Spatial Interpolants

Aws Albargouthi[1], Josh Berdine[2], Byron Cook[3], and Zachary Kincaid[4]

University of Wisconsin-Madison[1], Microsoft Research[2], University College London[3], University of Toronto[4]

Abstract. We propose SplInter, a new technique for proving properties of heap-manipulating programs that marries (1) a new *separation logic–based* analysis for heap reasoning with (2) an *interpolation-based* technique for refining heap-shape invariants with data invariants. SplInter is *property directed*, *precise*, and produces counterexample traces when a property does not hold. Using the novel notion of *spatial interpolants modulo theories*, SplInter can infer complex invariants over general recursive predicates, e.g., of the form *all elements in a linked list are even* or a *binary tree is sorted*. Furthermore, we treat interpolation as a black box, which gives us the freedom to encode data manipulation in any suitable theory for a given program (e.g., bit vectors, arrays, or linear arithmetic), so that our technique immediately benefits from any future advances in SMT solving and interpolation.

1 Introduction

Since the problem of determining whether a program satisfies a given property is undecidable, every verification algorithm must make some compromise. There are two classical schools of program verification, which differ in the compromise they make: the *static analysis* school gives up refutation soundness (i.e., may report *false positives*); and the *software model checking* school gives up the guarantee of termination. In the world of integer program verification, both schools are well explored and enjoy cross-fertilization of ideas: each has its own strengths and uses in different contexts. In the world of heap-manipulating programs, the static analysis school is well-attended [11, 13, 15, 36], while the software model checking school has remained essentially vacant. This paper initiates a program to rectify this situation, by proposing one of the first path-based software model checking algorithms for proving combined shape-and-data properties.

The algorithm we propose, SplInter, marries two celebrated program verification ideas: McMillan's *lazy abstraction with interpolants* (Impact) algorithm for software model checking [26], and *separation logic*, a program logic for reasoning about shape properties [33]. SplInter (like Impact) is based on a path-sampling methodology: given a program P and safety property φ, SplInter constructs a proof that P is memory safe and satisfies φ by sampling a finite number of paths through the control-flow graph of P, proving them safe, and then assembling proofs for each sample path into a proof for the whole program. The key technical advance which enables SplInter is an algorithm for *spatial interpolation*, which is used to construct proofs in *separation logic* for the sample traces (serving the same function as *Craig interpolation* for first-order logic in Impact).

SplInter is able to prove properties requiring integrated heap and data (e.g., integer) reasoning by strengthening separation logic proofs with *data refinements* produced by classical Craig interpolation, using a technique we call *spatial interpolation modulo theories*. Data refinements are *not tied to a specific logical theory*, giving us a rather generic algorithm and freedom to choose an appropriate theory to encode a program's data.

Fig. 1 summarizes the high-level operation of our algorithm. Given a program with no heap manipulation, SplInter only computes theory interpolants and behaves exactly like Impact, and thus one can thus view SplInter as a proper extension of Impact to heap manipulating programs. At the other extreme, given a program with no data manipulation, SplInter is a new shape analysis that uses path-based relaxation to construct memory safety proofs in separation logic.

There is a great deal of work in the static analysis school on shape analysis and on combined shape-and-data analysis, which we will discuss further in Sec. 8. We do not claim superiority over these techniques (which have had the benefit of 20 years of active development). SplInter, as the first member of the software model checking school, is not *better*; however, it *is* fundamentally *different*. Nonetheless, we will mention two of the features of SplInter (not enjoyed by any previous verification algorithm for shape-and-data properties) that make our approach worthy of exploration: path-based refinement and property-direction.

- *Path-based refinement*: This supports a progress guarantee by tightly correlating program exploration with refinement, and by avoiding imprecision due to lossy join and widening operations employed by abstract domains. SplInter does not report false positives, and produces counterexamples for violated properties. This comes, as usual, at the price of potential divergence.
- *Property-direction*: Rather than seeking the strongest invariant possible, we compute one that is *just strong enough* to prove that a desired property holds. Property direction enables scalable reasoning in rich program logics like the one described in this paper, which combines separation logic with first-order data refinements.

We have implemented an instantiation of our generic technique in the T2 verification tool [38], and used it to prove correctness of a number of programs, partly drawn from open source software, requiring combined data and heap invariants. Our results indicate the usability and promise of our approach.

Contributions We summarize our contributions as follows:

1. A generic property-directed algorithm for verifying and falsifying safety of programs with heap and data manipulation.
2. A precise and expressive separation logic analysis for computing memory safety proofs of program paths using a novel technique we term *spatial interpolation*.
3. A novel interpolation-based technique for strengthening separation logic proofs with data refinements.
4. An implementation and an evaluation of our technique for a fragment of separation logic with linked lists enriched with linear arithmetic refinements.

The extended version [2] of this paper contains additional details and material.

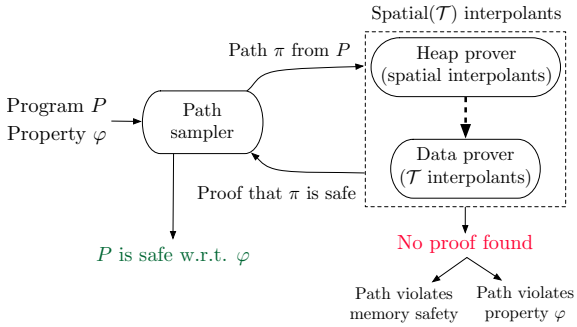

Fig. 1. Overview of SplInter verification algorithm.

2 Overview

In this section, we demonstrate the operation of SplInter (Fig. 1) on the simple linked list example shown in Fig. 2. We assume that integers are unbounded (i.e., integer values are drawn from $\mathbb{Z}$ rather than machine integers) and that there is a `struct` called `node` denoting a linked list node, with a next pointer `N` and an integer (data) element `D`. The function `nondet()` returns a nondeterministic integer value. This program starts by building a linked list in the loop on location `2`. The loop terminates if the initial value of `i` is $\geqslant 0$, in which case a linked list of size `i` is constructed, where data elements `D` of list nodes range from `1` to `i`. Then, the loop at location `3` iterates through the linked list asserting that the data element of each node in the list is $\geqslant 0$. Our goal is to prove that the assertion at location `4` is never violated.

```
1: int i = nondet();
   node* x = null;
2: while (i != 0)
     node* tmp = malloc(node);
     tmp->N = x;
     tmp->D = i;
     x = tmp;
     i--;
3: while (x != null)
4:   assert(x->D >= 0);
     x = x->N;
```

Fig. 2. Illustrative Example

Sample a Program Path To start, we need a path π through the program to the assertion at location `4`. Suppose we start by sampling the path `1,2,2,3,4`, that is, the path that goes through the first loop once, and enters the second loop arriving at the assertion. This path is illustrated in Fig. 3 (where `2a` indicates the second occurrence of location `2`). Our goal is to construct a Hoare-style proof of this path: an annotation of each location along the path with a formula describing reachable states, such that location `4` is annotated with a formula implying that `x->D >= 0`. This goal is accomplished in two phases. First, we use *spatial interpolation* to compute a memory safety proof for the path π (Fig. 3(b)). Second, we use *theory refinement* to strengthen the memory safety proof and establish that the path satisfies the post-condition `x->D >= 0` (Fig. 3(c)).

Compute Spatial Interpolants The first step in constructing the proof is to find *spatial interpolants*: a sequence of separation logic formulas *approximating* the shape of the heap at each program location, and forming a Hoare-style memory safety proof of the path. Our spatial interpolation procedure is a two

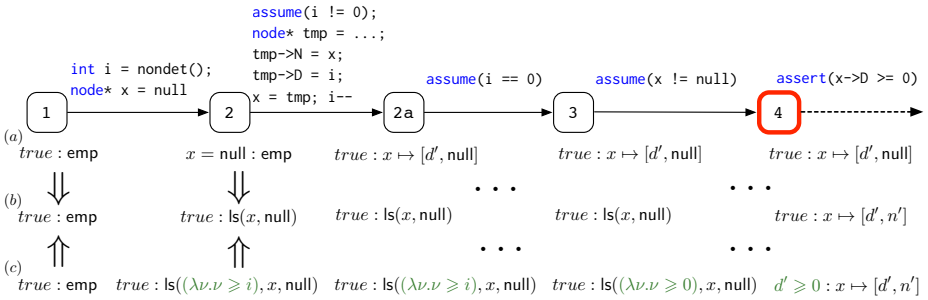

Fig. 3. Path through program in Fig. 2, annotated with (a) results of forward symbolic execution, (b) spatial interpolants, and (c) spatial($\mathcal{T}$) interpolants, where $\mathcal{T}$ is linear integer arithmetic. Arrows $\Rightarrow$ indicate implication (entailment) direction.

step process that first symbolically executes the path in a forward pass and then derives a weaker proof using a backward pass. The backward pass can be thought of as an under-approximate weakest precondition computation, which uses the symbolic heap from the forward pass to guide the under-approximation.

We start by showing the *symbolic heaps* in Fig. 3(a), which are the result of the forward pass obtained by symbolically executing *only* heap statements along this program path (i.e., the strongest postcondition along the path). The separation logic annotations in Fig. 3 follow standard notation (e.g., [15]), where a formula is of the form $\Pi : \Sigma$, where Π is a Boolean first-order formula over heap variables (pointers) as well as data variables (e.g., $x = \mathsf{null}$ or $i > 0$), and Σ is a *spatial conjunction* of *heaplets* (e.g., emp, denoting the empty heap, or $Z(x, y)$, a recursive predicate, e.g., that denotes a linked list between x and y). For the purposes of this example, we assume a recursive predicate $\mathsf{ls}(x, y)$ that describes linked lists. In our example, the symbolic heap at location `2a` is $true : x \mapsto [d', \mathsf{null}]$, where the heap consists of a node, pointed to by variable x, with null in the `N` field and the (implicitly existentially quantified) variable d' in the `D` field (since so far we are only interested in heap shape and not data).

The symbolic heaps determine a memory safety proof of the path, but it is too strong and would likely not generalize to other paths. The goal of spatial interpolation is to find a sequence of annotations that are weaker than the symbolic heaps, but that still prove memory safety of the path. A sequence of spatial interpolants is shown in Fig. 3(b). Note that all spatial interpolants are implicitly spatially conjoined with true; for clarity, we avoid explicitly conjoining formulas with true in the figure. For example, location `2` is annotated with $true : \mathsf{ls}(x, \mathsf{null}) * \mathsf{true}$, indicating that there is a list on the heap, as well as other potential objects not required to show memory safety. We compute spatial interpolants by going backwards along the path and asking questions of the form: *how much can we weaken the symbolic heap while still maintaining memory safety?* We will describe how to answer such questions in Section 4.

Refine with Theory Interpolants Spatial interpolants give us a memory safety proof as an approximate heap shape at each location. Our goal now is to strengthen these heap shapes with data refinements, in order to prove that the

assertion at the end of the path is not violated. To do so, we generate a system of Horn clause constraints from the path in some first-order theory admitting interpolation (e.g., linear arithmetic). These Horn clauses carefully encode the path's data manipulation along with the spatial interpolants, which tell us heap shape at each location along the path. A solution of this constraint system, which can be solved using off-the-shelf interpolant generation techniques (e.g., [27, 35]), is a *refinement* (strengthening) of the memory safety proof.

In this example, we encode program operations over integers in the theory of linear integer arithmetic, and use Craig interpolants to solve the system of constraints. A solution of this system is a set of linear arithmetic formulas that refine our spatial interpolants and, as a result, imply the assertion we want to prove holds. One possible solution is shown in Fig. 3(c). For example, location `2a` is now labeled with $true : \mathsf{ls}((\lambda\nu.\nu \geqslant i), x, \mathsf{null})$, where the green parts of the formula are those added by refinement. Specifically, after refinement, we know that *all* elements in the list from x to `null` after the first loop have data values greater than or equal to i, as indicated by the predicate $(\lambda\nu.\nu \geqslant i)$. (In Section 3, we formalize recursive predicates with data refinements.)

Location `4` is now annotated with $d' \geqslant 0 : x \mapsto [d', n'] * \mathsf{true}$, which implies that `x->D >= 0`, thus proving that the path satisfies the assertion.

From Proofs of Paths to Proofs of Programs We go from proofs of paths to whole program proofs implicitly by building an *abstract reachability tree* as in Impact [26]. To give a flavour for how this works, consider that the assertions at `2` and `2a` are identical: this implies that this assertion is an inductive invariant at line `2`. Since this assertion also happens to be strong enough to prove safety of the program, we need not sample any longer unrollings of the first loop. However, since we have not established the inductiveness of the assertion at `3`, the proof is not yet complete and more traces need to be explored (in fact, exploring one more trace will do: consider the trace that unrolls the second loop once and shows that the second time `3` is visited can also be labeled with $true : \mathsf{ls}((\lambda\nu.\nu \geqslant 0), x, \mathsf{null})$).

Since our high-level algorithm is virtually the same as Impact [26], we will not describe it further in the paper. For the remainder of this paper, we will concentrate on the novel contribution of our algorithm: computing spatial interpolants with theory refinements for program paths.

3 Preliminaries

3.1 Separation Logic

We define RSep, a fragment of separation logic formulas featuring points-to predicates and general recursive predicates refined by theory propositions.

Fig. 4 defines the syntax of RSep formulas. In comparison with the standard list fragment used in separation logic analyses (e.g., [4, 14, 28]), the differentiating features of RSep are: (1) General *recursive predicates*, for describing unbounded pointer structures like lists, trees, etc. (2) Recursive predicates are augmented with a vector of *refinements*, which are used to constrain the data values appearing on the data structure defined by the predicate, detailed below. (3) Each heap cell (points-to predicate), $E \mapsto [\vec{A}, \vec{E}]$, is a *record* consisting of

$$
\begin{array}{lll}
x, y \in \mathsf{HVar} & & \text{(Heap variables)} \\
a, b \in \mathsf{DVar} & & \text{(Data variables)} \\
A \in \mathsf{DTerm} & & \text{(Data terms)} \\
\varphi \in \mathsf{DFormula} & & \text{(Data formulas)} \\
Z \in \mathsf{RPred} & & \text{(Rec. predicates)} \\
\theta \in \mathsf{Refinement} & ::= \lambda \vec{a}.\varphi & \\
X \subseteq \mathsf{Var} & ::= x \mid a &
\end{array}
\qquad
\begin{array}{rl}
E, F \in \mathsf{HTerm} & ::= \mathsf{null} \mid x \\
\text{Æ} & ::= A \mid E \\
\Pi \in \mathsf{Pure} & ::= \mathit{true} \mid E = E \mid E \neq E \mid \\
& \quad \varphi \mid \Pi \wedge \Pi \\
H \in \mathsf{Heaplet} & ::= \mathsf{true} \mid \mathsf{emp} \mid E \mapsto [\vec{A}, \vec{E}] \mid Z(\vec{\theta}, \vec{E}) \\
\Sigma \in \mathsf{Spatial} & ::= H \mid H * \Sigma \\
P \in \mathsf{RSep} & ::= (\exists X.\ \Pi : \Sigma)
\end{array}
$$

Fig. 4. Syntax of RSep formulas.

data fields (a vector $\vec{A}$ of DTerm) followed by *heap* fields (a vector $\vec{E}$ of HTerm). (Notationally, we will use d_i to refer to the ith element of the vector $\vec{d}$, and $\vec{d}[t/d_i]$ to refer to the vector $\vec{d}$ with the ith element modified to t.) (4) Pure formulas contain heap and first-order data constraints.

Our definition is (implicitly) parameterized by a first-order theory $\mathcal{T}$. DVar denotes the set of theory variables, which we assume to be disjoint from HVar (the set of heap variables). DTerm and DFormula denote the sets of theory terms and formulas, and we assume that heap variables do not appear in theory terms.

For an RSep formula P, $\mathsf{Var}(P)$ denotes its free (data and heap) variables. We treat a Spatial formula Σ as a multiset of heaplets, and consider formulas to be equal when they are equal as multisets. For RSep formulas $P = (\exists X_P.\ \Pi_P : \Sigma_P)$ and $Q = (\exists X_Q.\ \Pi_Q : \Sigma_Q)$, we write $P * Q$ to denote the RSep formula

$$P * Q = (\exists X_P \cup X_Q.\ \Pi_P \wedge \Pi_Q : \Sigma_P * \Sigma_Q)$$

assuming that X_P is disjoint from $\mathsf{Var}(Q)$ and X_Q is disjoint from $\mathsf{Var}(P)$ (if not, then X_P and X_Q are first suitably renamed). For a set of variables X, we write $(\exists X.\ P)$ to denote the RSep formula

$$(\exists X.\ P) = (\exists X \cup X_P.\ \Pi_P : \Sigma_P)$$

Recursive predicates Each recursive predicate $Z \in \mathsf{RPred}$ is associated with a definition that describes how the predicate is unfolded. Before we formalize these definitions, we will give some examples.

The definition of the list segment predicate from Sec. 2 is:

$$\begin{aligned}
\mathsf{ls}(R, x, y) \equiv\ & (x = y : \mathsf{emp}) \vee \\
& (\exists d, n'.\ x \neq y \wedge R(d) : x \mapsto [d, n'] * \mathsf{ls}(R, n', y))
\end{aligned}$$

In the above, R is a *refinement variable*, which may be instantiated to a concrete refinement $\theta \in \mathsf{Refinement}$. For example, $\mathsf{ls}((\lambda a.a \geqslant 0), x, y)$ indicates that there is a list from x to y where every element of the list is at least 0.

A refined binary tree predicate is a more complicated example:

$$\begin{aligned}
\mathsf{bt}(Q, L, R, x) = (x = \mathsf{null} : \mathsf{emp}) & \\
\vee\ (\exists d, l, r.\ Q(d) : x \mapsto [d, l, r] & \\
* \mathsf{bt}((\lambda a.Q(a) \wedge L(d, a)), L, R, l) & \\
* \mathsf{bt}((\lambda a.Q(a) \wedge R(d, a)), L, R, r)) &
\end{aligned}$$

This predicate has three refinement variables: a unary refinement Q (which must be satisfied by every node in the tree), a binary refinement L (which is a relation that must hold between every node and its descendants to the left), and a binary

refinement R (which is a relation that must hold between every node and its descendants to the right). For example,

$$\mathsf{bt}((\lambda a.\mathit{true}), (\lambda a, b.a \geqslant b), (\lambda a, b.a \leqslant b), x)$$

indicates that x is the root of a *binary search tree*, and

$$\mathsf{bt}((\lambda a.a \geqslant 0), (\lambda a, b.a \leqslant b), (\lambda a, b.a \leqslant b), x)$$

indicates that x is the root of a *binary min-heap* with non-negative elements.

To formalize these definitions, we first define *refinement terms* and *refined formulas*: a refinement term τ is either (1) a refinement variable R or (2) an abstraction $(\lambda a_1, \ldots, a_n.\Phi)$, where Φ is a refined formula. A *refined formula* is a conjunction where each conjunct is either a data formula ($\mathsf{DFormula}$) or the application $\tau(\vec{A})$ of a refinement term to a vector of data terms (DTerm).

A *predicate definition* has the form

$$Z(\vec{R}, \vec{x}) \equiv (\exists X_1.\ \Pi_1 \wedge \Phi_1 : \Sigma_1) \vee \cdots \vee (\exists X_n.\ \Pi_n \wedge \Phi_n : \Sigma_n)$$

where $\vec{R}$ is a vector of refinement variables, $\vec{x}$ is a vector of heap variables, and where refinement terms may appear as refinements in the spatial formulas Σ_i. We refer to the disjuncts of the above formula as the *cases* for Z, and define $\mathit{cases}(Z(\vec{R}, \vec{x}))$ to be the set of cases of Z. $\vec{R}$ and $\vec{x}$ are bound in $\mathit{cases}(Z(\vec{R}, \vec{x}))$, and we will assume that predicate definitions are closed, that is, for each case of Z, the free refinement variables belong to $\vec{R}$, the free heap variables belong to $\vec{x}$, and there are no free data variables. We also assume that they are well-typed in the sense that each refinement term τ is associated with an arity, and whenever $\tau(\vec{A})$ appears in a definition, the length of $\vec{A}$ is the arity of τ.

Semantics The semantics of our logic, defined by a satisfaction relation $s, h \models Q$, is essentially standard. Each predicate $Z \in \mathsf{RPred}$ is defined to be the least solution[1] to the following equivalence:

$$s, h \models Z(\vec{\theta}, \vec{E}) \iff \exists P \in \mathit{cases}(Z(\vec{R}, \vec{x}))\ .\ s, h \models P[\vec{\theta}/\vec{R}, \vec{E}/\vec{x}]$$

Note that when substituting a λ-abstraction for a refinement variable, we implicitly β-reduce resulting applications. For example, $R(b)[(\lambda a.a \geqslant 0)/R] = b \geqslant 0$.

Semantic entailment is denoted by $P \models Q$, and provable entailment by $P \vdash Q$. When referring to a proof that $P \vdash Q$, we will mean a sequent calculus proof.

3.2 Programs

A program $\mathcal{P}$ is a tuple $\langle V, E, v_\mathrm{i}, v_\mathrm{e} \rangle$, where

- V is a set of control locations, with a distinguished *entry* node $v_\mathrm{i} \in V$ and *error* (exit) node $v_\mathrm{e} \in V$, and
- $E \subseteq V \times V$ is a set of directed edges, where each $e \in E$ is associated with a program command e^c.

We impose the restriction that all nodes $V \setminus \{v_\mathrm{i}\}$ are reachable from v_i via E, and all nodes can reach v_e. The syntax for program commands appears below. Note that the allocation command creates a record with n data fields,

[1] Our definition does not preclude ill-founded predicates; such predicates are simply unsatisfiable, and do not affect the technical development in the rest of the paper.

$D_1, \ldots, D_n$, and m heap fields, $N_1, \ldots, N_m$. To access the ith data field of a record pointed to by `x`, we use `x->D`$_i$ (and similarly for heap fields). We assume that programs are well-typed, but not necessarily memory safe.

Assignment: `x := E`	**Assumption**: `assume(`Π`)`	**Allocation**: `x := new(`n, m`)`
Heap store: `x->N`$_i$ `:= E`	**Data store**: `x->D`$_i$ `:= A`	**Disposal**: `free(x)`
Heap load: `y := x->N`$_i$	**Data load**: `y := x->D`$_i$	

As is standard, we compile assert commands to reachability of v_{e}.

4 Spatial Interpolants

In this section, we first define the notion of spatial path interpolants, which serve as memory safety proofs of program paths. We then describe a technique for computing spatial path interpolants. This algorithm has two phases: the first is a (forwards) *symbolic execution* phase, which computes the strongest memory safety proof for a path; the second is a (backwards) *interpolation* phase, which weakens the proof so that it is more likely to generalize.

Spatial path interpolants are bounded from below by the strongest memory safety proof, and (implicitly) from above by the weakest memory safety proof. Prior to considering the generation of inductive invariants using spatial path interpolants, consider what could be done with only one of the bounds, in general, with either a path-based approach or an iterative fixed-point computation. Without the upper bound, an interpolant or invariant could be computed using a standard forward transformer and widening. But this suffers from the usual problem of potentially widening too aggressively to prove the remainder of the path, necessitating the design of analyses which widen conservatively at the price of computing unnecessarily strong proofs. The upper bound neatly captures the information that must be preserved for the future execution to be proved safe. On the other hand, without the lower bound, an interpolant or invariant could be computed using a backward transformer (and lower widening). But this suffers from the usual problem that backward transformers in shape analysis explode, due to issues such as not knowing the aliasing relationship in the pre-state. The lower bound neatly captures such information, heavily reducing the potential for explosion. These advantages come at the price of operating over full paths from entry to error. Compared to a forwards iterative analysis, operating over full paths has the advantage of having information about the execution's past and future when weakening at each point along the path. A forwards iterative analysis, on the other hand, trades the information about the future for information about many past executions through the use of join or widening operations.

The development in this section is purely spatial: we do not make use of data variables or refinements in recursive predicates. Our algorithm is thus of independent interest, outside of its context in this paper. We use Sep to refer to the fragment of RSep in which the only data formula (appearing in pure assertions and in refinements) is *true* (this fragment is equivalent to classical separation logic). An RSep formula P, in particular including those in recursive predicate definitions, determines a Sep formula $\underline{P}$ obtained by replacing all refinements (both variables and λ-abstractions) with $(\lambda \vec{a}.\mathit{true})$ and all DFormulas

$\mathsf{exec}(\texttt{x := new}(k,l),\ (\exists X.\ \Pi : \Sigma)) = (\exists X \cup \{x', \vec{d}, \vec{n}\}.\ (\Pi : \Sigma)[x'/x] * x \mapsto [\vec{d}, \vec{n}])$
where $x', \vec{d}, \vec{n}$ are fresh, $\vec{d} = (d_1, \ldots, d_k)$, and $\vec{n} = (n_1, \ldots, n_l)$.

$\mathsf{exec}(\texttt{free(x)},\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}]) = (\exists X.\ \Pi \wedge \Pi^{\neq} : \Sigma)$
where $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$ and $\Pi^{\neq}$ is the conjunction of all disequalities $x \neq y$ s.t $y \mapsto [\_, \_] \in \Sigma$.

$\mathsf{exec}(\texttt{x := E},\ (\exists X.\ \Pi : \Sigma)) = (\exists X \cup \{x'\}.\ (x = E[x'/x]) * (\Pi : \Sigma)[x'/x])$
where x' is fresh.

$\mathsf{exec}(\texttt{assume}(\Pi'),\ (\exists X.\ \Pi : \Sigma)) = (\exists X.\ \Pi \wedge \underline{\Pi'} : \Sigma)$.

$\mathsf{exec}(\texttt{x->N}_i\ \texttt{:= E},\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])) = (\exists X.\ \Pi : \Sigma * x \mapsto [\vec{d}, \vec{n}[E/n_i]])$
where $i \leqslant |\vec{n}|$ and $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$.

$\mathsf{exec}(\texttt{y := x->N}_i,\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])) =$
$(\exists X \cup \{y'\}.\ (y = n_i[y'/y]) * (\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])[y'/y])$
where $i \leqslant |\vec{n}|$ and $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$, and y' is fresh.

Fig. 5. Symbolic execution for heap statements. Data statements are treated as skips.

in the pure part of P with *true*. Since recursive predicates, refinements, and DFormulas appear only positively, $\underline{P}$ is no stronger than any refinement of P. Since all refinements in Sep are trivial, we will omit them from the syntax (e.g., we will write $Z(\vec{E})$ rather than $Z((\lambda \vec{a}.true), \vec{E})$).

4.1 Definition

We define a *symbolic heap* to be a Sep formula where the spatial part is a *-conjunction of points-to heaplets and the pure part is a conjunction of pointer (dis)equalities. Given a command c and a symbolic heap S, we use $\mathsf{exec}(\mathsf{c}, S)$ to denote the symbolic heap that results from symbolically executing c starting in S (the definition of exec is essentially standard [4], and is shown in Fig. 5).

Given a program path $\pi = e_1, \ldots, e_n$, we obtain its strongest memory safety proof by symbolically executing π starting from the empty heap emp. We call this sequence of symbolic heaps the symbolic execution sequence of π, and say that a path π is *memory-feasible* if every formula in its symbolic execution sequence is consistent. The following proposition justifies calling this sequence the strongest memory safety proof.

Proposition 1. *For a path π, if the symbolic execution sequence for π is defined, then π is memory safe. If π is memory safe and memory-feasible, then its symbolic execution sequence is defined.*

Recall that our strategy for proving program correctness is based on sampling and proving the correctness of several program paths (*á la* Impact [26]). The problem with *strongest* memory safety proofs is that they do not generalize well (i.e., do not generate inductive invariants).

One solution to this problem is to take advantage of property direction. Given a desired postcondition P and a (memory-safe and -feasible) path π, the goal is to come up with a proof that is weaker than π's symbolic execution sequence, but still strong enough to show that P holds after executing π. Coming up with

such "weak" proofs is how traditional path interpolation is used in IMPACT. In light of this, we define *spatial path interpolants* as follows:

Definition 1 (Spatial path interpolant). *Let $\pi = e_1, \ldots, e_n$ be a program path with symbolic execution sequence $S_0, \ldots, S_n$, and let P be a* Sep *formula (such that $S_n \models P$). A* spatial path interpolant *for π is a sequence $I_0, \ldots, I_n$ of* Sep *formulas such that*

- *for each $i \in [0, n]$, $S_i \models I_i$;*
- *for each $i \in [1, n]$, $\{I_{i-1}\}\ e_i^c\ \{I_i\}$ is a valid triple in separation logic; and*
- *$I_n \models P$.*

Our algorithm for computing spatial path interpolants is a backwards propagation algorithm that employs a *spatial interpolation* procedure at each backwards step. Spatial interpolants for a single command are defined as:

Definition 2 (Spatial interpolant). *Given* Sep *formulas S and I' and a command* c *such that $\mathsf{exec}(\mathsf{c}, S) \models I'$, a* spatial interpolant *(for S,* c*, and I') is a* Sep *formula I such that $S \models I$ and $\{I\}\ \mathsf{c}\ \{I'\}$ is valid.*

Before describing the spatial interpolation algorithm, we briefly describe how spatial interpolation is used to compute path interpolants. Let us use $\mathsf{itp}(S, \mathsf{c}, I)$ to denote a spatial interpolant for S, c, I, as defined above. Let $\pi = e_1, \ldots, e_n$ be a program path and let P be a Sep formula. First, symbolically execute π to compute a sequence $S_0, \ldots, S_n$. Suppose that $S_n \vdash P$. Then we compute a sequence $I_0, \ldots, I_n$ by taking $I_n = P$ and (for $k < n$) $I_k = \mathsf{itp}(S_k, e_{k+1}^c, I_{k+1})$. The sequence $I_0, \ldots, I_n$ is clearly a spatial path interpolant.

4.2 Bounded Abduction

Our algorithm for spatial interpolation is based on an abduction procedure. Abduction refers to the inference of explanatory hypotheses from observations (in contrast to deduction, which derives conclusions from given hypotheses). The variant of abduction we employ in this paper, which we call *bounded abduction*, is simultaneously a form of abductive and deductive reasoning. Seen as a variant of abduction, bounded abduction adds a constraint that the abduced hypothesis be at least weak enough to be derivable from a given hypothesis. Seen as a variant of deduction, bounded abduction adds a constraint that the deduced conclusion be at least strong enough to imply some desired conclusion. Formally, we define bounded abduction as follows:

Definition 3 (Bounded abduction). *Let L, M, R be* Sep *formulas, and let X be a set of variables. A solution to the* bounded abduction problem

$$L \vdash (\exists X.\ M * [\ ]) \vdash R$$

is a Sep *formula A such that $L \models (\exists X.\ M * A) \models R$.*

Note how, in contrast to bi-abduction [11] where a solution is a pair of formulas, one constrained from above and one from below, a solution to bounded abduction problems is a single formula that is simultaneously constrained from above and below. The fixed lower and upper bounds in our formulation of abduction give

considerable guidance to solvers, in contrast to bi-abduction, where the bounds are part of the solution.

Sec. 6 presents our bounded abduction algorithm. For the remainder of this section, we will treat bounded abduction as a black box, and use $L \vdash (\exists X.\ M * [A]) \vdash R$ to denote that A is a solution to the bounded abduction problem.

4.3 Computing Spatial Interpolants

We now proceed to describe our algorithm for spatial interpolation. Given a command c and Sep formulas S and I' such that $\mathsf{exec}(\mathsf{c}, S) \vdash I'$, this algorithm must compute a Sep formula $\mathsf{itp}(S, \mathsf{c}, I')$ that satisfies the conditions of Definition 2. Several examples illustrating this procedure are given in Fig. 3.

This algorithm is defined by cases based on the command c. We present the cases for the spatial commands; the corresponding data commands are similar.

Allocate Suppose c is $\texttt{x := new}(n, m)$. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists x.\ A)$, where A is obtained as a solution to $\mathsf{exec}(\mathsf{c}, S) \vdash (\exists \vec{a}, \vec{z}.\ x \mapsto [\vec{a}, \vec{z}] * [A]) \vdash I'$, and $\vec{a}$ and $\vec{z}$ are vectors of fresh variables of length n and m, respectively.

Deallocate Suppose c is $\texttt{free(x)}$. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ I' * x \mapsto [\vec{a}, \vec{z}])$, where $\vec{a}$ and $\vec{z}$ are vectors of fresh variables whose lengths are determined by the unique heap cell which is allocated to x in S.

Assignment Suppose c is $\texttt{x := E}$. We take $\mathsf{itp}(S, \mathsf{c}, I') = I'[E/x]$.

Store Suppose c is $\texttt{x->N}_i\ \texttt{:= E}$. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ A * x \mapsto [\vec{a}, \vec{z}])$, where A is obtained as a solution to $\mathsf{exec}(\mathsf{c}, S) \vdash (\exists \vec{a}, \vec{z}.\ x \mapsto [\vec{a}, \vec{z}[E/z_i]] * [A]) \vdash I'$ and where $\vec{a}$ and $\vec{z}$ are vectors of fresh variables whose lengths are determined by the unique heap cell which is allocated to x in S.

Example 1. Suppose that S is $t \mapsto [4, y, \mathsf{null}] * x \mapsto [2, \mathsf{null}, \mathsf{null}]$ where the cells have one data and two pointer fields, c is $\texttt{t->N}_0\ \texttt{:= x}$, and I' is $\mathsf{bt}(t)$. Then we can compute $\mathsf{exec}(\mathsf{c}, S) = t \mapsto [4, x, \mathsf{null}] * x \mapsto [2, \mathsf{null}, \mathsf{null}]$, and then solve the bounded abduction problem

$$\mathsf{exec}(\mathsf{c}, S) \vdash (\exists a, z_1.\ t \mapsto [a, x, z_1] * [\,]) \vdash I'\ .$$

One possible solution is $A = \mathsf{bt}(x) * \mathsf{bt}(z_1)$, which yields

$$\mathsf{itp}(S, \mathsf{c}, I') = (\exists a, z_0, z_1.\ t \mapsto [a, z_0, z_1] * \mathsf{bt}(z_1) * \mathsf{bt}(x))\ . \qquad \lrcorner$$

Load Suppose c is $\texttt{y := x->N}_i$. Suppose that $\vec{a}$ and $\vec{z}$ are vectors of fresh variables of lengths $|\vec{A}|$ and $|\vec{E}|$ where S is of the form $\Pi : \Sigma * w \mapsto [\vec{A}, \vec{E}]$ and $\Pi : \Sigma * w \mapsto [\vec{A}, \vec{E}] \vdash x = w$ (this is the condition under which $\mathsf{exec}(\mathsf{c}, S)$ is defined, see Fig. 5). Let y' be a fresh variable, and define $\overline{S} = (y = z_i[y'/y]) * (\Pi : \Sigma * w \mapsto [\vec{a}, \vec{z}])[y'/y]$. Note that $\overline{S} \vdash (\exists y'.\ \overline{S}) \equiv \mathsf{exec}(\mathsf{c}, S) \vdash I'$.

We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ A[z_i/y, y/y'] * x \mapsto [\vec{a}, \vec{z}])$ where A is obtained as a solution to $\overline{S} \vdash (\exists \vec{a}, \vec{z}.\ x[y'/y] \mapsto [\vec{a}, \vec{z}] * [A]) \vdash I'$.

Example 2. Suppose that S is $y = t : y \mapsto [1, \mathsf{null}, x] * x \mapsto [5, \mathsf{null}, \mathsf{null}]$, c is $\texttt{y := y->N}_1$, and I' is $y \neq \mathsf{null} : \mathsf{bt}(t)$. Then $\overline{S}$ is

$$y = x \wedge y' = t : y' \mapsto [1, \mathsf{null}, x] * x \mapsto [5, \mathsf{null}, \mathsf{null}]$$

We can then solve the bounded abduction problem

$$\overline{S} \vdash (\exists a, z_0, z_1.\ y' \mapsto [a, z_0, z_1] * [\,]) \vdash I'$$

A possible solution is $y \neq \mathsf{null} \wedge y' = t : \mathsf{bt}(z_0) * \mathsf{bt}(z_1)$, yielding

$\mathsf{itp}(S, \mathsf{c}, I') = (\exists a, z_0, z_1.z_1 \neq \mathsf{null} \wedge y = t : \mathsf{bt}(z_0) * \mathsf{bt}(z_1) * y \mapsto [a, z_0, z_1])$. ⌟

Assumptions The interpolation rules defined up to this point cannot introduce recursive predicates, in the sense that if I' is a *-conjunction of points-to predicates then so is $\mathsf{itp}(S, \mathsf{c}, I')$.[2] A *-conjunction of points-to predicates is *exact* in the sense that it gives the full layout of some part of the heap. The power of recursive predicates lies in their ability to be *abstract* rather than exact, and describe only the shape of the heap rather than its exact layout. It is a special circumstance that $\{P\}\ \mathsf{c}\ \{I'\}$ holds when I' is exact in this sense and P is not: intuitively, it means that by executing c we somehow gain information about the program state, which is precisely the case for `assume` commands.

For an example of how spatial interpolation can introduce a recursive predicate at an `assume` command, consider the problem of computing an interpolant

$$\mathsf{itp}(S, \texttt{assume(x} \neq \mathsf{null}\texttt{)}, (\exists a, z.\ x \mapsto [a, z] * \mathsf{true}))$$

where $S \equiv x \mapsto [d, y] * y \mapsto [d', \mathsf{null}]$: a desirable interpolant may be $\mathsf{ls}(x, \mathsf{null}) * \mathsf{true}$. The disequality introduced by the assumption ensures that one of the *cases* of the recursive predicate $\mathsf{ls}(x, \mathsf{null})$ (where the list from x to null is empty) is impossible, which implies that the other case (where x is allocated) must hold.

Towards this end, we now define an auxiliary function intro which we will use to introduce recursive predicates for the `assume` interpolation rules. Let P, Q be Sep formulas such that $P \vdash Q$, let Z be a recursive predicate and $\vec{E}$ be a vector of heap terms. We define $\mathsf{intro}(Z, \vec{E}, P, Q)$ as follows: if $P \vdash (\exists \emptyset.\ Z(\vec{E}) * [A]) \vdash Q$ has a solution and $A \not\vdash Q$, define $\mathsf{intro}(Z, \vec{E}, P, Q) = Z(\vec{E}) * A$. Otherwise, define $\mathsf{intro}(Z, \vec{E}, P, Q) = Q$.

Intuitively, the abduction problem has a solution when P implies $Z(\vec{E})$ and $Z(\vec{E})$ can be *excised* from Q. The condition $A \not\vdash Q$ is used to ensure that the excision from Q is non-trivial (i.e., the part of the heap that satisfies $Z(\vec{E})$ "consumes" some heaplet of Q).

To define the interpolation rule for assumptions, suppose c is `assume(E` $\neq$ `F)` (the case of equality assumptions is similar). Letting $\{\langle Z_i, \vec{E}_i \rangle\}_{i \leqslant n}$ be an enumeration of the (finitely many) possible choices of Z and $\vec{E}$, we define a formula M to be the result of applying intro to I' over all possible choices of Z and $\vec{E}$:

$$M = \mathsf{intro}(Z_1, \vec{E}_1, S \wedge E \neq F, \mathsf{intro}(Z_2, \vec{E}_2, S \wedge E \neq F, \dots))$$

where the innermost occurrence of intro in this definition is $\mathsf{intro}(Z_n, \vec{E}_n, S \wedge E \neq F, I')$. Since intro preserves entailment (in the sense that if $P \vdash Q$ then $P \vdash \mathsf{intro}(Z, \vec{E}, P, Q)$), we have that $S \wedge E \neq F \vdash M$. From a proof of $S \wedge E \neq F \vdash M$, we can construct a formula M' which is entailed by S and differs from M only

[2] But if I' *does* contain recursive predicates, then $\mathsf{itp}(S, \mathsf{c}, I')$ may also.

in that it renames variables and exposes additional equalities and disequalities implied by S, and take $\mathsf{itp}(S, \mathsf{c}, I')$ to be this M'.

The construction of M' from M is straightforward but tedious. *The procedure is detailed in the extended version [2]; here, we will just give an example to give intuition on why it is necessary.* Suppose that S is $x = w : y \mapsto z$ and I' is $\mathsf{ls}(w, z)$, and c is $\texttt{assume(}x \texttt{ = } y\texttt{)}$. Since there is no opportunity to introduce new recursive predicates in I', M is simply $\mathsf{ls}(w, z)$. However, M is not a valid interpolant since $S \not\models M$, so we must expose the equality $x = w$ and rename w to y in the list segment in $M' \equiv x = w : \mathsf{ls}(y, z)$.

In practice, it is undesirable to enumerate all possible choices of Z and $\vec{E}$ when constructing M (considering that if there are k in-scope data terms, a recursive predicate of arity n requires enumerating k^n choices for $\vec{E}$). A reasonable heuristic is to let Π be the strongest pure formula implied by S, and enumerate only those combinations of Z and $\vec{E}$ such that there is some $\Pi' : \Sigma' \in \mathit{cases}(Z(\vec{R}, \vec{x}))$ such that $\underline{\Pi'}[\vec{E}/\vec{x}] \wedge \Pi \wedge x \neq y$ is unsatisfiable. For example, for $\texttt{assume(x} \neq \texttt{y)}$, this heuristic means that we enumerate only $\langle x, y\rangle$ and $\langle y, x\rangle$ (i.e, we attempt to introduce a list segment from x to y and from y to x).

We conclude this section with a theorem stating the correctness of our spatial interpolation procedure.

Theorem 1. *Let S and I' be Sep formulas and let c be a command such that $\mathsf{exec}(\mathsf{c}, S) \vdash I'$. Then $\mathsf{itp}(S, \mathsf{c}, I')$ is a spatial interpolant for S, c, and I'.*

5 Spatial Interpolation Modulo Theories

We now consider the problem of *refining* (or *strengthening*) a given separation logic proof of memory safety with information about (non-spatial) data. This refinement procedure results in a proof of a conclusion stronger than can be proved by reasoning about the heap alone. In view of our example from Fig. 3, this section addresses how to derive the third sequence (Spatial Interpolants Modulo Theories) from the second (Spatial Interpolants).

The input to our spatial interpolation modulo theories procedure is a path π, a separation logic (Sep) proof ζ of the triple $\{\mathit{true} : \mathsf{emp}\}\ \pi\ \{\mathit{true} : \mathsf{true}\}$ (i.e., a memory safety proof for π), and a postcondition φ. The goal is to transform ζ into an RSep proof of the triple $\{\mathit{true} : \mathsf{emp}\}\ \pi\ \{\varphi : \mathsf{true}\}$. The high-level operation of our procedure is as follows. First, we traverse the memory safety proof ζ and build (1) a corresponding *refined* proof ζ' where refinements may contain second-order variables, and (2) a constraint system $\mathcal{C}$ which encodes logical dependencies between the second-order variables. We then attempt to find a solution to $\mathcal{C}$, which is an assignment of data formulas to the second-order variables such that all constraints are satisfied. If we are successful, we use the solution to instantiate the second-order variables in ζ', which yields a valid RSep proof of the triple $\{\mathit{true} : \mathsf{emp}\}\ \pi\ \{\varphi : \mathsf{true}\}$.

Horn Clauses The constraint system produced by our procedure is a recursion-free set of Horn clauses, which can be solved efficiently using existing first-order interpolation techniques (see [34] for a detailed survey). Following [18], we define a *query* to be an application $Q(\vec{a})$ of a second-order variable Q to a vector of

Entailment rules

STAR
$$\frac{\mathcal{C}_0 \;\blacktriangleright\; \Pi \wedge \Phi : \Sigma_0 \vdash \Pi' \wedge \Phi' : \Sigma_0' \qquad \mathcal{C}_1 \;\blacktriangleright\; \Pi \wedge \Phi : \Sigma_1 \vdash \Pi' \wedge \Phi' : \Sigma_1'}{\mathcal{C}_0; \mathcal{C}_1 \;\blacktriangleright\; \Pi \wedge \Phi : \Sigma_0 * \Sigma_1 \vdash \Pi' \wedge \Phi' : \Sigma_0' * \Sigma_1'}$$

POINTS-TO
$$\frac{\Pi \models \Pi'}{\Phi' \leftarrow \Phi \;\blacktriangleright\; \Pi \wedge \Phi : E \mapsto [\vec{A}, \vec{F}] \vdash \Pi' \wedge \Phi' : E \mapsto [\vec{A}, \vec{F}]}$$

FOLD
$$\frac{\mathcal{C} \;\blacktriangleright\; \Pi : \Sigma \vdash \Pi' : \Sigma' * P[\vec{\tau}/\vec{R}, \vec{E}/\vec{x}]}{\mathcal{C} \;\blacktriangleright\; \Pi : \Sigma \vdash \Pi' : \Sigma' * Z(\vec{\tau}, \vec{E})} \; P \in \mathit{cases}(Z(\vec{R}, \vec{x}))$$

UNFOLD
$$\frac{\begin{array}{c}\mathcal{C}_1 \;\blacktriangleright\; \Pi : \Sigma * P_1[\vec{\tau}/\vec{R}, \vec{E}/\vec{x}] \vdash \Pi' : \Sigma' \quad \cdots \\ \mathcal{C}_n \;\blacktriangleright\; \Pi : \Sigma * P_n[\vec{\tau}/\vec{R}, \vec{E}/\vec{x}] \vdash \Pi' : \Sigma'\end{array}}{\mathcal{C}_1; \ldots; \mathcal{C}_n \;\blacktriangleright\; \Pi : \Sigma * Z(\vec{\tau}, \vec{E}) \vdash \Pi' : \Sigma'} \; \begin{array}{l}\{P_1, \ldots, P_n\} = \\ \mathit{cases}(Z(\vec{R}, \vec{x}))\end{array}$$

PREDICATE
$$\frac{\Pi \models \Pi'}{\begin{array}{l}\Phi' \leftarrow \Phi; \Psi_1' \leftarrow \Psi_1 \wedge \Phi; \ldots; \Psi_{|\vec{\tau}|}' \leftarrow \Psi_{|\vec{\tau}|} \wedge \Phi \;\blacktriangleright \\ \Pi \wedge \Phi : Z(\vec{\tau}, \vec{E}) \vdash \Pi' \wedge \Phi' : Z(\vec{\tau'}, \vec{E})\end{array}} \; \begin{array}{l}\text{Where } \tau_i = (\lambda \vec{a}_i.\Psi_i) \\ \text{and } \tau_i' = (\lambda \vec{a}_i.\Psi_i')\end{array}$$

Execution rules

DATA-ASSUME
$$\frac{\mathcal{C} \;\blacktriangleright\; P \wedge \varphi \vdash Q}{\mathcal{C} \;\blacktriangleright\; \{P\}\ \texttt{assume(}\varphi\texttt{)}\ \{Q\}}$$

FREE
$$\frac{\mathcal{C} \;\blacktriangleright\; P \vdash \Pi \wedge \Phi : \Sigma * x \mapsto [\vec{A}, \vec{E}]}{\mathcal{C} \;\blacktriangleright\; \{P\}\ \texttt{free(x)}\ \{\Pi \wedge \Phi : \Sigma\}}$$

SEQUENCE
$$\frac{\mathcal{C}_0 \;\blacktriangleright\; \{P\}\ \pi_0\ \{\widehat{O}\} \qquad \mathcal{C}_1 \;\blacktriangleright\; \{\widehat{O}\}\ \pi_1\ \{Q\}}{\mathcal{C}_0; \mathcal{C}_1 \;\blacktriangleright\; \{P\}\ \pi_0; \pi_1\ \{Q\}}$$

DATA-LOAD
$$\frac{\begin{array}{c}\mathcal{C}_0 \;\blacktriangleright\; P \vdash (\exists X.\ \Pi \wedge \widehat{\Phi} : \widehat{\Sigma} * x \mapsto [\vec{A}, \vec{E}]) \\ \mathcal{C}_1 \;\blacktriangleright\; (\exists X, a'.\ \Pi[a'/a] \wedge \widehat{\Phi}[a'/a] \wedge a = A_i[a'/a] : (\widehat{\Sigma} * x \mapsto [\vec{A}, \vec{E}])[a'/a]) \vdash Q\end{array}}{\mathcal{C}_0; \mathcal{C}_1 \;\blacktriangleright\; \{P\}\ \texttt{a := x->D}_i\ \{Q\}}$$

DATA-ASSIGN
$$\frac{\mathcal{C} \;\blacktriangleright\; (\exists a'.\ \Pi \wedge \Phi[a'/a] \wedge a = A[a'/a] : \Sigma[a'/a] \vdash Q)}{\mathcal{C} \;\blacktriangleright\; \{\Pi \wedge \Phi : \Sigma\}\ \texttt{a := A}\ \{Q\}}$$

DATA-STORE
$$\frac{\begin{array}{c}\mathcal{C}_0 \;\blacktriangleright\; P \vdash (\exists X.\ \Pi \wedge \widehat{\Phi} : \widehat{\Sigma} * x \mapsto [\vec{A}, \vec{E}]) \\ \mathcal{C}_1 \;\blacktriangleright\; (\exists X, a'.\ \Pi \wedge \widehat{\Phi} \wedge a' = A : \widehat{\Sigma} * x \mapsto [\vec{A}[a'/A_i], \vec{E}]) \vdash Q\end{array}}{\mathcal{C}_0; \mathcal{C}_1 \;\blacktriangleright\; \{P\}\ \texttt{x->D}_i\ \texttt{:= A}\ \{Q\}}$$

ALLOC
$$\frac{\mathcal{C} \;\blacktriangleright\; (\exists x', \vec{a}, \vec{x}.\ \Pi[x'/x] \wedge \Phi : \Sigma[x'/x] * x \mapsto [\vec{a}, \vec{x}]) \vdash Q}{\mathcal{C} \;\blacktriangleright\; \{\Pi \wedge \Phi : \Sigma\}\ \texttt{x := new(}n\texttt{,}m\texttt{)}\ \{Q\}}$$

Fig. 6. Constraint generation.

Refined memory safety proof ζ'	**Constraint system $\mathcal{C}$**	**Solution σ**
$\{R_0(i) : \mathsf{true}\}$	$R_0(i') \leftarrow \mathit{true}$	$R_0(i) : \mathit{true}$
`i = nondet(); x = null`	$R_1(i') \leftarrow R_0(i)$	$R_1(i) : \mathit{true}$
$\{R_1(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}1}(\nu,i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_2(i') \leftarrow R_1(i) \wedge i \neq 0 \wedge i' = i+1$	$R_2(i) : \mathit{true}$
`assume(i != 0); ...; i--;`	$R_3(i) \leftarrow R_2(i) \wedge i = 0$	$R_3(i) : \mathit{true}$
$\{R_2(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}2}(\nu,i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_4(i,d') \leftarrow R_3(i) \wedge R_{\mathsf{ls}3}(d',i)$	$R_4(i,d') : d' \geqslant 0$
`assume(i == 0)`	$R_{\mathsf{ls}2}(\nu,i') \leftarrow R_1(i) \wedge R_{\mathsf{ls}1}(\nu,i) \wedge i \neq 0 \wedge i' = i+1$	$R_{\mathsf{ls}1}(\nu,i) : \nu \geqslant i$
$\{R_3(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}3}(\nu,i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_{\mathsf{ls}2}(\nu,i') \leftarrow R_1(i) \wedge \nu = i \wedge i \neq 0 \wedge i' = i+1$	$R_{\mathsf{ls}2}(\nu,i) : \nu \geqslant i$
`assume(x != null)`	$R_{\mathsf{ls}3}(\nu,i) \leftarrow R_2(i) \wedge R_{\mathsf{ls}2}(\nu,i) \wedge i = 0$	$R_{\mathsf{ls}3}(\nu,i) : \nu \geqslant 0$
$\{(\exists d', y.\ R_4(i,d') : x \mapsto [d', y] * \mathsf{true})\}$	$d' \geqslant 0 \leftarrow R_4(i,d')$	

Fig. 7. Example constraints.

(data) variables, and define an *atom* to be either a data formula $\varphi \in \mathsf{DFormula}$ or a query $Q(\vec{a})$. A *Horn clause* is of the form $h \leftarrow b_1 \wedge \cdots \wedge b_N$ where each of $h, b_1, \ldots, b_N$ is an atom. In our constraint generation rules, it will be convenient to use a more general form which can be translated to Horn clauses: we will allow constraints of the form $h_1 \wedge \cdots \wedge h_M \leftarrow b_1 \wedge \cdots \wedge b_N$ (shorthand for the set of Horn clauses $\{h_i \leftarrow b_1 \wedge \cdots \wedge b_N\}_{1 \leqslant i \leqslant M}$) and we will allow queries to be of the form $Q(\vec{A})$ (i.e., take arbitrary data terms as arguments rather than variables). If $\mathcal{C}$ and $\mathcal{C}'$ are sets of constraints, we will use $\mathcal{C}; \mathcal{C}'$ to denote their union.

A *solution* to a system of Horn clauses $\mathcal{C}$ is a map σ that assigns each second-order variable Q of arity k a $\mathsf{DFormula}$ Q^σ with free variables drawn from $\vec{\nu} = \langle \nu_1, \ldots, \nu_k \rangle$ such that for each clause $h \leftarrow b_1 \wedge \cdots \wedge b_N$ in $\mathcal{C}$ the implication $\forall A.(h^\sigma \Leftarrow (\exists B.b_1^\sigma \wedge \cdots \wedge b_N^\sigma))$ holds, where A is the set of free variables in h and B is the set of variables free in some b_i but not in h. In the above, for any data formula φ, φ^σ is defined to be φ, and for any query $Q(\vec{a})$, $Q(\vec{a})^\sigma$ is defined to be $Q^\sigma[a_1/\nu_1, \ldots, a_k/\nu_k]$ (where k is the arity of Q).

Constraint Generation Calculus We will present our algorithm for spatial interpolation modulo theories as a calculus whose inference rules mirror the ones of separation logic. The calculus makes use of the same syntax used in recursive predicate definitions in Sec. 3. We use τ to denote a *refinement term* and Φ to denote a *refined formula*. The calculus has two types of judgements. An *entailment judgement* is of the form

$$\mathcal{C} \ \blacktriangleright \ (\exists X.\ \Pi \wedge \Phi : \Sigma) \vdash (\exists X'.\ \Pi' \wedge \Phi' : \Sigma')$$

where Π, Π' are equational pure assertions over heap terms, Σ, Σ' are refined spatial assertions, Φ, Φ' are refined formulas, and $\mathcal{C}$ is a recursion-free set of Horn clauses. Such an entailment judgement should be read as "for any solution σ to the set of constraints $\mathcal{C}$, $(\exists X.\ \Pi \wedge \Phi^\sigma : \Sigma^\sigma)$ entails $(\exists X'.\ \Pi' \wedge \Phi'^\sigma : \Sigma'^\sigma)$," where Φ^σ is Φ with all second order variables replaced by their data formula assignments in σ (and similarly for Σ^σ).

Similarly, an *execution judgement* is of the form

$$\mathcal{C} \ \blacktriangleright \ \{(\exists X.\ \Pi \wedge \Phi : \Sigma)\} \ \pi \ \{(\exists X'.\ \Pi' \wedge \Phi' : \Sigma')\}$$

where π is a path and $X, X', \Pi, \Pi', \Phi, \Phi', \Sigma, \Sigma'$, and $\mathcal{C}$ are as above. Such an execution judgement should be read as "for any solution σ to the set of constraints $\mathcal{C}$,

$$\{(\exists X.\ \Pi \wedge \Phi^\sigma : \Sigma^\sigma)\} \ \pi \ \{(\exists X'.\ \Pi' \wedge \Phi'^\sigma : \Sigma'^\sigma)\}$$

is a valid triple."

Let π be a path, let ζ be a separation logic proof of the triple $\{true : \mathsf{emp}\}\ \pi\ \{true : \mathsf{true}\}$ (i.e., a memory safety proof for π), and let $\varphi \in \mathsf{DFormula}$ be a postcondition. Given these inputs, our algorithm operates as follows. We use $\vec{v}$ to denote a vector of all data-typed program variables. The triple is *rewritten with refinements* by letting R and R' be fresh second-order variables of arity $|\vec{v}|$ and conjoining $R(\vec{v})$ and $R'(\vec{v})$ to the pre and post. By recursing on ζ, at each step applying the appropriate rule from our calculus in Fig. 6, we derive a judgement

$$\frac{\zeta'}{\mathcal{C} \ \blacktriangleright\ \{true \wedge R(\vec{v}) : \mathsf{true}\}\ \pi\ \{true \wedge R'(\vec{v}) : \mathsf{true}\}}$$

and then compute a solution σ to the constraint system

$$\mathcal{C}; \quad R(\vec{v}) \leftarrow true; \quad \varphi \leftarrow R'(\vec{v})$$

(if one exists). The algorithm then returns ζ'^{σ}, the proof obtained by applying the substitution σ to ζ'.

Intuitively, our algorithm operates by recursing on a separation logic proof, introducing refinements into formulas on the way down, and building a system of constraints on the way up. Each inference rule in the calculus encodes both the downwards and upwards step of this algorithm. For example, consider the Fold rule of our calculus: we will illustrate the intended reading of this rule with a concrete example. Suppose that the input to the algorithm is a derivation of the following form:

$$\dfrac{\dfrac{\zeta_0}{x \mapsto [a, \mathsf{null}] \vdash (\exists b, y.\ x \mapsto [b, y] * \mathsf{ls}(y, \mathsf{null}))}}{Q(i) : x \mapsto [a, \mathsf{null}] \vdash R(i) : \mathsf{ls}((\lambda a.S(x, a)), x, \mathsf{null})}\ \textsc{Fold}$$

(i.e., a derivation where the last inference rule is an application of Fold, and the conclusion has already been rewritten with refinements). We introduce refinements in the premise and recurse on the following derivation:

$$\frac{\zeta_0}{Q(i) : x \mapsto [a, \mathsf{null}] \vdash \qquad (\exists b, y.\ R(i) \wedge S(i, b) : x \mapsto [b, y] * \mathsf{ls}((\lambda a.S(x, a)), y, \mathsf{null}))}$$

The result of this recursive call is a refined derivation ζ_0' as well as a constraint system $\mathcal{C}$. We then return both (1) the refined derivation obtained by catenating the conclusion of the Fold rule onto ζ_0' and (2) the constraint system $\mathcal{C}$.

A crucial point of our algorithm is hidden inside the hat notation in Fig. 6 (e.g, $\widehat{O}$ in Sequence): this notation is used to denote the introduction of fresh second-order variables. For many of the inference rules (such as Fold), the refinements which appear in the premises follow fairly directly from the refinements which appear in the conclusion. However, in some rules entirely new formulas appear in the premises which do not appear in the conclusion (e.g., in the Sequence rule in Fig. 6, the intermediate assertion $\widehat{O}$ is an arbitrary formula which has no obvious relationship to the precondition P or the postcondition Q). We refine such formula O by introducing a fresh second-order variable for the pure assertion and for each refinement term that appears in O. The following offers a concrete example.

Example 3. Consider the trace π in Fig. 3. Suppose that we are given a memory safety proof for π which ends in an application of the SEQUENCE rule:

$$\frac{\begin{array}{c}\{true : \mathsf{emp}\}\ \pi_0\ \{true : \mathsf{ls}(x, \mathsf{null})\} \\ \{true : \mathsf{ls}(x, \mathsf{null})\}\ \pi_1\ \{(\exists b, y.\ true : x \mapsto [b, y])\}\end{array}}{\{Q(i) : \mathsf{emp}\}\ \pi_0; \pi_1\ \{(\exists b, y.\ R(i, b) : x \mapsto [b, y])\}}\ \textsc{Sequence}$$

where π is decomposed as $\pi_0; \pi_1$, π_0 is the path from 1 to 3, and π_1 is the path from 3 to 4. Let $O = true : \mathsf{ls}(x, \mathsf{null})$ denote the intermediate assertion which appears in this proof. To derive $\widehat{O}$, we introduce two fresh second order variables, S (with arity 1) and T (with arity 2), and define $\widehat{O} = S(i) : \mathsf{ls}((\lambda a.T(i, a)), x, \mathsf{null})$. The resulting inference is as follows:

$$\frac{\begin{array}{c}\{Q(i) : \mathsf{emp}\}\ \pi_0\ \{S(i) : \mathsf{ls}((\lambda a.T(i, a)), x, \mathsf{null})\} \\ \{S(i) : \mathsf{ls}((\lambda a.T(i, a)), x, \mathsf{null})\}\ \pi_1\ \{(\exists b, y.\ R(i, b) : x \mapsto [b, y])\}\end{array}}{\{Q(i) : \mathsf{emp}\}\ \pi_0; \pi_1\ \{(\exists b, y.\ R(i, b) : x \mapsto [b, y])\}} \quad \lrcorner$$

The following example provides a simple demonstration of our constraint generation procedure:

Example 4. Recall the example in Fig. 3 of Sec. 2. The row of spatial interpolants in Fig. 3 is a memory safety proof ζ of the program path. Fig. 7 shows the refined proof ζ', which is the proof ζ with second-order variables that act as placeholders for data formulas. ***For the sake of illustration, we have simplified the constraints by skipping a number of intermediate annotations in the Hoare-style proof.***

The constraint system $\mathcal{C}$ specifies the logical dependencies between the introduced second-order variables in ζ'. For instance, the relation between R_2 and R_3 is specified by the Horn clause $R_3(i) \leftarrow R_2(i) \wedge i = 0$, which takes into account the constraint imposed by `assume (i == 0)` in the path. The Horn clause $d' \geqslant 0 \leftarrow R_4(i, d')$ specifies the postcondition defined by the assertion `assert(x->D >= 0)`, which states that the value of the data field of the node x should be $\geqslant 0$.

Replacing second-order variables in ζ' with their respective solutions in σ produces a proof that the assertion at the end of the path holds (last row of Fig. 3). $\lrcorner$

Soundness and Completeness The key result regarding the constraint systems produced by these judgements is that any solution to the constraints yields a valid refined proof. The formalization of the result is the following theorem.

Theorem 2 (Soundness). *Suppose that π is a path, ζ is a derivation of the judgement $\mathcal{C} \blacktriangleright \{P\}\ \pi\ \{Q\}$, and that σ is a solution to $\mathcal{C}$. Then ζ^σ, the proof obtained by applying the substitution σ to ζ, is a (refined) separation logic proof of $\{P^\sigma\}\ \pi\ \{Q^\sigma\}$.*

Another crucial result for our counterexample generation strategy is a kind of completeness theorem, which effectively states that the strongest memory safety proof always admits a refinement.

Theorem 3 (Completeness). *Suppose that π is a memory-feasible path and ζ is a derivation of the judgement $\mathcal{C} \blacktriangleright \{R_0(\vec{v}) : \mathit{emp}\}\ \pi\ \{R_1(\vec{v}) : \mathit{true}\}$ obtained by symbolic execution. If φ is a data formula such that $\{true : \mathit{emp}\}\ \pi\ \{\varphi : \mathit{true}\}$ holds, then there is a solution σ to $\mathcal{C}$ such that $R_1^\sigma(\vec{v}) \Rightarrow \varphi$.*

$$\textsc{Empty}\ \frac{\Pi \models \Pi'}{\Pi : [\mathsf{emp}]^c \vdash \Pi' : \langle [\mathsf{emp}]^c \trianglelefteq \mathsf{emp} \rangle} \qquad \textsc{Star}\ \frac{\Pi : \Sigma_0 \vdash \Pi' : \Sigma_0' \qquad \Pi : \Sigma_1 \vdash \Pi' : \Sigma_1'}{\Pi : \Sigma_0 * \Sigma_1 \vdash \Pi' : \Sigma_0' * \Sigma_1'}$$

$$\textsc{Points-to}\ \frac{\Pi \models \Pi'}{\Pi : [E \mapsto [a, F]]^c \vdash \Pi' : \langle [E \mapsto [a, F]]^c \trianglelefteq E \mapsto [a, F] \rangle} \qquad \textsc{True}\ \frac{\Pi \models \Pi'}{\Pi : \Sigma \vdash \Pi' : \langle [\mathsf{true}]^c \trianglelefteq \mathsf{true} \rangle}$$

$$\textsc{Substitution}\ \frac{\Pi[E/x] : \Sigma[E/x] \vdash \Pi'[E/x] : \Sigma'[E/x] \qquad \Pi \models x = E}{\Pi : \Sigma \vdash \Pi' : \Sigma'} \qquad \exists\text{-RIGHT}\ \frac{P \vdash Q[\text{Æ}/x]}{P \vdash (\exists x.\ Q)}$$

Fig. 8. Coloured strengthening. All primed variables are chosen fresh.

6 Bounded Abduction

In this section, we discuss our algorithm for bounded abduction. Given a bounded abduction problem

$$L \vdash (\exists X.\ M * [\,]) \vdash R$$

we would like to find a formula A such that $L \vdash (\exists X.\ M * A) \vdash R$. Our algorithm is sound but not complete: it is possible that there exists a solution to the bounded abduction problem, but our procedure cannot find it. In fact, there is in general no complete procedure for bounded abduction, as a consequence of the fact that we do not pre-suppose that our proof system for entailment is complete, or even that entailment is decidable.

High level description Our algorithm proceeds in three steps:

1. Find a *colouring* of L. This is an assignment of a colour, either *red* or *blue*, to each heaplet appearing in L. Intuitively, red heaplets are used to satisfy M, and blue heaplets are left over. This colouring can be computed by recursion on a proof of $L \vdash (\exists X.\ M * \mathsf{true})$.
2. Find a *coloured strengthening* $\Pi : [M']^{\mathrm{r}} * [A]^{\mathrm{b}}$ of R. (We use the notation $[\Sigma]^{\mathrm{r}}$ or $[\Sigma]^{\mathrm{b}}$ to denote a spatial formula Σ of red or blue colour, respectively.) Intuitively, this is a formula that (1) entails R and (2) is coloured in such a way that the red heaplets correspond to the red heaplets of L, and the blue heaplets correspond to the blue heaplets of L. This coloured strengthening can be computed by recursion on a proof of $L \vdash R$ using the colouring of L computed in step 1.
3. Check $\Pi' : M * A \models R$, where Π' is the strongest pure formula implied by L. This step is necessary because M may be weaker than M'. If the entailment check fails, then our algorithm fails to compute a solution to the bounded abduction problem. If the entailment check succeeds, then $\Pi'' : A$ is a solution, where Π'' is the set of all equalities and disequalities in Π' which were actually used in the proof of the entailment $\Pi' : M * A \models R$ (roughly, all those equalities and disequalities which appear in the leaves of the proof tree, plus the equalities that were used in some instance of the Substitution rule).

First, we give an example to illustrate these high-level steps:

Example 5. Suppose we want to solve the following bounded abduction problem:

$$L \vdash \mathsf{ls}(x, y) * [\,] \vdash R$$

where $L = x \mapsto [a, y] * y \mapsto [b, \mathsf{null}]$ and $R = (\exists z.\ x \mapsto [a, z] * \mathsf{ls}(y, \mathsf{null}))$. Our algorithm operates as follows:

1. Colour L: $[x \mapsto [a, y]]^{\mathrm{r}} * [y \mapsto [b, \mathsf{null}]]^{\mathrm{b}}$
2. Colour R: $(\exists z.\ [x \mapsto [a, z]]^{\mathrm{r}} * [\mathsf{ls}(y, \mathsf{null})]^{\mathrm{b}})$
3. Prove the entailment

$$x \neq \mathsf{null} \wedge y \neq \mathsf{null} \wedge x \neq y : \mathsf{ls}(x, y) * \mathsf{ls}(y, \mathsf{null}) \models R$$

This proof succeeds, and uses the pure assertion $x \neq y$.
Our algorithm computes $x \neq y : \mathsf{ls}(y, \mathsf{null})$ as the solution to the bounded abduction problem. ⌟

We now elaborate our bounded abduction algorithm. We assume that L is quantifier free (without loss of generality, since quantified variables can be Skolemized) and *saturated* in the sense that for any pure formula Π', if $L \vdash \Pi'$, where $L = \Pi : \Sigma$, then $\Pi \vdash \Pi'$.

Step 1 The first step of the algorithm is straightforward. If we suppose that there exists a solution, A, to the bounded abduction problem, then by definition we must that have $L \models (\exists X.\ M * A)$. Since $(\exists X.\ M * A) \models (\exists X.\ M * \mathsf{true})$, we must also have $L \models (\exists X.\ M * \mathsf{true})$. We begin step 1 by computing a proof of $L \vdash (\exists X.\ M * \mathsf{true})$. If we fail, then we abort the procedure and report that we cannot find a solution to the abduction problem. If we succeed, then we can colour the heaplets of L as follows: for each heaplet $E \mapsto [\vec{A}, \vec{F}]$ in L, either $E \mapsto [\vec{A}, \vec{F}]$ was used in an application of the Points-to axiom in the proof of $L \vdash (\exists X.\ M * \mathsf{true})$ or not. If yes, we colour $E \mapsto [\vec{A}, \vec{F}]$ red; otherwise, we colour it blue. We denote a heaplet H coloured by a colour c by $[H]^c$.

Step 2 The second step is to find a coloured strengthening of R. Again, supposing that there is some solution A to the bounded abduction problem, we must have $L \models (\exists X.\ M * A) \models R$, and therefore $L \models R$. We begin step 2 by computing a proof of $L \vdash R$. If we fail, then we abort. If we succeed, then we define a coloured strengthening of R by recursion on the proof of $L \vdash R$. Intuitively, this algorithm operates by inducing a colouring on points-to predicates in the leaves of the proof tree from the colouring of L (via the Points-to rule in Fig. 8) and then only folding recursive predicates when all the folded heaplets have the same colour.

More formally, for each formula P appearing as the consequent of some sequent in a proof tree, our algorithm produces a mapping from heaplets in P to coloured spatial formulas. The mapping is represented using the notation $\langle \Sigma \trianglelefteq H \rangle$, which denotes that the heaplet H is mapped to the coloured spatial formula Σ. For each recursive predicate Z and each $(\exists X.\ \Pi : H_1 * \cdots * H_n) \in \mathit{cases}(Z(\vec{R}, \vec{x}))$, we define two versions of the fold rule, corresponding to when $H_1, \ldots, H_n$ are coloured homogeneously (Fold1) and heterogeneously (Fold2):

$$\textsc{Fold1}\ \frac{(\Pi : \Sigma \vdash \Pi' : \Sigma' * \langle [H_1]^c \trianglelefteq H_1 \rangle * \cdots * \langle [H_n]^c \trianglelefteq H_n \rangle)[\vec{E}/\vec{x}]}{\Pi : \Sigma \vdash \Pi' : \Sigma' * \langle [Z(\vec{E})]^c \trianglelefteq Z(\vec{E}) \rangle}$$

$$\textsc{Fold2}\ \frac{(\Pi : \Sigma \vdash \Pi' : \Sigma' * \langle \Sigma'_1 \trianglelefteq H_1 \rangle * \cdots * \langle \Sigma'_n \trianglelefteq H_n \rangle)[\vec{E}/\vec{x}]}{\Pi : \Sigma \vdash \Pi' : \Sigma' * \langle \Sigma'_1 * \cdots * \Sigma'_n \trianglelefteq Z(\vec{E}) \rangle}$$

The remaining rules for our algorithm are presented formally in Fig. 8.[3] To illustrate how this algorithm works, consider the FOLD1 and FOLD2 rules. If a given (sub-)proof finishes with an instance of FOLD that folds $H_1 * \cdots * H_n$ into $Z(\vec{E})$, we begin by colouring the sub-proof of

$$\Pi : \Sigma \vdash \Pi' : \Sigma' * H_1 * \cdots * H_n$$

This colouring process produces a coloured heaplet Σ_i for each H_i. If there is some colour c such that each Σ'_i is $[H_i]^c$, then we apply FOLD1 and $Z(\vec{E})$ gets mapped to $[Z(\vec{E})]^c$. Otherwise (if there is some i such that Σ_i is not H_i or there is some i, j such that Σ_i and Σ_j have different colours), we apply FOLD2, and map $Z(\vec{E})$ to $\Sigma_1 * \cdots * \Sigma_n$.

After colouring a proof, we define A to be the blue part of R. That is, if the colouring process ends with a judgement of
$\Pi : [\Sigma_1]^{\mathrm{r}} * [\Sigma_2]^{\mathrm{b}} \vdash \Pi' : \langle [\Sigma'_{11}]^{\mathrm{r}} * [\Sigma_{12}]^{\mathrm{b}} \trianglelefteq H_1 \rangle * \cdots * \langle [\Sigma'_{n1}]^{\mathrm{r}} * [\Sigma_{n2}]^{\mathrm{b}} \trianglelefteq H_n \rangle$
(where for any coloured spatial formula Σ, its partition into red and blue heaplets is denoted by $[\Sigma_1]^{\mathrm{r}} * [\Sigma_2]^{\mathrm{b}}$), we define A to be $\Pi' : \Sigma_{12} * \cdots * \Sigma_{n2}$. This choice is justified by the following lemma:

Lemma 1. *Suppose that*
$\Pi : [\Sigma_1]^{\mathrm{r}} * [\Sigma_2]^{\mathrm{b}} \vdash \Pi' : \langle [\Sigma'_{11}]^{\mathrm{r}} * [\Sigma_{12}]^{\mathrm{b}} \trianglelefteq H_1 \rangle * \cdots * \langle [\Sigma'_{n1}]^{\mathrm{r}} * [\Sigma_{n2}]^{\mathrm{b}} \trianglelefteq H_n \rangle$
is derivable using the rules of Fig. 8, and that the antecedent is saturated. Then the following hold:

- $\Pi' : \Sigma_{11} * \Sigma_{12} * \cdots * \Sigma_{n2} \models \Pi' : H_1 * \cdots * H_n$*;*
- $\Pi : \Sigma_1 \models \Pi' : \Sigma_{11} * \cdots * \Sigma_{n1}$*; and*
- $\Pi : \Sigma_2 \models \Pi' : \Sigma_{12} * \cdots * \Sigma_{n2}$.

Step 3 The third step of our algorithm is to check the entailment $\Pi : M * A \models R$. To illustrate why this is necessary, consider the following example:

Example 6. Suppose we want to solve the following bounded abduction problem:

$$x \neq y : x \mapsto [a, y] \vdash \mathsf{ls}(x, y) * [\,] \vdash x \mapsto [a, y] \ .$$

In Step 1, we compute the colouring $x \neq y : [x \mapsto [a, y]]^{\mathrm{r}} * [\mathsf{emp}]^{\mathrm{b}}$ of the left hand side. In step 2, we compute the colouring $[x \mapsto [a, y]]^{\mathrm{r}} * [\mathsf{emp}]^{\mathrm{b}}$ of the right hand side. However, emp is not a solution to the bounded abduction problem. In fact, there is no solution to the bounded abduction problem. Intuitively, this is because M is too weak to entail the red part of the right hand side. ⌟

7 Implementation and Evaluation

Our primary goal is to study the feasibility of our proposed algorithm. To that end, we implemented an instantiation of our generic algorithm with the linked list recursive predicate ls (as defined in Sec. 3) and refinements in the theory of linear arithmetic (QF_LRA). The following describes our implementation and evaluation of SPLINTER in detail.

[3] Note that some of the inference rules are missing. This is because these rules are inapplicable (in the case of UNFOLD and INCONSISTENT) or unnecessary (in the case of NULL-NOT-LVAL and *-PARTIAL), given our assumptions on the antecedent.

Implementation We implemented SplInter in the T2 safety and termination verifier [38]. Specifically, we extended T2's front-end to handle heap-manipulating programs, and used its safety checking component (which implements McMillan's Impact algorithm) as a basis for our implementation of SplInter. To enable reasoning in separation logic, we implemented an entailment checker for RSep along with a bounded abduction procedure.

We implemented a constraint-based solver using the linear rational arithmetic interpolation techniques of Rybalchenko and Stokkermans [35] to solve the non-recursive Horn clauses generated by SplInter. Although many off-the-shelf tools for interpolation exist (e.g., [27]) we implemented our own solver for experimentation and evaluation purposes to allow us more flexibility in controlling the forms of interpolants we are looking for. We expect that SplInter would perform even better using these highly tuned interpolation engines.

Our main goal is to evaluate the feasibility of our proposed extension of interpolation-based verification to heap and data reasoning, and not necessarily to demonstrate performance improvements against other tools. Nonetheless, we note that there are two tools that target similar programs: (1) Thor [23], which computes a memory safety proof and uses off-the-shelf numerical verifiers to strengthen it, and (2) Xisa [13], which combines shape and data abstract domains in an abstract interpretation framework. Thor cannot compute arbitrary refinements of recursive predicates (like the ones demonstrated here and required in our benchmarks) unless they are manually supplied with the required theory predicates. Instantiated with the right abstract data domains, Xisa can in principle handle most programs we target in our evaluation. (Xisa was unavailable for comparison [12].) Sec. 8 provides a detailed comparison with related work.

Benchmarks To evaluate SplInter, we used a number of linked list benchmarks that require heap and data reasoning. First, we used a number of simple benchmarks: `listdata` is similar to Fig. 2, where a linked list is constructed and its data elements are later checked; `twolists` requires an invariant comparing data elements of two lists (all elements in list A are greater than those in list B); `ptloop` tests our spatial interpolation technique, where the head of the list must not be folded in order to ensure its data element is accessible; and `refCount` is a reference counting program, where our goal is to prove memory safety (no double free). For our second set of benchmarks, we used a cut-down version of BinChunker (`http://he.fi/bchunk/`), a Linux utility for converting between different audio CD formats. BinChunker maintains linked lists and uses their data elements for traversing an array. Our property of interest is thus ensuring that all array accesses are within bounds. To test our approach, we used a number of modifications of BinChunker, `bchunk_a` to `bchunk_f`, where `a` is the simplest benchmark and `f` is the most complex one.

Heuristics We employed a number of heuristics to improve our implementation. First, given a program path to prove correct, we attempt to find a similar proof to previously proven paths that traverse the same control flow locations. This is similar to the *forced covering* heuristic of [26] to force path interpolants to generalize to inductive invariants. Second, our Horn clause solver uses Farkas'

Benchmark	#ProvePath	Time (s)	$\mathcal{T}$ Time	Sp. Time
listdata	5	1.37	0.45	0.2
twolists	5	3.12	2.06	0.27
ptloop	3	1.03	0.28	0.15
refCount	14	1.6	0.59	0.14
bchunk_a	6	1.56	0.51	0.25
bchunk_b	18	4.78	1.7	0.2
bchunk_c	69	31.6	14.3	0.26
bchunk_d	23	9.3	4.42	0.27
bchunk_e	52	30.1	12.2	0.25
bchunk_f	57	22.4	12.0	0.25

Table 1. Results of running SplInter on our benchmark set.

lemma to compute linear arithmetic interpolants. We found that minimizing the number of non-zero *Farkas coefficients* results in more generalizable refinements. A similar heuristic is employed by [1].

Results Table 1 shows the results of running SplInter on our benchmark suite. Each row shows the number of calls to ProvePath (number of paths proved), the total time taken by SplInter in seconds, the time taken to generate Horn clauses and compute theory interpolants ($\mathcal{T}$ Time), and the time taken to compute spatial interpolants (Sp. Time). SplInter proves all benchmarks correct w.r.t. their respective properties. As expected, on simpler examples, the number of paths sampled by SplInter is relatively small (3 to 14). In the bchunk_* examples, SplInter examines up to 69 paths (bchunk_c). It is important to note that, in all benchmarks, almost half of the total time is spent in theory interpolation. We expect this can be drastically cut with the use of a more efficient interpolation engine. The time taken by spatial interpolation is very small in comparison, and becomes negligible in larger examples. The rest of the time is spent in checking entailment of RSep formulas and other miscellaneous operations.

Our results highlight the utility of our proposed approach. Using our prototype implementation of SplInter, we were able to verify a set of realistic programs that require non-trivial combinations of heap and data reasoning. We expect the performance of our prototype implementation of SplInter can greatly improve with the help of state-of-the-art Horn clause solvers, and more efficient entailment checkers for separation logic.

8 Related Work

Abstraction Refinement for the Heap To the best of our knowledge, the work of Botincan et al. [8] is the only separation logic shape analysis that employs a form of abstraction refinement. It starts with a family of separation logic domains of increasing precision, and uses spurious counterexample traces (reported by forward fixed-point computation) to pick a more precise domain to restart the analysis and (possibly) eliminate the counterexample. Limitations of this technique include: (1) The precision of the analysis is contingent on the set of abstract domains it is started with. (2) The refinement strategy (in contrast to SplInter) does not guarantee progress (it may explore the same path repeatedly), and may report false positives. On the other hand, given a program

path, SplInter is guaranteed to find a proof for the path or correctly declare it an unsafe execution. (3) Finally, it is unclear whether refinement with a powerful theory like linear arithmetic can be encoded in such a framework, e.g., as a set of domains with increasingly more arithmetic predicates.

Podelski and Wies [31] propose an abstraction refinement algorithm for a shape-analysis domain with a logic-based view of three-valued shape analysis (specifically, first-order logic plus transitive closure). Spurious counterexamples are used to either refine the set of predicates used in the analysis, or refine an imprecise abstract transformer. The approach is used to verify specifications given by the user as first-order logic formulas. A limitation of the approach is that refinement is syntactic, and if an important recursive predicate (e.g., there is a list from x to null) is not explicitly supplied in the specification, it cannot be inferred automatically. Furthermore, abstract post computation can be expensive, as the abstract domain uses quantified predicates. Additionally, the analysis assumes a memory safe program to start, whereas, in SplInter, we construct a memory safety proof as part of the invariant, enabling us to detect unsafe memory operations that lead to undefined program behavior.

Beyer et al. [6] propose using shape analysis information on demand to augment numerical predicate abstraction. They use shape analysis as a backup analysis when failing to prove a given path safe without tracking the heap, and incrementally refines TVLA's [7] three-valued shape analysis [36] to track more heap information as required. As with [31], [6] makes an *a priori* assumption of memory safety and requires an expensive abstract post operator.

Finally, Manevich et al. [24] give a theoretical treatment of counterexample-driven refinement in power set (e.g., shape) abstract domains.

Combined Shape and Data Analyses The work of Magill et al. [23] infers shape and numerical invariants, and is the most closely related to ours. First, a separation logic analysis is used to construct a memory safety proof of the whole program. This proof is then *instrumented* by adding additional user-defined integer parameters to the recursive predicates appearing in the proof (with corresponding user-defined interpretations). A numerical program is generated from this instrumented proof and checked using an off-the-shelf verification tool, which need not reason about the heap. Our technique and [23]'s are similar in that we both decorate separation logic proofs with additional information: in [23], the extra information is instrumentation variables; in this paper, the extra information is refinement predicates. Neither of these techniques properly subsumes the other, and we believe that they may be profitably combined. An important difference is that we synthesize data refinements automatically from program paths, whereas [23] uses a fixed (though user-definable) abstraction.

A number of papers have proposed abstract domains for shape and data invariants. Chang and Rival [13] propose a separation logic–based abstract domain that is parameterized by programmer-supplied *invariant checkers* (recursive predicates) and a data domain for reasoning about contents of these structures. McCloskey et al. [25] also proposed a combination of heap and numeric abstract domains, this time using 3-valued structures for the heap. While the

approaches to combining shape and data information are significantly different, an advantage of our method is that it does not lose precision due to limitations in the abstract domain, widening, and join.

Bouajjani et al. [9, 10] propose an abstract domain for list manipulating programs that is parameterized by a data domain. They show that by varying the data domain, one can infer invariants about list sizes, sum of elements, etc. Quantified data automata (QDA) [17] have been proposed as an abstract domain for representing list invariants where the data in a list is described by a regular language. In [16], invariants over QDA have been synthesized using language learning techniques from concrete program executions. Expressive logics have also been proposed for reasoning about heap and data [32], but have thus far been only used for invariant checking, not invariant synthesis. A number of decision procedures for combinations of the singly-linked-list fragment of separation logic with SMT theories have recently been proposed [29, 30].

Path-based Verification A number of works proposed path-based algorithms for verification. Our work builds on McMillan's Impact technique [26] and extends it to heap/data reasoning. Earlier work [20] used interpolants to compute predicates from spurious paths in a CEGAR loop. Beyer et al. [5] proposed *path invariants*, where infeasible paths induce program slices that are proved correct, and from which predicates are mined for full program verification. Heizmann et al. [19] presented a technique that uses interpolants to compute path proofs and generalize a path into a visibly push-down language of correct paths. In comparison with SplInter, all of these techniques are restricted to first-order invariants.

Our work is similar to that of Itzhaky et al. [22], in the sense that we both generalize from bounded unrollings of the program to compute ingredients of a proof. However, they compute proofs in a fragment of first-order logic that can only express linked lists and has not yet been extended to combined heap and data properties.

Bibliography

[1] Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: Sharygina and Veith [37]

[2] Albargouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. Tech. Rep. MSR-TR-2015-4 (Jan 2015), http://research.microsoft.com/apps/pubs/default.aspx?id=238328

[3] Ball, T., Jones, R.B. (eds.): CAV, LNCS, vol. 4144. Springer (2006)

[4] Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS. LNCS, vol. 3780. Springer (2005)

[5] Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI. ACM (2007)

[6] Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball and Jones [3]

[7] Bogudlov, I., Lev-Ami, T., Reps, T.W., Sagiv, M.: Revamping TVLA: Making parametric shape analysis competitive. In: Damm, W., Hermanns, H. (eds.) CAV. LNCS, vol. 4590. Springer (2007)

[8] Botincan, M., Dodds, M., Magill, S.: Abstraction refinement for separation logic program analyses, http://www.cl.cam.ac.uk/~mb741/papers/abs_ref_draft.pdf

[9] Bouajjani, A., Dragoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV. LNCS, vol. 6174. Springer (2010)

[10] Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI. LNCS, vol. 7148. Springer (2012)
[11] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) POPL. ACM (2009)
[12] Chang, B.Y.E.: Personal communication
[13] Chang, B.E., Rival, X.: Relational inductive shape analysis. In: Necula, G.C., Wadler, P. (eds.) POPL. ACM (2008)
[14] Cook, B., Haase, C., Ouaknine, J., Parkinson, M.J., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J., König, B. (eds.) CONCUR. LNCS, vol. 6901. Springer (2011)
[15] Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS, LNCS, vol. 3920. Springer (2006)
[16] Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina and Veith [37]
[17] Garg, P., Madhusudan, P., Parlato, G.: Quantified data automata on skinny trees: An abstract domain for lists. In: Logozzo, F., Fähndrich, M. (eds.) SAS. LNCS, vol. 7935. Springer (2013)
[18] Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS. LNCS, vol. 7078. Springer (2011)
[19] Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo and Palsberg [21]
[20] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) POPL. ACM (2004)
[21] Hermenegildo, M.V., Palsberg, J. (eds.): POPL. ACM (2010)
[22] Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559. Springer (2014)
[23] Magill, S., Tsai, M., Lee, P., Tsay, Y.: Automatic numeric abstractions for heap-manipulating programs. In: Hermenegildo and Palsberg [21]
[24] Manevich, R., Field, J., Henzinger, T.A., Ramalingam, G., Sagiv, M.: Abstract counterexample-based refinement for powerset domains. In: Reps, T.W., Sagiv, M., Bauer, J. (eds.) Prog. Analysis and Compilation. LNCS, vol. 4444. Springer (2006)
[25] McCloskey, B., Reps, T.W., Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In: Cousot, R., Martel, M. (eds.) SAS. LNCS, vol. 6337. Springer (2010)
[26] McMillan, K.L.: Lazy abstraction with interpolants. In: Ball and Jones [3]
[27] McMillan, K.L.: Interpolants from Z3 proofs. In: Bjesse, P., Slobodová, A. (eds.) FMCAD. FMCAD Inc. (2011)
[28] Pérez, J.A.N., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: Hall, M.W., Padua, D.A. (eds.) PLDI. ACM (2011)
[29] Pérez, J.A.N., Rybalchenko, A.: Separation logic modulo theories. In: Shan, C. (ed.) APLAS. LNCS, vol. 8301. Springer (2013)
[30] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina and Veith [37]
[31] Podelski, A., Wies, T.: Counterexample-guided focus. In: Hermenegildo and Palsberg [21]
[32] Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Boehm, H., Flanagan, C. (eds.) PLDI. ACM (2013)
[33] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. IEEE Computer Society (2002)
[34] Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving horn clauses for verification. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE. LNCS, vol. 8164. Springer (2013)
[35] Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI. LNCS, vol. 4349. Springer (2007)
[36] Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Appel, A.W., Aiken, A. (eds.) POPL. ACM (1999)
[37] Sharygina, N., Veith, H. (eds.): CAV, LNCS, vol. 8044. Springer (2013)
[38] T2. http://research.microsoft.com/en-us/projects/t2/