Delta Logics: Logics for Change

Adithya Murali and P. Madhusudan

Department of Computer Science, University of Illinois, Urbana-Champaign

Abstract. We define delta logics, a new class of logics designed to express verification conditions of basic blocks that destructively manipulates a heap. Delta logics separately describes the part of the heap that changes and its unchanged context. Utilizing this simplicity and separation, we develop an expressive decidable delta logic that states properties of lists using a variety of measures on them, including their heaplets, their length, the multiset of keys stored in them, the min/max keys stored in them, and their sortedness. We show that delta logics and the associated decision procedure yield a practically viable verification engine through an implementation of the technique and an evaluation of it.

1 Introduction

¡¡ Introduce deductive verification ¿¿

Classical logics, such as first-order logic with least fixpoints or higher order logics, are often static in the sense that formulas describe the state of a single world. Verification conditions of imperative programs describe typically at least two different worlds— the pre-state and the post-state of a program snippet. Consequently, expressing validity of verification conditions naturally involves the challenge of expressing the evolving worlds of program states in a static logic. The classical notion of strongest post-condition for programs with scalar variables does precisely this— it expresses the precondition and the intermediate states of the program and the post-state using *auxilliary* first-order variables that capture the scalar variables at these states. The weakest precondition, again for programs with scalar variables, solves the same problem.

The focus of this paper is in generating logical formulations of verification conditions for program snippets that manipulate the heap. In this setting, the heap consists, minimally, of a set of pointer fields that are modeled by *first-order functions*, and the program's execution alters these functions. Consequently, the translation of validity of verification conditions to validity of logic formulas changes considerably.

Let us consider a set of pointer fields $\overline{p}$ and a recursive definition of a unary predicate or function $R(x)$ defined using least fixpoints over $\overline{p}$. Typical functions include properties such as "x points to a linked list segment ending at the location z", "the length of the list pointed to be x", "x points to a binary search tree", "the set of keys stored in the tree pointed to by x", "the heaplet defined by the tree pointed to by x", etc. Consider a Hoare triple of the form

$@pre(\overline{x}, \overline{p}, R)$ S $@post(\overline{x}, \overline{p}, R)$, the pre- and post-conditions use the recursively defined predicate/function R.

One simple approach is to capture the precondition using logic and use the *frame rule* to reason soundly (but incompletely) about the post-state— i.e., we can simply ignore the definition of R on the transformed heap, and just say that $R'(x)$ is guaranteed to hold if it held in the pre-state and the Δ portion of the heap modified did not intersect with the heaplet of the definition of R. This is, in practice, a very *convenient and simple* reasoning that often works, but is incomplete. For example, consider the following Hoare triple with pre/post conditions written in FO+lfp, where $list(x)$ means x points to a list, and in which case, $hlist(x)$ stands for the set of locations in that list:

$$\{list(x) \wedge list(w) \wedge y \notin hlist(w)\} \quad \texttt{y.next := w} \quad \{list(x)\}$$

In this case, the Hoare triple does hold, but frame reasoning is insufficient to argue it— since y can be in $hlist(x)$ in the pre-state, the frame rule is insufficient in inferring that the postcondition continues to hold. (The above can also be formulated in separation logic, of course, and frame reasoning alone will continue to be ineffective.)

Let us now focus on basic blocks S that do not involve function calls. We would like to generate precise verification conditions in such cases, while falling back on frame reasoning for function calls. There are several approaches in the literature that argue for this— for example the Grasshopper suite of tools handle such blocks accurately [], and there is work on expressing weakest pre-conditions for separation logic formulas and such blocks using the magic wand.

One precise formulation of the verification condition is $@pre(\overline{x}, R) \wedge T(\overline{x}, \overline{x}', \overline{p}, \overline{p}') \wedge @post(\overline{x}', \overline{p}', R')$ where T describes the effect of the program on the stack and the heap, describing how the scalar variables $\overline{x}$ and pointer-fields $\overline{p}$ have evolved to the scalar variables $\overline{x}'$ and $\overline{p}'$. Most importantly, the new recursive definitions R' are *reformulated* by replacing $\overline{p}$ in the definition of R with $\overline{p}'$.

Though the above is a precise formulation of the verification condition, it has several drawbacks. First, there is a heaplet H modified by the program, and the formula will have conjuncts of the form $\forall y \notin H. p'(y) = p(y)$, for every $p \in \overline{p}$. This introduces universal quantification, which is harder to reason with automatically— however, for basic blocks that do not involve function calls, i.e, when H is finite, we can map this into a decidable quantifier free logic (modeling p as an array and p' as an update to the array). Second, the new definition of R' depends on $\overline{p}'$, which in turn depends on the various constraints introduced by the basic block. For example, pointer fields may change depending on complex properties involving the data elements stored in the heap. Reasoning with R' which involves least fixpoints, coupled with such constraints is daunting.

Surely, there must be a simpler formulation of the verification condition! The delta changes to the heap do cause global changes and can dramatically affect the valuation of recursively defined predicates/functions, which are global. But surely, the effect on the semantics of R' must be expressible in a simpler localized fashion.

In this paper, we describe a class of logics, called *delta logics,* that are precisely meant to solve the above issues. Formulas in delta logics are Boolean combinations of two distinct kinds of formulas— one kind, called delta-specific formulas, talk strictly about the modified portion of the heap (identifiied by a bounded set of location Δ) without using any recursive definitions, while the other kind, called contextual formulas, talk strictly talk about the unbounded heap different from Δ using recursive definitions. The common free variables X between these sets of formulas are used to *communicate* information between the Δ portion of the heap and its context. In particular, a recursive definition R over the unbounded context are *parameterized* over a set of first-order communication variables Par^R, where Par^R summarizes the value of R within Δ. These variables themselves can, of course, depend on the value of R outside Δ as well, setting up mutally restricting constraints.

We prove several key results. First, we show a *separability theorem* that shows that any quantifier-free FO formula with recursive definitions can be expressed in delta logics, i.e., as a Boolean combination of delta specific formulas and context formulas. We prove this by setting up communication between the two portions of the heap, and it turns out that a new set of recursive definitions and communication variables involving the *rank* for recursive definitions is needed to accurately capture least fixpoints. These ranks can however be constrained to be *bounded* integers, as opposed to ordinals, even though the heap is infinite.

Second, we argue that verification conditions of basic blocks without function calls can be expressed precisely using delta logics. The key idea here is to set up *two parameterized sets* of recursive definitions, expressing the pre-condition using one and the post-condition using the other. The change to the heaplet can be described using FOL as usual, and the pre and post conditions can be translated to delta logic formulas using the separation theorem above. Notice that this captures naturally captures the changing Δ portion of the heap while keeping the context unchanged, hence removes the need for the universally quantified formula asserting that pointers in the context have not changed.

We turn then to specific delta logics and explore decidability results for them. We define a delta logic that expresses properties of list segments along with a variety of measures on them, including their heaplets (for expressing separation properties), their lengths, the multisets of keys stored in them, the min/max keys stored in them, and their sortedness. We consider the context-logic corresponding to this delta logic, and by exploiting the simplicity of delta logics (namely, that the recursive definitions change only with respect to the communication parameters, which correspond to base cases of these definitions), we show that they can be transformed to equivalent formulas expressed as quantifier-free formulas *without* recursive definitions. This leads us to a decision procedure for delta logics for linked lists with all the six measures above, and hence a decidable logic for verification conditions of programs manipulating linked lists with pre- and post-conditions expressed using the above measures.

Finally, we implement and evaluate our technique by expressing VCs using delta logics and validating them using our decision procedure on a suite of pro-

grams, and show it to be effective. To the best of our knowledge, this is the most expressive decidable logic over lists in existing literature.

Whie our decidability result is interesting in its own right, we emphasize the main contribution of this paper is, in our view, the definition and advocacy of delta logics as a logic for verification conditions. Traditional ways of reducing validation of verification conditions to logic embed the verifcation in logics that have much more expressive power than necessary, and harder to reason with; we think it is a reduction from an easier problem to a harder problem! Delta-logics explicitly delineate the changed portion of the heap and only allow recursive definitions to be parameterized over first-order communication variables, and avoid introducing quantification, making them easier to reason with.

2 Delta Logics

In this section, we define a general delta-logic extending many-sorted first-order logic with least fixpoints with background axiomatizations of some of the sorts.

Let us fix a many-sorted first-order signature $\Sigma = (S, \mathcal{F}, \mathcal{P}, \mathcal{C}, \mathcal{G}, \mathcal{R})$ where $S = \{\sigma_0, \ldots, \sigma_n\}$ is a nonempty finite set of sorts, $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{C}$ are sets of function symbols, relation symbols, and constant symbols, respectively, and $\mathcal{G}$ and $\mathcal{R}$ are function and relation symbols that will be recursively defined. These symbols have implicity defined an appropriate arity and a type signature.

Let σ_0 be a special sort that we refer to as the *location* sort, which will model locations of the heap. The other sorts, which we refer to as background sorts, can be arbitrary and constrained to conform to some theory (such as a theory of arithmetic or a theory of sets).

We assume the following restrictions:

- We assume all functions in $\mathcal{F}$ map either from tuples of one sort to itself or from the foreground sort σ_0 to a background sort σ_i. Relations in $\mathcal{P}$ are over tuples of one sort only.
- The functions in $\mathcal{F}$ whose domain is over the foreground sort σ_0 are *unary*. Also, relations over the foreground sort σ_0 are unary relations.
- Recursively defined functions (in $\mathcal{G}$) are all unary functions from the foreground sort σ_0 to the foreground sort or a background sort. Recursively defined relations (in $\mathcal{R}$) are all unary relations on the foreground sort σ_0.

The restriction to have unary functions from the foreground sort (which models locations) is sufficient to model pointers on the heap (unary functions from σ_0 to σ_0) and to model data stored in the heap (like the key stored at locations modeled as a function from σ_0 to a background sort of integers). This restriction will greatly simplify the presentation of delta-logics below. The restriction of having unary recursively defined functions and relations will also simplify the notation. Note that recursive definitions such as $lseg(x, y)$ that are binary can be written recursively as unary relations such as $lseg_y(x)$ (i.e., parameterized over the variable y) with recursion on the variable x.

Define FO+LFP: syntax and semantics.

2.1 Delta Logics

We parameterize delta logics by a finite set of first-order variables $\Delta = \{v_1, \ldots v_n\}$.

Intuitively, a delta logic formula $\varphi(\boldsymbol{x})$ is a formula that evaluates on a model M while *ignoring* the functions/relations on the locations interpreted for the variables in Δ.

More precisely, a semantic definition of the delta logic over Σ with respect to Δ, is defined as below. First, let us define when a pair of models over the same universe and an interpretations differ only on Δ.

Definition 1 (Models differing only on Δ). *Let M and M' be two Σ-models with universe U that interpret constants the same way, and let I be an interpretation of variables over U. Then we say (M, I) and (M', I) differ only on Δ if:*

- *for every function symbol f and for every $l \in U$, $[\![f]\!]_M(l) \neq [\![f]\!]_{M'}(l)$ only if there exists some $v \in \Delta$ such that $I(v) = l$.*
- *for every relation symbol S and for every $l \in U$, $[\![S]\!]_M(l) \not\equiv [\![S]\!]_{M'}(l)$ only if there exists some $v \in \Delta$ such that $I(v) = l$.* □

Intuitively, the above says that the interpretation of the (unary) functions and relations of the two models are precisely the same for all elements in the universe that are not interpretations of the variables in Δ.

An FOL+lfp formula over Σ, $\varphi(\boldsymbol{x})$, is a delta-logic formula if the formula does not distinguish between models that differ only on Δ:

Definition 2 (Delta-logic formulas). *An FOL+lfp formula over Σ, $\varphi(\boldsymbol{x})$, is a delta-logic formula if for every two Σ-models M and M' with the same universe and every interpretation I such that (M, I) and (M', I) differ only on Δ, $M, I \models \varphi$ iff $M', I \models \varphi'$.* □

Delta-logic formulae can be easily written using syntactic restrictions where the formula (and the recursive definitions) are written so that every occurrence of $f(t)$ (where f is a function symbol) or $P(t)$ (where P is a relation symbol), where t is a term of type σ_0, is guarded by the clause "$t \notin \Delta$", which is short for $\bigwedge_{v \in \Delta} t \neq v$).
Let us now give some examples of delta-logic formulae.

Example: Let us fix a finite set Δ of first-order variables.
The recursive definition

$$ls(x) :=_{lfp} (x = nil \vee (x \neq nil \wedge x \notin \Delta \wedge ls(n(x))) \vee (x \neq nil \wedge \bigvee_{v \in \Delta} (x{=}v \wedge b_v)))$$

is a delta-logic definition with respect to Δ. The above defines lists where by "imbibing" facts about whether a location v in Δ is a list using the free Boolean variable b_v. The definition says that x points to a list if it either is equal to the constant nil, or is not equal to nil and either x is not in Δ and $n(x)$ is a list

or x is in Δ, and the corresponding Boolean variable holds. The least fixpoing semantics of the above definition gives a unique definition: $ls(u)$ is true iff there is a path using the pointer $n()$ that either ends in nil or ends in a node v in Δ where b_v is true. Note that the formula $n(x)$ is guarded by the check $x \in \Delta$, and hence this is a delta-logic formula. Changing the model to reinterpret $n()$ over Δ (but preserving the interpretation of the variables b_v, $v \in \Delta$) will not change the definition of ls in any way.

The formula $u \notin \Delta \wedge u'{=}n(u) \wedge ls(u')$ is a delta-logic formula that uses the above definition (note that the formula $n(u)$ is again guarded by a check ensuring u is not in Δ). Again, changing the interpretation of $n()$ on Δ will not affect the truth of this formula (provided the interpretation of u and u' do not change).

3 Translating Verification Conditions to Delta Logics

4 A Decidable Delta Logic on Lists with List Measures

In this section, we will define a delta logic on linked lists equipped with *list measures*— measures of list segments that include length, its heaplet, the multiset of keys stored in the list (say, in a data-field `key`), and the minimum and maximum keys stored in it. We prove that the quantifier-free first-order logic fragment of this delta logic is *decidable*. More precisely, we show that the logic can be translated to an equisatisfiable quantifier-free first-order formula that is decidable by using a Nelson-Oppen combination of decidable theories of arithmetic, sets, and uninterpreted functions. The translation in fact will allow us to decide formulas of the form $\alpha \wedge \beta$ obtained as verification conditions in the section above, where α is a delta-logic formula on lists and list measures and β is over a decidable Nelson-Oppen combinable quantifier-free theories, and where α and β share Boolean and first-order variables. At the end of the section, we outline some generalizations of our result.

A delta logic over list measures

As usual, let us fix a set of first-order variables Δ. Let us also fix a single pointer field n.

Definition 3 (Recursive Definitions for the Logic of List Measures (LM)). *Let us fix a set of* parameter variables *P that consists of a tuple of sets of variables: a set of Boolean variables LS^v_z, a set of variables with type set of locations HLS^v_z, a set of variables of type multiset of keys $MSKEYS^v_z$, and a set of variables of type integer Max^v_z and Min^v_z, where z, v range over Δ.*

The delta-logic of list measures (LM) wrt Δ and parameter variables P is defined using the following recursive definitions, which depend crucially on the parameter variables P.

- *We have unary relations ls^P_z that capture linked list segments that end in z, where the relation for a location v in Δ is imbibed using the Boolean variables*

LS^v_z ($v \in \Delta$), and where z is any element of Δ or the constant location nil. (The relation $ls_{nil}()$ captures whether a location points to a list ending with nil.)
This is defined as follows:

$$ls^P_z(x) :=_{lfp} \Big(x{=}z \vee \Big(x{\neq}z \wedge x{\neq}nil \wedge x \notin \Delta \wedge ls^P_z(n(x)) \Big) \vee \Big(x{\neq}z \wedge x \in \Delta \wedge \bigwedge_{v \in \Delta} (x = v \Rightarrow LS^v_z) \Big) \Big)$$

- We have recursive definitions that capture the heaplet of such list-segments, where the heaplet of list-segments from an element v in Δ to z (where $z \in \Delta \cup \{nil\}$) is imbibed from the set variable HLS^v_z:

$$hls^P_z(x) :=_{lfp} \begin{cases} \emptyset & \text{if } [\![x]\!]{=}[\![z]\!] \\ \{x\} \cup hls^P_z(n(x)) & \text{if } [\![x]\!]{\neq}[\![z]\!] \wedge [\![x]\!]{\neq}[\![nil]\!] \wedge [\![x]\!] \notin [\![\Delta]\!] \\ HLS^v_z & \text{if } [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta \end{cases}$$

- We have recursive definitions that capture the multiset of data elements (through a data-field key) stored in list segments, where again the multiset of data of list-segments from an element v in Δ to z (where $z \in \Delta \cup \{nil\}$) is imbibed from the set variable $MSKeys^v_z$:

$$mskeys^P_z(x) :=_{lfp} \begin{cases} \emptyset & \text{if } [\![x]\!]{=}[\![z]\!] \\ \{key(x)\} \cup_m mskeys^P_z(n(x)) & \text{if } [\![x]\!]{\neq}[\![z]\!] \wedge [\![x]\!]{\neq}[\![nil]\!] \wedge [\![x]\!] \notin [\![\Delta]\!] \\ MSKeys^x_z & \text{if } [\![x]\!]{\neq}[\![z]\!] \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta \end{cases}$$

- We have recursive definitions that capture the maximum/minimum element of data elements stored in list segments, where again the maximum/minimum element of list-segments from an element v in Δ to z where $z \in \Delta \cup \{nil\}$) is imbibed from the data variable Max^v_z (or Min^v_z). We assume the data-domain has a linear-order $\leq$, and that there are special constants $-\infty$ and $+\infty$ that are the minimum and maximum elements of this order. Let $max(r_1, r_2) \equiv ite(r_1 \leq r_2, r_2, r_1)$.

$$Max^P_z(x) :=_{lfp} \begin{cases} -\infty & \text{if } [\![x]\!]{=}[\![z]\!] \\ max(key(x), Max^P_z(n(x))) & \text{if } [\![x]\!]{\neq}[\![z]\!] \wedge [\![x]\!]{\neq}[\![nil]\!] \wedge [\![x]\!] \notin [\![\Delta]\!] \\ MSKeys^v_z & \text{if } [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta \end{cases}$$

The function Min^P_z is similarly defined.

- We have a recursive definition that captures sortedness, using the minimum measure.

$$Sorted^P_z(x) :=_{lfp} \Big(x{=}z \ \vee \Big(x{\neq}z \wedge x{\neq}nil \wedge x \notin \Delta \wedge min^P_z(x){\neq}\bot \wedge key(x) \leq min^P_z(x) \wedge Sorted^P_z(n(x)) \Big) \vee \Big(x{\neq}z \wedge x \in \Delta \wedge min^P_z(x){\neq}\bot \wedge key(x) \leq min^P_z(x) \wedge \bigwedge_{v \in \Delta} (x = v \Rightarrow SORTED^v_z) \Big) \Big)$$

The above definitions can be written in usual syntax using *ite* expressions; we omit this formulation.

We define the *context logic of list-measures (LM)* to be quantifier-free formulas that use only the recursive definitions of LM mentioned above, and combine them as described in Figure 1. The logic LM allows first-order variables that range over locations, keys, and integers. Note that dereferencing pointers of locations is completely disallowed— the delta-specific formula that combined with context logic formulas will refer to such dereferences, depending on the particular state modeled. Here Δ is assumed to be a subset of the free location variables in the context logic formula.

LM formulas are ; the subformulae of the formula in LM are however allowed to refer to *different* sets of parameter variables.

$$\begin{aligned}
\textit{Location Term } lt &::= x \mid p_i(y) \mid \texttt{nil} * && \textit{where } y \notin \Delta \\
\textit{Integer Term } it &::= c \mid len_z^P(lt) \mid it + it \\
\textit{Key Term } keyt &::= c \mid key(lt) \mid max_z^P(lt) \mid min_z^P(lt) \mid keyt + keyt \\
\textit{Heaplet Term } hlt &::= \emptyset \mid \{lt\} \mid hls_z^P(lt) \mid hlt \cup hlt \mid hlt \cap hlt \mid hlt \setminus hlt \\
\textit{MultisetKeys Term } mskt &::= \emptyset \mid mskeys_z^P(x) \mid mskt \cup_m mskt \mid mskt \cap_m mskt \mid mskt \setminus_m mskt
\end{aligned}$$

$$\begin{aligned}
\textit{Formulas } \varphi ::= {} & b \mid ls_z^P(x) \mid sorted_z^P(x) \mid lt = lt \mid lt \in hlt \mid hlt \subseteq hlt \mid hlt = \emptyset \mid \\
& it < it \mid it = it \mid keyt < keyt \mid keyt = keyt \mid \\
& keyt \in mskt \mid mskt \subseteq_m mskt \mid \varphi \vee \varphi \mid \neg\varphi
\end{aligned}$$

Fig. 1: The context logic LM of list measures involving list-segments, heaplets, multisets of keys, max, min, and sortedness.

4.1 Deciding the logic of list measures

We can now state the main result of this section:

Theorem 1. *Given a quantifier-free formula $\varphi(\boldsymbol{x})$ in LM with recursive definitions of ls and measures of length, heaplet, keys, max, and min, there is an effective procedure that constructs a quantifier-free FOL formula $\psi(\boldsymbol{x})$ over a decidable Nelson-Oppen combination of theories of quantifier-free Presburger arithmetic, sets with cardinality constraints, and uninterpreted functions such that for* any *interpretation of the variables $\boldsymbol{x}$, there is a model that satisfies φ iff there is a model that satisfies ψ.*

Corollary 1. *The satisfiability problem for quantifier-free LM formulas is decidable.*

Let us fix a set of sets of parameters $\mathcal{P} = \{P_1, \ldots P_k\}$ (we encourage the reader to fix $k = 2$ in their mind while reading the section, as it's the most common and the logic VCs translate to, as shown in Section **??**).

We will first describe the decision procedure and its proof of correctness for the fragment of LM that involves only the three recursive definitions ls_z^P, hls_z^P, and $rank_z^P$, where $P \in \{\mathcal{P}\}$, which we will refer to as $LM[ls, hls, rank]$. Then we will extend the procedure to handle the logic with all the other measures; this latter proof requires more expressive decision procedures and pseudo-measures that make its proof harder.

Let us assume a quantifier-free $LM[ls, hls, rank]$ formula φ which is a Δ-logic formula wrt a finite set of variables Δ. Assume the (free) location variables occurring in φ is $X = \{x_1, \ldots x_n\}$ with $\Delta \subseteq X$.

In order to determine whether there is a model satsifying φ, we need to construct a universe of locations, an interpretation of the variables in X, and the heap (with the single pointer field $n()$) on all locations *outside* Δ (the definition of $n()$ on Δ, by definition, does not matter).

Our decision procedure intuitively relies on the following observations. First, note that the locations reached by using the $n()$ pointer any number of times forms the relevant set of locations that φ's truth can depend on (as φ is quantifier-free and has recursive definitions that only use the n-pointer). When pursuing the paths using the n-pointer on a location x, there are three distinct cases that can happen: (a) the path may reach a node in Δ, (b) the path may reach a node that is reachable also from another location in X, or (c) the path may never reach a location in Δ nor a location that is reachable from another location in X.

The key idea is to *collapse* paths where the reachability of them from variables in X does not change. More precisely, let L be the set of all locations reachable from X such that l is in Δ or for every location l' reachable from X such that $n(l') = l$, the set of nodes in X that have a path to l' is different from the set of nodes in X that have a path to l.

It is easy to see that there are at most $|X|-1$ locations of the above kind that are distinct from Δ, since the paths can merge at most $|X| - 1$ times forming a tree-like structure. Our key idea is now to represent these list segments that connect these kinds of locations *symbolically*, summarizing the measures on these list segments. Since there are only a bounded number of such locations and hence list segments, we can compute recursive definitions of linear measures involving them using quantifier-free and recursive-definition-free formulae.

We construct a formula ψ that is satisfiable iff φ is satisfiable, as follows. First, we fix a new set (distinct from X) of location variables $V = v_1, \ldots v_{|X|-1}$, to stand for the merging locations described above. We introduce an uninterpreted function $T : V \cup (X \setminus \Delta) \longrightarrow V \cup X \cup \{\bot\}$. Let Z be the set of variables in Δ as well as the variables in $X \setminus \Delta$ such that the recursive definitions ls_z^P, hls_z^P, $rank_z^P$, for some $P \in \mathcal{P}$, occurs in φ.

ψ is the conjunct of the following formulas:

– The formula φ (but with recursive definitions treated as uninterpreted relations and functions).
– For every $z \in Z$, we introduce an uninterpreted function $Dist_z : V \cup (X \setminus \Delta) \longrightarrow \mathbb{N} \cup \{\bot\}$ that is meant to capture the distance from any location in $V \cup X$ to z, if z is reachable from that location without going through Δ, and is $\bot$ otherwise. We add the constraint:

$$\bigwedge_{v \in V \cup (X \setminus \Delta)} \big[(Dist_z(v) = 0 \Leftrightarrow v = z) \ \wedge$$

$$v \neq z \Rightarrow \big(\ ((T(v) = \bot \vee Dist_z(T(v)) = \bot) \Rightarrow Dist_z(v) = \bot)$$

$$\wedge \ ((T(v) \neq \bot \wedge Dist_z(T(v)) \neq \bot) \Rightarrow Dist_z(v) = Dist_z(T(v)) + 1) \ \big) \big]$$

– For every $x \in X$, and for every $P \in \mathcal{P}$, we have a conjunct:

$$ls_z^P(x) \Leftrightarrow (Dist_z(x) \neq \bot \vee \bigvee_{v \in \Delta} (Dist_v(x) \neq \bot \wedge LS_z^v))$$

– For every $x \in X$, $z \in Z$, and for every $P \in \mathcal{P}$, we have a conjunct:

$$\big(Dist_z(x) = \bot \Rightarrow rank_z^P(x) = \bot\big) \ \wedge \ \big(Dist_z(x) \neq \bot \Rightarrow rank_z^P(x) = RANK_z^P\big)$$

– We capture the heaplets of list-segments from $v \in V \cup (X \setminus \Delta)$ to $T(v)$ (excluding both end-points) using a set of locations $H(v)$ and constrain them so that they are pairwise disjoint and do not contain the locations X:

$$\bigwedge_{x \in X, v \in V \cup (X \setminus \Delta)} x \notin H(v) \ \wedge \bigwedge_{v, v' \in V \cup (X \setminus \Delta)} (v \neq v' \Rightarrow H(v) \cap H(v') = \emptyset)$$

– We can then precisely capture the heaplet $hls_z^P(x)$ by taking the union of all heaplets of list segments lying on its path to z. We do this using the following constraint, for each $v \in X \cup V$:

$$(Dist_z(v) = \bot \Rightarrow hls_z^P(v) = \emptyset) \wedge (hls_z^P(z) = \emptyset) \wedge$$

$$(Dist_z(v) \neq \bot \wedge v \neq z) \Rightarrow hls_z(v) = H(v) \cup \{v\} \cup hls_z(T(v))$$

Note that the formula ψ is quantifier-free and over the combined theory of arithmetic, uninterpreted functions, and sets.

We can show the correctness of the above translation:

Theorem 2. *For any quantifier-free formula $\varphi(\mathcal{P}, X)$ of $LM[ls, hls, rank]$, the quantifier-free and recursion-free formula $\psi(\mathcal{P}, X)$ obtained φ obtained from the translation above satisfies the following property: for any interpretation of the free variables in $\mathcal{P} \cup X$, there is a model for φ iff there is a model for ψ.*

We now turn to the more complex logic $LM[ls, hls, rank, len, mskeys, min, max, sorted]$, and show that any quantifier-free formula φ in the logic can be satisfied. First, we model the multi-set of keys, minimum and maximum values and sortedness of each list-segment from v to $T(v)$ (where $v \in (X \setminus \Delta) \cup V$), which is outside Δ, using multiset variables $mskeys\mu(v)$, integer variables $min\mu(v)$, $max\mu(v)$, and $len\mu(v)$ and Boolean variables $sorted\mu(v)$. We can also aggregate them, as above, to express the sets $mskeys_z(x)$, $min_z(x)$, $max_z(x)$, $len_z(x)$ and $sorted_z(x)$, for each $z \in Z$ and each $x \in (X \setminus \Delta) \cup V$, similar to definitions of $hls_z(x)$ as defined above. One point to note is that the recursive definition of sortedness across segments will by expressed by using both $Min_z(x)$ and $Max_z(x)$ definitions, though the definition of sortedness is defined using only minimum—this is needed as expressing when concatenation of sorted list segments is sorted requires the max value of the first segment. We skip these definitions as they are easy to derive.

The main problem that remains is in *constraining* these measures so that they can be the measures of the *same* list segment. The following constraints capture these constraints, for each $v \in (X \setminus \Delta) \cup V$:

- The cardinality of $hls\mu(v)$ must be $len\mu(v)$.
- The cardinality of $mskeys\mu(v)$ must be $len\mu(v)$.
- $min\mu(v)$ and $max\mu(v)$ must be the minimum and maximum elements of $mskeys\mu(v)$.
- If $min\mu(v) = max\mu(v)$ (and they are not $\perp$), then $sorted\mu(v)$ can only be true.

The intuition is that any measures meeting the above constraints can be realized using true list segments. As for the third clause above, notice that any list segment with minumum element different from maximum can be realized using either a sorted list or an unsorted list.

The above constraints on measures, though seemingly simple, are hard to shoehorn into existing decidable theories (though the third constraint can be easily expressed). The first constraint can be expressed using quantifier-free BAPA [] (Boolean Algebra with Presburger arithmetic) constraints, which is decidable. We can get around defining the minimum of list segments by having the set of keys store only offsets from the minimum (and including the key 0 always). However, capturing max and sortedness measures in addition while preserving decidability seems hard.

Consequently, we give a new decision procedure that exploits the setup we have here. First, note that we can restrict the formulas that use the keys stored in sets to involve only membership testing of free variables in them, combinations using union and intersection, and checking emptiness of derived sets. We can *disallow checking non-emptiness* as non-emptiness of a set S can always be captured by demanding $k \in S$, for a freshly introduced free variable k without affecting satisfiability.

Our primary observation is that we can then restrict the multiset of keys to be over a *bounded* universe of elements. This bounded universe will consist of one element for each free variable of type key in the formula (call this K), and in

addition will consist of one element for each Venn region formed by the multiset of keys for each segment $(v, T(v))$ of the context heap. The idea of introducing an element for each Venn region is not new, and is found in many works that deal with combinations of sets and cardinality constraints [].

One can show that if there is a satisfiable model, then by collapsing all elements that are different than K and that are in the same Venn region can be identified by the unique representative we have chosen for that region. Taking unions and intersections will not affect the multiplicities of elements in the Venn region and the formula cannot distinguish between them (as it is quantifier-free).

Once we have bounded the universe of keys, we can represent a multiset of keys using a set of natural numbers that represent the multiplicity of elements in the multiset, and write the effect of unions and intersections using Presburger arithmetic. The cardinality of the multiset can be expressed as the sum of these numbers and the minimum and maximum key can be expressed using the smallest and largest keys in the finite universe with multiplicity greater than 0. We can hence translate the formula into a quantifier-free formula, giving the required theorem.

Note that the above procedure introduces an exponential number of variables, and hence poses challenges to be effective in practice. There are several possible ways we see to mitigate it. First, there is existing work (see []) on reasoning with BAPA that argues and builds practical algorithms that introduce far smaller universes in practice. Second, in the case where we allow only combinations of sets using union (and not intersection), and allow checking subset constraints and emptiness, we can show that introducing a *single* new element in the universe other than K suffices. The reason is that when there are no intersections, the identity of the elements do not matter and their multiplicities are preserved by representing with only one element. It turns out that there are many instances in verification of programs where the intersection of multisets of keys is never called for (intersection of *heaplets* are very important as they are needed to model separation, but we handle them using true sets as described earlier). Third, as we explain in the evaluation section, we can use some approximations of the constraints to obtain faster procedures, and this already seems to work adequately in practice.

We end with the main theorem:

Theorem 3. *For any quantifier-free formula $\varphi(\mathcal{P}, X)$ of $LM[ls, hls, rank, len, min, max, sorted]$, the quantifier-free and recursion-free formula $\psi(\mathcal{P}, X)$ obtained φ obtained from the translation above satisfies the following property: for any interpretation of the free variables in $\mathcal{P} \cup X$, there is a model for φ iff there is a model for ψ.*

5 A Separability Theorem

In this section, we show a key result: that for any quantifier-free FO+*lfp* formula we can effectively find an equivalent quantifier-free delta-logic formula. We do this by reasoning separately with the elements of the formula that are specific to

Δ, and those that are oblivious to Δ. We bring these separate analyses together with a set of parameters that we shall describe and justify below.

First, let us consider a recursively defined function R. Let the set of functions $PF = \{p_i \mid 1 \leq i \leq k\}$ (for some k) model the pointer fields (we assume that have a clause $p_i(nil) = nil$ for every $1 \leq i \leq k$). We also assume that Δ is fixed for this discussion.

We define a set of variables $\{R^d \mid d \in \Delta\}$ of the type of the range of R. These variables are in the set of parameters P. Then if R is defined as $R(x) :=_{lfp} \varphi(x)$ we define a new function corresponding to R, namely R^P, that is recursively defined as follows:

$$\begin{aligned} R^P(x) :=_{lfp} \quad & R^d \ \textit{if } [\![x]\!] = [\![d]\!] \text{ for some } d \in \Delta && \text{(delta case)} \\ & \varphi[R^P/R] \ \textit{if } [\![x]\!] \notin [\![\Delta]\!] && \text{(recursive case)} \end{aligned}$$

It is easy to see that for a formula $R(x)$, writing it as $R^P(x)$ would be a formula in context-logic since any model of it would not depend on the valuation of PF over Δ.

To capture the semantics of the original *lfp* definition, we constrain these parameters R^P. This will yield definitions that are equivalent in FO+*lfp* under such constraints.

We do this by writing constraints that, effectively, unfold the recursive definition over Δ. However in doing so we would run into a problem with cycles; for instance simply imbibing the value of $lseg_z$ from the node pointed to will not work when we have a circular list.

We handle this by introducing the notion of the 'rank' of a location w.r.t R. In particular, for the example of a circular list, if we recursively defined rank as a natural number increasing on a list starting from 0 at the location nil, there is no way to provide a valuation of every element on the cycle as pointing to a list. However, since the rank will need to communicate through the elements outside Δ to maintain this order (pointer paths between elements interpreted in Δ need not lie within it), it will also be a similarly relativised *lfp* definition with a set of parameters $\{RANK^d_R \mid d \in \Delta\}$ which are also included in P. We choose to model the rank as a function to $\mathbb{N} \cup \{\bot\}$ ($\bot$ signifies undefined rank) as follows for the recursively-defined function R:

$$\begin{aligned} Rank_R(x) :=_{lfp} \ & RANK^d_R && \textit{if } [\![x]\!] = [\![d]\!] \text{ for some } d \in \Delta \text{(delta case)} \\ & 0 && \textit{if } [\![x]\!] \notin [\![\Delta]\!] \wedge \varphi(x)[\bot/R] \neq \bot \text{(base case)} \\ & \max_{1 \leq i \leq k} \{Rank_R(p_i(x))\} && \textit{if } [\![x]\!] \notin [\![\Delta]\!] \wedge R^P(x) \neq \bot \text{(recursive case)} \end{aligned}$$

Intuitively, $Rank_R(x)$ imbibes the value $RANK^d_R$ for x in Δ, and otherwise propagates the rank across the context. This will be used below to correctly infer the value of R on elements in Δ.

Finally, we define a delta-specifc formula β_R that 'computes' the value of R inside delta by propagating the definition of R using the pointer fields on Δ. To

this end, we define more parameters in P in similar vein as above to communicate from the context to delta. This will be on *boundary* variables in $p_i(\Delta) \setminus \Delta$ for some i, and are therefore named thus:

Let $R^{p_i(\Delta)} = \{R^{p_i(d)} \mid d \in \Delta\}$ of the type of range of R and $RANK_R^{p_i(\Delta)} = \{RANK_R^{p_i(d)} \mid d \in \Delta\} \subseteq \mathbb{N} \cup \{\bot\}$ for every $1 \leq i \leq k$.

We then denote the substitution $\varphi(x)[P_R/R]$ as replacing the term $R(p_i(x))$ with $R^{p_i(x)}$ for every $1 \leq i \leq k$, and $\varphi(x)[\bot/R]$ as replacing with $\bot$. With the above, we write the following delta-specific constraint β_R for a recursively defined function R:

$$\bigwedge_{d \in \Delta} \Big[\Big(\varphi(d)[\bot/R] \neq \bot \implies R^d = \varphi(d)[\bot/R] \wedge \Big(RANK_R^d = 0\Big)\Big) \text{ (base case)}$$
$$\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P_R/R] \neq \bot) \implies \Big(R^d = \varphi(d)[P_R/R]$$
$$\wedge \Big(RANK_R^d = \max_{1 \leq i \leq k}(\{RANK_R^{p_i(d)}\}) + 1\Big)\Big)\Big) \text{ (recursive case)}$$
$$\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P_R/R] = \bot) \implies \Big(R^d = \bot \wedge \Big(RANK_R^d = \bot\Big)\Big)\Big) \Big] \text{(undefined)}$$

The above constraints capture the values of R accurately on Δ:

- the first case simply constrains the parameter R^d at a node interpreting its corresponding variable to be the value provided by the recursive function definition R, and its rank to be 0 when the interpretation for that variables satisfies the base case of the recursive definition.
- the recursive case constrains the parameter (when it does not satisfy the base case) to be the value computed by one unfolding of the definition, where the values of the descendants are also denoted by their respective parameters (whether Δ or boundary) and its rank to be one more than the maximum rank among its descendants.
- the undefined case constrains the parameter to be undefined when it must be according to an unfolding of the definition, and its rank to be undefined as well.

Lastly, we must also have that the boundary variables do in fact communicate the values of the context-logic recursive definition to Δ, i.e that the placeholder parameters for their values are indeed the values provided by the context-logic *lfp* definition R^P:

$$\bigwedge_{1 \leq i \leq k} \big[p_i(d) \notin \Delta \implies \big(R^{p_i(d)} = R^P(p_i(d))\big) \wedge \Big(RANK_R^{p_i(d)} = Rank_R(p_i(d))\Big)\big]$$

Before stating the main theorem of this section, we prove a technical lemma. This lemma states that (a) there is always a valuation of the parameters P_R that satisfies the constraints above, and (b) any valuation of the parameters that satisfies the constraints above will make the context-logic definition R^{P_R} precisely the same as R.

Lemma 1. *For any recursively defined function R,* $(\exists P_R.\ \beta_R) \wedge \big(\forall P_R.\ \big(\beta_R \implies R^{P_R} = R\big)\big)$

We now can show that any FO+*lfp* formula has an equivalent delta-logic formula. Let the set of recursive functions/predicates mentioned in α be $\mathcal{R}$, and Δ be fixed. Let $\mathcal{R}^P = \{R^{P_R} \mid R \in \mathcal{R}\}$, and $P_\mathcal{R} = \bigcup_{R \in \mathcal{R}} P_R$. Then:

Theorem 4 (Separability). $\alpha \equiv \exists P_\mathcal{R}.\, \alpha[\mathcal{R}^P/\mathcal{R}] \wedge \left(\bigwedge_{R \in \mathcal{R}} \beta_R \right)$.

Proof. Consider that α holds. From lemma 1, for every $R \in \mathcal{R}$, we can pick a valuation for P_R such that β_R holds and therefore, $R^{P_R} = R$. Thus we have that $\alpha[\mathcal{R}^P/\mathcal{R}] \wedge \left(\bigwedge_{R \in \mathcal{R}} \beta_R \right)$ holds.

Conversely, let $\alpha[\mathcal{R}^P/\mathcal{R}] \wedge \left(\bigwedge_{R \in \mathcal{R}} \beta_R \right)$ hold. Again, from Lemma 1 we have that for every $R \in \mathcal{R}$, the valuation given by the model for P_R satisfies β_R, and therefore $R^{P_R} = R$. Therefore, $\alpha[\mathcal{R}^P/\mathcal{R}][\mathcal{R}/\mathcal{R}^P] = \alpha$ holds.

Observe that the latter formula in Theorem 4 is a formula in delta-logic, i.e, is a Boolean combination of context-logic formulae and delta-specific formulae.

6 Translating VCs to Delta Logic

Let us consider the Hoare Triple: (α_{pre}, T, α_{post}) such that α_{pre} and α_{post} are FO+*lfp* formulae. The program manipulates pointers and data fields, which we model using unary functions. We can then write α_{pre} over sets of variables, fields and recursively defined functions X, P and R respectively and similarly α_{post} over sets X', P' and R' such that for every $x \in X$, there is a corresponding $x' \in X'$ abd similarly for $f \in P$, there is a corresponding symbol $f' \in P'$. Intuitively, this is used to identify the values of program variables and distinguish the state of the pointer/data fields in the universe after the execution of the program. Consequently, for every recursively defined function $r \in R$, there is a corresponding $r' \in R'$ such that $r' = r[P'/P]$ is a substitution of f' for the corresponding f in the definition of r.

We can also write T, the formula describing the program transformation, over $X \cup X' \cup X_{tmp}$ and $P \cup P'$ such that $\Delta \subseteq X \cup X' \cup X_{tmp}$, where $T = T_1 \wedge T_2$ can be written as a conjuction of:

- A quantifier-free formula T_1 such that any subterm of the form $f(t)$ for some $f \in P \cup P'$ and some term t must have $t = v$ for some $v \in \Delta$ (this is to ensure that the pointer and data fields are only referenced at variables in Δ), and
- A formula T_2: $\bigwedge_{f \in P} (\forall z.\, z \notin \Delta \implies f'(z) = f(z))$ (this describes that the pointer and data fields are changed only on variables in Δ).

This is possible since the program changes the values of the data and pointer fields on elements only within Δ, and the values of the variables in the state resulting after the program execution can be written as expressions of the values in the state before, with only a finite number of temporary variables.

It is clear that the VC that captures the goven Hoare Triple will be $\alpha_{pre} \wedge T \wedge \neg\alpha_{post}$, which by the above is a formula in FO+*lfp* within our given signature. From Theorem 4, we have that an FO+*lfp* formula can be written equivalently as a delta-logic formula. Therefore, α_{pre} and α_{post} can be rewritten to equivalent delta-logic.
However, from the theorem we will also see in particular that the resursive definitions of functions and predicates in α_{post} use functions from P' only on arguments in Δ^c. Therefore, from T_2 we have that these can be replaced with corresponding functions in P since the pointer and data fields of (locations interpreted by) variables not in Δ are unaltered by the program. Then, we also remove T_2 since the symbols in P' are no longer referred to anywhere else on an argument not in Δ.

Therefore, the VC can be written equivalently as a delta-logic formula (since T does not contain any recursively defined functions as subterms). Since delta-logic formulae are boolean combinations of context-logic fomulae and delta-specific formulae, satisfiability of the VC then becomes the meaningful question of asking independently for a model of a context, a prior state and a resulting state (with a common valuation for finitely many shared first-order variables), such the context when applied over a model of the prior state satisfies the precondition, and applied over a model of the resulting state falsifies the postcondition.

7 Implementation and evaluation of decision procedure for LM

In this section we detail the results of implementing the above-mentioned decision procedure for $LM[ls, hls, rank, len, mskeys, min, max, sorted]$. We applied our technique on a small suite of list-manipulating programs, encoding the VC in Z3 [].

The specifications for the programs were the strongest possible expressions in our logic. For example, the program *copyall* copies a given list pointed to by x into one pointed to by y, and has the precondition $ls(x, nil)$, and the postcondition $(ls(y, nil)) \wedge (hls(x, nil) \cap hls(y, nil) = \emptyset) \wedge (mkeys(x, nil) = mkeys(y, nil))$.

program	#VCs	time(s)	bugs	time(s)
append(x: list, y: list)	4	57	buggy_append(x: list, y: list)	0.275
copyall(x: list)	4	183	buggy_copyall(x: list)	36
detect_cycle(x: list)	6	5	buggy_detect_cycle(x: list)	0.719
deleteall(x: list, k: key)	5	10	buggy_deleteall(x: list, k: key)	0.129
find(x: list, k: key)	3	6	buggy_find(x: list, k: key)	0.127

insert(x: list, k: key)	4	38	buggy_insert(x: list, k: key	0.198
insert_front(x: list, k: key)	1	3	buggy_insert_front(x: list, k: key)	0.116
insert_back(x: list, k: key)	4	20	buggy_insert_back(x: list, k: key)	0.158
reverse(x: list)	3	14	buggy_reverse(x: list)	0.129
sorted_append(x: list, y: list)	4	54	buggy_sorted_append(x: list, y: list)	0.229
sorted_deleteall(x: list, k: key)	5	2	buggy_sorted_deleteall(x: list, k: key)	1.131
sorted_insert(x: list, k: key)	4	17	buggy_sorted_insert(x: list, k: key)	1.150
sorted_reverse(x: list)	3	37	buggy_sorted_reverse(x: list)	0.537
sorted_merge(x: list, y: list)	8	300	buggy_sorted_merge(x: list, y: list)	0.682

Table 1: Decision procedure for LM: experimental results

The experiements were performed on a machine with an Intel®Core™i7-7600U with clock speeds upto 3.9GHz.

8 Related Work

The work on decidable fragments of logics with recursively defined functions that can be used to verify heap-manipulating programs runs quickly into the problem of finding decidable fragments of separation logic [16] with inductive predicates, upon which a lot of work has been done. While the first works such as [1,2,6,11] handled only list segments, there has been further improvement. [12] hanldes only list segments, but allows conjunctions with pure formulae from arbitrary decidable SMT theories. [4] argues decidablility for separation logic with arbitrary/user-defined inductive predicates, and [8, 9] adapts the idea of using equisatisfiable 'base' formulas instead of inductive predicates to include arithmetic constraints satisfying a certain property. However, none of these would be directly applicable to program verification since they do not address entailment, and do not contain an implication in their logic. [12] does contain implications, but it does not allow for spatial formulas other than points-to, list segment and in particular not provide support for multiple natural measures which our work provides. However, in the light of our broader work on context-logics and delta-logics, it is valuable to consider the satisfiability problem in the context-logic since it models the unchanging and does not need to address entailments, of which these works above provide useful decision procedures.
In constrast, [13, 14] do address program verification and provides a decidable logic on spatial formulas that when reduced to SMT can be extended with other decidable SMT theories. There is reasonable overlap in the spirit of this claim to our work, where allowing expressions in arbitrary decidable SMT theories on Δ would preserve decidability as well, and would fit well within the paradigm of delta-logics. But where [13] can support trees/tree segments, it does not provide support for measures. [7, 10] differ similarly **maybe add more? Where to I place [3] ?**.
Lastly there is work in the spirit of [5,15], where one dispenses with decidability and are therefore incomparable in that respect with our work. In particular, [15]

is a powerful method capable of verifying all of our examples, but such logics are incomplete and cannot give meaningful counterexamples like our work does.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 97–109. FSTTCS'04 (2004)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Proceedings of the Third Asian Conference on Programming Languages and Systems. pp. 52–68. APLAS'05 (2005)
3. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: CONCUR 2009 - Concurrency Theory. pp. 178–195 (2009)
4. Brotherston, J., Fuhs, C., Pérez, J.A.N., Gorogiannis, N.: A decision procedure for satisfiability in separation logic with inductive predicates. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 25:1–25:10. CSL-LICS '14 (2014)
5. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties. In: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007). pp. 307–320 (2007)
6. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Proceedings of the 22Nd International Conference on Concurrency Theory. pp. 235–249. CONCUR'11 (2011)
7. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: Proceedings of the 24th International Conference on Automated Deduction. pp. 21–38. CADE'13 (2013)
8. Le, Q.L., Sun, J., Chin, W.N.: Satisfiability modulo heap-based programs. In: Computer Aided Verification. pp. 382–404 (2016)
9. Le, Q.L., Tatsuta, M., Sun, J., Chin, W.N.: A decidable fragment in separation logic with inductive predicates and arithmetic. In: Computer Aided Verification (2017)
10. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 611–622. POPL '11 (2011)
11. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 556–566. PLDI '11 (2011)
12. Pérez, J.A.N., Rybalchenko, A.: Separation logic modulo theories. In: Programming Languages and Systems (APLAS). pp. 90–106. Springer International Publishing, Cham (2013)
13. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 711–728 (2014)
14. Piskac, R., Wies, T., Zufferey, D.: Grasshopper. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 124–139 (2014)

15. Qiu, X., Garg, P., Ştefănescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 231–242. PLDI '13 (2013)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02 (2002)