Delta Logics: Logics for Change

Adithya Murali and P. Madhusudan

Department of Computer Science
University of Illinois at Urbana Champaign
{adithya5,madhu}@illinois.edu

Abstract. We define delta logics, a new class of logics designed to express verification conditions of basic blocks that destructively manipulate a heap. Instead of monolothically describing the entire heaplet modified by the basic block, delta logic formulas describe separately the part of the heap that changes and its unchanged context. We prove that any formula in first-order logic with least fixed points can be written in delta logics. 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

Classical logics, such as first-order logic with least fixpoints (FO + *lfp*) or higher order logics, are often static in the sense that formulae 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. Consequently, expressing validity of verification conditions (VCs) naturally involves the challenge of expressing the evolving worlds of program states in a static logic. The classical notion of strongest postcondition for programs with scalar variables does precisely this— it expresses the precondition, the intermediate states of the program, and the post-state using *auxiliary* 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. The pre- and post-conditions are written in quantifier-free FO + *lfp*. Several complex and interesting properties of heap manipulating programs can be expressed using recursively defined functions and predicates, such as 'x points to a linked list segment ending at z' and 'the maximum element stored in the list pointed to by x'. 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 Hoare triples to validity of verification conditions is

considerably harder, in comparison with programs with only scalar variables (in fact, it is hard for any heap logic). This is further complicated by the presence of recursively defined functions/predicates that refer to pointer and data fields, which can express complex properties combining both shape and data on the heap.

The most straightforward way to accurately capture a Hoare triple involves defining two sets of pointer variables ($\boldsymbol{p}$ and $\boldsymbol{p'}$), one for the pre-state and one for the post-state, and two versions of recursive functions that use them ($\boldsymbol{R}$ using $\boldsymbol{p}$ and $\boldsymbol{R'}$ using $\boldsymbol{p'}$). Furthermore, a quantified formula can be used to say that all pointer fields on elements not modified by the code snippet remain the same in the post-state (for every $p \in \boldsymbol{p}$, $\forall x \notin Mod.\, p'(x) = p(x)$). All of this, especially the introduction of a new set of recursive functions, makes automated reasoning of the verification conditions extremely complex.

Another way to generate verification conditions is to avoid defining the new set of recursive functions ($\boldsymbol{R'}$) for the post-state. Instead, we use *frame reasoning* to argue that recursive functions on certain locations have not changed across the snippet as the modified heaplet is disjoint from the footprint of the definition ([46, 41]). Frame reasoning is at the heart of the design of separation logic [47, 39]. However, frame reasoning alone is not complete, as when the code snippet changes the semantics of a recursive definition, we do not have a way of reasoning with this semantics (see Section 8.1 for an example).

For program snippets that involve function calls, we believe that frame reasoning is appropriate, and continue recommending it. In modular verification, where we do not examine the implementation of the function called, all we can assume is the contract it provides. Hence, a verification condition that assumes the modified heaplet is changed arbitrarily and that nothing can be known about recursive functions involving the changed heaplet (beyond what the contract promises) is often effective.

However, for program snippets *without* function calls, where we know precisely the code snippet, we argue that an accurate encoding of the verification condition is possible that is also amenable to automated reasoning.

Delta Logics

In this paper, we identify a new class of logics called *Delta logics* for expressing verification conditions that addresses the above problem. Consider a Hoare triple of a program snippet, where the snippet changes a finite portion of the heaplet, Δ. Formulae in delta logics are Boolean combinations of two kinds of formulae. The first kind are *delta-specific* formulae that talk about the modified heaplet Δ, without using any recursive definitions. The second kind, called *contextual/context-logic* formulae, strictly talk about the unbounded portion excluding Δ, and use recursive definitions. A set of first-order interface variables (called *parameters*) are used to communicate information between the two kinds of formulae. The recursive definitions occurring in the contextual formulae use these parameters, which essentially summarize the values of the recursive def-

initions within Δ. These parameters themselves can depend on values of the recursive definitions outside Δ, setting up mutual constraints (see Figure 1).

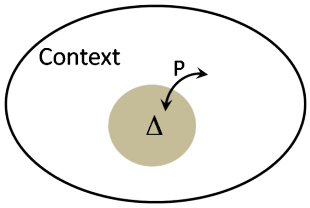

Fig. 1. Delta logic

We argue that delta logics are well suited for expressing verification conditions of program snippets without function calls. When translating Hoare triples to VCs, the semantics of the program snippet translates to a delta-specific formula. A technical challenge, however, is to translate the pre- and post-conditions (expressed in FO+*lfp*, not delta logics) to delta logic. We prove a key theorem, called the *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 formulae and contextual formulae. We shall see that this separation is nontrivial, and requires the definition of new parameterized recursive functions called *ranks* for each parameterized recursive definition.

The key advantage of expressing VCs in delta logic is that recursive definitions are allowed in contextual formulae, and hence we do not need to redefine them for the post-state. (More precisely, the recursive definition on the post-state depends on the first-order parameters of the post-state but not on the first-order pointer functions of the post-state).

We emphasize that though delta logics are particularly meant for program snippets that do not involve function calls, they can be, as we show, seamlessly combined with frame reasoning for function calls, leading us to define a comprehensive verification technique for programs with function calls. (A more verbose discussion of the intuition behind delta logics and its relationship with frame reasoning, with an illustrative example showing the separation into contextual and delta-specific formulae, as well as an overview of the VC generation mechanism, is given in Appendix 8.1, due to lack of space).

A Decidable Logic for List Measures

The second half of this paper is devoted to building decidable heap logics using the simplicity that delta logics bring in terms of expressing VCs. When VCs are expressed in delta logic, we need to reason with a single contextual universe over which there is a single set of recursive predicates and functions. This greatly simplifies reasoning with these recursive predicates and functions.

We define a delta logic LM that expresses properties of list segments along with a varied collection 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 (all these measures are expressed as recursive functions). Our technique for decidability is to design a decision procedure for the contextual logic with these measures. The procedure aims to find a *single* infinite heap model for the context where the VC may get violated by summarizing measures on list segments between locations where list segments merge. Our algorithm works by translating delta logic formulae to equisatisfiable quantifier-free formulae *without* recursive definitions. This leads us to a decision procedure for delta logics for linked lists with all the six measures above. The decision procedure crucially relies on the fact that contextual formulas are on a universe where the pointer fields do not change, resulting in a static universe that is far easier to reason with. To the best of our knowledge, this is the most expressive decidable program logic for list manipulating programs.

Finally, we implement and evaluate our technique by expressing VCs using delta logics and validating them using our decision procedure on a suite of programs, and show it to be effective both for verifying correct programs and finding bugs in incorrect ones.

2 Delta Logics

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

Quantifer-free First Order logic with least fixpoints (FO + *lfp*)

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 have recursive definitions. These symbols have an appropriate arity and a type signature implicitly defined.

Let σ_0 be a special sort that we refer to as the *foreground* sort, which will model locations on 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, $i > 0$. 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. 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 keys stored at locations modeled as a function *key* from σ_0 to a background sort of integers, the set of keys stored in a linked list pointed to by a location as a function from the foreground sort to sets 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 $ls(x, y)$ that are binary can be written recursively as unary relations such as $ls_y(x)$ (parameterized over the variable y).

The logic of quantifier-free FO+*lfp* formulae that we use consists of a pair $(Defs, \varphi)$ where *Defs* is set of mutually recursive definitions for (unary) predicates $\mathcal{R}$ and functions $\mathcal{G}$ (with least fixpoint semantics), and φ is a quantifier-free formula that uses these definitions. A definition of a recursive function G will be of the form: $G :=_{lfp} \varphi(x, \mathcal{G}(p_1(x)), \ldots, \mathcal{G}(p_n(x)), \mathcal{R}(p_1(x)), \ldots, \mathcal{R}(p_n(x)))$ where p_i are unary functions in $\mathcal{F}$ from σ_0 to σ_0, $\mathcal{G}(p_i(x))$ represents terms of the form $G'(p_i(x))$ for any $G' \in \mathcal{G}$; $\mathcal{R}(p_i(x))$ is defined similarly. We also require that φ be monotonic, i.e., the terms $G(p_i(x))$ and $R(p_i(x))$ occur in the definition under an even number of negations. Similarly, the definition of a recursive relation will be of the form $R :=_{lfp} \varphi(x, \mathcal{R}(p_1(x)), \ldots, \mathcal{R}(p_n(x)), \mathcal{G}(p_1(x)), \ldots, \mathcal{G}(p_n(x)))$.

The lattice for computing the least fixpoints for predicates is $\{\top, \bot\}$ with $\bot < \top$. For functions whose range is $\mathcal{D}$, this lattice is $\mathcal{D} \cup \{\bot\}$, where $\bot < d$ for every d in $\mathcal{D}$. The semantics of atomic formulas is that they evaluate to *false* whenever they involve a term involving $\bot$, when the formula is under an even number of negations; and to *true* otherwise.

Example 1. Consider the recursive definition: $ls_y(x) :=_{lfp} ((x = y) \vee ls_y(n(x)))$ This defines the property that x points to a list-segment ending in y. The least fixpoint semantics ensures that $ls_y(x)$ is true if and only if y is reachable from x. The formula $key(x) > 10 \wedge ls_y(x)$ is one that uses the above definition.

Delta Logics

We now define delta logics, which are a special kind of quantifier-free FO+lfp formulas. We place several restrictions on quantifier-free FO+lfp formulas for a simpler exposition. First, we restrict the kind of recursively defined predicates to have definitions $R :=_{lfp} \rho(x, e(x))$ where there is a conjunctive formula $e(x)$ that uses recursively defined functions and relations over $p_i(x)$ such that ρ is a positive Boolean combination of e and atomic formulae that do not use recursively defined functions/relations. This restricts definitions to have a unique dependence on their successive calls, and most natural definitions for heaps satisfy this property. Similarly, the restriction on recursively defined functions is that they have definitions of the form $G(x) :=_{lfp} t(x, e(x))$ where t can involve

ite (if-then-else) expressions but is free of recursively defined functions and predicates except for the use of e, which is a term that is allowed to use recursively defined functions.

We also disallow the compositional application of multiple functions defined on the foreground sort (modeling pointer fields); and hence do not have terms of the form $p_1(p_2(x))$. This is not really a restriction, since a recursive definition with such terms can be rewritten into multiple recursive definitions, each one satisfying our requirements.

We parameterize delta logics by a finite set of first-order variables $\Delta = \{v_1, \ldots v_n\}$. Delta logic formulas are Boolean combinations of *contextual* formulae and *delta-specific* formulae. We now define the former.

Contextual Formulae Intuitively, a contextual 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 contextual logic over Σ with respect to Δ is as below. First, we shall define when a pair of models over the same universe and interpretation "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 FO+lfp formula over Σ, $\varphi(\boldsymbol{x})$, is a contextual formula if the formula does not distinguish between models that differ only on Δ; we define this semantically:

Definition 2 (Contextual formulae). *A set of recursive definitions Defs over Σ is contextual for a given Δ if for every two Σ-models M and M' and every interpretation I such that (M, I) and (M', I) differ only on Δ, the semantics of the recursive definitions are the same.*

A quantifier-free FO+lfp formula (Defs, φ) *is a contextual formula over Σ, if Defs are contextual and for every two Σ-models M and M' and every interpretation I such that (M, I) and (M', I) differ only on Δ, $M, I \models \varphi$ iff $M', I \models \varphi$.* □

Contextual formulae can be easily defined syntactically as well— we can demand that every occurrence of an atomic formula ψ, in definitions or in φ, occurs in a subformula of the form $\psi \wedge \bigwedge_{t|f(t) \text{ occurs in } \psi} t \notin \Delta$, where t is of type

σ_0 and $t \notin \Delta$ is short for the formula $\bigwedge_{v \in \Delta} t \neq v$. We avoid restricting to this particular syntax as it is too cumbersome and contextual formula can be written in various other ways where it is easy to see that it is independent of Δ.
Let us now give some examples of contextual formulae.

Example 2. Let us fix a finite set Δ of first-order variables. The definition

$$ls^*_y(x) :=_{lfp} (x = y \vee (x \notin \Delta \wedge ls^*_y(n(x))))$$

is a contextual definition with respect to Δ. ls^*_y defines a list-segment ending at y that does not pass through Δ. The use of $n(x)$ is guarded by the check $x \in \Delta$, and hence the above is a contextual formula. Changing the model to reinterpret n over Δ will not change the definition of ls^*_y.

The formula $ls^*_y(z) \wedge z_1 \neq z_2$ where ls^*_y is defined as above is a contextual formula. The formula $ls^*_y(z) \wedge n(z_1) = z_2$ is contextual if $z_1 \notin \Delta$, but is not contextual otherwise.

Definition 3 (Delta-specific formula). *A delta-specific formula for a given Δ is a quantifier-free formula where for every occurrence of a term of the form $f(t)$, we have $t \in \Delta$. Furthermore, we require that the formula does not refer to any of the recursively defined functions/predicates.* □

Definition 4 (Delta Logic formula). *A quantifier-free FO+lfp formula $(Defs, \varphi)$ is delta logic formula for a given Δ if all definitions in Defs are contextual for that Δ, and φ is a Boolean combination of contextual formulae and delta-specific formulae.* □

Example 3. For $\Delta = \{z_1\}$, the formula $ls^*_y(z) \wedge n(z_1) = z_2$ is a delta logic formula that uses the above definition of ls^*_y.

3 A Separability Theorem

In this section, we show a key result: that for any quantifier-free FO+*lfp* formula we can effectively find an equisatisfiable delta logic formula. We do this by separately reasoning 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 P that we shall describe in parts and justify below. A thorough example of the constructions that follow is worked out in Appendix 8.3.

For simplicity for exposition let us consider a signature that has a single recursively defined function R. The results in this section extend smoothly to arbitrary signatures. Let the set of functions $PF = \{p_i \mid 1 \leq i \leq k\}$ (for some k) model the pointer fields (we assume that we have a clause $p_i(nil) = nil$ for every $1 \leq i \leq k$). We also assume that Δ is fixed for this discussion.

First, let us see how to convert R to a parameterized contextual definition R^P that captures R when parameters P capture certain relevant properties of Δ.

Let R be defined as $R(x) =_{lfp} \varphi(x)$. We define a set of first-order variables $\{R^d \mid d \in \Delta\}$ of the type of the range of R. These variables are part of the set of parameters P. We define a contextual recursive function corresponding to R, named R^P, that is defined as follows:

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

R^P evaluates on a location $x \in \Delta$ by simply imbibing the value of the parameter R^x associated with x. On locations in the context, it mimics the definition of R. It is easy to see that R^P is a contextual definition since its semantics does not depend on the valuation of PF over Δ (see definition of ls^*_y in Section 2 for example). If the parameters were constrained to match the semantics of R on Δ, then R^P would precisely capture R.

Second, let us constrain P so that it precisely captures R on Δ. We do this by writing constraints that, effectively, unfold the recursive definition of R over Δ. If this constraint leads us outside Δ, we would imbibe the value from R^P. Consequently, P and R^P are mutually constrained.

However, naively doing the above will not work because of cyclic dependencies (even within Δ, but also cycles created by mutual dependence with the context). We handle this by introducing the notion of the 'rank' of a location w.r.t R. In particular, for the example of a recursive definition of a 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 for every element on a cyclic list.

We hence want to define a rank for every location (in Δ and in the context). In order to express this rank definition in delta logic, we introduce a contextual rank definition, called $RANK_R$ that captures the rank on locations in the context. This definition will in turn have its own set of parameters to capture ranks for locations within Δ. Let us fix these parameters: $\{RANK^d_R \mid d \in \Delta\}$; these are added to the parameter set P.

Since a least fixed point definition of $RANK_R$ for the context will anyway ensure that cyclic dependencies are handled correctly, we will define rank such that it increases only when definitions are used within Δ but stays the same across the context. This significantly simplifies ranks (we need only a finite number of ranks, and do not deal with least fixedpoints that would otherwise require ordinal ranks).[1]

Let *Dep* be the set of all p in PF such that φ contains $R(p(x))$. We choose to model the rank as a function to $\mathbb{N} \cup \{\bot\}$ ($\bot$ signifies undefined rank) as follows

[1] Even though heaps are finite, in an FO setting, we have to consider infinite models as one cannot express finiteness of models in FO.

for the recursively defined function R (the ranks will naturally get bound by $|\Delta|$):

$$
\begin{aligned}
Rank_R(x) :=_{lfp} \quad & RANK_R^d \ \textit{if } [\![x]\!] = [\![d]\!] \text{ for some } d \in \Delta \quad \text{(delta case)} \\
& 0 \ \textit{if } [\![x]\!] \notin [\![\Delta]\!] \wedge \varphi(x)[\bot/R] \neq \bot \quad \text{(base case)} \\
& \max_{p \in Dep} \{Rank_R(p(x))\} \ \textit{if } [\![x]\!] \notin [\![\Delta]\!] \wedge \varphi(x)[\bot/R] = \bot \wedge R^P(x) \neq \bot \text{(recursive case)} \\
& \bot \ \textit{if } R^P(x) = \bot \quad \text{(undefined case)}
\end{aligned}
$$

Intuitively, $Rank_R(x)$ imbibes the value $RANK_R^d$ for x in Δ, and otherwise propagates the rank across the context. This will be used below to infer correctly the value of R on elements in Δ. In the second line above, $\varphi(x)[\bot/R]$ stands for the formula obtained by taking the definition φ of R and replacing occurrences of R with $\bot$. This corresponds to the base case. In the third line above, if $R^P(x)$ is defined and is not the base case, then the rank is the maximum of the ranks of all the locations the definition of R utilizes. Note that we do not increment the rank as this is a contextual definition. Lastly, if R^P is not defined then the rank is undefined as well.

Finally, we define a delta logic formula β_R so that the parameters R^d get constrained to compute the values for R on Δ by unfolding the definition of R using the pointer fields on Δ. When $p_i(d)$ is within Δ for d in Δ, we can compute the value of R^d using the other parameters. However, when $p_i(d)$ is not within Δ, we need to imbibe the value from the context. To this end, we define a set of *boundary variables* that correspond to locations of the form $p_i(d)$ for $d \in \Delta$: $\{b_{p_i(d)} \mid d \in \Delta, 1 \leq i \leq k\}$. For simplicity, we shall have that $p_i(nil) = nil$ for any i. To imbibe the value from boundary variables we introduce a corresponding set of parameters for the boundary variables as we did for variables in Δ: $B = \{R^{p_i(d)} \mid d \in \Delta\}$ of the type of range of R and $\{RANK_R^{p_i(d)} \mid d \in \Delta\} \subseteq \mathbb{N} \cup \{\bot\}$ for every $1 \leq i \leq k$. These parameters together with the previously defined sets of parameters constitute P.

We then denote the substitution $\varphi(x)[P/R]$ 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 $\bot$. Using the above we introduce a formula β_R for a recursively defined function R which is defined in Figure 2.

This constraint captures accurately the values of R on Δ. The former conjunct is a delta-specific formula that has the following cases:

- the base case: when the interpretation for d satisfies the base case of R, we constrain R^d to be the appropriate value and $RANK_R^d$ to be 0.
- the recursive case constrains the R^d (when d does not satisfy the base case) to be the value computed by one unfolding of the definition, where the values on the locations it depends are modeled using their respective parameters (whether Δ or boundary). We also constrain $RANK_R^d$ to be one more than the maximum rank among the locations on which d depends.
- the undefined case: when unfolding the definition on d yields $\bot$, we constrain R^d and $RANK_R^d$ to be $\bot$.

$$\bigwedge_{d \in \Delta} \Big[\Big(\varphi(d)[\bot/R] \neq \bot \implies \Big(R^d = \varphi(d)[\bot/R] \wedge RANK_R^d = 0 \Big) \Big) \quad \text{(base case)}$$

$$\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P/R] \neq \bot) \implies \quad \text{(recursive case)}$$

$$\Big(R^d = \varphi(d)[P/R] \ \wedge \ RANK_R^d = \max_{p \in Dep} \ (\{RANK_R^{p(d)}\}) + 1 \Big) \Big)$$

$$\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P/R] = \bot) \implies \Big(R^d = \bot \wedge \Big(RANK_R^d = \bot \Big) \Big) \Big) \Big]$$

(undefined case)

$$\wedge \bigwedge_{d \in \Delta, \, (1 \leq i \leq k} \Big((p_i(d) = b_{p_i(d)}) \wedge \Big[b_{p_i(d)} \notin \Delta \implies \Big(R^{p_i(d)} = R^P(b_{p_i(d)}) \Big)$$

$$\wedge \Big(RANK_R^{p_i(d)} = Rank_R(b_{p_i(d)}) \Big) \Big] \Big)$$

Fig. 2. β_R constraints that make R^P equivalent to R.

Lastly we must also have that the parameters for boundary variables do in fact imbibe their values from the contextual recursive definitions R^P and $Rank_R$. We therefore fix the boundary variables as those dereferenced from elements in Δ and constrain them as in the second conjunct (last line) in Figure 2.

For a given quantifier-free FO+lfp formula α that uses the recursive definition R, the resulting delta logic formula will be the conjunction of α with R^P replaced for R and the β_R formula. The additional boundary variables introduced and the parameters will be existentially quantified.

Theorem 1 (Separability). $\alpha \equiv \exists B. \exists P. \alpha[R^P/R] \wedge \beta_R$

We prove this theorem in the Appendix, Section 8.2.

Observe that the formula within the quantifiers on the right hand side in Theorem 1 is a formula in delta logic, i.e, is a Boolean combination of contextual formulae and delta-specific formulae. If we do not existentially quantify the variables, we can say that for every FO+*lfp* formula there exists an *equisatisfiable* delta logic formula, and it is indeed this equisatisfiable formula that is used in the section on decidability (Section 5). Further, as we shall show in our proof of Theorem 1, given a model, the valuation for the parameters and the boundary variables is *uniquely determined* for satisfying β_R. Therefore the quantifier in Theorem 1 can be universal as well, yielding:

Theorem 2 (Alternate statement of Separability). $\alpha \equiv \forall B. \forall P. \beta_R \Rightarrow \alpha[R^P/R]$

4 Translating Verification Conditions to Delta Logics

We now explain the crux of the motivation behind delta logics, namely that they can naturally express verification conditions, without introducing extra

quantification or complex reformulation of recursive definitions. This is explained more informally in Section 8.1 in the Appendix.

Consider a Hoare triple: $\langle \alpha_{pre}(X, P, \mathcal{R}),\ S,\ \alpha_{post}(X, P, \mathcal{R})\rangle$, such that α_{pre} and α_{post} utilise a set of relations and functions $\mathcal{R}$ with recursive definitions defined using FO+*lfp*. The program manipulates a set of scalar variables X, and pointer and data fields P, the latter being modeled as unary functions.

The verification condition is then of the form $\alpha_{pre}(X, P, \mathcal{R}) \wedge T(X, X', P, P') \Rightarrow \alpha_{post}(X', P', \mathcal{R}')$, where T captures the semantics of the program snippet S, which has no function calls. T can be expressed as a delta-specific formula T_{Δ} conjoined with the formula by modeling the pointer change in the program transformation as an array update. For example, if the program changes the pointer f on y to z, we can write the new pointer f' as $f' = f[y \mapsto z]$. This is an *implicitly* quantified formula since it must say $\forall x \notin \Delta.\, f'(x) = f(x)$. However, since Δ is finite, this function update can be expressed as a quantifier-free formula (say, using if-then-else expressions), and T_{Δ} is therefore a delta-specific formula. Notice that the definitions of $\mathcal{R}'$ are obtained from the definitions of $\mathcal{R}$ by substituting P' for P.

Let us now show how to express this VC using an equivalent quantifier-free delta logic formula. First, using the separability theorem, Theorem 1 and Theorem 2, we can write α_{pre} and α_{post} as delta logic formulae $\alpha^*_{pre}(U, X, P, \mathcal{R}^U)$ and $\alpha^*_{post}(V, X', P', (\mathcal{R}')^V)$ where the former is existentially quantified and the latter is universally quantified. Notice that the definition of R' uses the transformed fields P' and we translate using *different* sets of parameters: U and V. However, observe that since $(R')^V$ is a contextual definition, we know that it does not refer to the changed heaplet Δ. Consequently, we can replace P' with P uniformly in $(R')^V$, which yields R^V.

This leaves us with a verification condition in delta logic of the form:

$$\alpha^*_{pre}(U, X, P, \mathcal{R}^U) \wedge T_{\Delta}(X, X', P, P') \Rightarrow \alpha^*_{post}(V, X', P', \mathcal{R}^V)$$

Incorporating function calls with delta logic Thus far, we have focused on formulating the VC for basic blocks *without function calls* for heap manipulating programs and have shown a decidable logic of lists and list-measures that solves the delta logic VC. As we have stated earlier, we recommend the continued use of frame reasoning to perform an inference on the post-state across a function call. Incorporating reasoning about programs with function calls using frame reasoning along with delta logics is quite simple and is detailed in Appendix 8.5. This allows us to verify a large suite of programs which we will detail in the experiments section.

5 A Decidable Delta Logic on Lists with List Measures

In this section, we embark on the second goal of this paper—to define a decidable delta logic on linked lists equipped with *list measures*. We define a delta logic on list segments with various measures, including its length, 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 delta logic is *decidable*, crucially relying on the fact that pointer functions do not change on the context heap.

We prove decidability by first proving that the corresponding contextual logic formulae can be translated to equisatisfiable quantifier-free first-order formulae. Consequently, a delta logic formula, being a Boolean combination of contextual formulae and delta-specific formulae can be translated to quantifier-free formulae as well. The latter is decidable using a Nelson-Oppen combination [37] of decidable theories of arithmetic, sets, and uninterpreted functions.

5.1 The contextual logic of list measures

$$
\begin{aligned}
\textit{Location Term } lt &::= x \mid p_i(y) \mid \texttt{nil} \qquad \textit{where } y \notin \Delta \\
\textit{Integer Term } it &::= c \mid len_z^P(lt) \mid it + it \mid it - it \\
\textit{Key Term } keyt &::= c \mid key(lt) \mid max_z^P(lt) \mid min_z^P(lt) \mid keyt + keyt \mid keyt - keyt \\
\textit{Heaplet Term } hlt &::= \emptyset \mid \{lt\} \mid hls_z^P(lt) \mid hlt \cup hlt \mid hlt \cap hlt \mid hlt \setminus hlt \\
\textit{MultisetKeys Term } mskt &::= \emptyset \mid \{keyt\} \mid mskeys_z^P(x) \mid mskt \cup_m mskt \mid mskt \cap_m mskt \mid mskt \setminus_m mskt \\
\textit{Formulas } \varphi &::= true \mid false \mid ls_z^P(x) \mid sorted_z^P(x) \mid lt = lt \mid lt \in hlt \mid hlt \subseteq hlt \mid hlt = \emptyset \mid \\
&\quad it < it \mid it = it \mid keyt < keyt \mid keyt = keyt \mid \\
&\quad keyt \in mskt \mid mskt \subseteq_m mskt \mid \varphi \vee \varphi \mid \neg\varphi
\end{aligned}
$$

Fig. 3. The contextual logic LM of list measures involving list-segments, heaplets, multisets of keys, max, min, and sortedness.

We now define a contextual logic over list segments and measures over them. Notice from Section 3 that the separation of formulae into delta logic introduces recursive definitions in the contextual logic (parameterized over various sets of variables) and additionally a rank function for each such definition. However, it is easy to see that for the definitions of lists and the measures we define on them, all the rank functions (over a given set of parameters) coincide, since the existence of a meaningful value for each of these measures is predicated upon the referred location pointing to a list. This motivates the following recursive definitions for our contextual logic of lists and measures. As usual, let us fix a set of first-order variables Δ. Let us also fix a single pointer field n.

Definition 5 (Recursive Definitions for the Logic of List Measures (LM)). *Let us fix a set of* parameters *P that consists of the following of sets of variables: a set of Boolean variables LS_z^v, a set of variables with type set of locations HLS_z^v, a set of variables of type integer LEN_z^v, a set of variables of type multiset of keys $MSKEYS_z^v$, and a set of variables of type integer (type of keys in general) Max_z^v and Min_z^v, where z, v range over Δ.*

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

- *We have unary relations ls_z^P that capture linked list segments that end in z, where the relation for a location v in Δ is imbibed using the Boolean parameter LS_z^v, 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:*

$$
\begin{aligned}
& ls_z^P(x) :=_{lfp} \\
& \Bigg(x = z \vee \Big(x \neq z \wedge x \neq nil \wedge x \notin \Delta \wedge ls_z^P(n(x)) \Big) \vee \\
& \Bigg(x \neq z \wedge x \in \Delta \wedge \bigwedge_{v \in \Delta} (x = v \Rightarrow LS_z^v) \Bigg) \Bigg)
\end{aligned}
$$

- *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 parameter HLS_z^v:*

$$
\begin{aligned}
hls_z^P(x) :=_{lfp} & \\
\emptyset & \quad \textit{if } [\![x]\!] = [\![z]\!] \\
\{x\} \cup hls_z^P(n(x)) & \quad \textit{if } [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] \neq [\![nil]\!] \\
& \quad \wedge [\![x]\!] \notin [\![\Delta]\!] \\
HLS_z^v & \quad \textit{if } [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] = [\![v]\!] \\
& \quad \wedge v \in \Delta
\end{aligned}
$$

- *We have similar recursive definitions that capture the length of list segments, the multiset of data elements stored in list segments, the maximum/minimum element stored in a list segment, and a predicate capturing sortedness of list segments. See Appendix 8.4.*

We define the *contextual logic of list-measures (LM)* to be quantifier-free formulae that use only the recursive definitions of LM mentioned above and combine them as in Figure 3. The logic LM allows first-order variables that range over locations, keys, and integers. Note that dereferencing pointers of locations captured by Δ is completely disallowed— the delta-specific formula will refer to such dereferences. Locations may just be compared for equality or disequality. Integer terms can be constructed from the length measure on list segments or arbitrary integer constants, and can have linear arithmetic constraints on them. Key terms can be constructed and constrained similarly. Heaplet terms can be constructed from arbitrary location terms, the heaplet of a list segment and can be combined using union, intersection or set difference. We can check membership with arbitrary location terms, subset relationships and emptiness. *mskt*

terms with the sort of multiset of keys can also be constructed and constrained similar to heaplet terms, with corresponding multiset operations. LM also has a recursive predicate for sortedness of list segments.

The formulae in the LM are then boolean combinations of these atomic formulae. Here Δ is assumed to be a subset of the free location variables in the contextual formula. Note that the formulae in LM are allowed to refer to the recursive predicates/functions defined using over various sets of parameters.

5.2 Translating contextual formulae to quantifier-free recursion-free formulae

Recall that we can show that the delta logic with list measures is decidable if we can provide an effective translation of contextual formulae to quantifier-free recursion-free formulas over standard background theories of sets, integers, and uninterpreted functions. The goal of this section is to provide such a translation.

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 is the most common and the logic VCs translate to as described in Section 4).

We will first show a translation of contextual logic formulas for a fragment of LM, $LM[ls, hls, rank]$, that involves only the three recursive definitions ls_z^P, hls_z^P, and $rank_z^P$, where $P \in \{\mathcal{P}\}$. Then we will extend the procedure to handle the logic with all the other measures; this latter translation requires more expressive logics (that are however decidable) and "pseudo-measures" that make it harder.

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

Recall that a model for a contextual formula consists of an unbounded 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 translation relies intuitively 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). There are three distinct cases to consider when pursuing paths using the n pointer on a location x: (a) the path may reach a node that is reachable also from another location in X, (b) the path may not be of the former kind but reach a node in Δ, or (c) the path may be neither of the two kinds above, staying forever outside Δ.

The key idea (see Figure 4) is to *collapse* list segments into single edges when the reachability of the locations in the segment from locations in X does not change. More precisely, let U be the set of locations reachable from some locations in X. Let L contain all locations in Δ as well as the following locations: locations l such that there exists *some* $l' \in U, x \in X$ with $n(l') = l$ such that l is reachable from x but l' is not reachable from x. Intuitively, L consists of all the locations where lists *merge* in the context-heap (as shown in Figure 4).

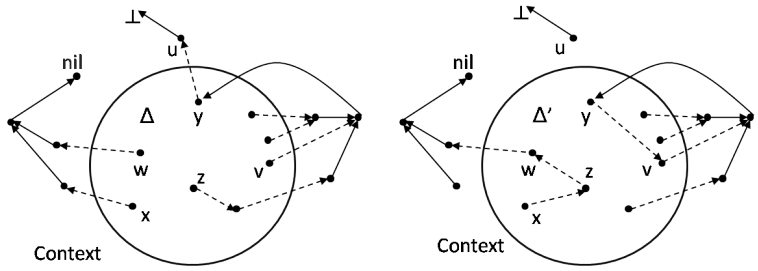

Fig. 4. Decidable delta logic of lists and list-measures. The dotted arrows represent the n and n' relations on Δ and Δ' respectively and the normal arrows represent the T relation, where an arrow to $\perp$ represents that the path further does not intersect any other point in the model, including elements in Δ or nil. Every T-related segment is associated with summaries of measures. The contextual model witnesses nontrivial merging points such as the one where the paths from x and w intersect. For unsatisfiability one must find a single context which together with Δ satisfies the precondition and with Δ' falsifies the postcondition. For example, the above context would witness the unsatisfiability of $ls_u(v)$ but would not witness the unsatisfiability of $ls_{nil}(x)$ (in the post-state).

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. Note, however, that these list measures are not necessarily *independent* of each other, and we will need to constrain their interdependence as well. We detail this construction in the Appendix, Section 8.6.

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 similarly translated. First, we model the multiset 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. A point of note is that the recursive definition of sortedness across segments is expressed by using both $Min_z(x)$ and $Max_z(x)$ definitions, though the recursive definition of sortedness uses only minimum—this is needed as expressing when the 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, i.e, be the attributes not of a pseudo-model. The following constraints capture this, 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 $min\mu(v) = max\mu(v) \neq \bot$, then $sorted\mu(v)$ can only be true.

The intuition is that any measures meeting the above constraints (including the ones stated earlier for $LM[ls, hls, rank]$) can be realized using true list segments. The final step in showing decidability is expressing these constraints using quantifier-free formulae in a combination of decidable SMT theories. To this end we use many different techniques including representing a collection of sets using representatives for each Venn region as in BAPA [24] as well as representing sets and multisets as arrays and operations on them as pointwise operations on the arrays [34]. This is quite involved and we detail the proof of realisability by true list segments and discuss the techniques for encoding using quantifier-free formulae in Appendix 8.6.

We can hence translate the formula into a quantifier-free formula, giving the required theorem. This proves the main theorem of this section:

Theorem 3. *For any quantifier-free contextual formula $\varphi(\mathcal{P}, X)$ of $LM[ls, hls, rank, len, min, max, sorted]$, there exists a quantifier-free and recursion-free formula $\psi(\mathcal{P}, X)$ that can be efficiently computed and is equisatisfiable given a common valuation for $\mathcal{P}$*

From the remarks at the beginning of this section, we have:

Corollary 1. *The delta logic of linear measures $LM[ls, hls, rank, len, min, max, sorted]$ is decidable.*

6 Implementation and evaluation of decision procedure for LM

We implemented the decision procedure for the delta logic LM with all the measures using the reduction to SMT described in Appendix 8.6, solved using Z3 [35]. We applied our technique to a suite of list-manipulating programs. The experiments were performed on a machine with an Intel® Core™i7-7600U processor with clock speeds upto 3.9GHz.

The experimental results are summarized in Figure 5. The left column shows results of verifying correct programs, while the column on the right shows results on buggy variants of these programs. The buggy programs were obtained

Correct programs	**#VCs**	**time(s)**	**Buggy programs**	**time(s)**
append(x: list, y: list)	4	57	buggy_append(x: list, y: list)	0.2
copyall(x: list)	4	183	buggy_copyall(x: list)	36
detect_cycle(x: list)	6	5	buggy_detect_cycle(x: list)	0.7
deleteall(x: list, k: key)	5	10	buggy_deleteall(x: list, k: key)	0.1
find(x: list, k: key)	3	6	buggy_find(x: list, k: key)	0.1
insert(x: list, k: key)	4	38	buggy_insert(x: list, k: key	0.2
insert_front(x: list, k: key)	1	3	buggy_insert_front(x: list, k: key)	0.1
insert_back(x: list, k: key)	4	20	buggy_insert_back(x: list, k: key)	0.2
reverse(x: list)	3	14	buggy_reverse(x: list)	0.1
sorted_append(x: list, y: list)	4	54	buggy_sorted_append(x: list, y: list)	0.2
sorted_deleteall(x: list, k: key)	5	2	buggy_sorted_deleteall(x: list, k: key)	1.1
sorted_insert(x: list, k: key)	4	17	buggy_sorted_insert(x: list, k: key)	1.2
sorted_reverse(x: list)	3	37	buggy_sorted_reverse(x: list)	0.5
sorted_merge(x: list, y: list)	8	300	buggy_sorted_merge(x: list, y: list)	0.7
insert_back_rec(x: list, k: key)	2	1.1	buggy_insert_back_rec(x: list, k: key)	0.2
deleteall_rec(x: list, k: key)	3	1.3	buggy_deleteall_rec(x: list, k: key)	0.25
even_split_rec(x: list)	2	0.3	buggy_even_split_rec(x: list)	0.45
sorted_merge_rec(x: list, y: list)	4	0.5	buggy_sorted_merge_rec(x: list, y: list)	0.43

Fig. 5. Experimental results for the decision procedure for the delta logic LM; extended with frame reasoning for function calls

by expressing weaker invariants/preconditions, or by introducing errors in the programs. Our tool works well on all these examples and for the buggy programs, the satisfying valuation it provided was sufficiently informative to diagnose the error.

The programs were written with strong functional correctness specifications. Specifications for a sample of four programs are provided in Figure 7 in the Appendix, and these are the strongest specifications that can be expressed in our logic. There are several other tools that can handle such programs but with much weaker specifications. For instance, in the experiments reported on the GitHub page for the tool GRASShopper [45], many contracts for list manipulating programs are very simple. For instance, for the method that copies a list, they only assert that the input is a list and the output is a list disjoint with the first one, with no mention of the requirement that the contents of the second list is identical to the contents of the first nor that the original list is unmodified. However the GRASShopper tool handles some complex data structures (implemented using lists) which we do not handle.

To the best of our knowledge, ours is the only decidable verification tool that can handle this suite of examples with its specifications.

7 Related Work

There are several ways of defining properties of heaps and reasoning with them. Heap reasoning can be performed using classical logic (first-order logic, first-order logic with recursive definitions that have least fixed point semantics, higher order logics, etc.).

Separation logic has emerged as a popular logic to state properties of heaps, and is designed to facilitate frame reasoning. Frame reasoning allows to conclude that a property holds across a computation when the computation's footprint does not affect the property. Frame reasoning considerably simplifies reasoning with heaps, and separation logic supports such a reasoning by implicitly defining the footprint of formulas describing heaps and endowing the logic with special operators for spatial combinations [47, 39, 38, 13]. There has been work also in classical logic, such as the work on dynamic frames [22, 23] which is used by tools such as Dafny [28] and Region Logic [3, 1, 2] that achieve similar goals. Recent work in reasoning automatically with separation logic combines both worlds, providing ways to translate separation logic properties to classical logics and reason with them [44, 45, 43, 41, 29, 11].

There is a rich literature on decidable logics for heaps. Decision procedures for lists were the first, and were for deciding shape properties of lists and combinations with SMT theories [5, 12, 36, 42, 6]. There have been decidable logics that do not allow expressing entailment (and hence are not suited for verification) [51, 15, 16, 14], including separation logic with user-defined inductive predicates and arithmetic constraints [10, 27, 26]. While the above logics are suited for reasoning with separation logic, they are not particularly suited to express verification conditions. However, we believe that when VCs are translated to delta logics, the above methods could be helpful in reasoning with contextual logics, and this is an interesting future direction.

One of the most expressive decidable logics for reasoning with programs with separation logic annotations is Grasshopper [43, 45], and works similar to ours, by a translation to SMT. Extensions to this work also support reasoning with trees and tree-segments [44]. The decidable logic for linked lists is based on a base logic called GRASS, a logic for graph reachability and stratified sets. The logic is expressive for reasoning with shape properties of lists, and through local theory extensions [43, 4, 48] can support properties such as sortedness. However, the logic does not support properties such as the length of lists and the multiset of keys stored in a list (see Section 6 for a comparison of specifications). In our decision procedure, we capture measures of list-segments independently and then constrain them with respect to each other (for example, that the cardinality of multiset of keys in a segment must equal its length). These mutual dependencies are very particular to the semantics of the measures, and we do not know of a way of realizing them as local theory extensions. The idea of collapsing list segments and using summaries, however, is not new [7, 8]

A lot of work is also present on reduction to decidability of MSO over trees [52] or other regular tree structures [54, 17, 18, 53]. PALE [32] uses MSO over strings and trees and reduces problems to MONA [33]. Works such as [30,

31, 9] combine shape properties with data. In comparison with our work, these works can neither handle the aggregation of data properties nor can they handle length.

There is work on using EPR (Effectively Propositional Reasoning) logics for encoding heap verification properties to obtain decidability [19–21], and is similar to local theory extensions. These logics can support shapes of lists, doubly-linked lists and properties of an order on data stored in them (including sortedness). EPR is however, very restricted. The use of even a function pointer (as opposed to reachability) is allowed only in limited contexts, ordering on data elements is abstracted to an arbitrary total order, length-measures and multisets-of-keys measures are not allowed, heaplet measures are not explicit, and consequently the annotations in the experiments are much weaker than ours. Correctness of programs that rely on true semantics of integers (with lengths as in the even_split program in our experiments) are particularly problematic.

The work in [49] investigated the verification of functional programs manipulating abstract data structures. The decision procedure for a fragment of the logic relies upon the unfolding of recursive definitions followed by an abstraction as uninterpreted functions. This idea of unfolding and usage of uninterpreted functions is also the basis of the work in sound and incomplete verification for imperative heap-manipulating programs [41, 29, 11]. In particular, [46] is a powerful method capable of verifying all of the correct programs in our experiments, but cannot disprove programs nor give counterexamples.

References

1. Banerjee, A., Naumann, D.: Local reasoning for global invariants, part ii: Dynamic boundaries **60** (06 2013)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008 – Object-Oriented Programming. pp. 387–411. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part i: Region logic. J. ACM **60**(3), 18:1–18:56 (Jun 2013). https://doi.org/10.1145/2485982, http://doi.acm.org/10.1145/2485982
4. Bansal, K., Reynolds, A., King, T., Barrett, C.W., Wies, T.: Deciding local theory extensions via e-matching. In: CAV (2015)
5. 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)
6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: Proceedings of the 4th International Conference on Formal Methods for Components and Objects. pp. 115–137. FMCO'05, Springer-Verlag, Berlin, Heidelberg (2006)
7. Bornat, R., et al.: Local reasoning, separation and aliasing (2004)
8. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 517–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
9. 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)
10. 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)
11. 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)
12. 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)
13. Demri, S., Deters, M.: Separation logics and modalities: a survey. Journal of Applied Non-Classical Logics **25**, 50–99 (2015)
14. Enea, C., Sighireanu, M., Wu, Z.: On automated lemma generation for separation logic with inductive definitions. In: ATVA (2015)
15. Gao, C., Chen, T., Wu, Z.: Separation logic with linearly compositional inductive predicates and set data constraints. ArXiv **abs/1811.00699** (2018)
16. Gu, X., Chen, T., Wu, Z.: A complete decision procedure for linearly compositional separation logic with data constraints. In: IJCAR (2016)
17. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive-closure logics. In: Marcinkowski, J., Tarlecki, A. (eds.) Computer Science Logic. pp. 160–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

18. 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)
19. Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A., Sagiv, M.: Modular reasoning about heap paths via effectively propositional formulas. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 385–396. POPL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535854, http://doi.acm.org/10.1145/2535838.2535854
20. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 756–772. CAV'13, Springer-Verlag, Berlin, Heidelberg (2013)
21. Itzhaky, S., BjØrner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 35–51. Springer-Verlag, Berlin, Heidelberg (2014)
22. Kassios, I.T.: The dynamic frames theory. Form. Asp. Comput. **23**(3), 267–288 (May 2011)
23. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Proceedings of the 14th International Conference on Formal Methods. pp. 268–283. FM'06, Springer-Verlag, Berlin, Heidelberg (2006)
24. Kuncak, V., Nguyen, H.H., Rinard, M.: An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In: Proceedings of the 20th International Conference on Automated Deduction. pp. 260–277. CADE' 20 (2005)
25. Kuncak, V., Rinard, M.: Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In: Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction. pp. 215–230. CADE-21 (2007)
26. Le, Q.L., Sun, J., Chin, W.N.: Satisfiability modulo heap-based programs. In: Computer Aided Verification. pp. 382–404 (2016)
27. 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)
28. Leino, R.: Specification and verification of object-oriented software (June 2008), https://www.microsoft.com/en-us/research/publication/specification-verification-object-oriented-software/
29. Löding, C., Madhusudan, P., Peña, L.: Foundations for natural proofs and quantifier instantiation. Proc. ACM Program. Lang. **2**(POPL), 10:1–10:30 (Dec 2017). https://doi.org/10.1145/3158098, http://doi.acm.org/10.1145/3158098
30. 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)
31. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using strand. In: Yahav, E. (ed.) Static Analysis. pp. 43–59. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
32. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. SIGPLAN Not. **36**(5), 221–231 (May 2001). https://doi.org/10.1145/381694.378851, http://doi.acm.org/10.1145/381694.378851
33. Møller, N.K.A.: Mona version 1.4 user manual. In: BRICS Notes Series (2001)
34. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: 2009 Formal Methods in Computer-Aided Design. pp. 45–52 (Nov 2009). https://doi.org/10.1109/FMCAD.2009.5351142

35. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
36. 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)
37. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (Oct 1979). https://doi.org/10.1145/357073.357079, http://doi.acm.org/10.1145/357073.357079
38. O'Hearn, P.W.: A primer on separation logic (and automatic program verification and analysis). In: Software Safety and Security (2012)
39. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the 15th International Workshop on Computer Science Logic. pp. 1–19. CSL '01, Springer-Verlag, London, UK, UK (2001), http://dl.acm.org/citation.cfm?id=647851.737404
40. Parkinson, M.F.: When separation logic met java (by example) (2006)
41. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in c using separation logic. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 440–451. PLDI '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594325, http://doi.acm.org/10.1145/2594291.2594325
42. Pérez, J.A.N., Rybalchenko, A.: Separation logic modulo theories. In: Programming Languages and Systems (APLAS). pp. 90–106. Springer International Publishing, Cham (2013)
43. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using smt. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 773–789. CAV'13, Springer-Verlag, Berlin, Heidelberg (2013)
44. 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)
45. Piskac, R., Wies, T., Zufferey, D.: Grasshopper. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg (2014)
46. 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)
47. 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)
48. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20. pp. 219–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
49. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. SIGPLAN Not. **45**(1), 199–210 (Jan 2010). https://doi.org/10.1145/1707801.1706325, http://doi.acm.org/10.1145/1707801.1706325
50. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications **5** (06 1955)

51. Tatsuta, M., Le, Q.L., Chin, W.N.: Decision procedure for separation logic with inductive definitions and presburger arithmetic. In: APLAS (2016)
52. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathematical systems theory **2**(1), 57–81 (Mar 1968). https://doi.org/10.1007/BF01691346, https://doi.org/10.1007/BF01691346
53. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. pp. 476–491. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
54. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) Foundations of Software Science and Computation Structures. pp. 94–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

8 Appendix

8.1 Overview of delta logics and an illustrative example

In this section, we shall provide the motivation for delta logics as a logic for expressing verification conditions, an overview of our technique, and illustrate the method with an example.

Motivation: frame reasoning, delta changes and verification conditions In this part we shall describe the need for a new logic for expressing verification conditions for Hoare triples involving snippets of code that modify a bounded set of heap locations.

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 by 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 postconditions 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 infer that $R(x)$, for any x, holds in the post-state if it held in the pre-state and the modified portion of the heap did not intersect with the underlying heaplet of $R(x)$. This is, in practice, a very *convenient and simple* reasoning that often works and is one of the foundational ideas that separation logic facilitates [47, 39]. However, vanilla frame reasoning can be incomplete, as we shall argue in Section 8.1.

The focus of this paper is on the generation of precise verification conditions for basic blocks that do not involve function calls. For basic blocks that involve function calls, our recommendation is to use frame reasoning, *à la* separation logic.

Let us now consider basic blocks that do not involve function calls. We would like to generate precise verification conditions in such cases. There are several approaches in the literature that argue for this: for example the Grasshopper suite of tools handle such blocks accurately for certain logics [44], and there is work on expressing weakest preconditions in separation logic for such blocks using the magic wand [47]; see section on related work. The goal of this paper is to accurately formulate verification conditions for very expressive logics (FO+*lfp*) that can also be reasoned with effectively, especially in the context of decidable logics.

One precise formulation of the verification condition (as mentioned in the introduction) is of the form:

$$(@pre(\overline{x}, \overline{p}, R) \wedge T(\overline{x}, \overline{x}', \overline{p}, \overline{p}')) \Rightarrow @post(\overline{x}', \overline{p}', R')$$

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 $\overline{x}'$ and $\overline{p}'$ respectively. Most importantly, the above requires *new* recursive definitions R' that are formulated 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 Mod modified by the program, and the formula will have conjuncts of the form $\forall y \notin Mod.p'(y) = p(y)$, 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, Mod is finite, and 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' automatically (which involves least fixpoints), coupled with such constraints, is daunting.

Surely, there must be a simpler formulation of the verification condition! Small changes to the heap do cause global changes that dramatically affect the valuation of recursively defined predicates/functions. For example, executing a statement $x.next := y$ can make an element far away from x a list. But surely, the effect on the semantics of R changing into R' must be expressible in a simpler localized fashion.

The goal of this paper is to identify such a logic.

Overview of using Delta Logics for Verification Conditions

Delta logics In this paper, we describe a class of logics, called *delta logics*, that are logics for writing verification conditions of basic blocks without function calls and are precisely meant to address the issues mentioned above. In particular, formulations in delta logics will avoid the need for *two* different recursive definitions R and R' that depend on different sets of functions modeling pointers. Instead, both R and R' will be expressed as the same recursive definition, but

parameterized using different sets of first-order variables (as opposed to being parameterized over two different sets of first-order *functions*, $\overline{p}$ and $\overline{p'}$ as above).

Formulae in delta logics are Boolean combinations of two distinct kinds of formulae: one kind, called *delta-specific* formulae, strictly talk about the modified portion of the global heap (identified by a bounded set of locations Δ) without using any recursive definitions; the other kind, called *contextual* formulae, strictly talk about the unbounded portion excluding Δ using recursive definitions (see Figure 1). A set of first-order interface variables are used to communicate information between Δ and the rest of the heap (its 'context'). In particular, a recursive definition R over the unbounded context is *parameterized* over a set of first-order communication variables P, where P summarizes the values of R within Δ. These variables themselves can, of course, depend on the value of R outside Δ as well, setting up mutually dependent constraints.

Example 4. Consider $\Delta = \{x\}$, and where we want to express the property that w points to a linked list (ending with nil). We can write this in delta logic as follows. First, we introduce what we call a *boundary* variable z, which is $n(x)$ (we model that $n(nil) = nil$ and assume that it is part of the given formula). Then we write the fact that w points to a list by breaking into *list segments* that do not intersect with Δ. In particular, we have two list segment definitions $ls_{nil}^{\neg x}$, which is a recursively defined predicate that checks if its argument points to a list ending with nil but does not pass through x, and ls_x, which is again a recursively defined predicate and checks if its argument points to a list segment that ends at x. Note that these are context-logic formulae as they do not dereference x. We also introduce two Boolean parameters, P_x and P_z; P_x is meant to capture whether x points to a linked list, and P_z is meant to capture whether z points to a linked list that does not go through x.

Now we can write the property that w points to a list ending with nil as:

$$\big(z = n(x) \wedge \big([\, ls_{nil}^{\neg x}(w) \vee (ls_x(w) \wedge P_x)] \wedge [P_x \iff (x = nil \vee P_z\,)] \wedge [P_z \iff ls_{nil}^{\neg x}(z)]\,\big)\,\big)$$

The above is a delta logic formula as the recursive definitions are purely contextual, and there is no atomic formula that uses $n(x)$ and a recursive definition.

Advantages of reasoning with Delta logics Expressing verification conditions in delta logics has a distinct advantage in automated reasoning. We formulate verification conditions in delta logics in the following manner (see Figure 6). We can view the program's transformation of the pre-heap to the post-heap as a static model consisting of three different submodels: one is the context heap, the second is the pre-heap restricted to Δ, and the third is the post-heap restricted to Δ. Note that there is only one context heap as it does not change, and this context heap is infinite, in general. However, the other two heaps are finite. The recursive properties of the heap are then defined on the context-heap, parameterized with the communication variables P for expressing properties of the pre-heap, and parameterized over the communication variables P' for expressing properties of the post-heap.

The key advantage of the above model is that two of the submodels (both over Δ) are finite, and reasoning about them and the data-fields accessible from them can be automated using standard SMT solvers. The context heap is the single unbounded submodel that poses automation challenges. In this paper, we exploit this simplicity of dealing only with one infinite context model (as opposed to the naive formulation, which would have two infinite context models) to build new powerful classes of decidable logics over lists and list-measures.

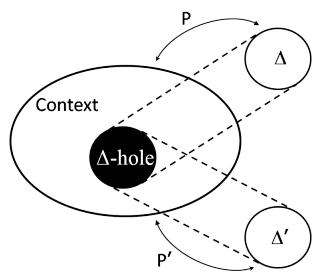

Fig. 6. VC expressed in delta logics

An Illustrative Example We shall now illustrate our method on an example. Let us consider the following Hoare triple with pre/post conditions written in quantifier-free FO+*lfp*:

$$\{list(x) \wedge n(y) = y_1\} \quad \texttt{y.n := nil} \quad \{list(x)\}$$

where $list(x)$ is a recursive definition that holds when x points to a list using the pointer *next*:

$$list(x) :=_{lfp} x = nil \vee (x \neq nil \wedge list(n(x)))$$

where by $n()$ in the logical formula above we mean a function that models the *next* pointer in our program semantics.

Incompleteness of frame reasoning It is easy to see that this Hoare Triple is valid. There are two cases: if y does not belong to the list pointed to by x, then clearly x continues to point to a list. Since y is not in the heaplet of $list(x)$, frame reasoning would prove this fact. However, in the case where y belongs to the list pointed to by x we cannot use frame reasoning to immediately conclude that x continues to point to a list. However x does point to a list in this case as well since x eventually reaches y which points to nil. Therefore, frame reasoning *alone* is insufficient to conclude the postcondition.

A Precise VC A precise way to derive a VC for this program is to use a new *next* pointer (say, $next'$) to model the post-state of the program, redefining the

recursive functions and predicates using $next'$ to talk about valuations on the post-state, and observing that $next$ has not changed on any other location except for the one stored in y:

$$\Big(\big(list(x) \wedge n(y) = y_1\big) \wedge \\ \big(\,(n'(y) = nil \wedge y' = y \wedge x' = x \wedge y'_1 = y_1) \wedge \\ (\,\forall u.\, u \neq y \Rightarrow n'(u) = n(u)\,)\,\big)\Big) \Rightarrow (list'(x'))$$

where the quantifier is interpreted to range over all locations on the heap and $list'$ is defined as:

$$list'(x) :=_{lfp} x = nil \vee (x \neq nil \wedge list'(n'(x)))$$

The fact that the recursive definitions on the post-state vary depending on the basic block makes reasoning with the VC hard.

A Precise VC using Delta Logics In the Hoare triple above, the set of modified locations is $\Delta = \{y\}$ [2] and the context is the complement. To write the VC in delta logic, we first convert the pre- and post- conditions to delta logic.The separability theorem (Section 3) allows us to do this by giving an equivalent delta logic formula given a quantifier-free FO+*lfp* formula.

The precondition stated in delta logic can be done by breaking up lists into list segments to separate them into contextual and delta-specific formulae (similar to the example in Section 8.1):

$$\begin{aligned} \exists z.\exists P_1 P_2.\big(z = n(y) \wedge \big([\, ls^{\neg y}_{nil}(x) \vee (ls_y(x) \wedge P_y)] \\ \wedge [P_y \iff (y = nil \vee (z = n(y) \wedge P_z)\,)] \\ \wedge [P_z \iff ls^{\neg y}_{nil}(z)]\,\big) \\ n(y) = y_1\,\big) \end{aligned}$$

We shall call this delta logic formula α_{pre}. This formula is a simplified version of what would be automatically generated by the separability theorem (including a contextual definition that is essentially $(ls_b(a))$.

Similarly, we transform the postcondition to another delta logic formula α_{post} which uses a *different* set of parameters.

The VC can then be written as:

[2] In our work, we consider the heaplet to be any logical location such that any pointer field f modeled as a function changes on this location. One can do a more refined heaplet by considering pairs of the form (x, f) to denote that only the f field of x has changed, and is common in separation logic [47, 39, 40]. Such a finer heaplet can be modeled in our framework as well, by having *field lookup functions* L_f, and writing $x.L_f.f$ instead of $x.f$. These lookup pointer fields will not change and hence x will never be in Δ, and when f is modified, the location stored at $x.L_f$ will be in the heaplet, as desired.

$$\alpha_{pre}(n, ls_{nil}^{\neg y}, ls_y) \wedge \mathit{transform}(n, n') \Rightarrow \alpha_{post}(n', ls_{nil}^{\neg y\prime}, ls'_y)$$

where *transform* is the program transformation formula:

$$n' \equiv n[y \mapsto nil] \wedge y' = y \wedge x' = x \wedge y'_1 = y_1$$

Observe that this formula is already a Boolean combination of delta-specific formulae and contextual formulae.

First, observe that since $ls_{nil}^{\neg y}$, ls_y and $ls_{nil}^{\neg y\prime}$, ls'_y are contextual formulae defined outside Δ where $n' \equiv n$, we can also identify the recursive definitions as $ls_{nil}^{\neg y} \equiv ls_{nil}^{\neg y\prime}$ and $ls_y \equiv ls'_y$. The VC can hence be written using one set of recursive definitions:

$$\alpha_{pre}(n, ls_{nil}^{\neg y}, ls_y) \wedge \mathit{transform}(n, n') \Rightarrow \alpha_{post}(n', ls_{nil}^{\neg y}, ls_y)$$

The subformula $n' \equiv n[y \mapsto nil]$ is really a quantified formula of the form $n'(y) = nil \wedge \forall u \neq y.\, n'(u) = n(u)$. However, we can write the VC without this quantifier by replacing all occurrences of $n'(t)$ for any term t with the if-then-else expression $ite(t = y, nil, n(t))$. This is possible, in general, since Δ is finite. The VC now becomes:

$$\big(\alpha_{pre}(n, ls_{nil}^{\neg y}, ls_y) \wedge \mathit{transform}'(n, n') \Rightarrow \alpha_{post}(n', ls_{nil}^{\neg y\prime}, ls'_y)\big)[n'(t) \mapsto ITE(y, nil, n(t))]$$

where $\mathit{transform}'$ is: $y' = y \wedge x' = x$.

Finally, there is a small technical issue. Recall that in α_{pre} and α_{post}, the separability theorem that converted the formulae into delta logics introduced quantification over parameters and auxiliary locations. However, these parameters and locations are actually *determined* (there is only one choice for them to make the formula true, if at all). Consequently, they can be introduced either using existential or universal quantification with appropriate modification. We hence write α_{pre} using existential quantification and α_{post} using universal quantification which makes the VC purely universally quantified, i.e, the VC is quantifier-free.

In the above illustrative example, we transformed formulae into delta logic using multiple recursive definitions. However the separability theorem, which handles general formulae, is more technically involved and creates a single set of recursive definitions and an auxiliary set of recursive *rank* definitions in order to achieve this transformation.

8.2 Separability Lemma and Theorem

In this Section we will prove the Separability theorem, which is a key result of this paper. In order to prove this, we need the following technical lemma that says that the β_R constraints (Figure 2) force our modified definition R^P to be equivalent to the given recursive definition R.

Lemma 1. *Fix a model M. There exists a valuation for variables in B and P over M such that β_R is satisfied. Moreover, for any valuation of the variables in M that satisfies β_R, the semantics of R^P under this valuation is precisely the semantics of R.*

Proof. The Tarski-Knaster theorem [50] gives that for monotonic functions over a lattice, an iterative procedure that starts from the bottom element of the lattice and iteratively applies the function will converge to the least fixpoint. However, in general, the iterations can have levels corresponding to limit ordinals. However, in the case of our recursive definitions observe that this procedure will converge in ω steps, i.e., the valuation for every element can be corresponded to a finite iteration level. This is because given the form of our recursive definitions, we can see that the valuation on an element x depends only on finitely many other elements, namely the $p(x)$, $p \in Dep$. So suppose the opposite. Then, there is an element x_ω whose valuation is obtained at the iteration level ω but no earlier. But, we have that the various $p(x_\omega)$ that determine the valuation at x_ω must have obtained their valuation some finite level, the max of which would be a finite number j. Then, by definition, x_ω obtained its valuation at the level $j+1$, which is a contradiction.

We prove this lemma in two parts. First, we prove that given a model if at all there exists a valuation of the parameters P and B satisfying β_R, then it is the only valuation. We then prove that there exists such a valuation and that it identifies R^P and R. It is clear why the proof holds for elements in B, given that the $b_{p_i(d)}$ are constrained exactly to be $p_i(d)$ for $d \in \Delta$ and $1 \leq i \leq k$, where $p_i(nil) = nil$ for any i. We shall now show the theorem for P.

Part I. *Uniquely determined parameters*

We prove the first part by induction over the value of $Rank_R$. Let there be given a model and a valuation of the parameters satisfying β_R. We prove a stronger claim, namely that the valuation of $Rank_R$ and R^P is uniquely determined for all elements in the model with a certain valuation for $Rank_R$ in the given model. We first induct over ranks not equal to $\perp$.

Base case: The base case is when rank is 0. But recall that the definitions must converge in ω iterations, and therefore in particular all elements with rank 0 must converge on their valuations in ω iterations. We prove that the valuation on elements with rank 0 must be uniquely determined by induction over the iterations. The base case or iteration 0 is simply the base case of the recursive formula φ (which defines R). It is easy to see from the definition of $Rank_R$ and β_R that given an element that satisfies the base case of the recursive formula φ, the valuation is as the definition determines and its rank must be 0, and can therefore not differ in any other valuation for the parameters satisfying β_R.

For the inductive case for an element x belonging to iteration $j > 0$, observe $x \notin \Delta$, since from the β_R constraints we have that rank is 0 for $y \in \Delta$ if and only if y satisfies the base case. However, from the definition of $Rank_R$ we have that for $x \notin \Delta$, the expression $\max_{p \in Dep} \{Rank_R(p(x))\}$ can evaluate to 0 if and

only if the rank of every $p(x)$ for $p \in Dep$ is 0, which must belong to iterations lower than j and by the inductive hypothesis are uniquely determined. Therefore the valuation on x for $Rank_R$ is uniquely determined as well. The proof of the valuation for R^P being uniquely determined on these elements also follows this induction over iterations, since R^P is uniquely determined for the base case of the recursive formula φ.

Inductive case: For the inductive case of PART I of the proof, let there be x such the valuation of $Rank_R$ on it is $r > 0$. We prove this similarly as above by induction over the iterations. Then, the base case is an element in Δ such that $r = \max_{p \in Dep} (\{RANK_R^{p(x)}\}) + 1)$. Then, by the inductive hypothesis we have that the valuation for $Rank_R$ and R_P is determined since they must have ranks lower than r. The inductive case for iterations $j > 0$ proceeds similarly as for the inductive case for elements with rank 0 above, as does the proof of the valuation for R^P being uniquely determined.

$\perp$ *case:* Observe that if the valuations on elements with $Rank_R$ not equal to $\perp$ are uniquely determined, then the valuation on the elements with rank $\perp$ must also be uniquely determined. Were it not, there must be a valuation where the rank is not $\perp$, but from above we have that all other valuations must do the same. The proof for this case concludes by noting that if $Rank_R$ is $\perp$, then the valuation for R^P must also be $\perp$.

Thus, we have that the valuation for R^P and $Rank_R$ is *uniquely determined* for all elements in the given model, and in particular the parameters are uniquely determined.

Part II. *Existence of parameter valuation satisfying* β_R

Consider the following definition:

$$
\begin{aligned}
TrueRank_R(x) :=_{lfp} & \\
& 0 \ \textit{if}\ [\![x]\!] \notin [\![\Delta]\!] \wedge \\
& \quad \varphi(x)[\perp/R] \neq \perp \text{ (base case)} \\
& \max_{p \in Dep} \{TrueRank_R(p(x))\} \ \textit{if}\ [\![x]\!] \notin [\![\Delta]\!] \wedge R(x) \neq \perp \\
& \quad \text{(context recursive case)} \\
& \max_{p \in Dep} \{TrueRank_R(p(x))\} + 1 \ \textit{if}\ [\![x]\!] \notin [\![\Delta]\!] \wedge R(x) \neq \perp \\
& \quad \text{(delta recursive case)} \\
& \perp \ \textit{if}\ R(x) = \perp \\
& \quad \text{(undefined case)}
\end{aligned}
$$

Observe that this is well-defined on any given model without needing a valuation for any parameters.

The proof of this part is trivial by choosing, in any given model, the valuation for the parameters at an element x to be equal to the valuations of R and

$TrueRank_R$ respectively at x, and observing that this makes $R^P = R$, $Rank_R = TrueRank_R$, and that the β_R constraints are satisfied.

This concludes the proof of Lemma 1 . □

Using the above lemma we can prove Theorem 1.

Proof of Theorem 1. Consider a model where α holds. From Lemma 1, we can pick a valuation for P and B such that β_R holds and for that valuation (again by Lemma 1) R^P is equivalent to R. Thus we have that $\exists B. \exists P. \alpha[R^P/R] \wedge \beta_R$ holds.

Conversely, consider a model where $\exists B. \exists P. \alpha[R^P/R] \wedge \beta_R$ holds. Since we have that the valuation given by the model for P and B satisfies β_R, From Lemma 1 we have that R^P is equivalent to R. Therefore, α holds in M. □

8.3 A Complete Example of the Separability Theorem

Let us illustrate the Separability Theorem (Theorem 1) by translating the formula $list(x)$ for $\delta = \{y\}$ into an equivalent Delta Logic formula.

Recall:

$list(x) :=_{lfp} x = nil \vee (x \neq nil \wedge list(n(x)))$

We fix P to be the following set of variables: $\{LS^y, LS^{n(y)}\}$ of Boolean type, and $\{RANK^y, RANK^{n(y)}\}$ of integer type. We then generate the following definitions:

$$list^P(x) :=_{lfp} \begin{cases} LS^y & x = y \\ x = nil \vee (x \neq nil \wedge list^P((n(x))) & x \neq y \end{cases}$$

The purely contextual definition above has the following semantics: $list^P(x)$ is true if and only if either x reaches nil without going through y, or x reaches y and the Boolean variable LS^y is set to true. Note that our transformation hence introduces 'magically' the notion of list-segments from the definition of lists, automatically.

Next, the contextual rank definition is as follows:

$$Rank^P(x) :=_{lfp} \begin{cases} RANK^y & \text{if } x = y \\ 0 & \text{if } x = nil \\ Rank^P(n(x)) & \text{if } x \neq nil \wedge list^P(x) \neq \bot \\ \bot & \text{if } list^P(x) = \bot \end{cases}$$

$Rank^P(x)$ is 0 if x points to a list that does not intersect y. It is equal to $RANK^y$ when x reaches y. It is $\bot$ if neither case applies.

The (simplified) β_{list} constraints are then as follows:

$$
\begin{array}{l}
(y = nil \Rightarrow (LS^y \wedge RANK^y = 0)) \wedge \\
((y \neq nil \wedge LS^{n(y)}) \Rightarrow (LS^y \wedge RANK^y = RANK^{n(y)} + 1)) \wedge \\
((y \neq nil \wedge \neg LS^{n(y)}) \Rightarrow (\neg LS^y \wedge RANK^y = \bot)) \wedge \\
((n(y) = b_{n(y)}) \\
\quad \wedge (y \neq b_{n(y)} \Rightarrow (LS^{n(y)} = list^P(b_{n(y)}) \\
\qquad\qquad\qquad \wedge RANK^{n(y)} = Rank^P(b_{n(y)}))))
\end{array}
$$

The *resulting Delta Logic formula* from transforming $list(x)$ is:
$list^P(x) \wedge \beta_{list}$

where we existentially quantify P in the above formula to yield an equivalent formula to $list(x)$. In any satisfying model for this formula, $list^P(x)$ will hold iff x points to a list (and hence captures the original formula precisely). $Rank^P(x)$ will be (i) 0 if x points to a list without going through y (ii) 1 if x points to a list going through y and (iii) $\bot$ if neither case applies.

Intuitively the contextual formula and the delta-specific formula essentially compute lfps on their separate universes, communicating between themselves in stages, where the stage numbers are captured by ranks.

8.4 Definitions of list measures

We provide in this section the contextual recursive definitions of the various measures omitted from Section 5.

- We have recursive definitions that capture the length of list segments, where the length of the list segment from an element v in Δ to z (where $z \in \Delta \cup \{nil\}$) is imbibed from the integer variable LEN_z^v:

$$
\begin{array}{rl}
len_z^P(x) :=_{lfp} & \\
0 & \textit{if } \ [\![x]\!]=[\![z]\!] \\
len_z^P(n(x)) + 1 & \textit{if } [\![x]\!] \neq [\![z]\!] \\
& \quad \wedge [\![x]\!] \neq [\![nil]\!] \\
& \quad \wedge [\![x]\!] \notin [\![\Delta]\!] \\
LEN_z^x & \textit{if } [\![x]\!] \neq [\![z]\!] \\
& \quad \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta
\end{array}
$$

- 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 $MSKeys_z^v$:

$$
\begin{aligned}
mskeys_z^P(x) :=_{lfp} & \\
& \emptyset \ \ if \ \ [\![x]\!]=[\![z]\!] \\
\{key(x)\} \cup_m mskeys_z^P(n(x)) \ \ & if \ [\![x]\!] \neq [\![z]\!] \\
& \wedge [\![x]\!] \neq [\![nil]\!] \\
& \wedge [\![x]\!] \notin [\![\Delta]\!] \\
MSKeys_z^x \ \ & if \ [\![x]\!] \neq [\![z]\!] \\
& \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta
\end{aligned}
$$

– 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_z^v (or Min_z^v). 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 $max(r_1, r_2) \equiv ite(r_1 \leq r_2, r_2, r_1)$.

$$
\begin{aligned}
Max_z^P(x) :=_{lfp} & \\
-\infty \ \ & if \ [\![x]\!]=[\![z]\!] \\
max(key(x), Max_z^P(n(x))) \ \ & if \ [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] \neq [\![nil]\!] \\
& \wedge [\![x]\!] \notin [\![\Delta]\!] \\
MAX_z^v \ \ & if \ [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] = [\![v]\!] \\
& \wedge v \in \Delta
\end{aligned}
$$

The function Min_z^P is similarly defined.

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

$$
\begin{aligned}
& Sorted_z^P(x) :=_{lfp} \\
& \Big(\ x = z \ \vee \\
& \ \big(x \neq z \wedge x \neq nil \wedge x \notin \Delta \wedge min_z^P(x) \neq \bot \\
& \ \wedge key(x) \leq min_z^P(x) \wedge Sorted_z^P(n(x))\big) \ \vee \\
& \ (x \neq z \wedge x \in \Delta \wedge min_z^P(x) \neq \bot \wedge key(x) \leq min_z^P(x) \\
& \ \wedge \bigwedge_{v \in \Delta} (x = v \Rightarrow SORTED_z^v)) \ \Big)
\end{aligned}
$$

8.5 Incorporating function calls with delta logic

In this section, we show how to generate VCs for a general setup that has both basic blocks and function calls, handling delta-changes using delta logics and

Program	Precondition, Postcondition and Loop Invariant
copyall (x:Loc) returns y:Loc	$@pre\ : ls_{nil}(x) \wedge mskeys_{nil}(x) = M \wedge hls_{nil}(x) = H$ $@post : ls_{nil}(x) \wedge ls_{nil}(y) \wedge hls_{nil}(x) \cap hls_{nil}(y) = \emptyset \wedge$ $mskeys_{nil}(x) = mskeys_{nil}(y) \wedge mskeys_{nil}(x) = M \wedge$ $hls_{nil}(x) = H$ $@inv\ : ls_{nil}(x) \wedge ls_{temp\_x}(x) \wedge ls_{temp\_y}(y) \wedge n(temp\_y) = nil \wedge$ $hls_{nil}(x) \cap hls_{nil}(y) = \emptyset \wedge$ $mskeys_{nil}(x) = mskeys_{nil}(temp\_x) \cup_m mskeys_{nil}(y) \wedge$ $mskeys_{nil}(x) = M \wedge hls_{nil}(x) = H$
sorted_listrev(x:Loc) returns ret:Loc	$@pre\ : ls_{nil}(x) \wedge sorted_{nil}(x) \wedge hls_{nil}(x) = H \wedge mskeys_{nil}(x) = M$ $@post : ls_{nil}(ret) \wedge revsorted_{nil}(ret) \wedge hls_{nil}(ret) = H \wedge$ $mskeys_{nil}(ret) = M$ $@inv\ : ls_{nil}(x) \wedge ls_{nil}(ret) \wedge sorted_{nil}(x) \wedge revsorted_{nil}(ret) \wedge$ $hls_{nil}(x) \cap hls_{nil}(y) = \emptyset \wedge$ $((x \neq nil \wedge ret \neq nil) \Rightarrow (min_{nil}(x) \geq max_{nil}(ret))) \wedge$ $hls_{nil}(x) \cup hls_{nil}(ret) = H \wedge$ $mskeys_{nil}(x) \cup_m mskeys_{nil}(ret) = M$
even_split_rec(x:Loc,ff:Loc,ss:Loc) recursive function	$@pre\ : ls_{nil}(x) \wedge hls_{nil}(x) = H \wedge mskeys_{nil}(x) = M$ $@post : ls_{nil}(ff) \wedge ls_{nil}(ss) \wedge hls_{nil}(ff) \cup hls_{nil}(ss) = H \wedge$ $mskeys_{nil}(ff) \cup_m mskeys_{nil}(ss) = M \wedge$ $len_{nil}(ff) \neq len_{nil}(ss) \Rightarrow len_{nil}(ff) = len_{nil}(ss) + 1$
sorted_merge(x:Loc, y:Loc) returns ret:Loc	$@pre\ : ls_{nil}(x) \wedge ls_{nil}(y) \wedge sorted_{nil}(x) \wedge sorted_{nil}(y) \wedge$ $hls_{nil}(x) \cap hls_{nil}(y) = \emptyset \wedge hls_{nil}(x) \cup hls_{nil}(y) = H \wedge$ $mskeys_{nil}(x) \cup mskeys_{nil}(y) = M$ $@post : ls_{nil}(ret) \wedge sorted_{nil}(ret) \wedge hls(ret, nil) = H \wedge$ $mskeys(ret, nil) = M$ $@inv\ : ls_{nil}(x) \wedge ls_{nil}(y) \wedge ls_{temp}(ret) \wedge n(temp) = nil \wedge$ $sorted_{nil}(x) \wedge sorted_{nil}(y) \wedge sorted_{nil}(ret) \wedge$ $hls_{nil}(x) \cap hls_{nil}(y) = \emptyset \wedge hls_{nil}(x) \cap hls_{nil}(ret) = \emptyset \wedge$ $hls_{nil}(y) \cap hls_{nil}(ret) = \emptyset \wedge$ $(x \neq nil) \Rightarrow max_{nil}(ret) \leq min_{nil}(x) \wedge$ $(y \neq nil) \Rightarrow max_{nil}(ret) \leq min_{nil}(y) \wedge$ $hls_{nil}(x) \cup hls_{nil}(y) \cup hls_{nil}(ret) = H \wedge$ $mskeys_{nil}(x) \cup_m mskeys_{nil}(y) \cup_m mskeys_{nil}(ret) = M$

Fig. 7. Examples of specifications in the delta logic of lists and list-measures

function calls using frame reasoning. To illustrate the key idea of our technique better, we shall restrict ourselves to the simple case of the delta logic of lists and list-measures introduced earlier. A formulation for general delta logics is a natural extension.

For simplicity of exposition, let us consider generating a VC for the case of a basic block of the form: S_1; `foo`$(\overline{y})$. Let the pre and post conditions be denoted by φ_{pre} and φ_{post}. We already know how to encode the transformation obtained from S_1 in a delta logic formula, say T_1. We also know to write the pre and post conditions of the program in delta logic as well. Let these be $\varphi_{pre-delta}$ and $\varphi_{post-delta}$ respectively expressed as in Section 4, with the precondition quantified existentially and the postcondition quantified universally.

In order to encode frame reasoning for the function call, we do the following:

- We introduce a new function n' (modeling the new *next* pointer), similarly new data fields (say, key') and new recursive definitions, $list'$, etc., defined using the new functions.
- We add constraints that model frame reasoning, saying that a recursive predicate/function has the same valuation as it did before the function call, *for the program variables in the basic block*, if the heaplet corresponding to the recursive definition did not intersect the modified heaplet of the function call. We also express a similar constraint for the function n'. Note that these are quantifier-free formulae.
- We then express that the post-condition of the function `foo` holds, using the new recursive definitions.

Observe that the postcondition of the program would be encoded by using these new recursive definitions that use n'. We can then write these formulae in delta logic, by using parameterized versions of these new recursive definitions.

In the case of our decidable delta logic of lists and list-measures, we model the abstraction of the context in the post-state of the function call with a function T', analogous to the abstraction of the context in the pre-state using the uninterpreted function T (as explained in Section 5). We then express similar constraints on summarizing segments in the post-state's context using T' and reason with the resulting VC. Notice that the decidability result for the delta logic of lists and list-measures extends naturally to the addition of a second version of the recursive definitions defined over n' instead of n (and similarly for the data fields) and summarized by T' instead of T since the relationships between these due to the program transformation are all expressible by quantifier-free formulae.

Let the pre and post conditions of `foo` be denoted by FC_{pre} and FC_{post} respectively. Then, if the the VC for this program would be of the form:

$$\begin{gathered}\varphi_{pre-delta}(\mathcal{R}^{P1}, n, \overline{x}) \wedge T_1(\mathcal{R}^{P1}, \mathcal{R}^{P2}, n, \overline{x}) \wedge \\ \Big(FC_{pre}(\mathcal{R}^{P2}, n, \overline{x}) \Rightarrow FC_{post}({\mathcal{R}'}^{P3}, n', \overline{x}')\Big) \wedge \\ \textit{frame-reasoning}(\text{heaplet}(FC_{pre})) \Rightarrow \varphi_{post-delta}({\mathcal{R}'}^{P3}, n', \overline{x}')\end{gathered}$$

where $\mathcal{R}^{P_1}$, $\mathcal{R}^{P_2}$ refer to the parameterized recursive functions in the state of the program before the function call: one set of parameters each for the states before and after S_1, and ${\mathcal{R}'}^{P_3}$ refers to the parameterized recursive functions that are defined on the state of the program after the function call. n and n' refer to the respective pointers, and $\overline{x}$ and $\overline{x}'$ refer to the total set of program variables in the pre- and post-states of the function call. Finally, *frame-reasoning* is a formula that systematically infers valuations of recursive functions on the post-state of the function call from the corresponding valuations on the pre-state, depending on the heaplet of the function call. Observe that this is a quantifier-free delta logic formula.

It is easy to see that this technique can be extended to any number of basic blocks interrupted by function calls. This gives us a generalized VC generation technique that helps us verify more interesting programs, examples of which are discussed in Section 6.

8.6 Decidability of LM Logic

In this Section we will see the constraints that we introduce to translate a quantifier-free formula the contextual logic of list measures into an equisatisfiable quantifier-free recursion-free formula, the encoding of these constraints into SMT and a proof sketch of the decidability of the delta logic of list measures.

We will first consider the simpler logic $LM[ls, hls, rank]$. Recall the definitions of X and L from Section 5. We construct a quantifier-free recursive-definition-free 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 L described in Section 5. We introduce an uninterpreted function $T : V \cup (X \setminus \Delta) \longrightarrow V \cup X \cup \{\bot\}$ ($\bot$ is used to signify that the n-path on the location never intersects $X \cup L$). Let $Z \subseteq X$ such that the recursive definitions ls_z^P, hls_z^P, $rank_z^P$, for some $P \in \mathcal{P}$, occur in φ.

ψ is the conjunct of the following formulae:

- 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:

$$
\begin{aligned}
&\wedge_{v \in V \cup (X \setminus \Delta)} \big[(Dist_z(v) = 0 \Leftrightarrow v = z) \wedge \\
&v \neq z \Rightarrow \big(\ ((T(v) = \bot \vee Dist_z(T(v)) = \bot) \\
&\qquad \Rightarrow Dist_z(v) = \bot) \\
&\qquad \wedge \ ((T(v) \neq \bot \wedge Dist_z(T(v)) \neq \bot) \\
&\qquad \Rightarrow Dist_z(v) = Dist_z(T(v)) + 1) \ \ \big) \big]
\end{aligned}
$$

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

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

– 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 $v \in (X \setminus \Delta) \cup V$:

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

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

We show correctness of the above translation:

Theorem 4. *For any quantifier-free contextual formula $\varphi(\mathcal{P}, X)$ of $LM[ls, hls, rank]$, the quantifier-free and recursion-free formula $\psi(\mathcal{P}, X)$ obtained from the translation above is equisatisfiable given a common valuation for $\mathcal{P}$.*

So far in this section we specified various constraints on the *measure summaries* that yield the decidability of the delta logic of lists and list-measures. In this section, we shall detail the encoding of these constraints into SMT and sketch a proof of the decidability.

The decidability of the logic follows from the decidability of the contextual logic $LM[ls, hls, rank, len, mskeys, min, max, sorted]$. This in turn is proved by expressing constraints that ensure that the summaries correspond to true lists, and the fact that these constraints can be expressed using a combination of decidable SMT theories.

Recall first that the constraints for $LM[ls, hls, rank]$ require that the heaplet measures for disjoint segments be disjoint, that the heaplet of a location be the union of the heaplet of the first summarised segment (ending at $T(v)$) with the heaplet of $T(v)$ and finally some constraints relating the values of the recursive definitions with the parameters and the $Dist$ function. Given the 'tree-like' structure of X (see Figure 4) it is easy to see that any valuation of measures satisfying these constraints is realisable by true list segments. This is because the $Dist$ function accurately captures the reachability and distance on X using applications of T which, combined with the constraint for the list, heaplet and rank recursive definitions imply that they are computed accurately provided that the heaplet measures segments are realisable by true heaplets. This is ensured by the constraint on disjoint segments (segments with different initial locations).

We can also prove a similar result for $LM[ls, hls, rank, len, mskeys, min, max, sorted]$. Recall that the constraints are that 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 $min\mu(v) = max\mu(v) \neq \bot$, then $sorted\mu(v)$ can only be true.

Similarly as above, the first two cardinality constraints imply that we can realise list segments with appropriate lengths and keysas given by the valuation. The third constraint determines the $min\mu$ and $max\mu$ measures. It only remains to determine the order of the keys in each list segment to realise true segments. For this we simply realise the keys in a sorted order or otherwise depending on the valuation of sortedness measure for the segment. Note that any list segment with minimum element different from maximum can be realized by either a sorted or an unsorted list. Therefore the fourth constraint handles the degenerate case when the minimum is to be equal to the maximum where the sortedness measure must necessarily evaluate to $true$. Clearly the list segments we have constructed above have the same measures as that of the valuation and are true list segments, which is what we desire.

The above constraints, though seemingly simple, are hard to shoehorn into existing decidable theories (though the fourth constraint can be easily expressed). The first two constraints can be expressed using quantifier-free BAPA [24] 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 as well 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 formulae that use sets containing keys 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 fresh free variable k.

Our primary observation is that we can then restrict the multiset of keys to be over a *bounded* universe of elements. This bounded universe consists 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's 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 [24].

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, and write the effect of unions and intersections using Presburger arithmetic. The cardinality of the multiset is 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.

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 of mitigating this. First, there is existing work (see [25]) 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 without intersections, the identity of the elements do not matter and their multiplicities are preserved by representing with only one element. We exploit this in our implementation.

We also note here that in our experiments, we use a slightly more expressive logic (see Figure 7), which allows free set variables to denote expressions of the appropriate type, such as $H = hls_{nil}(x) \cup hls_{nil}(y)$, where H is a set variable over locations. We add parameterized predicates for sortedness in the descending order and handle it the same way as our $sorted_z^P$ predicate, as well as a new constructor for integer terms $\#(mskt, keyt)$ that denotes the multiplicity of $keyt$ in $mskt$ (constrained to be non-negative). The results that for the logic LM extend smoothly to this extended logic as well.