Delta Logics: Logics for Change

ANONYMOUS AUTHOR(S)

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

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

Classical logics, such as first-order logic with least fixpoints 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*. A lot of 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', '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. This is further complicated by the presence of recursively defined functions/predicates that refer to data fields, which can express complex properties combining both shape and data on the heap.

There are two general ways of translating Hoare triples to verification conditions in this setting. The first is to accurately capture the Hoare triple; this typically involves two versions of the recursive functions, one for the pre-state and one for the post-state, where the two are parameterized by different sets of pointer fields. Furthermore, the verification conditions introduces quantification to precisely capture the heaplet that is changed by the program snippet. All of this makes automated reasoning of the verification condition extremely complex.

The second way is to model the heap change as an arbitrary change of the modified portion of the heap, and use *frame reasoning* to argue properties are conserved across the heap transformation. This is at the heart of the design of separation logic. However, frame reasoning alone is not complete.

For program snippets that involve function calls, we believe that frame reasoning is appropriate, and continue recommending it. However, for program snippets without function calls, we argue in

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

this paper that an accurate encoding of the verification condition is possible that is also amenable to automated reasoning.

In this paper, we identify a new class of logics called *Delta-logics* for expressing verification conditions that addresses the above problem. Formulae in delta-logics are Boolean combinations of two kinds of formulae: one, *delta-specific* formulae, talk about the modified portion of the global heap, which we call Δ, without using any recursive definitions; the other kind, called *contextual/context-logic* formulae, strictly talk about the unbounded portion excluding Δ, and uses recursive definitions. A set of first-order interface variables (called *parameters*) are used to communicate information between Δ and the rest of the heap.

We motivate the desirable aspects of delta-logics for expressing verification conditions, and provide an overview of the mechanism of VC generation using delta-logics in Section 2.

When translating Hoare triples to VCs in delta-logic, the program snippets translate naturally as delta-logic clearly delineates the modified heaplet from its context, and the program's transformation is a delta-specific formula. A technical challenge, however, is to express the pre- and post-conditions which are on both the modified and unmodified portions of the heap. 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 simplicity of expressing VCs in delta logics enables us to design powerful *decidable* program logics. We define a delta-logic that expresses properties of list segments along with a varied collection of of measures on them, including their heaplets (for expressing separation properties), their lengths, the multisets of keys stored in them, the min/max keys stored in them, and their sortedness. We show that these delta-logic formulae can be translated 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. To the best of our knowledge, this is the most expressive decidable program logic for list manipulating programs.

We emphasize that though delta logics are particularly meant for program snippets that do not involve function calls, we show that it can be seamlessly combined with frame reasoning for function calls, leading us to define a comprehensive verification technique for programs with function calls.

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.

The main contributions of this paper are:

- A new class of logics called Delta-Logics, which are specially suited to expressing verification conditions of program manipulating a bounded set of locations.
- A nontrivial *Separability Theorem* that shows that quantifier-free FO+*lfp* formulae can be translated into delta-logic formulae.
- A technique for expressing verification conditions, of programs manipulating, heaps in delta-logics.
- A powerful decidable delta-logic over lists and list-measures using a reduction to standard SMT theories.
- An extension of the VC generation technique to incorporate frame reasoning for function calls with delta-logics, and consequently a decidability result for VCs of list-manipulating programs with function calls.

- An implementation and evaluation of the VC generation and decision procedure for the delta-logic of lists and measures, showing their efficacy on a suite of list-manipulating programs, both correct programs and faulty programs.

The paper is organized as follows. In Section 2, we motivate delta-logics, the separability theorem, an overview of VC generation using delta-logics, through an illustrative example. In Section ?? we formally define delta-logics and in Section ?? we prove the separability theorem. Section ?? discusses the VC generation mechanism for delta-logics from Hoare triples for basic blocks without function calls. In Section ?? we define the decidable delta-logic of lists and list-measures and provide its decision procedure. Combined with the VC generation technique, this provides a decision procedure for the program logic of list-manipulating programs without function calls. In Section ?? we extend the VC generation technique to verify list-manipulating programs with function calls by incorporating frame-reasoning for function calls with delta-logics. In Section ?? we discuss the implementation and evaluation of our technique on a suite of list-manipulating programs. Finally, in Section ?? we discuss and compare our work with existing literature. We then conclude in Section ?? with some comments on future work.

2 OVERVIEW AND MOTIVATING EXAMPLE

In this section, we shall provide the motivation for our paper, an overview of our technique, and illustrate the method on a motivating example.

2.1 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 $R(x)$ 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 [?]. However, vanilla frame reasoning can be incomplete, as we shall argue through a motivating example (see Section ??).

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 [?], and there is work on expressing weakest preconditions in separation logic for such blocks using the magic wand [?]; 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 is of the form:

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

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 H modified by the program, and the formula will have conjuncts of the form $\forall y \notin H.p'(y) = p(y)$, for every $p \in \overline{p}$. This introduces universal quantification, which is harder to reason with automatically. However, for basic blocks that do not involve function calls, H 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 and can dramatically affect the valuation of recursively defined predicates/functions, which are global. 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.

2.2 Overview of method

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 in the above section. In particular, formulations in delta-logics will avoid the need for *two* diferent recursive definitions R and R'. 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 **??**). 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^R, where P^R 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.

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 **??**). We can view the program's transformation of 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_R for expressing properties of

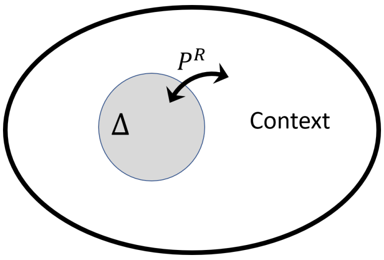

Fig. 1. Delta-logic

the pre-heap, and parameterized over the communication variables P'_R 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 submodel (as opposed to the naive formulation, which would have two infinite submodels) to build new powerful classes of decidable logics over lists and list-measures.

Expressing verification conditions in Delta-logics. With the above model of verification conditions in mind, let us examine how to write VCs corresponding to Hoare triples in delta-logics. The key idea is to set up parameterized sets of recursive definitions with two sets of parameters P_R and P'_R, as described above, expressing the precondition using one and the postcondition using the other. The program transformation of the Δ portion of the pre-heap to that of the post-heap can be described in quantifier-free FO, since they are finite.

The main challenge that remains is to express the precondition and postcondition using Delta logics. In order to do this, we prove a general technical result for delta-logics, called the *separability theorem* in Section **??**. This theorem shows that any quantifier-free FO formula with recursive definitions (with *lfp* semantics) can be expressed in delta-logics: as a boolean combination of delta-specific formulae and contextual formulae.

The separation theorem sets up communication between the two portions: the Δ portion of the heap and its context. The context is handled using the parameterized recursive definitions and the Δ portion, being finite, is handled by 'unfolding' the recursive definition across Δ. This unfolding would stop at the 'boundary' of Δ and the context, which is where we use the interface variables to communicate the valuations across these disjoint portions. These are the mutual constraints that were referred to earlier in this section. However, it turns out that a new set of recursive definitions and communication variables involving a notion of *rank* for each (parameterized) recursive definition is needed to accurately capture least fixpoints, for the parameterized recursive functions are now merely defined on the context and the *lfp* semantics cannot be preserved by a naive unfolding of the definition, even if the portion on which the unfolding is done is finite (for instance, the familiar definition of lists as either being *nil* or pointing to a location that points to a list is incorrect with just fixpoint semantics, as it would then be acceptable to call a cycle a list as well). These ranks can, however, be constrained to be *bounded integers* as opposed to ordinals (though the heap is infinite).

Decidable Delta-logics. As a culmination of the above arguments, we demonstrate the efficacy of delta-logics to build a powerful program logic for reasoning with manipulations of lists and list-measures. The decision procedure crucially relies on the simplicity of VCs expressed in delta-logics— having *one* infinite contextual heap, two finite heaplets and communication between them using only first-order variables.

2.3 Motivating Example

We shall now illustrate our method on a motivating example. Let us consider the following Hoare triple with pre/post conditions written in FO+*lfp*:

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

where $list(x)$ is a recursive definition that that holds when x points to a list, and in which case, $hlist(x)$, another recursive definition, captures the set of locations in that list. We omit these definitions.

It is easy to see that this Hoare Triple is valid. First notice that frame reasoning argues that w continues to point to a list. There are then two cases: if y does not belong to the list pointed to by x, then since y is not in the heaplet, frame reasoning would prove that x continues to point to a list. However, in the case when y does belong to the list pointed to by x, we cannot use frame reasoning to immediately conclude that x will continue to point to a list. However, notice that x will point to a list in this case as well, since y will point to w which points to a list. Therefore, frame reasoning *alone* will not be sufficient to conclude the postcondition from the precondition and the program transformation at this point.

The naive VC for this program is expressed by using 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 that pointed to by y:

$$\Big(\big(list(x) \wedge list(w) \wedge y \notin hlist(w)\big) \wedge \big((next'(y) = w \wedge y' = y \wedge w' = w \wedge x' = x) \wedge (\ \forall u.\, u \neq y \Rightarrow next'(u) = next(u)\ )\big)\Big) \Rightarrow\ (list'(x'))$$

where the quantifier is interpreted to range over all locations on the heap. We shall now convert this to a VC in delta-logic.

In this case, the set of modified locations is $\Delta = \{y\}$. To write the VC in delta-logic we shall first use the separability theorem (Section **??**) to write the pre- and and post-conditions in delta-logic. The separability theorem introduces new parameterized *rank* functions for each parameterized recursive definition. For the precondition, this yields a delta-logic formula of the kind:

$$\big(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)\big)\ \wedge\ \beta(P)$$

where $list^P()$ and $hlist^P()$ are the parameterized versions with the parameter set P. The domain of all of these definitions is the context, and therefore the formula $\big(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)\big)$ is a contextual formula. β is a delta-logic formula that accurately captures the least fixpoint semantics of the original definitions using the ranks and communication variables P. Similarly, we can also write the post-condition in delta-logic as $list'^Q(x') \wedge \beta(Q)$.

Substituting these into the VC gives:

$$\Big(\big(\big(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)\big) \wedge \beta(P)\big) \wedge \big((next'(y) = w \wedge y' = y \wedge w' = w \wedge x' = x) \wedge (\ \forall u.\, u \neq y \Rightarrow next'(u) = next(u)\ )\big)\Big) \Rightarrow\ \big(list'^Q(x') \wedge \beta(Q)\big)$$

We now come to the crux of the versatility of delta-logics in expressing VCs. Notice that $list'^Q(x')$ on the right of the implication above is a *contextual* formula defined using $next'$. However, on the context, $next'$ is precisely *next*. Hence we can replace $list'^Q(x')$ with $list^Q(x')$, the latter being defined in terms of $next$. Since $next'$ is never used on the context, we can drop the universally quantified clause, giving us the VC:

$$\Big(\big(\,(list^P(x) \wedge list^P(w) \wedge y \notin hlist^P(w)) \wedge \beta(P)\big) \wedge \big(\,(next'(y) = w \wedge y' = y \wedge w' = w \wedge x' = x)\,\big)\Big) \Rightarrow \big(list^Q(x') \wedge \beta(Q)\big)$$

It is clear that this is a *quantifier-free* delta-logic formula, and there is only one recursive definition for *list* which is defined using *next*, though it is parameterized with P and Q.

The problem now reduces to solving for the validity of this VC in delta-logic. In the decidability section (Section **??**), we show exactly how to build an effective procedure to decide the validity of delta-logic formulae over lists with various measures (which, in fact solve this particular example as well; see Section **??**). The problem really is of finding a decision procedure for the solution of contextual formulae. To do this we observe that the context can be abstracted by a finite set of locations, and the list segments between them, if they exist, summarized in a way that can be encoded in SMT. We shall describe this in further detail in Section **??**

3 DELTA-LOGICS

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

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

Let σ_0 be a special sort that we refer to as the *location* sort, which will model locations 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. Relations in $\mathcal{P}$ are over tuples of one sort only.
- The functions in $\mathcal{F}$ whose domain is over the foreground sort σ_0 are *unary.* Also, relations over the foreground sort σ_0 are unary relations.
- Recursively defined functions (in $\mathcal{G}$) are all unary functions from the foreground sort σ_0 to the foreground sort or a background sort. Recursively defined relations (in $\mathcal{R}$) are all unary relations on the foreground sort σ_0.

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

The logic FO+*lfp* that we use consists of a set of recursive definitions of (unary) predicates and functions (with least fixpoint semantics) and a quantifier-free formula that uses these definitions. For a simpler exposition, a definition of a recursive function R will be of the form: $R ::=_{lfp} \varphi(x, R(p_1(x)), ..R(p_n(x)))$ where p_i are unary functions and φ is monotonic, i.e, the $R(p_i(x))$ occur in the defintion under an even number of negations.

The latttice that we have for computing the least fixpoints for predicates is $\{\top, \bot\}$ with $\bot < \top$. For functions whose range is a domain $\mathcal{D}$, 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.

Furthermore, we restrict the kind of recursively defined predicates to have definitions of the form $R(x) ::= \varphi(x, e)$ where e is a set of conjunctions involving $R(p_i(x))$. This restricts definitions to have a unique dependence on their successors, and most natural definitions for heaps satisfy this property. Similarly, the restriction on recursively defined functions is that they have definitions of the form $G(x) ::= t(x, e)$ for some term t, such that e is a term involving terms of the form $G(p_i(x))$.

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

Contextual Formulae. Intuitively, a contextual formula $\varphi(\vec{x})$ 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 3.1 (Models differing only on Δ). Let M and M' be two Σ-models with universe U that interpret constants the same way, and let I be an interpretation of variables over U. Then we say (M, I) and (M', I) differ only on Δ if:

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

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

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

Definition 3.2 (Contextual formulae). An FOL+lfp formula over Σ, $\varphi(\vec{x})$, is a contextual formula if for every two Σ-models M and M' with the same universe and every interpretation I such that (M, I) and (M', I) differ only on Δ, $M, I \models \varphi$ iff $M', I \models \varphi$. □

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

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

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

is a contextual formula/definition with respect to Δ. The above defines lists but by "imbibing" facts about whether a location v in Δ is a list using the free Boolean variable b_v. These Boolean variables play the role of the interface variables/parameters mentioned earlier. The least fixpoint semantics of the above definition gives a unique definition: $ls(u)$ is true iff there is a path using the pointer $n()$ that either ends in nil or ends in a node v in Δ where b_v is true. Note that the formula $n(x)$ is guarded by *the check $x \in \Delta$, and hence is a contextual formula. Changing the model to reinterpret $n()$ over Δ (but preserving the interpretation of the variables b_v, $v \in \Delta$) will not change the definition of ls.*

Delta-specific formulae: A delta-specific formula is a quantifier-free formula where every occurrence of a term of the form $f(t)$, (for an uninterpreted function f) $t \in \Delta$. Furthermore the formula does not refer to any of the recursively defined functions/predicates.

Delta-logic formulae: A delta-logic formula is a Boolean combination of contextual formulae and delta specific-formulae.

Example: The formula $u'{=}n(u) \wedge ls^*(u')$ is a delta-logic formula that uses the above definition of ls.

4 A SEPARABILITY THEOREM

In this section, we show a key result: that for any quantifier-free FO+*lfp* formula we can effectively find an equivalent quantifier-free delta-logic formula. We do this by 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.

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

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

$$R^P(x) :=_{lfp} \quad R^d \;\; if\, [\![x]\!] = [\![d]\!] \text{ for some } d \in \Delta \qquad \text{(delta case)}$$
$$\varphi[R^P/R] \;\; if\, [\![x]\!] \notin [\![\Delta]\!] \qquad \text{(recursive case)}$$

It is easy to see that for a formula $R(x)$, $R^P(x)$ would be a contextual formula since any model of it would not depend on the valuation of PF over Δ (see definition of ls^* in Section **??** for example).

To capture the semantics of the original *lfp* definition, we constrain these parameters. This will yield a definition that is equivalent in FO+*lfp* under such constraints.

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

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

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

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.

Finally, we define a delta-logic formula β_R that 'computes' the value of R on Δ by unfolding the definition of R using the pointer fields on Δ. To this end, we define the final sets of parameters to be included in P in similar vein as above to 'communicate' from the context to Δ. These parameters will correspond to *boundary* variables: $p_i(\Delta) \setminus \Delta$ for some i, and are therefore named thus:

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

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

$$\begin{aligned}
\bigwedge_{d\in\Delta} \Big[&\Big(\varphi(d)[\bot/R] \neq \bot \implies R^d = \varphi(d)[\bot/R] \wedge \Big(RANK_R^d = 0\Big)\Big) \text{ (base case)} \\
&\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P/R] \neq \bot) \implies \Big(R^d = \varphi(d)[P/R] \\
&\qquad \wedge \Big(RANK_R^d = \max_{1\le i\le k}(\{RANK_R^{p_i(d)}\}) + 1\Big)\Big)\Big) \text{ (recursive case)} \\
&\wedge \Big((\varphi(d)[\bot/R] = \bot \wedge \varphi(d)[P/R] = \bot) \implies \Big(R^d = \bot \wedge \Big(RANK_R^d = \bot\Big)\Big)\Big) \Big]\text{(undefined)}
\end{aligned}$$

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

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

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

$\bigwedge_{1\le i\le k} \Big[p_i(d) \notin \Delta \implies \Big(R^{p_i(d)} = R^P(p_i(d))\Big) \wedge \Big(RANK_R^{p_i(d)} = Rank_R(p_i(d))\Big)\Big]$ This formula in conjunction with the above constraints forms the delta-logic formula β_R introduced above.

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

Lemma 4.1. *For any recursively defined function R,* $(\exists P_R.\ \beta_R) \wedge \Big(\forall P_R.\ \Big(\beta_R \implies R^{P_R} = R\Big)\Big)$

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

Theorem 4.2 (Separability). $\alpha \equiv \exists P_{\mathcal{R}}.\ \alpha[\mathcal{R}^P/\mathcal{R}] \wedge \Big(\bigwedge_{R\in\mathcal{R}} \beta_R\Big).$

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

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

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

5 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.

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 $\wedge_{f\in P} (\forall z.\, z \notin \Delta \Rightarrow f'(z) = f(z))$, which captures the fact that the fields of elements outside Δ have not changed. 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 delta-logic formula without introducing universal quantification.

First, using the separability theorem, Theorem ??, we can write α_{pre} and α_{post} as delta-logic formulae $\alpha^*_{pre}(U, X, P, \mathcal{R}^U)$ and $\alpha^*_{post}(V, X', P', (\mathcal{R}')^V)$. 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 $(R)^V$. Also, we can *remove* the universally quantified conjunct in T that says that fields for locations outside Δ are unaltered.

This leaves us with a verification condition of the form:

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

where T_Δ is a delta-specific formula that simply says how the variables X change to X', and the fields P change to P' over Δ. The VC is now clearly a delta-logic formula.

Notice that though we have two set of recursive definitions, the definitions do not rely on different sets of data and pointer fields (which are functions), but rather differ only on the first-order parameter variables U and V.

6 A DECIDABLE DELTA LOGIC ON LISTS WITH LIST MEASURES

In this section, we will define a delta-logic on linked lists equipped with *list measures*— measures of list segments that include 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 quantifier-free first-order logic fragment of this delta-logic is *decidable.*

We prove decidability by first proving that the corresponding contextual logic formuale 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, which is decidable using a Nelson-Oppen combination of decidable theories of arithmetic, sets, and uninterpreted functions.

The contextual logic of list measures

We now define a contextual logic over list segments and measures over them. Notice from Section **??** 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 measures, 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 6.1 (Recursive Definitions for the Logic of List Measures (LM)). Let us fix a set of *parameter variables 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 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 parameter variables P is defined using the following recursive definitions, which depend crucially on the parameter variables 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 variables LS_z^v ($v \in \Delta$), and where z is any element of Δ or the constant location nil. (The relation $ls_{nil}()$ captures whether a location points to a list ending with nil.)
  This is defined as follows:
  $$ls_z^P(x) :=_{lfp} \Big(x = z \vee \Big(x \neq z \wedge x \neq nil \wedge x \notin \Delta \wedge ls_z^P(n(x)) \Big) \vee$$
  $$\Big(x \neq z \wedge x \in \Delta \wedge \bigwedge_{v \in \Delta} (x = v \Rightarrow LS_z^v) \Big) \Big)$$
- We have recursive definitions that capture the heaplet of such list-segments, where the heaplet of list-segments from an element v in Δ to z (where $z \in \Delta \cup \{nil\}$) is imbibed from the set variable HLS_z^v:
  $$hls_z^P(x) :=_{lfp} \begin{array}{r} \emptyset \ \mathit{if} \ [\![x]\!] = [\![z]\!] \\ \{x\} \cup hls_z^P(n(x)) \ \mathit{if} \ [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] \neq [\![nil]\!] \wedge [\![x]\!] \notin [\![\Delta]\!] \\ HLS_z^v \ \mathit{if} \ [\![x]\!] \neq [\![z]\!] \wedge [\![x]\!] = [\![v]\!] \wedge v \in \Delta \end{array}$$
- We have similar recursive definitions that capture the multiset of data elements stored in list segments, the maximum/minimum element stored inthe list, and a predicate capturing sortedness of list segments. We omit these definitions: see Appendix for detailed definitions.

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 described in Figure 1. The logic *LM* allows first-order variables that range over locations, keys, and integers. Note that dereferencing pointers of locations is completely disallowed— the delta-specific formula will refer to such dereferences, depending on the particular state modeled. Here Δ is assumed to be a subset of the free location variables in the contextual formula. Formulae in *LM* are allowed to refer to recursive predicates/functions defined using over various sets of parameter variables.

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

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

Fig. 2. The context logic LM of list measures involving list-segments, heaplets, multisets of keys, max, min, and sortedness.

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

We can now state the main result of this section:

Theorem 6.2. *For any quantifier-free formula $\varphi(\mathcal{P}, X)$ of $LM[ls, hls, rank, len, min, max, sorted]$, the quantifier-free and recursion-free formula $\psi(\mathcal{P}, X)$ obtained φ obtained from the translation below satisfies the following property: for any interpretation of the free variables in $\mathcal{P} \cup X$, there is a model for φ iff there is a model for ψ.*

Corollary 6.3. *The delta-logic of linear measures is decidable.*

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

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

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

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

Our decision procedure 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 the paths using the n-pointer on a location x: (a) the path may reach a node in Δ, (b) the path may reach a node that is reachable also from another location in X, or (c) the path may never reach a location in Δ nor a location that is reachable from another location in X.

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

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

We construct a formula ψ that is satisfiable iff φ is satisfiable, as follows. First, we fix a new set (distinct from X) of location variables $V = v_1, \ldots v_{|X|-1}$, to stand for the merging locations described above. We introduce an uninterpreted function $T : V \cup (X \setminus \Delta) \longrightarrow V \cup X \cup \{\bot\}$ ($\bot$ is used to signify that the n()-path on the location never intersects $X \cup L$). Let Z be the set of variables in 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

We can show the correctness of the above translation:

Theorem 6.4. *The statement of Theorem ?? holds for the fragment LM[ls, hls, rank].*

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 can be realized using true list segments. As for the fourth clause above, notice that any list segment with minumum element different from maximum can be realized by either a sorted or an unsorted list.

The above constraints on measures, 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 [?] (Boolean Algebra with Presburger Arithmetic) constraints, which is decidable. We can get around defining the minimum of list segments by having the set of keys store only offsets from the minimum (and including the key 0 always). However, capturing max and sortedness measures 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 [?].

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. We can hence translate the formula into a quantifier-free formula, giving the required theorem. This proves Theorem **??**.

Note that the above procedure introduces an exponential number of variables, and hence poses challenges to be effective in practice. There are several possible ways of mitigating this. First, there

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

is existing work (see [?]) on reasoning with BAPA that argues and builds practical algorithms that introduce far smaller universes in practice. Second, in the case where we allow only combinations of sets using union (and not intersection), and allow checking subset constraints and emptiness, we can show that introducing a *single* new element in the universe other than K suffices. The reason is that 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 below.

7 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. 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 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; $\texttt{foo}(\overline{y})$. Let the pre and post conditions be denoted by φ_{pre} and φ_{post}.

Observe that 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.

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 **??**). 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:

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

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 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 **??**.

7.1 Implementation and evaluation of decision procedure for LM

We implemented the decision procedure for the delta-logic $LM[ls, hls, rank, len, mskeys, min, max, sorted]$ using the reduction to SMT described above, solved using Z3. We applied our technique to a suite of list-manipulating programs.

The specifications for the programs were strong. For example, programs such as `deleteall` which removed a certain key from a list were also verified with the additional requirement any other arbitrary key had to be preserved in multiplicity across the program. We could also verify such properties as the heaplet of the resulting list being a subset of the original heaplet. The `detect_cycle` program was able to express the existence of cycles using the predicates in *LM*, and used them to verify Floyd's tortoise and the hare algorithm.

The results are summarized in Figure ??. The left column shows results of verifying correct prorgrams, while the column on the right shows results on 'buggy' variants of these programs. The buggy programs were obtained by expressing weaker annotations. Our tool worked well on all these examples and for the buggy programs, the satisfying valuation it provided was sufficiently informative to diagnose the error. To the best of our knowledge, ours is the only decidable verification tool that can handle this suite of examples.

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

8 RELATED WORK

There is rich literature on decidable fragments of separation logic [?] with inductive predicates. While the first works such as [????] handled only list segments, there has been further improvement. The work in [?] handles only list segments, but allows conjunctions with pure formulae from arbitrary decidable SMT theories. The work in [?] argues decidablility for separation logic with arbitrary/user-defined inductive predicates, and [??] adapts the idea of using equisatisfiable 'base' formulas instead of inductive predicates to include arithmetic constraints satisfying a certain property. However, none of these would be directly applicable to program verification since they do not address entailment, and do not contain an implication in their logic. The work in [?] does contain implications, but it does not allow for spatial formulas other than points-to, list segment and in particular not provide support for multiple natural measures which our work provides. However, in the light of our broader work on context-logics and delta-logics, it is valuable to consider the satisfiability problem in the context-logic since it models the unchanging heap and does not need to address entailments, of which these works above provide useful decision procedures.

In constrast, works such as [??] do address program verification and provide a decidable logic on spatial formulas that when reduced to SMT can be extended with other decidable SMT theories. There is reasonable overlap in the spirit of this claim to our work, where allowing expressions in arbitrary decidable SMT theories on Δ would preserve decidability as well, and would fit well within the paradigm of delta-logics. But where [?] can support trees/tree segments, it does not provide support for measures. Works such as[??] differ similarly. Lastly there is work in the spirit of [??], where one dispenses with decidability and is therefore incomparable in that respect with our work. In particular, [?] is a powerful method capable of verifying all of our examples, but such logics are incomplete and cannot give meaningful counterexamples like our work does.

9 CONCLUSION AND FUTURE WORK

In conclusion, we presented a general delta-logic for generating precise verification conditions for heap-manipulating programs that are also amenable to automated reasoning. We proved a Separability Theorem to generate VCs in delta-logics, and used it to build a powerful decidable delta-logic on lists and list-measures. By incorporating frame-reasoning for function calls along with delta-logics, we have provided a tool for verifying a large suite of list-manipulating programs.

We seek to do the following as future work:

- Extend our decidability result to trees and measures on them, similar to our list-measures, such as heaplet, height, multiset of keys, predicates checking for binary search trees, etc.
- Improve our codebase towards a programmer-facing tool that can be used for verification.
- Incorporate our method into a full-fledged verification engine such as Boogie.

REFERENCES

Josh "Berdine, Cristiano Calcagno, and Peter W." O'Hearn. "2004". "A Decidable Fragment of Separation Logic". In *"Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science" ("FSTTCS'04")*. "97–109".

Josh "Berdine, Cristiano Calcagno, and Peter W." O'Hearn. "2005". "Symbolic Execution with Separation Logic". In *"Proceedings of the Third Asian Conference on Programming Languages and Systems" ("APLAS'05")*. "52–68".

James "Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos" Gorogiannis. "2014". "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)" ("CSL-LICS '14")*. "25:1–25:10".

"W. N. Chin, C. David, H. H. Nguyen, and S. Qin". "2007". "Automated Verification of Shape, Size and Bag Properties". In *"12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)"*. "307–320".

Byron "Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James" Worrell. "2011". "Tractable Reasoning in a Fragment of Separation Logic". In *"Proceedings of the 22Nd International Conference on Concurrency Theory" ("CONCUR'11")*. "235–249".

Radu "Iosif, Adam Rogalewicz, and Jiri" Simacek. "2013". "The Tree Width of Separation Logic with Recursive Definitions". In *"Proceedings of the 24th International Conference on Automated Deduction" ("CADE'13")*. "21–38".

Viktor "Kuncak, Huu Hai Nguyen, and Martin" Rinard. "2005". "An Algorithm for Deciding BAPA: Boolean Algebra with Presburger Arithmetic". In *"Proceedings of the 20th International Conference on Automated Deduction" ("CADE' 20")*. "260–277".

Viktor "Kuncak and Martin" Rinard. "2007". "Towards Efficient Satisfiability Checking for Boolean Algebra with Presburger Arithmetic". In *"Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction" ("CADE-21")*. "215–230".

Quang Loc "Le, Jun Sun, and Wei-Ngan" Chin. "2016". "Satisfiability Modulo Heap-Based Programs". In *"Computer Aided Verification"*. "382–404".

Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *Computer Aided Verification*.

P. "Madhusudan, Gennaro Parlato, and Xiaokang" Qiu. "2011". "Decidable Logics Combining Heap Structures and Data". In *"Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages" ("POPL '11")*. "611–622".

Juan Antonio "Navarro Pérez and Andrey" Rybalchenko. "2011". "Separation Logic + Superposition Calculus = Heap Theorem Prover". In *"Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation" ("PLDI '11")*. "556–566".

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2013. Separation Logic Modulo Theories. In *Programming Languages and Systems (APLAS)*. Springer International Publishing, Cham, 90–106.

Ruzica "Piskac, Thomas Wies, and Damien" Zufferey. "2014". "Automating Separation Logic with Trees and Data". In *"Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559"*. "711–728".

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper. In *Tools and Algorithms for the Construction and Analysis of Systems*. 124–139.

Xiaokang "Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy" Madhusudan. "2013". "Natural Proofs for Structure, Data, and Separation". In *"Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation" ("PLDI '13")*. "231–242".

John C." "Reynolds. "2002". "Separation Logic: A Logic for Shared Mutable Data Structures". In *"Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science" ("LICS '02")*. "55–74".

APPENDIX

We give here the detailed definitions of the other measures that were omitted in Section 4.

- 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 $MSKeys_z^v$:

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

- We have recursive definitions that capture the maximum/minimum element of data elements stored in list segments, where again the maximum/minimum element of list-segments from an element v in Δ to z where $z \in \Delta \cup \{nil\}$) is imbibed from the data variable Max_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 $max(r_1, r_2) \equiv ite(r_1 \leq r_2, r_2, r_1)$.

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

  The function Min_z^P is similarly defined.
- We have a recursive definition that captures sortedness, using the minimum measure.

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