Quantified Data Automata on Skinny Trees: An Abstract Domain for Lists

Pranav Garg[1], P. Madhusudan[1], and Gennaro Parlato[2]

[1] University of Illinois at Urbana-Champaign, USA
[2] University of Southampton, UK

Abstract. We propose a new approach to heap analysis through an abstract domain of automata, called *automatic shapes*. Automatic shapes are modeled after a particular version of *quantified data automata on skinny trees* (QSDAs), that allows to define universally quantified properties of programs manipulating acyclic heaps with a single pointer field, including data-structures such singly-linked lists. To ensure convergence of the abstract fixed-point computation, we introduce a subclass of QSDAs called elastic QSDAs, which forms an abstract domain. We evaluate our approach on several list manipulating programs and we show that the proposed domain is powerful enough to prove a large class of these programs correct.

1 Introduction

The abstract analysis of heap structures is an important problem in program verification as dynamically evolving heaps are ubiquitous in modern programming, either in terms of low level pointer manipulation or in object-oriented programming. Abstract analysis of the heap is hard because abstractions need to represent the heap which is of unbounded size, and must capture both the *structure* of the heap as well as the unbounded *data* stored in the heap. While several data-domains have been investigated for data stored in static variables, the analysis of unbounded structure and unbounded data that a heap contains has been less satisfactory. The primary abstraction that has been investigated is the rich work on *shape analysis* [25]. However, unlike abstractions for data-domains (like intervals, octagons, polyhedra, etc.), shape analysis requires carefully chosen *instrumentation* predicates to be given by the user, and often are particular to the program that is being verified. Shape analysis techniques typically *merge* all nodes that satisfy the same unary predicate, achieving finiteness of the abstract domain, and interpret the other predicates using a 3-valued (must, must not, may) abstraction. Moreover, these instrumentation predicates often require to be encoded in particular ways (for example, capturing binary predicates as particular kinds of unary predicates) so as to not lose precision.

For instance, consider a sorting algorithm that has an invariant of the form:

$$\forall x, y.\ (\ (x \rightarrow^*_{next} y \wedge y \rightarrow^*_{next} i) \Rightarrow d(x) \leq d(y)\ )$$

which says that the sub-list before pointer i is sorted. In order to achieve a shape-analysis algorithm that discovers this invariant (i.e., captures this

F. Logozzo and M. Fähndrich (Eds.): SAS 2013, LNCS 7935, pp. 172–193, 2013.

invariant precisely during the analysis), we typically need instrumentation predicates such as $p(z) = z \rightarrow^*_{next} i$, $s(x) = \forall y.((x \rightarrow^*_{next} y \wedge y \rightarrow^*_{next} i) \Rightarrow d(x) \leq d(y))$, etc. The predicate $s(x)$ says that the element that is at x is less than or equal to the data stored in every cell between x and i. These instrumentation predicates are clearly too dependent on the precise program and property being verified.

In this paper, we investigate an abstract domain for heaps that works *without user-defined instrumentation predicates* (except we require that the user fix an abstract domain for data, like octagons, for comparing data elements).

We propose a radically new approach to heap analysis through an abstract domain of automata, called *automatic shapes* (automatic because we use automata). Our abstract domain are modeled after a particular kind of automata, called *quantified data automata*, that define, logically, universally quantified properties of heap structures. In this paper, we restrict our attention to acyclic heap structures that have only *one pointer field*; our analysis is hence one that can be used to analyze properties of heaps containing lists, with possible aliasing (merging) of them, especially at intermediate stages in the program. One-pointer acyclic heaps can be viewed as *skinny trees* (trees where the number of branching nodes is bounded).

Automata, in general, are classical ways to capture an infinite set of objects using finite means. A class of (regular) skinny trees can hence be represented using tree automata, capturing the structure of the heap. While similar ideas have been explored before in the literature [14], our main aim is to also represent properties of the *data* stored in the heap, building automata that can express universally quantified properties on lists, in particular those of the form

$$\bigwedge_i \forall \overline{x}.\,(Guard_i(\overline{p}, \overline{x}) \Rightarrow Data_i(d(\overline{p}), d(\overline{x})))$$

where $\overline{p}$ is the set of static pointer variables in the program. The $Guard_i$ formulas express structural constraints on the universally quantified variables and the pointer variables, while the $Data_i$ formulas express properties about the data stored at the nodes pointed to by these pointers. In this paper, we investigate an abstract domain that can infer such quantified properties, parameterized by an abstract numerical domain $\mathcal{F}_d$ for the data formulas and by the number of quantified variables $\overline{x}$.

The salient aspect of the automatic shapes that we build is that (a) there is no requirement from the user to define instrumentation predicates for the structural *Guard* formulas; (b) since the abstraction will not be done by merging unary predicates and since the automata can define how data stored at *multiple* locations on the heap are related, there is no need for the user to define carefully crafted unary predicates that relate structure and data (e.g., the unary predicate $s(x)$ defined above that says that the location x is sorted with respect to all successive locations that come after x but before i). Despite this lack of help from the user, we show how our abstract domain can infer properties of a large number of list-manipulating programs adequately to prove interesting quantified properties.

The crux of our approach is to use a class of automata, called quantified data automata on skinny trees (QSDA), to express a class of single-pointer heap structures and the data contained in them. QSDAs read skinny trees with data along with *all* possible valuations of the quantified variables, and for each of them check whether the data stored in these locations (and the locations pointed to by pointer variables in the program) relate in particular ways defined by the abstract data-domain $\mathcal{F}_d$.

We show, for a simple heap-manipulating programming language, that we can define an abstract post operator over the abstract domain of QSDAs. This abstract post preserves the structural aspects of the heap *precisely* (as QSDAs can have an arbitrary number of states to capture the evolution of the program) and that it soundly abstracts the quantified data properties. The abstract post is nontrivial to define and show effective as it requires automata-theoretic operations that need to simultaneously preserve structure as well as data properties; this forms the hardest technical aspect of our paper. We thus obtain an effective sound transfer function for QSDAs. However, it turns out that QSDAs are not complete lattices (infinite sets may not have least upper-bounds), and hence do not form an abstract domain to interpret programs. Furthermore, typically, in each iteration, the QSDAs obtained would grow in the number of states, and it is not easy to find a fixed-point.

Traditionally, in order to handle loops and reach termination, abstract domains require some form of widening. Our notion of widening is founded on the principle that lengths of stretches of the heap that are neither pointed to by program variables nor by the quantified variables (in one particular instantiation of them) must be ignored. We would hence want the automaton to check the same properties of the instantiated heap no matter how long these stretches of locations are. This notion of abstraction is also suggested by our earlier work where we have shown that such abstractions lead to *decidability*; in other words, properties of such abstracted automata fall into decidable logical theories [10,18]. Assume that the programmer computes a QSDA as an invariant for the program at a particular point, where there is an assertion expressed as a quantified property p over lists (such as “the list pointed to by *head* is sorted”). In order to verify that the abstraction proves the assertion, we will have to check if the language of lists accepted by the QSDA is contained in the language of lists that satisfy the property p. However, this is in general *undecidable*. However, this inclusion problem is decidable if the automata abstracts the lengths of stretches as above. Our aim is hence to *over-approximate* the QSDA into a larger language accepted by a particular kind of data automata, called *elastic* QSDA (EQSDA) that ignores the stretches where variables do not point to, and where ”merging” of the pointers do not occur [10, 18].

This *elastification* will in fact serve as the basis for widening as well, as it turns out that there are only a *finite* number of elastic QSDAs that express structural properties, discounting the data-formulas. Consequently, we can combine the elastification procedure (which over-approximates a QSDA into an elastic QSDA) and widening over the numerical domain for the data in order to obtain widening procedures that can be used to accelerate the computation for loops. In fact, the domain of EQSDAs is an abstract domain and a complete lattice (where infinite sets also have least upper-bounds), and there is a natural abstract interpretation between sets of concrete heap configurations and EQSDAs, where the EQSDAs permit widening procedures. We show a unique elastification theorem that shows that for any QSDA, there is a unique elastic QSDA that over-approximates it. This allows us to utilize the abstract transfer function on QSDAs (which is more precise) on a linear block of statements, and then elastify them to EQSDAs at join points to have computable fixed-points.

We also show that EQSDA properties over lists can be translated to a decidable fragment of the logic Strand [18] over lists, and hence inclusion checking an elastic QSDA with respect to any assertion that is also written using the decidable sublogic of Strand over lists is decidable. The notion of QSDAs and elasticity are extensions of recent work in [10], where such notions were developed for *words* (as opposed to trees) and where the automata were used for *learning* invariants from examples and counter-examples.

We implement our abstract domain and transformers and show, using a suite of list-manipulating programs, that our abstract interpretation is able to prove the naturally required (universally-quantified) properties of these programs. While several earlier approaches (such as shape analysis) can tackle the correctness of these programs as well, our abstract analysis is able to do this *without* requiring program-specific help from the user (for example, in terms of instrumentation predicates in shape analysis [25], and in terms of guard patterns in the work by Bouajjani et al [5]).

Related Work. Shape analysis [25] is the one of the most well-known technique for synthesizing invariants about dynamically evolving heaps. However, shape analysis requires user-provided instrumentation predicates which are often too particular to the program being verified. Hence coming up with these instrumentation predicates is not an easy task. In recent work [5,6,12,21], several abstract domains have been explored which combine the shape and the data constraints. Though some of these domains [6, 21] can handle heap structures more complex than singly-linked lists, all these domains require the user to provide a set of data predicates [12] or a set of structural guard patterns [5] or predicates over both the structure and the data constraints [6, 21]. In contrast, the only assistance our technique requires from the user is specifying a numerical domain over data formulas and the number of universally quantified variables.

For singly-linked lists, [20] introduces a family of abstractions based on a set of instrumentation predicates which track uninterrupted list segments. However these abstractions only handle structural properties and not the more-complex quantified data properties. Several separation logic based shape analysis techniques have also been developed over the years [3, 4, 9, 13]. But they too mostly handle only the shape properties (structure) of the heap.

Our automaton model for representing quantified invariants over lists is inspired by the decidable fragment of Strand [18] and can track invariants with guard constraints of the form $y \leq t$ or $t \leq y$ for a universal variable y and some term t. These structural constraints on the guard are very similar to array partitions in [8,11,15]. However, our automata model is more general. For instance, none of these related works can handle sortedness of arrays which requires quantification over more than one variable.

Techniques based on *Craig's interpolation* have recently emerged as an orthogonal way for synthesizing quantified invariants over arrays and lists [1,17,22,26]. These methods use different heuristics like term abstraction [26] or introduction of existential ghost variables [1] or finding interpolants over a restricted language [17,22] to ensure the convergence of the interpolant from a small number of spurious counter-examples. The shape analysis proposed in [24] is also counter-example driven. [24] requires certain quantified predicates to be provided by the user. Given these predicates, it uses a CEGAR-loop for incrementally improving the precision of the abstract transformer and also discovering new predicates on the heap objects that are part of the invariant.

Automata based abstract interpretation has been explored in the past [14] for inferring shape properties about the heap. However, in this paper we are interested in strictly-richer universally quantified properties on the data stored in the heap. [2] introduces a streaming transducer model for algorithmic verification of single-pass list-processing programs. However the transducer model severely constrains the class of programs it can handle; for example, [2] disallows repeated or nested list traversals which are required in sorting routines, etc.

In this paper we introduce a class of automata called quantified skinny-tree data automata (QSDA) to capture universally quantified properties over skinny-trees. The QSDA model is an extension of recent work in [10] where a similar automata model was introduced for words (as opposed to trees). Also, the automata model in [10] was parameterized by a *finite* set of data formulas and was used for *learning* invariants from examples and counter-examples. In contrast, we extend the automata in [10] to be instantiated with a (possibly-infinite) abstract domain over data formulas and develop a theory of abstract interpretation over QSDAs.

2 Programs Manipulating Heap and Data

We consider sequential programs manipulating acyclic singly-linked data structures. A *heap structure* is composed of locations (also called nodes). Each location is endowed with a *pointer field* `next` that points to another location or it is undefined, and a *data field* called `data` that takes values from a potentially infinite domain $\mathbb{D}$ (i.e. the set of integers). For simplicity we assume a special location, called *dirty*, that models an un-allocated memory space. We assume that the `next` pointer field of *dirty* is undefined. Besides the heap structure, a program also has a finite number of *pointer variables* each pointing to a location in the heap structure, and a finite number of *data variables* over $\mathbb{D}$. In our programming language we do not have procedure calls, and we handle non-recursive procedures calls by inlining the code at call points. In the rest of the section we formally define the syntax and semantics of these programs.

$\langle prgm\rangle ::=$ pointer $p_1,\ldots,p_k$; data $d_1,\ldots,d_\ell$; $\langle pc\_stmt\rangle^+$
$\langle pc\_stmt\rangle ::= pc : \langle stmt\rangle;$
$\langle stmt\rangle ::= \langle ctrl\_stmt\rangle \mid \langle heap\_stmt\rangle$
$\langle ctrl\_stmt\rangle ::= d_i := \langle data\_expr\rangle \mid \texttt{skip} \mid \texttt{assume}(\langle pred\rangle)$
$\mid$ `if` $\langle pred\rangle$ `then` $\langle pc\_stmt\rangle^+$ `else` $\langle pc\_stmt\rangle^+$ `fi`
$\mid$ `while` $\langle pred\rangle$ `do` $\langle pc\_stmt\rangle^+$ `od`
$\langle heap\_stmt\rangle ::= \texttt{new}\ p_i \mid p_i := \texttt{nil} \mid p_i := p_j$
$\mid\ p_i := p_j \rightarrow \texttt{next} \mid p_i \rightarrow \texttt{next} := \texttt{nil} \mid p_i \rightarrow \texttt{next} := p_j$
$\mid\ p_i \rightarrow \texttt{data} := \langle data\_expr\rangle$

Fig. 1. Simple programming language

Syntax. The syntax of programs is defined by the grammar of Figure 1. A program starts with the declaration of pointer variables followed by a declaration of data variables. Data variables range over a potentially infinite data domain $\mathbb{D}$. We assume a language of data expressions built from data variables and terms of the form $p_i \rightarrow \texttt{data}$ using operations over $\mathbb{D}$. Predicates in our language are either data predicates built from predicates over $\mathbb{D}$ or structural predicates concerning the heap built from atoms of the form $p_i == p_j$, $p_i == \texttt{nil}$, $p_i \rightarrow \texttt{next} == p_j$ and $p_i \rightarrow \texttt{next} == \texttt{nil}$, for some $i, j \in [1, k]$. Thereafter, there is a non-empty list of labelled statements of the form $pc\!:\!\langle stmt\rangle$ where pc is the *program counter* and $\langle stmt\rangle$ defines a language of either C-like statements or statements which modify the heap. We do not have an explicit statement to *free* locations of the heap: when a location is no longer reachable from any location pointed by a pointer variable we assume that it automatically disappears from the memory. For a program P, we denote with PC the set of all program counters of P statements. Figure 2(a) shows the code for program *sorted list-insert* which is a running example in the paper. The program inserts a *key* into the sorted list pointed to by variable *head.*

Semantics. A *configuration* C of a program P with set of pointer variables PV and data variables DV is a tuple $\langle pc, H, pval, dval\rangle$ where

- $pc \in PC$ is the program counter of the next statement to be executed;
- H is a *heap configuration* represented by a tuple $(Loc, \texttt{next}, \texttt{data})$ where (1) *Loc* is a finite set of heap locations containing special elements called *nil* and *dirty*, (2) $\texttt{next} : Loc \mapsto Loc$ is a partial map defining an edge relation among locations such that the graph $(Loc, \texttt{next})$ is acyclic, and (3) $\texttt{data} : Loc \mapsto \mathbb{D}$ is a map that associates each *non-nil* and *non-dirty* location of *Loc* with a data value in $\mathbb{D}$;
- $pval : \widehat{PV} \rightarrow Loc$, where $\widehat{PV} = PV \cup \{\texttt{nil}, \texttt{dirty}\}$, associates each pointer variable of P with a location in H. If $pval(p) = v$ we say that node v is *pointed* by variable p. Furthermore, each node in *Loc* is reachable from a node pointed by a variable in PV. There is no outgoing ($\texttt{next}$) edge from location *dirty* and there is a $\texttt{next}$ edge from the location pointed by $\texttt{nil}$ to *dirty* (henceforth we use PV everywhere instead of the $\widehat{PV}$);
- $dval : DV \rightarrow \mathbb{D}$ is a valuation map for the data variables.

Figure 2(b) graphically shows a program configuration which is reachable at program counter 8 of the program in Figure 2(a) (as explained later we encode the data variable *key* as a pointer variable in the heap configuration). The *transition relation* of a program P, denoted $\xrightarrow{stmt}_P$ for each statement *stmt* of P, is defined as usual. The control-flow statements update the program counter, possibly depending on a predicate (condition). The assignment statements update the variable valuation or the heap structure other than moving to the next program counter. Let us define the concrete transformer $F^{\natural} = \lambda\mathcal{C}.\{\mathcal{C}' \mid \mathcal{C} \xrightarrow{stmt}_P \mathcal{C}'\}$. The concrete semantics of a program is given as the least fixed point of a set of equations of the form $\psi = F^{\natural}(\psi)$.

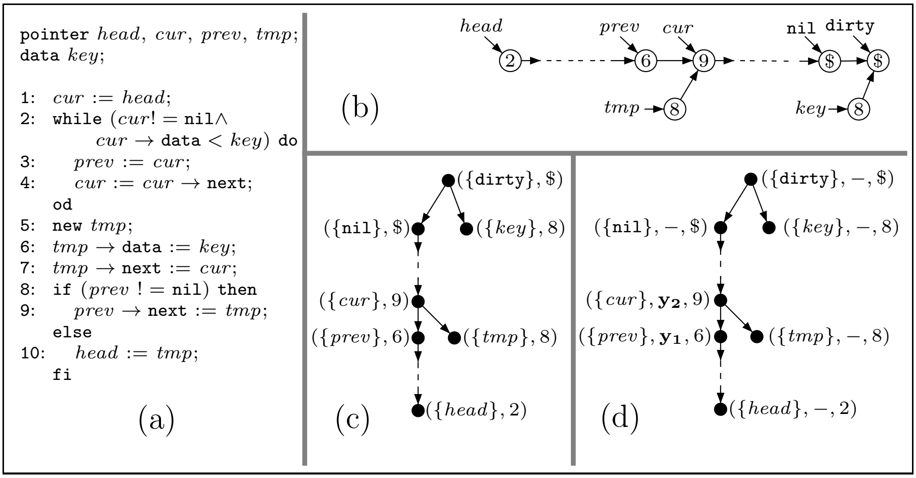
To simplify the presentation of the paper, we assume that our programs do not have data variables. This restriction, indeed, does not reduce their expressiveness: we can always transform a program P into an *equivalent* program P' by translating each data variable d into a pointer variable that will now point to a fresh node in the heap structure, in which the value d is now encoded by $d \rightarrow \texttt{data}$. The node pointed by d is not pointed by any other pointer, further, $d \rightarrow \texttt{next}$ points to *dirty*. Obviously, wherever d is used in P will now be replaced by $d \rightarrow \texttt{data}$ in P'.

3 Quantified Skinny-Tree Data Automata

In this section we define *quantified skinny-tree data automata* (QSDAs, for short), an accepting mechanism of program configurations (represented as special labeled trees) on which we can express properties of the form $\bigwedge_i \forall y_1, \ldots, y_\ell.\, Guard_i \Rightarrow Data_i$, where variables y_i range over the set of locations of the heap, $Guard_i$ represent quantifier-free structural constraints among the pointer variables and the universally quantified variables y_i, and $Data_i$ (called *data formulas*) are quantifier-free formulas that refer to the data stored at the locations pointed either by the universal variables y_i or the pointer variables, and compare them using operators over the data domain. In the rest of this section, we first define *heap skinny-trees* which are a suitable labeled tree encodings for program configurations; we then define *valuation trees* which are heap skinny-trees by adding to the labels an instantiation of the universal variables. *Quantified skinny-tree data automata* is a mechanism designed to recognize valuation trees. The *language* of a QSDA is the set of all heap skinny-trees such that all valuation trees deriving from them are accepted by the QSDA. Intuitively, the heap skinny-trees in the language defined by the QSDA are all the program configurations that verify the formula $\bigwedge_i \forall y_1, \ldots, y_\ell. Guard_i \Rightarrow Data_i$.

Let T be a tree. A node u of T is *branching* whenever u has more than one child. For a given natural number k, T is *k-skinny* if it contains at most k branching nodes.

Heap Skinny-Trees. Let PV be the set of pointer variables of a program P and $\Sigma = 2^{PV}$ (let us denote the empty set with a blank symbol b). We associate with each P configuration $C = \langle pc, H, pval, dval\rangle$ with $H = (Loc, \texttt{next}, \texttt{data})$, the $(\Sigma \times \mathbb{D})$-labeled graph $\mathcal{H} = (T, \lambda)$ whose nodes are those of Loc, and where (u, v) is an edge of T iff $\texttt{next}(v) = u$ (essentially we reverse all $\texttt{next}$ edges). From the definition of program configurations, since all locations are required to be reachable from some program variable, it is easy to see that T is a k-skinny tree



```
pointer head, cur, prev, tmp;
data key;

1:  cur := head;
2:  while (cur! = nil∧
          cur → data < key) do
3:     prev := cur;
4:     cur := cur → next;
    od
5:  new tmp;
6:  tmp → data := key;
7:  tmp → next := cur;
8:  if (prev ! = nil) then
9:     prev → next := tmp;
    else
10:    head := tmp;
    fi
```

Fig. 2. **(a)** *sorted list-insert* program P; **(b)** shows a P configuration at program counter 8; **(c)** is the heap skinny-tree associated to (b); **(d)** is a valuation tree of (c)

where $k = |PV|$. The labeling function $\lambda : Loc \rightarrow (\Sigma \times \mathbb{D})$ is defined as follows: for every $u \in Loc$, $\lambda(u) = (S, d)$ where S is the set of all pointer variables p such that $pval(p) = u$, and $d = \mathtt{data}(u)$. We call $\mathcal{H}$ the *heap skinny-tree* of C.

Heap skinny-trees are formally defined as follows.

Definition 1 (Heap Skinny-Trees). *A* heap skinny-tree *over a set of pointer variables PV and data domain $\mathbb{D}$, is a $(\Sigma \times \mathbb{D})$-labeled k-skinny tree (T, λ) with $\Sigma = 2^{PV}$ and $k = |PV|$, such that:*

- *for every leaf v of T, $\lambda(v) = (S, d)$ where $S \neq \emptyset$;*
- *for every $p \in PV$, there is a unique node v of T such that $\lambda(v) = (S, d)$ with $p \in S$ and some $d \in \mathbb{D}$;*
- *for a node v of T such that $\lambda(v) = (S, d)$, if $\mathtt{nil} \in S$ then v is one of the children of the root of T; if v is the root of T then $S = \{\mathtt{dirty}\}$.* □

Figure 2(c) shows the heap skinny-tree corresponding to the program configuration of Figure 2(b). Note that though the program handles a singly linked list, in the intermediate operations we can get trees. However they are special trees with bounded branching. This example illustrates that program configurations of list manipulating programs naturally correspond to heap skinny-trees. It also motivates why we need to extend automata over words introduced in [10] to quantified data automata over skinny-trees. We now define valuation trees.

Valuation Trees. Let us fix a finite set of *universal* variables Y. A *valuation tree* over Y of a heap skinny-tree $\mathcal{H}$ is a $(\Sigma \times (Y \cup \{-\}) \times \mathbb{D})$-labeled tree obtained from $\mathcal{H}$ by adding an element from the set $Y \cup \{-\}$ to the label, in which every element in Y occurs exactly once in the tree. We use the symbol '$-$' at a node v if there is no variable from Y labeling v. A valuation tree corresponding to the heap skinny-tree of Figure 2(c) is shown in Figure 2(d).

Quantified skinny-tree data automata are a mechanism to accept skinny-trees. To express properties on the data present in the nodes of the skinny-trees, QSDAs are parameterized by a set of data formulas F over $\mathbb{D}$ which form a complete-lattice $\mathcal{F} = (F, \sqsubseteq_{\mathcal{F}}, \sqcup_{\mathcal{F}}, \sqcap_{\mathcal{F}}, \mathit{false}, \mathit{true})$ where $\sqsubseteq_{\mathcal{F}}$ is the partial-order on the data-formulas, $\sqcup_{\mathcal{F}}$ and $\sqcap_{\mathcal{F}}$ are the least upper bound and the greatest lower bound and *false* and *true* are formulas required to be in F and correspond to the bottom and the top elements of the lattice, respectively. Also, we assume that whenever $\alpha \sqsubseteq_{\mathcal{F}} \beta$ then $\alpha \Rightarrow \beta$. Furthermore, we assume that any pair of formulas in F are non-equivalent. For a logical domain as ours, this can be achieved by having a canonical representative for every set of equivalent formulas. Let us now formally define QSDAs.

Definition 2 (Quantified Skinny-Tree Data Automata). *A quantified skinny-tree data automaton (*QSDA*) over a set of pointer variables PV (with $|PV| = k$), a data domain $\mathbb{D}$, a set of universal variables Y, and a formula lattice $\mathcal{F}$, is a tuple $\mathcal{A} = (Q, \Pi, \Delta, \mathcal{T}, f)$ where:*

- *Q is a finite set of states;*
- *$\Pi = \Sigma \times \widehat{Y}$ is the alphabet where $\Sigma = 2^{PV}$ and $\widehat{Y} = Y \cup \{-\}$;*
- *$\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ where, for every $i \in [1, k]$, $\Delta_i : (Q^i \times \Pi) \mapsto Q$ is a partial function and defines a (deterministic) transition relation;*
- *$\mathcal{T} : Q \to 2^{PV \cup Y}$ is the type associated with every state $q \in Q$;*
- *$f : Q \to \mathcal{F}$ is a final-evaluation.* □

A valuation tree (T, λ) over Y of a program P, where N is the set of nodes of T, is *recognized* by a QSDA $\mathcal{A}$ if there exists a node-labeling map $\rho : N \mapsto Q$ that associates each node of T with a state in Q such that for each node t of T with $\lambda(t) = (S, y, d)$ the following holds (here $\lambda'(t) = (S, y)$ is obtained by projecting out the data values from $\lambda(t)$):

- if t is a leaf then $\Delta_0(\lambda'(t)) = \rho(t)$ and $\mathcal{T}(\rho(t)) = S \cup \{y\} \setminus \{-\}$.
- if t is an internal node, with sequence of children $t_1, t_2, \ldots, t_i$ then
  - $\Delta_i\,(\,(\rho(t_1), \ldots, \rho(t_i)),\ \lambda'(t)\,) = \rho(t)$;
  - $\mathcal{T}(\rho(t)) = S \cup \{y\} \setminus \{-\} \cup \left(\bigcup_{j\in[1,i]} \mathcal{T}(\rho(t_j))\right)$.
- if t is the root then the formula $f(\rho(t))$, obtained by replacing all occurrences of terms $y \to \texttt{data}$ and $p \to \texttt{data}$ with their corresponding data values in the valuation tree, holds true.

A QSDA can be thought as a *register* automaton that reads a valuation tree in a bottom-up fashion and stores the data at the positions evaluated for Y and locations pointed by elements in PV, and checks whether the formula associated to the state at the root holds true by instantiating the data values in the formula with those stored in the registers. Furthermore, the role of map $\mathcal{T}$ is that of enforcing that each element in $PV \cup Y$ occurs exactly once in the valuation tree.

A QSDA $\mathcal{A}$ *accepts* a heap skinny-tree $\mathcal{H}$ if $\mathcal{A}$ recognizes all valuation trees of $\mathcal{H}$. The *language* accepted by $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of all heap skinny-trees $\mathcal{H}$ accepted by $\mathcal{A}$. A language $\mathcal{L}$ of heap skinny-trees is *regular* if there is a QSDA $\mathcal{A}$ such that $\mathcal{L} = L(\mathcal{A})$. Similarly, a language $\mathcal{L}$ of valuation trees is *regular* if there is a QSDA $\mathcal{A}$ such that $\mathcal{L} = L_{\mathrm{v}}(\mathcal{A})$, where $L_{\mathrm{v}}(\mathcal{A})$ is the set of all valuation trees recognized by $\mathcal{A}$.

QSDAs are a generalization of *quantified data automata* introduced in [10] which handle only lists, as opposed to QSDAs that handle skinny-trees. We now introduce various characterizations of QSDAs which are used later in the paper.

Unique Minimal QSDA**.** In [10] the authors show that it is not possible to have a unique minimal (with respect to the number of states) quantified data automaton over words which accepts a given language over linear heap configurations. The proof gives a set of heap configurations over a linear heap-structure that is accepted by two different automata having the same number of states. Since QSDAs are a generalization of quantified data automata, the same counter-example language holds for QSDAs. However, under the assumption that all data formulas in $\mathcal{F}$ are pairwise non-equivalent, there does exist a canonical automaton at the level of *valuation trees.* In [10], the authors prove the canonicity of quantified data automata, and their result extends to QSDAs in a straight forward manner.

Theorem 1. *For each* QSDA $\mathcal{A}$ *there is a unique minimal* QSDA $\mathcal{A}'$ *such that* $L_v(\mathcal{A}) = L_v(\mathcal{A}')$*.*

We give some intuition behind the proof of Theorem 1. First, we introduce a central concept called *symbolic trees.* A symbolic tree is a $(\Sigma \times (Y \cup \{-\}))$-labeled tree that records the positions of the universal variables and the pointer variables, but does not contain concrete data values (hence the word symbolic). A valuation tree can be viewed as a symbolic tree augmented with data values at every node in the tree. There exists a unique tree automaton over the alphabet Π that accepts a given regular language over symbolic trees. It can be shown that if the set of formulas in $\mathcal{F}$ are pairwise non-equivalent, then each state q in the tree automaton, at the root, can be decorated with a unique data formula $f(q)$ which extends the symbolic trees with data values such that the corresponding valuation trees are in the given language of valuation trees.

Hence, a language of valuation trees can be viewed as a function that maps each symbolic tree to a uniquely determined formula, and a QSDA can be viewed as a Moore machine (an automaton with output function on states) that computes this function. This helps us separate the structure of valuation trees (the height of the trees, the cells the pointer variables point to) from the data contained in the nodes of the trees. We formalize this notion by introducing *formula trees.*

Formula Trees. A *formula tree* over pointer variables PV, universal variables Y and a set of data formulas $\mathcal{F}$ is a tuple of a $\Sigma \times (Y \cup \{-\})$-labeled tree (or in other words a symbolic tree) and a data formula in $\mathcal{F}$. For a QSDA which captures a universally quantified property of the form $\bigwedge_i \forall y_1, \ldots, y_\ell . Guard_i \Rightarrow Data_i$, the symbolic tree component of the formula tree corresponds to guard formulas like $Guard_i$ which express structural constraints on the pointers pointing into the valuation tree. The data formula in the formula trees correspond to $Data_i$ which express the data values with which a symbolic tree (read $Guard_i$) can be extended so as to get a valuation tree accepted by the QSDA. In our running example, consider a QSDA with a formula tree which has the same symbolic tree as the valuation tree in Figure 2(d) (but without the data values in the nodes) and a data-formula $\varphi = y_1 \rightarrow \texttt{data} \leq y_2 \rightarrow \texttt{data} \wedge y_1 \rightarrow \texttt{data} < key \wedge y_2 \rightarrow \texttt{data} \geq key$. This formula tree represents all valuation trees (including the one shown in Figure 2(d)) which extend the symbolic tree with data values which satisfy φ.

By introducing formula trees we explicitly take the view of a QSDA as an automaton that reads symbolic trees and outputs data formulas. We say a formula tree (t, φ) is accepted by a QSDA $\mathcal{A}$ if $\mathcal{A}$ reaches the state q after reading t and $f(q) = \varphi$. Given a QSDA $\mathcal{A}$, the language of valuation trees accepted by $\mathcal{A}$ gives an equivalent language of formula trees accepted by $\mathcal{A}$ and vice-versa. We denote the set of formula trees accepted by $\mathcal{A}$ as $L_f(\mathcal{A})$. A language over formula trees is called regular if there exists a QSDA accepting the same language.

Theorem 2. *For each* QSDA $\mathcal{A}$ *there is a unique minimal* QSDA $\mathcal{A}'$ *that accepts the same set of formula trees.*

4 A Partial Order over QSDAs

In the previous section we introduced quantified skinny-tree data automata as an automaton model for expressing universally quantified properties over heap skinny-trees. In this section, we first establish a partial order over the class of QSDAs and then show that QSDAs do not form a complete lattice with respect to this partial order. This motivates us to introduce a subclass of QSDAs called elastic QSDAs which we show, in Section 6, form a complete lattice and can be to compute the semantics of programs. The partial order over EQSDAs with respect to which they form a lattice is the same as the partial order over QSDAs we introduce in this section.

Given a set of pointer variables PV and universal variables Y, let $\mathcal{Q}_{\mathcal{F}}$ be the class of all QSDAs over the lattice of data formulas $\mathcal{F}$. Clearly $\mathcal{Q}_{\mathcal{F}}$ is a partially-ordered set where the most natural partial order is the set-inclusion over the language of QSDAs. However, QSDAs are not closed under unions. Thus, a *least upper bound* for a pair of QSDAs does not exist with respect to this partial order. So we consider a new partial-order on QSDAs which allows us to define a least upper bound for every pair of QSDAs.

If we view a QSDA as a mapping from symbolic trees to formulas in $\mathcal{F}$, we can define a new partial-order relation on QSDAs as follows. We say $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if $L_f(\mathcal{A}_1) \subseteq L_f(\mathcal{A}_2)$, which means that for every symbolic tree t if $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$ then $\varphi_1 \sqsubseteq_{\mathcal{F}} \varphi_2$. Note that, whenever $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ implies that $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$. QSDAs, with respect to this partial order, form a lattice. Unfortunately, QSDAs do not form a complete lattice with respect to this above defined partial order (infinite sets of QSDAs may not have least upper-bounds). Consequently, we invent a subclass of QSDAs called elastic QSDAs (or EQSDAs) which we show form a complete lattice with respect to the above defined partial order. We also show that EQSDAs form an abstract domain by establishing an abstraction function and a concretization function between a set of heap skinny-trees and EQSDAs and showing that they form a Galois-connection. Even though QSDAs do not form a complete-lattice, we describe next a sound abstract transformer over QSDAs, a variant of which we use in Section 6 for abstracting the semantics of programs over EQSDAs.

5 Abstract Transformer over QSDAs

In this section we describe an abstract transformer over QSDAs which soundly over-approximates the concrete transformer over heap skinny-trees. We will later use a variant of this transformer when we compute the semantics of programs abstractly over EQSDAs.

Given a QSDA $\mathcal{A}$, the concrete transformer $F^{\natural}$ guesses a pre-state accepted by $\mathcal{A}$ (which involves existential quantification), and then constrains the post-state with respect to this guessed pre-state according to the semantics of the statement. For instance, consider the statement $p_i := p_j$. Given a QSDA accepting a universally quantified property $\forall y_1, \ldots, y_\ell . \psi$, its strongest post-condition with respect to this statement is the formula: $\exists p'_i . \forall y_1, \ldots, y_\ell . \psi[p_i/p'_i] \ \wedge \ p_i = p_j$. Note that, an interpretation of the existentially quantified variable p'_i in a model of this formula gives the location node pointed to by variable p_i in the pre-state, such that the formula $\forall y_1, \ldots, y_\ell . \psi$ was satisfied by the pre-state. However it is not possible to express these precise post-conditions, which are usually of the form $\exists^* \forall^* \psi$, in our automaton model. So we over-approximate these precise post-conditions by a QSDA which semantically moves the existential quantifiers inside the universally quantified prefix – $\forall y_1 \ldots y_\ell . \exists p'_i . \psi[p_i/p'_i] \ \wedge p_i = p_j$. The existential quantifier can now be eliminated using a combination of automata based quantifier elimination, for the structure, and the quantifier elimination procedures for the data-formula lattice $\mathcal{F}$. In the above example, intuitively, the abstract post-condition QSDA guesses a position for the pointer variable p_i for every valuation of the universal variables, such that the valuation tree augmented with this guessed position is accepted by the precondition QSDA. More generally, the abstract transformer computes the most precise post-condition over the language of valuation trees accepted by a QSDA, instead of computing the precise post-condition over the language of heap skinny-trees. In fact, we go beyond valuation trees to formula trees; the abstract transformer evolves the language of formula trees accepted by a QSDA by tracking the precise set of symbolic trees to be accepted in the post-QSDA and their corresponding data formulas.

We assume that the formula lattice $\mathcal{F}$ supports quantifier-elimination. We encourage the reader to keep in mind numerical domains over the theory of integers with constants (0, 1, etc.), addition, and the usual relations (like $<, \leq, =$)

Table 1. Abstract Transformer $F_f^\sharp$ over the language of formula trees. The abstract transformer over QSDAs $F^\sharp(\mathcal{A}) = \mathcal{A}'$ where $\mathcal{A}'$ is the unique minimal QSDA such that $L_f(\mathcal{A}') = (F_f^\sharp)\ L_f(\mathcal{A})$. The predicate *update* and the set *label* are defined below.

Statements	Abstract Transformer $F_f^\sharp$ on a regular language over formula trees
$p_i := \mathit{nil}$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t,\varphi) \in L_f,$ $\mathit{update}(t, p_i := \mathit{nil}, t')\}\big\}$
$p_i := p_j$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = (p_i \to \texttt{data} = p_j \to \texttt{data}) \sqcap$ $\bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t,\varphi) \in L_f,$ $\mathit{update}(t, p_i := p_j, t')\}\big\}$
$p_i := p_j \to \texttt{next}$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t,\varphi) \in L_f,$ $\mathit{update}(t, p_i := p_j \to \texttt{next}, t')\}$ $\bigsqcap\{p_i \to \texttt{data} = v \to \texttt{data} \mid v \in \mathit{label}(t', p_i)\}\big\}$
$p_i \to \texttt{next} := \mathit{nil}$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = \bigsqcup\{\varphi \mid (t,\varphi) \in L_f, \mathit{update}(t, p_i \to \texttt{next} := \mathit{nil}, t')\}\big\}$
$p_i \to \texttt{next} := p_j$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = \bigsqcup\{\varphi \mid (t,\varphi) \in L_f, \mathit{update}(t, p_i \to \texttt{next} := p_j, t')\}\big\}$
$p_i \to \texttt{data} :=$ $\mathit{data\_expr}$	$\lambda L_f.\ \big\{(t,\varphi') \mid \varphi' = \exists d.\big(\varphi[v_1 \to \texttt{data}/d, \ldots, v_\ell \to \texttt{data}/d] \sqcap$ $\bigsqcap\{v \to \texttt{data} = \mathit{data\_expr}[v_1 \to \texttt{data}/d, \ldots, v_\ell \to \texttt{data}/d] \mid v \in V\}\big),$ $V = \{v_1, \ldots, v_\ell\} = \mathit{label}(t, p_i), (t,\varphi) \in L_f\big\}$
$\texttt{assume}\ \psi_{struct}$	$\lambda L_f.\ \big\{(t',\varphi') \mid (t',\varphi') \in L_f,\ t' \models \psi_{struct}\big\}$
$\texttt{assume}\ \psi_{data}$	$\lambda L_f.\ \big\{(t',\varphi') \mid \varphi' = \varphi \sqcap \psi_{data}, (t',\varphi) \in L_f\big\}$
$\texttt{new}\ p_i$	$\lambda L_f.\big\{(t',\varphi') \mid \varphi' = (y \to \texttt{data} = p_i \to \texttt{data}) \sqcap$ $\bigsqcup\{\exists d_1 d_2.\varphi[p_i \to \texttt{data}/d_1, y \to \texttt{data}/d_2] \mid (t,\varphi) \in L_f,$ $\mathit{update}(t, \texttt{new}^{\{y\}}\ p_i, t')\},\ \ y \in Y\big\}$ $\bigcup\ \big\{(t',\varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t,\varphi) \in L_f,$ $\mathit{update}(t, \texttt{new}^{\{-\}}\ p_i, t')\}\big\}$

as an example of the formula lattice. Table 1[1] gives the abstract transformer $F_f^\sharp$ which takes a regular language over formula trees L_f and gives, as output, a set of formula trees. We know from Theorem 2 that for any regular set of formula trees there exists a unique minimal QSDA that accepts it. We show below (see Lemma 2) that for a QSDA $\mathcal{A}$, the language over formula trees given by $(F_f^\sharp)\ L_f(\mathcal{A})$ is regular. Hence, we can define the abstract transformer $F^\sharp$ as $F^\sharp = \lambda\mathcal{A}.\mathcal{A}'$ where $\mathcal{A}'$ is the unique minimal QSDA such that $L_f(\mathcal{A}') = (F_f^\sharp)\ L_f(\mathcal{A})$.

In Table 1, $\mathit{label}(t, p_i)$ is the set of pointer and universal variables which label the same node in t as variable p_i. The predicate $\mathit{update}(t, \mathit{stmt}, t')$ is true if symbolic trees t and t' are related such that the execution of statement *stmt* updates precisely the symbolic tree t to t'. As an example, the abstract transformer for the statement $p_i := \mathit{nil}$ in the first row of Table 1 states that the post-QSDA maps the symbolic tree t' to the data-formula φ' where φ' is the join of all formulas of the form $\exists d.\varphi[p_i \to \texttt{data}/d]$ where φ is the data-formula associated with symbolic tree t in the pre-QSDA such that $\mathit{update}(t, p_i := \mathit{nil}, t')$ is true.

[1] The abstract transformer defined in Table 1 assumes that there are no memory errors in the program. It can be extended to handle memory errors.

We now briefly describe the predicate $update(t, \mathtt{new}^{\{\hat{y}\}}\ p_i, t')$, where $\hat{y} \in Y \cup \{-\}$, which is used in the definition of the transformer for the `new` statement and is slightly more involved. The statement `new` p_i allocates a new memory location. After the execution of this statement, pointer p_i points to this allocated node. Besides, the universal variables also need to valuate over this new node apart from the valuations over the previously existing locations in the heap. The superscript $\{y\}$ in the predicate $update(t, \mathtt{new}^{\{y\}}\ p_i, t')$ tracks the case when variable $y \in Y$ valuates over the newly allocated node (analogously, the superscript {-} tracks the case when no universal variable valuates over the newly allocated node). Hence, if $update(t, \mathtt{new}^{\{y\}}\ p_i, t')$ holds true then the symbolic trees t and t' agree on the locations pointed to by all variables except p_i and the universal variable y; both these variables point, in t', to a new location v which is not in t and a new edge exists in t' from the root to v.

An important point to note is that the abstract transformer for the statement $p_i \rightarrow \mathtt{next}$ (i.e., the predicate $update(t, p_i \rightarrow \mathtt{next} := p_j, t')$) assumes that the program does not introduce cycles in the heap configurations.

From the construction in Table 1 it can be observed that given a language of valuation trees obtained uniquely from a language of formula trees, $F_f^\sharp$ applies the most-precise concrete transformer on each valuation tree in the language, and then constructs the smallest regular language of valuation trees (or equivalently formula trees) which approximates this set. As we have already discussed, the abstract transformer by reasoning over valuation/formula trees (and not heap skinny-trees) leads to a loss in precision. To regain some of the lost precision, we define a function *Strengthen* which takes a formula language L_f and finds a smaller language over formula trees, which accepts the same set of heap trees. Here $t \downharpoonleft_y$ stands for a $\Pi \backslash \{y\}$ -labeled tree which agrees with t on the locations pointed to by all variables except y.

$$\begin{aligned} \mathit{Strengthen} = \lambda y. \lambda L_f. \big\{ (t', \varphi') \mid \varphi' = \varphi'' \sqcap \phi,\ (t', \varphi'') \in L_f, \\ \phi = \bigsqcap \{ \exists d. \varphi[y \rightarrow \mathtt{data}/d] \mid (t, \varphi) \in L_f, t \downharpoonleft_y = t' \downharpoonleft_y \} \big\} \end{aligned}$$

We now reason about the soundness of the operator *Strengthen*. Fix a $y \in Y$. Consider a QSDA $\mathcal{A}$ with a language over formula trees L_f and consider all symbolic trees t such that $t \downharpoonleft_y = t' \downharpoonleft_y$. This implies that the trees t have the pointer variables pointing to the same positions as t' and have the same valuations for variables in $Y \backslash \{y\}$. Since our automaton model has a universal semantics, any heap tree accepted by $\mathcal{A}$ should satisfy the data formulas annotated at the final states reached for every valuation of the universal variables. If we look at a fixed valuation for variables in $Y \backslash \{y\}$ (which is same as that in t') and different valuations for y, any heap tree accepted should satisfy the formula $\exists d. \varphi[y \rightarrow \mathtt{data}/d]$ for all such $(t, \varphi) \in L_f$. Hence the *Strengthen* operator can safely strengthen the formula φ'' associated with the symbolic tree t' to $\varphi'' \sqcap \phi$. It can be shown that for a given universal variable y and a regular language L_f, the language over formula trees (*Strengthen*) y L_f is regular. In fact, the QSDA accepting the language (*Strengthen*) y $L_f(\mathcal{A})$ for a QSDA $\mathcal{A}$ can be easily constructed. The abstract transformer $F_f^\sharp$ can thus be soundly strengthened by an application of *Strengthen* at each step, for each variable $y \in Y$.

It is clear that the abstract transformer $F_f^\sharp$ in Table 1 as well as the function *Strengthen* are monotonic. We now show that the language over formula trees given by $(F_f^\sharp)L_f(\mathcal{A})$ is a regular language for any QSDA $\mathcal{A}$. This helps us to construct the abstract transformer $F^\sharp : \mathcal{Q}_\mathcal{F} \rightarrow \mathcal{Q}_\mathcal{F}$. Finally, we show that this abstract transformer is a sound approximation of the concrete transformer $F^\natural$.

Lemma 1. *The abstract transformer $F_f^\sharp$ is sound with respect to the concrete semantics.*

Lemma 2. *For a* QSDA *$\mathcal{A}$, the language $(F_f^\sharp)$ $L_f(\mathcal{A})$ over formula trees is regular.*

From Lemma 2 and Theorem 2, it follows that there exists a QSDA $\mathcal{A}'$ such that $\mathcal{A}' = (F^\sharp)\mathcal{A}$. The monotonicity of $F^\sharp$, with respect to the partial order defined in Section 4 over QSDAs, follows from the monotonicity of $F_f^\sharp$. The soundness of $F^\sharp$ can be stated as the following theorem.

Theorem 3. *The abstract transformer $F^\sharp$ is sound with respect to the concrete transformer $F^\natural$.*

Hence $F^\sharp$ is both monotonic, and sound with respect to the concrete transformer $F^\natural$. In the next section we introduce elastic QSDAs, a subclass of QSDAs, which form an abstract domain and we use the above defined transformer $F^\sharp$ over QSDAs to define an abstract transformer over elastic QSDAs. Note that the abstract transformer $F^\sharp$, in general, might require a powerset construction over the input QSDA, very similar to the procedure for determinizing a tree automaton. Hence the worst-case complexity of the abstract transformer is exponential in the size of the QSDA. However our experiments show that this worst-case is not achieved for most programs in practice.

6 Elastic Quantified Skinny-Tree Data Automata

As we saw in Section 4, a least upper bound might not exist for an infinite set of QSDAs. Therefore, we identify a sub-class of QSDAs called elastic quantified skinny-tree data automata (EQSDAs) such that elastic QSDAs form a complete lattice and provide a mechanism to compute the abstract semantics of programs.

Let us denote the symbol $(b, -) \in \Pi$ indicating that a position does not contain any variable by $\underline{b}$. A QSDA $A = (Q, \Pi, \Delta, \mathcal{T}, f)$ where $\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ is called elastic if each transition on $\underline{b}$ in Δ_1 is a self loop i.e. $\Delta_1(q_1, \underline{b}) = q_2$ implies $q_1 = q_2$.

We first show that the number of states in a minimal EQSDA is bounded for a fixed set PV and Y. Consider all skinny-trees where a blank symbol $\underline{b}$ occurs only at branching points. Since the number of branching points is bounded and since every variable can occur only once, there are only a bounded number of such trees. Consider any minimal EQSDA. Consider all states that are part of the run of the EQSDA on the trees of the kind above. Clearly, there are only a bounded number of states in this set. Now, we argue that on *any* tree, the run on that tree can only use these states. For any tree t, consider the tree t' obtained by removing the nodes of degree one marked by blank. The run on tree t will label common states of t and t' identically, and the nodes that are removed will be labeled by the state of its child, since blank transitions cannot cause state-change. Since in any minimal automaton, for any state, there must be some tree that uses this state, we know that the number of state is bounded.

We next show the following result that every QSDA $\mathcal{A}$ can be *most precisely over-approximated* by a language of valuation trees (or equivalently formula trees) that can be accepted by an EQSDA $\mathcal{A}_{\text{el}}$. We will refer to this construction, which we outline below, as *elastification.* This result is an extension of the unique over-approximation result for quantified data automata over words [10]. Using this result, we can show that elastic QSDAs form a complete lattice and there exists a Galois-connection $\langle \alpha, \gamma \rangle$ between a set of heap skinny trees and EQSDAs. This lets us define an abstract transformer over the abstract domain EQSDAs such that the semantics of a program can be computed over EQSDAs in a sound manner.

Let $\mathcal{A} = (Q, \Pi, \Delta, \mathcal{T}, f)$ be a QSDA such that $\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ and for a state q let $R_{\underline{b}}(q) := \{q' \mid q' = q \text{ or } \exists q''.q'' \in R_{\underline{b}}(q) \text{ and } \Delta_1(q'', \underline{b}) = q'\}$ be the set of states reachable from q by a (possibly empty) sequence of $\underline{b}$-unary-transitions. For a set $S \subseteq Q$ we let $R_{\underline{b}}(S) := \bigcup_{q \in S} R_{\underline{b}}(q)$.

The set of states of $\mathcal{A}_{\text{el}}$ consists of sets of states of $\mathcal{A}$ that are reachable by the following transition function Δ^{el} (where $\Delta_i(S_1, \ldots, S_i, a)$ denotes the standard extension of the transition function of $\mathcal{A}$ to sets of states):

$$\Delta_0^{\text{el}}(a) = R_{\underline{b}}(\Delta_0(a))$$

$$\Delta_1^{\text{el}}(S, a) = \begin{cases} R_{\underline{b}}(\Delta_1(S, a)) & \text{if } a \neq \underline{b} \\ S & \text{if } a = \underline{b} \text{ and } \Delta_1(q, \underline{b}) \text{ is defined for some } q \in S \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\Delta_i^{\text{el}}(S_1, \ldots, S_i, a) = R_{\underline{b}}(\Delta_i(S_1, \ldots, S_i, a)) \text{ for } i \in [2, k]$$

Note that this construction is similar to the usual powerset construction except that in each step we apply the transition function of $\mathcal{A}$ to the current set of states and take the $\underline{b}$-closure. However, if the input letter is $\underline{b}$ on a unary transition, $\mathcal{A}_{\text{el}}$ loops on the current set if a $\underline{b}$-transition is defined for some state in the set.

It can be argued inductively, starting from the leaf states, that the type for all states in a set is the same. Hence we define the type of a set S as the type of any state in S. The final evaluation formula for a set is the least upper bound of the formulas for the states in the set: $f^{\text{el}}(S) = \bigsqcup_{q \in S} f(q)$. We can now show that $\mathcal{A}_{\text{el}}$ is the *most precise over-approximation* of the language of valuation trees accepted by QSDA $\mathcal{A}$.

Theorem 4. *For every* QSDA $\mathcal{A}$*, the* EQSDA $\mathcal{A}_{el}$ *satisfies* $L_v(\mathcal{A}) \subseteq L_v(\mathcal{A}_{el})$*, and for every* EQSDA $\mathcal{B}$ *such that* $L_v(\mathcal{A}) \subseteq L_v(\mathcal{B})$*,* $L_v(\mathcal{A}_{el}) \subseteq L_v(\mathcal{B})$ *holds.*

The proof of Theorem 4 is similar to the proof of a similar theorem in [10] for the case of words. The above theorem can also be stated over a language of formula trees in the same way, that $\mathcal{A}_{\text{el}}$ is the most precise over-approximation of the language of formula trees accepted by QSDA $\mathcal{A}$.

We can now show that EQSDAs form a complete lattice $({\mathcal{Q}_{\mathcal{F}}}^{el}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The partial order on EQSDAs is the same as the partial order on QSDAs. For EQSDAs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if $L_f(\mathcal{A}_1) \subseteq L_f(\mathcal{A}_2)$, meaning that for every symbolic tree t if $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$ then $\varphi_1 \sqsubseteq_{\mathcal{F}} \varphi_2$. Given EQSDAs $\mathcal{A}_1$ and $\mathcal{A}_2$ and a symbolic tree t such that $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$, the meet $\mathcal{A}_1 \sqcap \mathcal{A}_2$ is the EQSDA that maps t to the unique formula $\varphi_1 \sqcap_{\mathcal{F}} \varphi_2$, and can be realized using a product construction. The meet for EQSDAs, $\mathcal{A}_1 \sqcup \mathcal{A}_2$, is obtained by constructing a QSDA which maps the symbolic tree t to the formula $\varphi_1 \sqcup_{\mathcal{F}} \varphi_2$ followed by its unique elastification to obtain an EQSDA. We can also similarly compute $\sqcup$ and $\sqcap$ for an infinite number of EQSDAs— we build a product automaton, which can potentially have infinitely many states, but because of the restriction that these are EQSDAs, we can show that the number of states of this product automaton is also bounded as above.

We can now view the space of EQSDAs as an abstraction over sets of heap skinny trees. Let us define an abstraction function $\alpha : \mathcal{H} \to {\mathcal{Q}_{\mathcal{F}}}^{el}$ and a concretization function $\gamma : {\mathcal{Q}_{\mathcal{F}}}^{el} \to \mathcal{H}$ such that $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ form a Galois-connection. Note that, abstract interpretation [7] requires that the abstraction function α maps a concrete element (a language of heap skinny-trees) to a unique element in the abstract domain and that α be surjective; similarly γ should be an injective function. Also note that given a regular language of heap skinny-trees there might be several QSDAs (and thus EQSDAs) accepting that language. In such a case defining a surjective function α is not possible. Therefore, we first restrict ourselves to a set of EQSDAs in ${\mathcal{Q}_{\mathcal{F}}}^{el}$ where each EQSDA accepts a different language. Under this assumption, we define an α and a γ as follows: for a set of heap configurations H, $\alpha(H) = \bigsqcap\{\mathcal{A} \mid H \subseteq L(\mathcal{A})\}$ and $\gamma(\mathcal{A}) = L(\mathcal{A})$. Note that both α and γ are order-preserving; α is surjective and γ is an injective function. Also for a set of heap configurations H, $H \subseteq \gamma(\alpha(H))$ and for an EQSDA $\mathcal{A}$, $\mathcal{A} = \alpha(\gamma(\mathcal{A}))$. Hence $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ forms a Galois-connection.

Theorem 5. *Let $(\mathcal{H}, \subseteq)$ be the class of sets of heap skinny-trees and $({\mathcal{Q}_{\mathcal{F}}}^{el}, \sqsubseteq)$ be the class of* EQSDA*s (accepting pairwise inequivalent languages) over data formulas $\mathcal{F}$, then $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ forms a Galois-connection.*

Let us define the abstract transformer over EQSDAs as $F^{\sharp}_{el} : {\mathcal{Q}_{\mathcal{F}}}^{el} \to {\mathcal{Q}_{\mathcal{F}}}^{el} = F_{el} \circ F^{\sharp}$ where F_{el} is the *elastification* operator which returns the most precise EQSDA over-approximating a language of valuation trees accepted by a QSDA. The soundness of $F^{\sharp}_{el}$ follows from the soundness of $F^{\sharp}$ (and the fact that F_{el} is extensive, i.e., $F_{el}(\mathcal{A}) \sqsupseteq \mathcal{A}$). Similarly its monotonicity follows from the monotonicity of $F^{\sharp}$ and the monotonicity of F_{el}. The semantics of a program can be thus computed over the abstract domain ${\mathcal{Q}_{\mathcal{F}}}^{el}$ as the least fix-point of a set of equations of the form $\psi = F^{\sharp}_{el}(\psi)$. Since the number of states in an EQSDA is bounded (for a given set of program variables PV and universal variables Y), this least fix-point computation terminates (modulo the convergence of the data formulas in the formula lattice $\mathcal{F}$ in which case termination can be achieved by defining a suitable widening operator on the data formula lattice).

6.1 From EQSDAs to a Decidable Fragment of STRAND

In this section we show that EQSDAs can be converted to formulas that fall in a decidable fragment of first order logic, in particular the decidable fragment of Strand over lists. Hence, once the abstract semantics has been computed over EQSDAs, the invariants expressed by the EQSDAs can be used to validate assertions in the program that are also written using the decidable sublogic of Strand over lists. We assume that the assertions in our programs express quantified properties over *disjoint* lists, like sortedness of lists, etc. and properties relying on mutual sharing or aliasing of list-structures are not allowed.

Given an EQSDA $\mathcal{A}$ and for every pointer variable p, we construct a QSDA over words that are projections of trees accepted by $\mathcal{A}$ and where the first node is p. A key property in the decidable fragment of Strand is that universal quantification is not permitted to be over elements that are only a bounded distance away from each other. In other words universally quantified variables are only allowed to be related by elastic relations. As a result, we can safely elastify the constructed QSDA over words and obtain an EQSDA over words expressing quantified properties in the decidable sublogic of Strand. [10] details the translation from an EQSDA over words to a quantified formula in the decidable fragment of Strand over lists. The formula, thus obtained, can be used to validate assertions in the program and thus prove the program correct.

7 Experimental Evaluation

We implemented the abstract domain over EQSDAs presented in this paper, and evaluated it on several list-manipulating programs. We now first present the implementation details followed by our experimental results. Our prototype implementation along with the experimental results and programs can be found at `http://web.engr.illinois.edu/~garg11/qsdas.html`.

Implementation Details. Given a program P we compute the abstract semantics of the program over the abstract domain ${\mathcal{Q}_{\mathcal{F}}}^{el}$ over EQSDAs. A program is a sequence of statements as defined by the grammar in Figure 1. In addition to those statements, a program is also annotated with a pre-condition and a bunch of assertions. The pre-condition formulas belong to the decidable fragment of Strand over lists and can express quantified properties over disjoint lists (aliasing of two list-structures is not allowed), like sortedness of lists, etc. Given a pre-condition formula φ, we construct the EQSDA which accepts all the heap skinny-trees which satisfy φ. This EQSDA precisely captures the set of initial configurations of the program. Starting from these configurations we compute the abstract semantics of the program over ${\mathcal{Q}_{\mathcal{F}}}^{el}$. The assert statements in the program are ignored during the fix-point computation. Once the convergence of the fix-point has been achieved, the EQSDAs can be converted back into

Table 2. Experimental results. Property checked — List: the return pointer points to a list; Init: the list is properly initialized with some key; Max: returned value is the maximum of all data values in the list; Gek: the list (or some parts of the list) have data values greater than or equal to a key k; Sort: the list is sorted; Last: returned pointer is the last element of the list; Empty: the returned list is empty.

Programs	#PV	#Y	#DV	Property checked	#Iter	Max. size of QSDA	Time (s)
INIT	2	1	1	Init, List	4	19	0.0
ADD-HEAD	2	1	1	Init, List	-	11	0.1
ADD-TAIL	3	1	1	Init, List	4	29	0.1
DELETE-HEAD	2	1	1	Init, List	-	10	0.0
DELETE-TAIL	4	1	1	Init, List	5	51	0.5
MAX	2	1	1	Max, List	4	19	0.1
CLONE	4	1	1	Init, List	4	44	0.7
FOLD-CLONE	5	1	1	Init, List	5	57	3.2
COPY-GE5	4	1	0	Gek, List	9	53	2.6
FOLD-SPLIT	3	1	1	Gek, List	4	33	0.3
CONCAT	4	1	1	Init, List	5	44	0.7
SORTED-FIND	2	2	2	Sort, List	5	38	0.3
SORTED-INSERT	4	2	1	Sort, List	6	163	5.8
BUBBLE-SORT	4	2	1	Sort, List	5/18	191	42.8
SORTED-REVERSE	3	2	0	Sort, List	5	43	1.5
EXPRESSOS-LOOKUP-PREV	3	2	1	Sort, List	6	73	2.2
GSLIST-APPEND	4	0	1	List	8	3	0.0
GSLIST-PREPEND	2	0	1	List	-	3	0.0
GSLIST-LAST	3	0	0	Last, List	3	7	0.0
GSLIST-FREE	3	0	0	Empty, List	1	3	0.0
GSLIST-POSITION	4	0	0	List	3	13	0.0
GSLIST-REVERSE	3	0	0	List	3	5	0.0
GSLIST-CUSTOM-FIND	3	1	1	Gek, List	4	29	0.1
GSLIST-NTH	3	0	1	List	3	7	0.0
GSLIST-REMOVE	4	0	1	List	4	10	0.0
GSLIST-REMOVE-LINK	5	0	0	List	4	16	0.0
GSLIST-REMOVE-ALL	5	1	1	Gek, List	5	51	0.6
GSLIST-INSERT-SORTED	5	2	1	Sort, List	6	279	27.4

decidable Strand formula over lists (as described in Section 6.1) and the Strand decision procedure can be used for validating the assertions.

We recall that the abstract transformer $F^{\sharp}_{el}$ is a function composition of the abstract transformer $F^{\sharp}$ over QSDAs and the unique elastification operator F_{el}. So that we are as precise as possible, for every statement in the program we apply the more precise transformer $F^{\sharp}$ (and not $F^{\sharp}_{el}$). However, we apply the elastification operator F_{el} at the header of loops before the join to ensure convergence of the computation of the abstract semantics. The intermediate semantic facts (QSDAs) in our analysis are thus not necessarily elastic.

Our abstract domains are parameterized by a quantifier-free domain $\mathcal{F}$ over the data formulas. In our experiments, we instantiate $\mathcal{F}$ with the octagon abstract domain [23] from the Apron library [16]. It is sufficient to capture the pre/post-conditions and the invariants of all our programs.

Experimental Results. We evaluate our abstract domain on a suite of list-manipulating programs (see Table 2). For every program we report the number of pointer variables (PV), the number of universal variables (Y), the number of data variables (DV) and the property being checked for the program. We also report the number of iterations required for the fixed-point to converge, the maximum size of the intermediate QSDAs and finally the time taken, in seconds, to analyze the programs.

The names of the programs in Table 2 are self-descriptive, and we only describe some of them. The program COPY-GE5, from [5], copies all those entries from a list whose data value is greater than or equal to 5. Similarly, the program FOLD-SPLIT [5] splits a list into two lists – one which has entries whose data values are greater than or equal to a key k and the other list with entries whose data value is less than k. The program EXPRESSOS-LOOKUP-PREV is a method from the module cachePage in a verified-for-security platform for mobile applications [19]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. This method returns the correct position in the cache at which a disc page could be inserted. The programs in the second part of the table are various methods adapted from the Glib list library which comes with the GTK+ toolkit and the Gnome desktop environment. The program GSLIST-CUSTOM-FIND finds the first node in the list with a data value greater or equal to k and GSLIST-REMOVE-ALL removes all elements from the list whose data value is greater or equal to k. The programs GSLIST-INSERT-SORTED and SORTED-INSERT insert a key into a sorted list.

All experiments were completed on an Intel Core i5 CPU at 2.4GHz with 6Gb of RAM. The number of iterations is left blank for programs which do not have loops. BUBBLE-SORT program converges on a fix-point after 18 iterations of the inner loop and 5 iterations of the outer loop. The size of the intermediate QSDAs depends on the number of universal variables and the number of pointer variables and largely governs the time taken for the analysis of the programs. For all programs, our prototype implementation computes their abstract semantics in reasonable time. Moreover we manually verified that the final EQSDAs in all the programs were sufficient for proving them correct (this validity check for assertions can be mechanized in the future). The results show that the abstract domain we propose in this paper is reasonably efficient and powerful enough to prove a large class of programs manipulating singly-linked list structures.

Acknowledgements. This work is partially supported by NSF CAREER award #0747041.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012)

2. Alur, R., Černý, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: POPL, pp. 599–610 (2011)
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
4. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
5. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI, pp. 578–589 (2011)
6. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
8. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
9. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
10. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning Universally Quantified Invariants of Linear Data Structures. In: CAV (2013) (to appear)
11. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL, pp. 338–350 (2005)
12. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246 (2008)
13. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI, pp. 256–265 (2007)
14. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011)
15. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI, pp. 339–348 (2008)
16. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
17. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
18. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622 (2011)
19. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in ExpressOS. In: ASPLOS, pp. 293–304 (2013)
20. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
21. McCloskey, B., Reps, T., Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)

22. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
23. Miné, A.: The octagon abstract domain. In: WCRE, pp. 310–319 (2001)
24. Podelski, A., Wies, T.: Counterexample-guided focus. In: POPL, pp. 249–260 (2010)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
26. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)