learning

Our approach to learning predicates from positive-only examples can easily make use of components from the existing ASP-based ILP systems [14, 18], seamlessly. In this section, we describe both the basic ILP learning loop by ASP and the procedure of our positive-only learning.

Algorithm 1 (with the light grey cover) shows the pseudocode of classic ILP learning loop, which follows the "generate-test-constrain" approach. It takes (1) background knowledge `bk`, (2) positive

Algorithm 1 The positive-only learning (POL) algorithm v.s. the standard ASP-based ILP

```
Require: Search space: initial constraints in_cons, size limit max_size
Require: User inputs: background knowledge bk, pos/neg examples exs
 1: procedure POL OR ILP (bk, exs, in_cons, max_size)
 2:     cons = in_cons, size = 1, sol = True
 3:     while size≠ max_size do
 4:         h = CLINGO(cons, size)                                    ▷ generate
 5:         if h is UNSAT then
 6:             size += 1
 7:         else
 8:             outcome = PROLOG(h, exs, bk)                          ▷ test
 9:             if outcome is complete and consistent then
10:                 sol = h
11:                 break
12:             cons += pruning(h, outcome)                           ▷ prune
13:             if outcome is complete and sol ⊭ h then
14:                 sol = update_sol(h, comparable(h, sol))
15:             cons += new_pruning(h, outcome)                       ▷ prune (updated)
16:     return sol
```

and negative examples exs, and (3) learning bias defining parameters of the search max_size, such as the maximum size of a predicate to be searched and other customisable parameters (*e.g.*, whether to enable mutually-recursive definitions), summarised as in_cons. After encoding the iterative deepening search problem into an answer set program, it first uses the ASP solver Clingo [24] to generate a hypothesis (line 4), and then uses Prolog to test it against the provided examples (line 8). A hypothesis is *complete* if it entails all positive examples, and *consistent* if it entails no negative examples. ILP returns a hypothesis as the solution if it is complete and consistent (line 11); otherwise, it *prunes* the search space using the test outcome (line 12) and continues the search. The test-based pruning works as follows. Whenever hypothesis testing shows that not all positive examples are true (incomplete), the pruning on *specialisation* is applied; whenever part of the negative examples are true (inconsistent), the pruning on *generalisation* is applied—ideas of the pruning are illustrated in Sec. 2.2 and are detailed in [18].

For the case of *positive-only* learning (*cf.* Algorithm 1 with the dark grey cover), we follow the approach from the work on *specification synthesis* [52] to output a set of non-comparable (*i.e.*, the element in the set is not entailed by any other) solutions as a conjunction: there are possibly several solutions satisfying the positive examples where none of them is more specific than the others, so the conjunction of them is the most specific specification. The main difference from classic ILP starts from line 13, when a hypothesis h is complete and is not entailed by the existing solution set sol, the solution set is updated to adding h to the set, together with removing all solutions in sol that are comparable (semantically more general in our case) to h (line 14). The set of most specific solutions in the search space is returned when the search is exhausted (line 16). The advanced pruning procedure (line 15) is enabled by domain knowledge (*i.e.*, Separation Logic predicates in our case—*cf.* Sec. 4.2).

We conclude this section by proving the soundness of positive-only learning in Algorithm 1.

THEOREM 3.1. *The hypothesis set returned by the positive-only learning in Algorithm 1 contains all non-comparable, most specific predicates that is complete (*i.e.*, entailing all examples) in the search space defined by the algorithm's initial constraints (in_cons) and the size limit (max_size) parameters.*

$$
\begin{aligned}
\langle\text{predicate}\rangle &::= \langle\text{main\_pred}\rangle \mid \langle\text{main\_pred}\rangle\langle\text{invented\_pred}\rangle* \\
\langle\text{main\_pred}\rangle &::= \langle\text{base\_clause}\rangle(\lceil\text{main\_head}\rfloor) \mid \langle\text{rec\_clause}\rangle(\lceil\text{main\_head}\rfloor)* \\
\langle\text{invented\_pred}\rangle &::= \langle\text{base\_clause}\rangle(\lceil\text{inv\_head}\rfloor) \mid \langle\text{rec\_clause}\rangle(\lceil\text{inv\_head}\rfloor)* \\
\langle\text{base\_clause}\rangle(H) &::= H(\texttt{This}, \langle\text{args}\rangle) \longleftarrow \langle\text{base\_lit}\rangle*, \langle\text{pure\_lit}\rangle* \\
\langle\text{rec\_clause}\rangle(H) &::= H(\texttt{This}, \langle\text{args}\rangle) \longleftarrow \langle\text{pointer\_lit}\rangle*, \langle\text{rec\_lit}\rangle*, \langle\text{pure\_lit}\rangle* \\
\langle\text{literal}\rangle(R) &::= R(\langle\text{args}\rangle) \\
\langle\text{base\_lit}\rangle &::= \langle\text{literal}\rangle(\lceil\text{base\_pred}\rfloor) && \texttt{\% Pre-defined for spatial relations} \\
\langle\text{pure\_lit}\rangle &::= \langle\text{literal}\rangle(\lceil\text{pure\_pred}\rfloor) && \texttt{\% Pre-defined for pure relations} \\
\langle\text{pointer\_lit}\rangle &::= \lceil\text{domain}\rfloor(\texttt{This}, \langle\text{var}\rangle) && \texttt{\% Extract from the memory graphs} \\
\langle\text{rec\_lit}\rangle &::= \langle\text{literal}\rangle(\langle\text{head}\rangle) \\
\langle\text{args}\rangle &::= \langle\text{var}\rangle \mid \langle\text{var}\rangle, \langle\text{args}\rangle \\
\langle\text{var}\rangle &::= \texttt{X1} \mid \texttt{X2} \mid \texttt{...} \mid \texttt{This} \\
\langle\text{head}\rangle &::= \lceil\text{main\_head}\rfloor \mid \lceil\text{inv\_head}\rfloor && \texttt{\% From the task or randomly generated}
\end{aligned}
$$

Fig. 5. The grammar of the SL predicates, in basic Backus–Naur form (BNF), extended with (1) meta-variables $(\cdot)$ for specialising the symbols, and (2) pre-defined atoms denoted by $\lceil X \rfloor$ (with comments of their origins).

Proof. By induction on the size limit `max_size` of the predicate: when `max_size` is 0, there is no predicate hypotheses, so `True` (the "always true predicate") is the only most specific one. Then assume that the theorem holds for `max_size` n, *i.e.*, `sol_i` is the most specific hypothesis set; we prove it for `max_size` $n + 1$.

When `max_size` is $n + 1$, based on the while loop in Algorithm 1, the search space for $n + 1$ is the search space for n plus when `size` is $n + 1$. By the induction hypothesis, `sol_i` is the most specific hypothesis set in the search space for n, and the output `sol` is either `sol_i` or containing more specific predicates of space $n + 1$ with all comparable predicates removed. Therefore, `sol` is the most specific hypothesis set in the search space with $n + 1$ as `max_size`.

□

4 Separation Logic Predicate Synthesis via Sippy

Having described the enhanced *general-purpose* predicate synthesis algorithm from positive-only examples, we now show how to instantiate it for synthesis of inductive SL predicates and improve the efficiency of the search algorithm by exploiting domain-specific SL insights. We further discuss the SL-validity of the synthesised predicates and the completeness of the search algorithm.

4.1 SL Predicates: Basics and Intricacies

We define the space of SL predicates in a way standard for Syntax-Guided Synthesis (SyGuS) [2]. The grammar of the SL predicates is shown in Fig. 5. An SL predicate is either having a shape with a single main predicate, or shaped by a main predicate together with a set of invented *auxiliary* predicates, which are required in the case of nested linked structures.

Specific to the predicates, both main predicate and invented predicates consist of the base and recursive clauses, where the base clause is the one that does not have any recursive calls, and the recursive clause is the one that has recursive calls. The head literal (*i.e.*, before $\longleftarrow$) in each clause has a fixed argument `This` that denotes the base address of the data structure (similar to the *this* reference in object-oriented programming). The body literals (*i.e.*, after $\longleftarrow$) in the clauses are defined in terms of different predicates: the base (and pure) predicates are pre-defined, but extensible, to capture the spatial relation among the pointer for the base clause (the pure constraints among variables in clauses, respectively); the domain predicates describe the points-to relations can be obtained from the memory graphs; the recursive predicates are the recursive calls to the main or invented predicates.

Three aspects in the grammar in Fig. 5 contribute to the infinite synthesis search space: (1) the length of clauses, (2) the number of sub-clauses for each predicate, (3) the arity of the invented predicates. For our task, we noticed that predicates for real-world structures rarely require more than 10 literals in their bodies; two sub-clauses for each predicate are sufficient to capture the common structures; and the arity of the invented predicates is set to be not more than the arity of the main predicates. Such bounds for hypothesis space are common in almost all synthesis-by-example tools (*e.g.*, [18, 39, 59]), not only to make the synthesis tractable, but also to avoid overfitting [51] (*e.g.*, a predicate disjointing facts of all examples).

Below, we discuss two challenges in make SL predicate synthesis effective and efficient, together with how we address them in Sippy.

4.1.1 Semantic-Based Pruning. In most existing syntax-guided synthesisers [2, 18, 59], the search is accelerated by pruning of the hypothesis search space by employing the general *syntax* restrictions. Other than limiting the syntax, we apply the following *semantic* properties' restriction of Separation Logic predicates to the search.

(1) *Basic reachability*: no points-to relation appears in the body other than the ones from the `this` pointer. Thus, the clause `p(X, Y) :- next(X, Y), next(Y, Z), ...` is not allowed as a candidate, because we expect all locations in the body to be accessible from `this` via fields.
(2) *Basic assumptions*: the base (non-recursive) clause restricts `this` pointer to either be **`null`** or to equal to another pointer parameter variable. *E.g.*, `p(X, Y) :- nullptr(X), ...` is allowed, but `p(X, Y) :- next(X, Y), ...` cannot be a base clause.
(3) *Restricted use of* **`null`**: if a variable `X` is a null-pointer (denoted by `nullptr(X)`), no more `X` occurs in the clause. *E.g.*, the clause `p(X, Y) :- nullptr(X), next(X, Y)` is not allowed.
(4) *Quasi-well-founded recursion of payload*: the pure argument passed to a recursive call should (non-strictly) decrease. *E.g.*, for a clause `p(X, S) :- next(X, Z), p(Z, S1), ...`, the set `S` should contains `S1`. This is a common assumption in recursive program synthesis [1, 39].
(5) *Heap functionality*: points-to relations of the same field should not target different locations. *E.g.*, a candidate clause cannot be `p(X, Y) :- next(X, Z), next(X, Z1), ...`.

This list of search constraints represents a combination of the properties implied by SL semantics (in a Java-style field-based memory model) as well as by common properties of data structures, which are essential for the efficient search of SL predicates. The exact encodings of these constraints in ASP are provided and explained in the appendix of the extended version of this paper [69].

4.1.2 Free Variables and Auxiliary Placeholders. Free variables are common in SL predicates, *e.g.*, the (implicitly existentially-quantified) location `Y` in the base clause of the doubly linked list below:

```
dll(X, Y) :- nullptr(X).
dll(X, Y) :- next(X, Z), prev(X, Y), dll(Z, X).
```

Unfortunately, completeness guarantees of pruning discussed in Sec. 3.2 do not hold for predicates with free variables in the sense that a complete (*i.e.*, valid) hypothesis with free variables might be wrongfully pruned during the search [18, §4.5]. To address this problem, we introduce *auxiliary placeholders* into the search as a way to express predicate clauses with free variables. For example, the following doubly linked list predicate can be regarded the same as the one above with `anypointer()` placeholder, and *can* be synthesised.

```
dll(X, Y) :- nullptr(X), anypointer(Y).
dll(X, Y) :- next(X, Z), prev(X, Y), dll(Z, X).
```

On a technical level, this requires adding an ASP constraint (shown in supplementary material) that forces the parameter of the placeholder predicate (`Y` here) to appear *twice* in the whole clause, so it could be later translated into a single occurrence of a free variable.

4.2 Ensuring SL Validity in Prolog

An astute reader can notice that the validity of the synthesised predicates is not immediate due to our treatment of Prolog clauses as SL assertions: the conjunction in Prolog does not guarantee the *separating conjunction* ($*$) in the SL sense. As an example, consider the following simplified Prolog predicate for binary trees:

```
bi_tree(X) :- nullptr(X).
bi_tree(X) :- t1(X, L), t2(X, R), bi_tree(L), bi_tree(R).
```

In this case, an instance of `bi_tree(X)` being evaluated to be *true* in Prolog can imply *false* under SL semantics that enforces heap disjointness: considering a memory graph with two nodes

```
t1(n1, n2). t2(n1, n2). t1(n2, null). t2(n2, null).
```

so that the graph fact `bi_tree(n1)` is provable in Prolog, but the clauses `bi_tree(L)` and `bi_tree(R)` are *non-disjoint*. Notice that, in our inductive synthesis setting, this situation would correspond to having *multiple* occurrences of the same points-to fact in a memory graph representing a positive example for the predicate, but should not be allowed by the definition of separating conjunction.

To avoid this source of unsoundness, we use a straightforward solution that enforces such separating conjunction semantics in Prolog: a valid SL predicate is a complete Prolog predicate where the positive examples succeed using each points-to fact *exactly* one time (a semantic property of SL assertions known as *linearity*). For the complete Prolog but invalid SL predicates, we also use the *specialisation* rule in Sec. 3.2 to prune them: if a predicate violates the linearity, then a more constrained one will also violate it; this contributes to the new pruning in line 15 of Algorithm 1.

We establish the following property of our SL-specific predicate synthesis stating that, for the predicates in Sippy's search space in Sec. 4.1, if a memory graph is provable in Prolog with linearity, then the corresponding heap is valid under SL semantics.

Theorem 4.1 (SL Validity). *Let* `F(h)` *denote the memory graph of a heap* `h`*. For any output predicate* `p(X)` *of Sippy and any heap* `h`*, the following fact holds:* $\mathtt{F(h)} \models_{\mathit{Prolog+Lin}} \mathtt{p(X)} \Rightarrow \mathtt{h} \models_{\mathit{SL}} \mathtt{p(X)}$.

4.3 The Sippy Algorithm

The only remaining step before putting all the pieces together is to select the desired predicate from the set of non-comparable solutions of positive-only learning. Even though predicates from POL can be conjuncted in general, the semantics of SL predicates following the definition in Sec. 4.1 is more restrictive and the conjunction of valid SL predicates may result in an ill-formed or a constantly false one. We found in practice that after the semantics-based normalisation from Sec. 3.1, the number of literals can serve as a *good enough* specificity metric among incomparable predicates, since containing more literals is likely to contain more information or constraints about the heap structure. Following this intuition, we define the synthesis algorithm with MAX_POL function, which obtains the largest predicate from POL as per Algorithm 1.

Algorithm 2 summarises the internal workings of Sippy. Our synthesiser takes as inputs memory graphs encoded as sets of logic facts (*e.g.*, `graph_bk`, such as `next(..)` and `value(..)` from Fig. 2), positive examples of heaps on which a predicate holds (*e.g.*, `exs` as `srtl(..)` from Fig. 2), so that the shape (matched with pre-defined shapes in Sec. 4.1) a set of ASP constraints (`graph_cons`) describing the information in the graphs (such as the arity and types of the predicates to be learned) are

Algorithm 2 The Sippy loop for inductive predicate synthesis

```
Require: memory graphs consist of graph_bk, exs
 1: procedure SIPPY(graph_bk, exs)
 2:     graph_cons, shapes = graph_info(graph_bk, exs)
 3:     max_var = max_body = 1
 4:     sol = True                                    ▹ The most general solution as initialisation
 5:     for shape in shapes do
 6:         max_size = maxsize(max_body, shape)
 7:         h = MAX_POL(graph_bk, exs, graph_cons, max_size)
 8:         if h $\prec$ sol then                        ▹ A more specific predicate is obtained
 9:             max_var, max_body = (var_of(h), size_of(h)) + $\delta$
10:             sol = h
11:         else if sol == True then                  ▹ No predicate is yet learned
12:             if max_var == upper_bound then
13:                 continue                          ▹ Try the next shape
14:             max_var, max_body += (1, 1)
15:         else
16:             break                                 ▹ No more specific predicate is found
17:     return sol
```

obtained (line 2). Two parameters (line 3) for positive-only learning (MAX_POL), (1) the maximum number of variables and (2) the maximum size of the body of a predicate clause for restricting the search space, are gradually increased during the search using the following empirical strategy: if no solution is valid (line 11), we either increase both parameters by one to enlarge the space until finding one (line 14), or the attempt on the current predicate shape fails (*i.e.*, the upper bound of the search space is reached), then Sippy will try synthesising using the next shape (line 13, *i.e.*, more auxiliary predicates); when obtaining one new better predicate than the existing, the search parameters are both increased by a parameter δ to find a possibly more specific predicate (line 9), and the solution is updated (line 10); if the learned predicate in the larger search space is not better than the previous, we stop the search and output (line 15-16). The role of the parameter δ is, thus, to provide a "margin" for the completeness of the search: it is not guaranteed that Sippy will find the most specific solution *across all possible search spaces*, but only in the search-space that is bound by the returned output's parameters *plus* δ.[2] Note that line 6 of Algorithm 2 features a function `maxsize()` that calculates the maximum size of the search space based on the maximum number of variables and the predicate shape setting.

Finally, we provide a correctness argument for Sippy. The soundness of synthesising *consistent* (*i.e.*, inhabited) and *well-formed* (*i.e.*, finitely provable) SL predicates is guaranteed by the soundness of classic ILP and Theorem 4.1. The following "local" completeness states that, given the output of Sippy, no more specific output can be discovered, *even in* the larger search space obtained by increasing the search parameters *once* by δ at the line 9 of Algorithm 2.

THEOREM 4.2 (LOCAL COMPLETENESS OF SIPPY). *If the output of Sippy is a predicate with the maximum number of variables* m *and the maximum length of the body* n*, then there is no predicate with the maximum length of the body* m' *and the maximum number of variables* n'*, where* $(m',\ n') - (m,\ n) = \delta$*, that is more specific than the output predicate.*

PROOF. Directly by contradiction and Theorem 3.1. Assume that the output solution sol is with size $(m,\ n)$, and it is not the most specific one in size $(m',\ n') = (m,\ n) + \delta$.

[2]We choose it to be (1,2) in our experiment from the natural observation: for our domain, we expect to have one body literal where the predicate is generating a new variable, and one more body literal that uses the new variable.

Algorithm 3 Generating random memory graphs and examples

```
Require: $p$: Program with assertions as tests
Require: $n$: Number of graphs to be generated
1: Initialize $GraphsAndExamples \leftarrow \emptyset$
2: for $sz \leftarrow 1$ to $max\_size$ do
3:     $cnt = 0$
4:     while $cnt < \lceil n/max\_size \rceil$ do
5:         $node_1, \ldots, node_{sz} \leftarrow init\_node()$
6:         $node_i.key \leftarrow$ random(int) for $i \in [1, sz]$
7:         $node_i.pointer \leftarrow$ random(node) for $i \in [1, sz]$
8:         if $p(node_1, \ldots, node_{sz})$ passes the tests then
9:             cnt $\leftarrow$ cnt + 1
10:            $example \leftarrow$ summarise_graph$(node_1, \ldots, node_{sz})$
11:            $GraphsAndExamples$.add(graph$(node_1, \ldots, node_{sz})$, $example$)
12: return $GraphsAndExamples$
```

Because `sol` is the output, the search space is set to be (m', n') after the loop it is obtained. With Theorem 3.1 and the assumption, there is a solution `sol`$'$ in (m', n') that is more specific than `sol`, which is a contradiction with the output `sol`. Thus, `sol` is the most specific one in (m', n'). □

5 Automated Heap Graph Generation via Grippy

As presented, Sippy requires input graphs to be provided explicitly. The final technical contribution of this work is Grippy—an auxiliary tool that allows one to *automatically generate valid structure graphs* via programs that take them as inputs and are most likely already available to the user.

Valid memory graphs can be obtained by extracting concrete heap states from the program execution, provided concrete inputs, as, *e.g.*, done by the SLING tool [38]. This approach is, however, only applicable if the program in question *generates* a data structure instance, and is problematic if it *expects* it as an input. Our idea for automated graph generating is inspired by works on fuzz-testing and uses a structure-expecting program as a *validator* for candidate graphs. Given a program with assertions, Grippy generates *arbitrary* input memory graphs, subject to basic validity constraints (*e.g.*, any node should be reachable from its root), and keeps the graphs that pass the assertions together with example facts summarised from those graphs by traversing them and accumulating their payload in a set, so they can be used as inputs for Sippy. The details are given in Algorithm 3.

One can argue that the generator defined this way can also effectively produce *negative* examples, thus removing the need for our positive-only learning. In practice, however, the number of the negative examples can be large due to the randomness of the generator that has no knowledge of the data structure (*cf.* Sec. 6.2.1). This makes it impractical to use them for ILP-based synthesis that cannot effectively discriminate good (*i.e.*, informative) negative examples from arbitrary junk.

6 Evaluation and Discussion

We implemented Sippy by extending Popper with the combined use of Python (~200 lines for modifying Popper), Prolog (14 rules for specificity of predicates, and 19 supported predicates for pure relations in first-order theories), and ASP (~200 lines for SL domain knowledge, and ~300 lines for Sippy's search space, where 46 rules are used to encode the minimisation rules of the 19 pure relations). The prototype of Grippy is implemented by ~50 lines of ASP (to generate the graphs) together with ~100 lines of Python. All experiments were conducted on an 8-core M1 MacBook Pro with 16GB RAM. The valid structure-specific input memory graphs were either manually/LLM-written, or automatically generated via Grippy as described in Sec. 5.

6.1 Benchmarking Predicate Synthesis

To assess Sippy's efficacy as a tool for heap predicate synthesis, our evaluation addresses the following research questions:

RQ 1.1 *How effective is Sippy in synthesising heap predicates?*
RQ 1.2 *What factors affect the synthesis efficiency?*
RQ 1.3 *How scalable is Sippy with respect to the input size?*

***RQ 1.1**: Effectiveness and Expressiveness.* We have assembled a set of benchmark for common and complex heap-based data structures. Tab. 1 summarises our (mostly)[3] successful case studies: 19 Separation Logic predicates synthesised by Sippy within the imposed 20 min time limit, which shows that Sippy can effectively produce SL predicates for a variety of heap-based data structures.

In our evaluation, we restricted the predicates in the search space to feature at *most one pure* theory, since for most predicates, the pure relations on variables do not influence each other's validity. For example, the *AVL tree* predicate should feature (1) a relation between the heights of a node's subtrees and (2) ordering relations on the node payload; any one of these can be encoded independently, leading to separate SL predicate definitions: `balanced(A, B)` for *balanced tree* and `bst(A, B)` for *binary search tree*. One can then *merge* those two SL predicates with identical spatial components into a united *AVL tree* `avl(A, B, C)` as follows:

(1) Merge the parameter lists by appending the pure variables;
(2) Merge pure relations with the overlapped variable renaming;
(3) Adapt the recursion in the spatial parts for the new parameter.

To show that our positive-only learning (in Sec. 3) and the SL-based optimisations (in Sec. 4.1.1) are effective, we compared the synthesis time of Sippy with the unoptimised classic ILP system Popper, Popper with the SL-based optimisations, and Sippy without the optimisations (from left to right in the table). The results show that (1) our positive-only learning effectively learns the predicates which classic ILP cannot discover, and (2) our optimisations help to reduce the synthesis time significantly for both classic ILP and POL for most cases. The only exception is #2, where the unoptimised Sippy is faster, is due to the small size of the predicate, where the cost of adding constraints for reducing the search space is larger than the benefit of the pruning.

We attempted to compare synthesis capabilities of Sippy to those of ShaPE [8], the most closely related work. Like Sippy, ShaPE synthesises SL predicates for data structures from memory graphs, allowing for positive-only synthesis without negative examples via meta-interpretive learning (MIL) [48]. However, unlike Sippy, ShaPE only allows one to synthesise a structure's shape constraints, without any restriction on the data payload or other pure constraints–this is why three data structures (SLL, BST, and a list of lists) with different pure relations are joined in Tab. 1 into one predicate per structure in the ShaPE column $\mathbf{T_S}$. It has been shown in the prior work that the MIL-based learning cannot define a complete search space for general logic programs [19], which indicates that ShaPE *cannot be extended*, even in principle, to express arbitrary data constraints, such as arithmetic ones. That is, theoretically, 9 (those not marked as NA) out of 16 predicates (with the joining taken into the account) in Tab. 1 can be synthesised by ShaPE. However, in practice, the bugs in ShaPE's learning loop implementation resulted in either (1) the search timing out (TO), (2) ShaPE terminating with an error (ERR), or (3) producing overfitted predicates, *e.g.*, 4 clauses instead of 2 for BST (WA). At the end, we were able to only synthesise 4 predicates with ShaPE.

***RQ 1.2**: Efficiency.* To understand what affects the synthesis efficiency, let us first look into the case studies with long synthesis times (more than 5 minutes): #13 (BST with list payload), #14 (balanced

[3]We will elaborate on the partially-successful binomial heap instance #15 in Sec. 6.3.

Table 1. Statistics on synthesised predicates and the comparison with other tool/setting. Columns following the predicate names are split by (1) the properties of the predicates: whether a predicate is invented (*i.e.*, nested data structure, PI), used pure relations Pure, the size of the output predicate Size (a triple of arity, the maximal number of literals, the maximal number of variables in the predicate); (2) the stats of Sippy's synthesis: the percentage of run time taken by testing the hypotheses Test%, the synthesis time of finding the expected hypothesis T_0, the synthesis time of exhaustive search in **P**ositive-only learning T_P, (3) the time T_I it took to synthesise a result when using Popper as the classic **I**LP to synthesise, the time T_{Io} when using classic **I**LP together with our **o**ptimisation to synthesise, and the time T_{Pd} of running **P**OL with SL-based optimisations **d**isabled optimisations; finally, (4) we report the runtimes T_S of ShaPE on the same tasks (but without pure relations). All times are in seconds, with TO for time-out, ERR for crashes, WA for wrong answers, and NA for not supported in principle.

No.	**Predicate**	**PI**	**Pure**	**Size**	**Test%**	T_0	T_P	T_I	T_{Io}	T_{Pd}	T_S
1	`singly linked list (payload)`	no	set	(2,8,5)	2%	<1	1	4	<1	3	<1
2	`singly linked list (length)`	no	int	(2,9,5)	1%	1	9	NA	NA	4	
3	`singly linked list segment`	no	set	(3,8,6)	5%	<1	1	TO	1	31	1
4	`doubly linked list`	no	set	(3,10,6)	1%	2	3	NA	NA	339	1
5	`doubly linked list segment`	no	set	(5,10,8)	1%	49	192	TO	10	TO	TO
6	`sorted singly linked list`	no	set	(2,9,5)	3%	<1	1	NA	NA	4	NA
7	`sorted doubly linked list`	no	set	(3,11,6)	2%	1	3	NA	NA	231	NA
8	`circular list`	no	set	(3,8,6)	23%	<1	1	18	<1	81	<1
9	`lasso list`	no	set	(3,8,6)	25%	<1	1	TO	1	TO	NA
10	`binary tree`	no	set	(2,11,8)	8%	2	13	TO	2	245	WA
11	`back-linked tree`	no	set	(3,13,9)	9%	33	176	TO	41	TO	TO
12	`binary search tree (set)`	no	set	(2,13,8)	15%	11	26	NA	NA	TO	NA
13	`binary search tree (list)`	no	list	(2,13,8)	39%	33	433	NA	NA	TO	
14	`balanced tree`	no	int	(2,12,7)	32%	34	458	NA	NA	TO	NA
15	`binomial heap (order)`	no	int	(3,14,8)	33%	37	1,087	NA	NA	TO	NA
16	`binomial heap (payload)`	no	set	(3,14,9)	2%	59	101	NA	NA	TO	NA
17	`list of lists (set)`	yes	set	(2,17,5)	1%	9	321	TO	33	TO	ERR
18	`list of lists (list)`	yes	list	(2,17,5)	1%	14	759	TO	47	TO	
19	`rose (n-ary) tree`	yes	set	(2,17,6)	1%	9	749	NA	NA	TO	ERR

tree), #15 (binomial heap with order), #18 (list of lists with list payload), and #19 (rose tree with set payload). All these case studies feature large predicate sizes, together with either nested data structures or complex pure theory (integers or lists). As witnessed by the last two columns of Tab. 1, the time Sippy takes to obtain the expected predicate is less than 1 minute, and most runtime is spent on the exhaustive search to give the guarantee of local completeness (*cf.* Theorem 4.2). It is natural for nested data structures to take long time, because the increment of search space is applied to both the synthesis of the auxiliary predicate and the whole predicate.

We also wondered about how the used pure theories affect the synthesis efficiency. To answer that question, we compare two pairs of predicates (#12 v. #13, #17 v. #18) using the same input memory graphs but different pure theories: sets v. lists. The former is more efficient: this is because the list is more expressive than set (*e.g.*, a set union with itself is eliminated, but appending a list to itself is producing a new list), so the search space is larger after the redundancy elimination. The same for integer theories: different combinations of integer operations lead to large search space.

***RQ 1.3**: Input and Scalability.* Our experiments demonstrate that 2-3 example graphs with tens of nodes in total (less than 20 node per case study on average in our benchmark suite) are sufficient to synthesise good predicates (we manually assessed the results' quality). We report the fraction of time it took to test the hypotheses during the search. In our case, it is not negligible for the

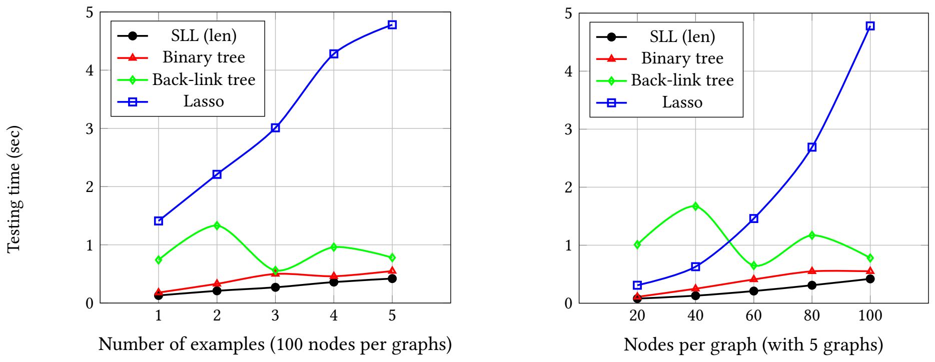


Fig. 6. Testing time with different input graphs (left); and with different number of nodes per graph (right).

examples with complex pure relations (*e.g.*, #13-15). In theory the test time should grow *slower than linearly* with the size of inputs. This is because only correct candidates require traversing all nodes by the Prolog unification. Since most candidate predicates do not satisfy all the examples, the testing is terminated when Prolog reaches the node that falsifies the example.

To show scalability of the tool *w.r.t.* the input size, Fig. 6 shows the testing time by graph number with 100 nodes per graph on its left, and the testing time by node number with 5 graphs on its right. The general trends are as expected: the testing time grows slower than linear in the graph number, and grows linearly in the node number (the percentage of nodes traversed in the examples should be constant with the same topology of the graph, thus linear).

Below we discuss the outliers. The reader can notice the substantial difference between the testing time of the lasso list case study and every other example. The reason for that is our current implementation of the linearity check, which is not optimised for *circular* data structures. Note that the testing of SL predicates consists of two parts: the Prolog validity check and the linearity check (in Sec. 4.2), we find that the all but circular data structures' linearity checking time is negligible. This inefficiency also explains the faster-than-linear growth of testing times in Fig. 6, and should be solvable by integrating the checking into Prolog's SLD resolution [33] (left as future work). Finally, the testing time of back tree is not even strictly increasing. The reason is: providing more or larger examples makes it possible to prune earlier, thereby reducing the overall synthesis time.

6.2 The Utility of Sippy

6.2.1 Verification. Let us demonstrate how the combination of Sippy and Grippy facilitates deductive program verification in SL-based provers. Specifically, our goal is to streamline the task of writing inductive predicates, by instead generating them automatically from programs to be verified. We are interested in the following research questions *w.r.t.* the effectiveness of the approach:

RQ 2.1 *How much human effort is required to infer the predicates?*
RQ 2.2 *How effective is Grippy for producing positive examples?*
RQ 2.3 *Are the inferred predicates the same as the expected (human-written) ones, so they can be used directly for the verification?*

RQ 2.1*: Required Human Effort.* For this experiment, we adopted the case studies from benchmark suites of three different deductive verifiers [30, 53, 54], containing heap-manipulating programs for different linked data structures, many of which come with non-trivial data constraints. Our aim is (1) to quantify the human effort for annotating selected program(s) with assertions as oracles for graph generation, and (2) to confirm that Grippy can produce good-quality graphs for Sippy to

Table 2. Statistics on generating input memory graphs for different predicates: the lines of codes of the `assert`-annotated function or its test, the number of graph instances (capped by 100) generated in 10 minutes, the number of asserted conjuncts, the ratio of the number of valid graphs to all randomly generated graphs, and whether the original program is verified with the synthesised predicates against its original specification. For VeriFast and GRASShopper, **Y** means verified; we did not manage to compile VCDryad, so = means equivalent to the expected predicate, and **?** means not equivalent and couldn't be checked.

Predicate	Programs	LOC_{prog}	LOC_{test}	Assert	Num	Ratio	Verified?
Singly Linked List	concat[2]	10	-	0	100	100%	**Y**
	reverse[2]	10	-	0	100	100%	**Y**
Sorted List	find[1]	10	-	1	100	14.4%	=
	insert_iter[1]	28	-	2	100	13.3%	=
	copy[2]	26	-	2	100	13%	**Y**
	double_all[2]	25	-	2	100	12.1%	**Y**
Doubly Linked List	append[1]	11	-	2	47	2.9%	=
	dispose[2]	9	-	2	41	2.5%	**Y**
	reverse[3]	16	-	2	47	2.9%	**Y**
	insert_front[1]	-	10	2	47	2.9%	=
Binary Search Tree	find[1]	16	-	3	100	11.5%	=
	insert[1]	22	-	6	100	8.0%	=
	free[3]	11	-	3	100	7.5%	**Y**
	remove[3]	43	-	3	100	12.3%	**Y**
Binomial Heap (order)	find_min[1] merge[1]	-	26	5	18	1.1%	**?**

[1] From VCDryad [53] [2] From GRASShopper [54] [3] From VeriFast [30]

synthesise the expected predicates within a reasonable time limit (10 min). For simplicity, we only consider the graphs without cycles unless being specified (doubly linked list in our case study).

Tab. 2 shows the statistics for generating up to 100 valid graphs, with up to five nodes, within the time limit of 10 minutes, to infer data structure predicates. In all cases, between 0 and 5 simple assertions (*i.e.*, one single comparison for most cases, except for BST, one of whose assertions is expressed by a comparison function) are enough for capturing the properties (*e.g.*, line 7 in Fig. 7).

RQ 2.2*: Effectiveness of the Graph Generator.* As shown in the table, in each experiment Grippy produced at least 18 valid graphs, which is sufficient for Sippy to infer the expected predicates. Unsurprisingly, the throughput of the generator (*i.e.*, the Num column in Tab. 2) correlates with the complexity of the predicates: *e.g.*, the binomial heap instance needs to satisfy not only the order relation between the nodes but also the heap property, which results in low chances for the generator to produce valid instances; generally, the throughput is similar (ranged by randomness) across programs manipulating the same structure. As expected, the fraction of valid graphs *w.r.t.* all randomly generated can be very low for complex predicates, which shows the importance of positive-only learning, as large numbers of trivially invalid graphs would slow down the synthesiser.

Even though in principle Grippy could be used for any programs, we found that writing assertions for certain functions (greyboxed in Tab. 2) is not an effective way to generate the valid instances. This is because the annotated node might simply not traverse "enough" of the structure to perform the validation. As an example, consider inserting a node to the front of a doubly linked list:

```
 1 DLNode * insert_front(DLNode * x, int k) {
 2   if (x == NULL) {
 3     DLNode * head = (DLNode *) malloc(sizeof(DLNode));
 4     head->key = k;
 5     head->next = NULL;
 6     head->prev = NULL;
 7     return head;
 8   } else {
 9     if(x->next != NULL) assert(x->next->prev == x);
10     DLNode * head = (DLNode *) malloc(sizeof(DLNode));
11     head->key = k;
12     head->next = x;
13     x->prev = head;
14     return head;
15   }
16 }
```

Testing the following incorrect example of a DLL heap graph, produced by Grippy, will not violate the assertion at line 9 of the code above.

```
next(n1,n2).   next(n2,n3). next(n3,null).
prev(n1,null). prev(n2,n1). prev(n3,null).
```

The reason is: the assertion at line 9 is only checked for node n1, but the offending node n2 is not checked because the function does not traverse the whole list. As a solution, in cases when no functions manipulating with the structure traverse the whole structure graph (so their LOCs are not informative, hence "-" in Tab. 2), one can instead write a standalone *traversal function* to be used as an oracle. Sizes of those additional functions are shown as LOC$_{\text{test}}$ in Tab. 2.

We conclude that Grippy is a useful front-end to Sippy, but its effectiveness depends on the "thoroughness" of the structure traversal done by the function that is used as an oracle.

***RQ 2.3**: Quality of Inferred Predicates.* With memory graphs obtained automatically, we synthesised the respective predicates with Sippy and translated them into the syntax of the corresponding SL-based verifiers (automatically or manually, based on how complex the concrete verifier's language is). Next, we verified the original programs with the inferred predicates, thus, demonstrating that the inferred predicates are equivalent to the expected ones. An inferred sorted list predicate of VCDryad, corresponding to the example from Sec. 2.2, is shown in Fig. 7. The only failing case is the binomial heap with order constraints because of the limitation of pre-defined predicates in

```
  int sorted_find(SNnode * l, int k){
    if (l == NULL) {
      return -1;
    } else if (l->key == k) {
      return 1;
    } else {
      if (l->next != NULL) assert(l->key <= l->next->key);
      int res = sorted_find(l->next, k);
      return res;
    } }
```

```
define pred sorted^(a):
 ((a l= nil) & emp) |
 ((a |-> loc next: c; int key: e) * sorted^(c) & (e lt-set keys^(c)))
```

Fig. 7. An example of the input program with assertions and inferred predicate for sorted list in VCDryad.

No.	Category	Program	Code/Spec	Time
1	Deallocate	sll	5.5x	0.2s
2		bst	8.0x	0.2s
3		dll_seg	2.4x	0.2s
4		multilist	16.0x	0.3s
5	Copy	lseg	2.0x	0.8s
6		bst	3.5x	3.3s
7		balanced tree	3.0x	1.9s
8	Size	sll_len	2.1x	0.4s
9		balanced tree	3.5x	0.6s
10	Transform	sll $\longrightarrow$ dlseg	2.5x	0.4s
11		srt_dll $\longrightarrow$ sll	3.1x	7.4s
12		dll $\longrightarrow$ bst	15.0x	42.8s
13		btree $\longrightarrow$ bktree	13.6x	11.8s
14		multilist $\longrightarrow$ sll	5.0x	8.8s
15		btree $\longrightarrow$ dll	9.6x	7.1s
16		bst $\longrightarrow$ srtl	11.6x	10.3s
17		dll $\longrightarrow$ srt_dll	7.3x	9.3s

Fig. 8. Example programs synthesised by SuSLik from SL specifications stated using predicates produced by Sippy.

```
// pre:  {f :-> x ** sorted_dll(x, z, s)}
// post: {f :-> y ** sll(y, s)}

void srt_dll_to_sll (loc f) {
loc x1 = READ_LOC(f, 0);
if (x1 == 0)  {
  WRITE_INT(f, 0, 0);
  return;
} else {
  int vx11 = READ_INT(x1, 0);
  loc nxtx11 = READ_LOC(x1, 1);
  loc z1 = READ_LOC(x1, 2);
  WRITE_LOC(f, 0, nxtx11);
  srt_dll_to_sll(f);
  loc y11 = READ_LOC(f, 0);
  loc y2 = (loc)malloc(2 * sizeof(loc));
  free(x1);
  WRITE_LOC(f, 0, y2);
  WRITE_LOC(y2, 1, y11);
  WRITE_INT(y2, 0, (int)vx11);
  return;
}}
```

Fig. 9. An example SuSLik output (#11): a C program for converting a sorted DLL to SLL. loc, READ_LOC, *etc* are macro-definitions around ordinary C types and operations.

Sippy (further explained in Sec. 6.3), where the synthesised predicates are partially correct but not strong enough, and need to be refined by manually adding the missing constraints.

To summarise, we found the combination of Grippy/Sippy effective for automatically producing SL predicates equivalent to human-written ones from either modestly-annotated programs to be verified, or with a help of a simple human-written traversal procedure for the data structure.

6.2.2 Deductive Synthesis. As another demonstration of Sippy's utility, we employed the synthesised SL predicates to automatically generate correct-by-construction heap-manipulating programs in C using a state-of-the-art deductive synthesiser SuSLik [56, 67]. The goal of this exercise was to demonstrate that, one can use Sippy together with SuSLik to obtain *provably correct* implementations of structure-specific procedures for copying, computing their size, and transformation without knowing how to specify SL predicates, but using heap graphs. Our case study includes 17 synthesis tasks involving the predicates from Tab. 1, producing programs *not* featured in any past works on SuSLik. The average code/spec AST size ratio is 4.1, and the average SuSLik synthesis time is 6.2 sec, which shows that Sippy produces predicates that are *immediately suitable* for proof-driven synthesis. Fig. 8 provides the detailed statistics. An example of the synthesised C program that transforms a sorted DLL into a singly linked list is given in Fig. 9.

Compared to the existing example-based heap-manipulating program synthesisers, SPT [61] and Synbad [58], the joint Sippy/SuSLik workflow does not require "fold/unfold" functions (as does SPT) or a template (as needed by Synbad) for the intended programs.

6.3 Failure Modes and Future Work

In its current version, Sippy failed to synthesise predicates for several intricate linked structures. The reasons for the failed tasks fall into one of the following three categories:

(1) Sippy's default settings cannot fully capture the structure's properties. Consider #15 from Tab. 1: the root and leaf nodes of binomial trees have different order relation with their siblings, but

our search space cannot express this distinction, so the synthesised predicate is *the best in this setting* (capturing nodes' ordering *w.r.t.* their children) but not the expected one.
(2) Nested data structures with multiple arguments or complex pure relations, For example, the needed search space of the *braced list segment* predicate [57] is too large to be fully explored within our time limits.
(3) An instance of a predicate cannot be proven by top-*down evaluation*, so that Prolog cannot evaluate them as expected in SL.

In the first case, the solution is naturally to allow richer search space: we might either extend the search space with more predicates (*e.g.*, judging a node is root or not), or enabling larger parameters in Sec. 4.1 to enrich the expressiveness of Sippy. This is in line with a common synthesis trade-off between the expressiveness and the efficiency; we leave it a future work to find general search space settings to enrich the expressiveness without much loss of performance.

In the second case, it is possible to optimise the synthesis of nested predicates using problem-specific knowledge. For instance, we can assume the inner data structure is not mutually recursive (as it is in the case of the braced list segment), reducing the search space by splitting the synthesis of the auxiliary predicate and the whole predicate. We leave this optimisation to the future work.

To explain the last issue, consider the following Prolog predicate:

```
p(X, Y) :- X == Y.
p(X, Y) :- next(X, Z), p(Z, Y), p(X, Z).
```

In plain words: if `X` and `Y` have the same location, then `p(X, Y)` is true; if not, then the `p(X, Y)` holds if the both segments `p(X, Z)` and `p(Z, Y)` are true, where `Z` is the next node of `X`. This predicate is valid in Separation Logic, but Prolog rejects it, because of its validity checking algorithm. To understand the difference that causes the rejection, consider the literal `p(a, b)`, in the case when `next(a, b)` is a fact that holds. In SL, the literal is true, because it is checked in a *bottom-up* way, *i.e.*, "whether the predicate is consistent if it is true". Therefore, `p(a, b)` holds because it is consistent with `next(a, b), p(b, b), p(a, b)`. However, the test of `p(a, b)` in Prolog is done in a top-down way, *i.e.*, "whether there is a variable unification that makes `next(a, Z), p(Z, b), p(a, Z)` true", which triggers a recursive test on the inner `p(a, b)`. It will be interesting to see whether replacing Prolog with the solvers from SL-COMP [60] can directly solve it without other overhead.

6.4 Why not just use Large Language Models?

Though Sippy has non-negligible runtime for synthesising complex predicates, its completeness guarantees (*cf.* Theorem 4.2) ensure that the synthesised predicates are the best (*i.e.*, the most specific ones) in the respective search space. Large Language Models (LLMs), as a powerful tool for learning, have been used extensively in the recent works for synthesising specifications [40, 68], with faster runtime but without completeness guarantees. One may wonder: why not just ask an LLM to synthesise a specification, and use Prolog to test it against the provided examples, mimicking the loop of Algorithm 1? To assess whether our synthesiser with proven completeness guarantees provides better solutions compared to a state-of-the-art LLM, we pose SL predicate synthesis as a task for the latter by designing a detailed prompt outlining our intentions, followed by a series of queries with inputs similar to what is required by Sippy (*i.e.*, positive examples).

Phase 1: Simple Prompt for Learning. Before the synthesis, we provide a detailed prompt to an LLMs as outlinining the required background knowledge, which includes the following parts:

(1) The task: synthesising SL predicates in Prolog for linked heap structures given the graphs.
(2) An example of the predicate "sll" for singly-linked list and its graphs, with the explanation.
(3) Other synthesis settings, such as the predicates that can be used for pure constraints, the option to invent auxiliary predicates, and the requirements on the size of the results.

Phase 2: Synthesising Complex Predicates. A synthesis prompt for each task is given via the following template, along with graphs and positive examples in the same format as taken by Sippy:

```
For the next task, here are the graphs: (Graphs)
And here are the positive examples: (Positive examples)
Please, synthesise the predicate.
```

Our experiments were done on latest ChatGPT-4o.[4] Below, we summarise the outcomes.

(1) We noticed that LLMs can correctly synthesise the predicates for simple cases (*e.g.*, doubly-linked list), but it fails to synthesise predicates for more structures with non-trivial constraints (*e.g.*, binary search trees and balanced tree): its result often don't type check or miss constraints.
(2) Unsurprisingly, an LLM benefitted from the predicate names we provided to it. For example, with providing the name `rose_tree` in the prompt, the output predicates are mostly correct.
(3) As LLMs have quick turnaround compared to running Sippy, one promising direction is to use LLMs for the initial exploration of the search space and then use Sippy to refine the results.

We conclude that LLMs can be used for synthesising simple predicates, and have a potential accelerate the synthesis process by deriving plausible candidates, which can be checked by Prolog and, possibly, repaired. That said, completeness guarantees of Sippy provide tangible benefits, allowing it derive correct solutions for complex examples, which an LLM failed to discover.

7 Related Work

Learning Data Structure Invariants. Other than ShaPE [8], discussed extensively in Sec. 6.1, earlier work on shape analysis also used inductive synthesis to generate shape predicates [28], but the input of the synthesis framework is a program that constructs the data structure instance, providing more information (*e.g.*, the recursion structure) compared to memory graphs. Similar to ShaPE, that work only considers the shape relation without the data properties. DOrder [71] and Evospex [43] are two later works on learning the data invariants from the constructors of the data structures (in OCaml or Java). Locust [10] infers shape predicates from pre-defined definitions with statistical machine learning, with no completeness guarantees. The work by Molina et al. [42] describes a deep learning-based framework that implements a binary classifier for the data structure invariants; unlike our work, it does not provide logical descriptions of data structures but merely tells valid structure instances from invalid ones. Though more machine learning methods [64] have shown to be effective in learning data structure, the training data is required to be large and diverse, which is not always available in practise. Dohrau [20] describes a black-box approach to infer SL specifications and predicates from programs based on ICE learning [22], while it can neither deal with, nor be easily extensible to nested data structures. SLING is a framework to infer program specifications in Separation Logic from memory graphs [38]; it does not infer new heap predicates and instead offers a number of pre-defined structure shapes, where Sippy can work as a complementary tool to help the user to pre-define new predicates.

Synthesising Declarative Representations. The approach of Sippy extends two lines of work on synthesis of declarative representations programs and data. The first one is inductive logic programming (ILP), which aims to learn logic programs from examples. Progol [47] is an early notable ILP system that achieves positive-only learning by Bayesian framework, which is not sound in general. Importantly, Progol does not support learning recursive logic programs. AMIE [21] is another knowledge rule-mining framework with positive-only examples, but the learning is in AMIE mainly targeted knowledge base graphs, so the learned rules were not as complex as in ILP settings. Other than those conventional ILP methods that synthesise Prolog programs, significant progress has been made recently on synthesising Datalog [62, 63] and ASP programs [35, 36] from

[4]The conversation snapshot is available at https://chatgpt.com/share/66f266a5-0338-8006-8bb9-1ef61c33d437.

examples. While the syntax of Separation Logic assertions and predicates can be expressed in the Datalog or ASP domain, the approaches developed in the past efforts are not immediately applicable, as they: (1) may require negative examples (in the case of Datalog synthesisers), and (2) limit the pre-defined predicates expressed only by grounded facts. In particular, the latter means that the predicates used in the synthesis cannot express arbitrary operations, because grounded facts are fully instantiated and do not contain variables. This limitation makes impossible the representation of general rules or operations that can be applied to a range of inputs. For example, a grounded fact can state that a specific element belongs to a set, but it cannot express a general rule for membership that applies to *any* element. At the same time, in Sippy, the predicates are written in Prolog, a Turing-complete language, which enables definitions of operations like list append, set union, *etc*,—a feature our approach has directly inherited from Popper [18].

The second relevant line of work is *specification synthesis*, which aims at synthesising formulas within various but pre-defined domains. Our graph generator (Sec. 5) follows the ideas of Precis [4], which similarly uses test cases as a learning oracle to generate positive example for synthesising program contracts. At the same time, the formal guarantees provided by positive only learning–generating all non-comparable and most specific predicates, are similar to those of Spyro [52], which defines a general framework for synthesising specifications for customisable domains. The main difference between Sippy and those works is that Sippy synthesises *predicates* that can contain *recursive definitions*—an aspect cannot be handled by the existing specification synthesisers.

The overlap between two lines of work above is the notion of *least general generalisation* (LGG) [55]. The example-based specification synthesisers can be considered as learning the LGG of the examples, which is exactly what early bottom-up ILP systems do. The limitation of existing LGG operations is well-known in ILP [15]: there is no LGG operations for recursive logic programs, which is the reason why modern ILP systems are defined in a top-down fashion.

Answer Set Programming v. Satisfiability Modulo Theories. ASP plays a crucial role in our work, similar to that of SMT solvers most contemporary synthesis tools. As mentioned by Bembenek et al. [5], ASP is effective at search tasks involving fixpoints: pruning in Sippy can be encoded easily with recursive logic predicates, some of which though can be expressed in SMT, need to be encoded in a more complex and hard-to-understand way. Another advantage of ASP is its efficiency when enumerating *all* solutions in a search space, which is crucial for exhaustively exploring the space of SL predicates. In contrast, in most state-of-the-art SMT solvers, only one model is returned at a time, so for obtaining a complete set of models, the user must either block an obtained model and re-run the solver to get the next one with high overhead, or use expert-level techniques [6]. On the other hand, SMT solvers usually come with a rich set of theories, whereas ASP modulo theories is still limited to basic theories like difference logic [31] and acyclicity constraints [7].

8 Conclusion

We presented the first approach for synthesising property-rich inductive predicates for data structures in Separation Logic (SL) from concrete heap graph examples, by positive-only learning via Answer Set Programming, with SL-based pruning. Our framework Sippy is capable of automatically learning predicates for complex structures with payload constraints and mutual recursion, facilitating applications of SL-based tools for deductive verification and program synthesis. In the future, we are planning to explore other possible applications of our predicate synthesiser for program repair [65], program comprehension [9], and Computer Science education [41].

Data Availability

The implementations of Sippy, Grippy, and the benchmark harness necessary for reproducing our experimental results in Sec. 6 are available online [70].

Acknowledgments

We thank Vladimir Gladshtein, Yunjeong Lee, Peter Müller, Hila Peleg, George Pîrlea, and Qiyuan Zhao for their feedback on earlier drafts of this paper. We also thank the reviewers of OOPSLA'25 for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV (LNCS, Vol. 8044)*. Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[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. https://doi.org/10.1109/FMCAD.2013.6679385

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

[4] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485481

[5] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2023. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems. *Proc. ACM Program. Lang.* 7, POPL (2023), 185–217. https://doi.org/10.1145/3571200

[6] Nikolaj Bjørner, Clemens Eisenhofer, and Laura Kovács. 2022. User-Propagation for Custom Theories in SMT Solving. In *Satisfiability Modulo Theories. 20th International Workshop*.

[7] Jori Bomanson, Martin Gebser, Tomi Janhunen, Benjamin Kaufmann, and Torsten Schaub. 2015. Answer Set Programming Modulo Acyclicity. In *LPNMR (LNCS, Vol. 9345)*. Springer, 143–150. https://doi.org/10.1007/978-3-319-23264-5_13

[8] Jan H. Boockmann and Gerald Luettgen. 2020. Learning Data Structure Shapes from Memory Graphs. In *LPAR (EPiC Series in Computing, Vol. 73)*. EasyChair, 151–168. https://doi.org/10.29007/dhpw

[9] Jan H. Boockmann and Gerald Lüttgen. 2022. Shape-analysis driven memory graph visualization. In *ICPC*. ACM, 298–308. https://doi.org/10.1145/3524610.3527913

[10] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *SAS (LNCS, Vol. 10422)*. Springer, 66–87. https://doi.org/10.1007/978-3-319-66706-5_4

[11] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (LNCS, Vol. 6617)*. Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[12] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[13] Domenico Corapi, Alessandra Russo, and Emil Lupu. 2011. Inductive Logic Programming in Answer Set Programming. In *ILP (LNCS, Vol. 7207)*. Springer, 91–97. https://doi.org/10.1007/978-3-642-31951-8_12

[14] Andrew Cropper. 2022. Learning Logic Programs Though Divide, Constrain, and Conquer. In *AAAI*. AAAI Press, 6446–6453. https://doi.org/10.1609/AAAI.V36I6.20596

[15] Andrew Cropper and Sebastijan Dumancic. 2022. Inductive Logic Programming At 30: A New Introduction. *J. Artif. Intell. Res.* 74 (2022), 765–850. https://doi.org/10.1613/JAIR.1.13507

[16] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. 2022. Inductive logic programming at 30. *Mach. Learn.* 111, 1 (2022), 147–172. https://doi.org/10.1007/s10994-021-06089-1

[17] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. 2020. Turning 30: New Ideas in Inductive Logic Programming. (2020), 4833–4839. https://doi.org/10.24963/IJCAI.2020/673

[18] Andrew Cropper and Rolf Morel. 2021. Learning programs by learning from failures. *Mach. Learn.* 110, 4 (2021), 801–856. https://doi.org/10.1007/s10994-020-05934-z

[19] Andrew Cropper and Stephen H. Muggleton. 2014. Logical Minimisation of Meta-Rules Within Meta-Interpretive Learning. In *ILP (LNCS, Vol. 9046)*. Springer, 62–75. https://doi.org/10.1007/978-3-319-23708-4_5

[20] Jérôme Dohrau. 2022. *Automatic Inference of Permission Specifications*. Ph. D. Dissertation. ETH Zurich, Zürich, Switzerland. https://doi.org/10.3929/ETHZ-B-000588977

[21] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. 2013. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*. ACM, 413–422. https://doi.org/10.1145/2488388.2488425

[22] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV (LNCS, Vol. 8559)*. Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[23] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00457ED1V01Y201211AIM019

[24] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR* abs/1405.3694 (2014). arXiv:1405.3694 http://arxiv.org/abs/1405.3694

[25] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1 (2019), 27–82. https://doi.org/10.1017/S1471068418000054

[26] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *ICLP*. MIT Press, 1070–1080.

[27] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[28] Bolei Guo, Neil Vachharajani, and David I. August. 2007. Shape analysis with inductive recursion synthesis. In *PLDI*. ACM, 256–265. https://doi.org/10.1145/1250734.1250764

[29] Peter L. Hammer and Alexander Kogan. 1993. Optimal Compression of Propositional Horn Knowledge Bases: Complexity and Approximation. *Artif. Intell.* 64, 1 (1993), 131–145. https://doi.org/10.1016/0004-3702(93)90062-G

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

[31] Tomi Janhunen, Roland Kaminski, Max Ostrowski, Sebastian Schellhorn, Philipp Wanko, and Torsten Schaub. 2017. Clingo goes linear over reals and integers. *Theory Pract. Log. Program.* 17, 5-6 (2017), 872–888. https://doi.org/10.1017/S1471068417000242

[32] Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485544

[33] Robert A. Kowalski. 1974. Predicate Logic as Programming Language. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 569–574.

[34] Michael Langowski. 2022. ASP - A Tutorial. https://madmike200590.github.io/asp-guide/tutorial.html

[35] Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. 2020. FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria. In *AAAI*. AAAI Press, 2877–2885. https://doi.org/10.1609/AAAI.V34I03.5678

[36] Mark Law, Alessandra Russo, and Krysia Broda. 2014. Inductive Learning of Answer Set Programs. In *JELIA (LNCS, Vol. 8761)*. Springer, 311–325. https://doi.org/10.1007/978-3-319-11558-0_22

[37] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527325

[38] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *PLDI*. ACM, 788–801. https://doi.org/10.1145/3314221.3314634

[39] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434335

[40] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. *CoRR* abs/2401.08807 (2024). https://doi.org/10.48550/ARXIV.2401.08807

[41] Mark Marron, César Sánchez, Zhendong Su, and Manuel Fähndrich. 2013. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Software Eng.* 39, 6 (2013), 774–786. https://doi.org/10.1109/TSE.2012.69

[42] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2019. Training binary classifiers as data structure invariants. In *ICSE*. IEEE / ACM, 759–770. https://doi.org/10.1109/ICSE.2019.00084

[43] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *ICSE*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[44] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. 2023. Message Chains for Distributed System Verification. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2224–2250. https://doi.org/10.1145/3622876

[45] Stephen H. Muggleton. 1991. Inductive Logic Programming. *New Gener. Comput.* 8, 4 (1991), 295–318. https://doi.org/10.1007/BF03037089

[46] Stephen H. Muggleton. 1995. Inverse entailment and Progol. *New generation computing* 13, 3 (1995), 245–286. https://doi.org/10.1007/BF03037227

[47] Stephen H. Muggleton. 1996. Learning from Positive Data. In *6th International Workshop on Inductive Logic Programming (LNCS, Vol. 1314)*. Springer, 358–376. https://doi.org/10.1007/3-540-63494-0_65

[48] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. 2014. Meta-interpretive learning: application to grammatical inference. *Mach. Learn.* 94, 1 (2014), 25–49. https://doi.org/10.1007/s10994-013-5358-3

[49] Thanh-Toan Nguyen, Quang-Trung Ta, Ilya Sergey, and Wei-Ngan Chin. 2021. Automated Repair of Heap-Manipulating Programs Using Deductive Synthesis. In *VMCAI (LNCS, Vol. 12597)*. Springer, 376–400. https://doi.org/10.1007/978-3-030-67067-2_17

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

[51] Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *CAV (LNCS, Vol. 11561)*. Springer, 315–334. https://doi.org/10.1007/978-3-030-25540-4_17

[52] Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023. Synthesizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 285:1–285:30. https://doi.org/10.1145/3622861

[53] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*. ACM, 440–451. https://doi.org/10.1145/2594291.2594325

[54] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (LNCS, Vol. 8413)*. Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

[55] Gordon D Plotkin. 1970. A note on inductive generalization. *Machine intelligence* 5, 1 (1970), 153–163.

[56] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30. https://doi.org/10.1145/3290385

[57] John C. Reynolds. 2008. *An Introduction to Separation Logic (Preliminary Draft)*. Technical Report. Carnegie Mellon University.

[58] Subhajit Roy. 2013. From Concrete Examples to Heap Manipulating Programs. In *SAS (LNCS, Vol. 7935)*. Springer, 126–149. https://doi.org/10.1007/978-3-642-38856-9_9

[59] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *ESEC/SIGSOFT FSE*. ACM, 515–527. https://doi.org/10.1145/3236024.3236034

[60] Mihaela Sighireanu, Juan Antonio Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siau-Cheng Khoo, Michal Cyprian, Adam Rogalewicz, Tomás Vojnar, Constantin Enea, Ondrej Lengál, Chong Gao, and Zhilin Wu. 2019. SL-COMP: Competition of Solvers for Separation Logic. In *TACAS (LNCS, Vol. 11429)*. Springer, 116–132. https://doi.org/10.1007/978-3-030-17502-3_8

[61] Rishabh Singh and Armando Solar-Lezama. 2012. SPT: Storyboard Programming Tool. In *CAV (LNCS, Vol. 7358)*. Springer, 738–743. https://doi.org/10.1007/978-3-642-31424-7_58

[62] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-guided synthesis of relational queries. In *PLDI*. ACM, 1110–1125. https://doi.org/10.1145/3453483.3454098

[63] Aalok Thakkar, Nathaniel Sands, George Petrou, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2023. Mobius: Synthesizing Relational Queries with Recursive and Invented Predicates. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1394–1417. https://doi.org/10.1145/3622847

[64] Muhammad Usman, Wenxi Wang, Kaiyuan Wang, Cagdas Yelen, Nima Dini, and Sarfraz Khurshid. 2019. A Study of Learning Data Structure Invariants Using Off-the-shelf Tools. In *SPIN (LNCS, Vol. 11636)*. Springer, 226–243. https://doi.org/10.1007/978-3-030-30923-7_13

[65] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. ACM, 151–162. https://doi.org/10.1145/3180155.3180250

[66] Ondřej Čepek. 1995. *Stuctural properties and minimization of Horn Boolean functions.* Ph. D. Dissertation. Rutgers University.

[67] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473589

[68] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *CAV (LNCS, Vol. 14682)*. Springer, 302–328. https://doi.org/10.1007/978-3-031-65630-9_16

[69] Ziyi Yang and Ilya Sergey. 2025. Inductive Synthesis of Inductive Heap Predicates – Extended Version. *CoRR* abs/2502.14478 (2025). arXiv:2502.14478 http://arxiv.org/abs/2502.14478

[70] Ziyi Yang and Ilya Sergey. 2025. *Sippy: the Artefact for the Paper "Inductive Synthesis of Inductive Heap Predicates"*. https://doi.org/10.5281/zenodo.14928260

[71] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically learning shape specifications. In *PLDI*. ACM, 491–507. https://doi.org/10.1145/2908080.2908125

Received 2024-10-11; accepted 2025-02-18