Constructing Specialized Shape Analyses for Uniform Change

Tal Lev-Ami[1]*, Mooly Sagiv[1], Neil Immerman[2]**, and Thomas Reps[3]***

[1] School of Computer Science, Tel Aviv University, {tla,msagiv}@post.tau.ac.il
[2] Department of Computer Science, UMass, Amherst, immerman@cs.umass.edu
[3] Computer Science Department, University of Wisconsin, Madison, reps@cs.wisc.edu

Abstract. This paper is concerned with one of the basic problems in abstract interpretation, namely, for a given abstraction and a given set of concrete transformers (that express the concrete semantics of a program), how does one create the associated abstract transformers? We develop a new methodology for addressing this problem, based on a syntactically restricted language for expressing concrete transformers. We use this methodology to produce best abstract transformers for abstractions of many important data structures.

1 Introduction

Abstraction and abstract interpretation [1] are key tools for automatically verifying both hardware and software systems. This paper is concerned with one of the basic problems in abstract interpretation, namely, for a given abstraction and a given set of concrete transformers (that express the concrete semantics of a program), how does one create the associated abstract transformers? We develop a new methodology for addressing this problem, based on a syntactically restricted language for expressing concrete transformers. Of particular interest is that—by employing previous results from dynamic algorithms and dynamic descriptive complexity [2]—our methods allow precise reachability information to be maintained for abstractions of data structures. We use this methodology to produce best abstract transformers for abstractions of many important data structures.

Shape Analysis, Canonical Abstraction, and Dynamic Descriptive Complexity. While our approach is quite general, the main application is to shape analysis (i.e., analysis of linked data structures) and to analyses based on canonical abstraction—the family of abstractions introduced by Sagiv, Reps, and Wilhelm [3] for analyzing programs that use dynamic data structures, including allocation and deallocation of memory cells and destructive updates of pointer-valued fields. In this approach, data structures are modeled using (3-valued) logical structures. Each element of the universe of the structure represents either a single memory cell, or, if the element is a *summary element*, it represents a set of memory cells.

The analysis simulates the program step-by-step, updating the structures appropriately, mimicking (i.e., approximating soundly) the semantics of program statements. When a fixpoint is reached, the resulting set of structures is a finite summary of relevant

* Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.
** Supported by NSF grants CCF-0514621 and CCF-0541018.
*** Supported by NSF grants CCF-0540955 and CCF-0524051.

properties of the data structures built by the program. Note that any resulting properties of the set of structures are thus proven to hold: they necessarily hold on all runs of the program. This analysis framework has been implemented in the TVLA system. (The acronym stands for **T**hree-**V**alued **L**ogic **A**nalyzer.)

A key technical difficulty concerns the summary elements. They are needed so that the unbounded-size set of unbounded-size concrete data structures that can arise are always abstracted to a finite set of finite-size logical structures, which guarantees that the analysis always reaches a fixpoint. The problem caused by summary nodes is that some relations between cells in memory can be true for some elements represented by a summary node and false for others. Hence a truth value of "$\frac{1}{2}$" is introduced, and the framework is based on 3-valued logic [3]. As the analysis propagates 3-valued structures, however, there is a tendency for logical values of $\frac{1}{2}$, i.e., "don't know", to increase, which limits the quality of information that the analysis can provide.

A good way to combat this problem is to maintain extra, auxiliary relations in the logical structures [3, 4]. The same approach is used in dynamic descriptive complexity, although the motivation is completely different:

– In dynamic descriptive complexity, we work with objects that undergo a series of inserts, deletes, changes, and queries; with each query, the goal is to return the answer with respect to the current object. The fundamental issue in dynamic descriptive complexity is one of *efficiency*: "What auxiliary information should be maintained to answer the query *quickly*?" The goal of maintaining extra information is to avoid recomputing each answer from scratch.
– In static analysis based on 3-valued logic, the issue is not so much to save computation time, but instead to preserve high-quality information, i.e., definite truth values—"0"s and "1"s, rather than "$\frac{1}{2}$"s—whenever possible.

A second key technical difficulty concerns reachability information, which is needed to express connectivity and separation properties of data structures. There has been extensive work in dynamic descriptive complexity on how to efficiently maintain reachability information. For example, Dong and Su showed that for acyclic graphs reachability may be maintained by first-order formulas [5]. Of particular interest to us is the result of Hesse that reachability for (not-necessarily acyclic) functional graphs can be maintained by quantifier-free formulas [6].

Our New Methodology. As explained above, TVLA maintains abstract (3-valued) structures, $\mathcal{A}$, that represent sets of concrete (2-valued) structures, $\gamma(\mathcal{A})$. We say that an abstract structure, $\mathcal{A}$, is **feasible** iff $\gamma(\mathcal{A}) \neq \emptyset$. Let β be the abstraction operator on individual concrete structures, i.e., $\beta(\mathcal{C})$ is the abstract representation of $\mathcal{C}$, so β and γ are (approximate) inverse operations (adjoined functions).

For each program statement, st, TVLA has an update formula τ_{st} so that on any concrete structure, $\mathcal{C}$, $\tau_{st}(\mathcal{C})$ is the concrete structure produced by executing statement st. Furthermore, the update formula is always **safe** on abstract structures, meaning that $\tau_{st}(\gamma(\mathcal{A})) \subseteq \gamma(\tau_{st}(\mathcal{A}))$.

Given an abstraction, the gold standard of abstract transformers is called the **best transformer** [1], and satisfies the property, $bt_{st}(\mathcal{A}) \quad = \quad \left\{\beta(\tau_{st}(\mathcal{C})) \;\middle|\; \mathcal{C} \in \gamma(\mathcal{A})\right\}$. However, because $\gamma(\mathcal{A})$ may be infinite, the equation above does not provide an algorithm for computing the best transformer.

TVLA employs heuristics to efficiently compute a safe transformer that is not necessarily the best transformer. In this paper, we introduce a syntactic condition called **monadic uniform** with the following property (see also Thm. 11):

Main Theorem: *If the update formulas for a data structure are monadic uniform and we have an algorithm that given an abstract structure, $\mathcal{A}$, decides whether $\mathcal{A}$ is feasible, then we can automatically compute the best transformers for the operations on the data structure.*

We then show that our main theorem applies to many important situations:

– We use and modify known results from dynamic descriptive complexity to create monadic-uniform update formulas for many important classes of data structures, including linked lists, cyclic linked lists, doubly-linked lists, cyclic doubly-linked lists, trees, shared trees, directed graphs with no undirected cycles, and also some of the above data structures when arbitrary unary relations and an ordering relation are included.
– We also present efficient feasibility algorithms for most of the above. Thus, for these data structures we can implement best abstract transformers automatically.

Our vision is to build specialized shape analyses for many of the available programs and observed properties. This paper is an important step in this direction because it shows that it is possible to build — in a systematic manner — specialized shape analyses with good theoretical properties for many important data structures.

Predicate Abstraction. Our results are not limited to the TVLA context; in particular, they provide a way to improve the predicate-abstraction method given by Rakamaric et al. [7]. Their linked-list abstraction uses the relation $between(x, y, z)$ to capture whether there is a path from x to z through y. Rakamaric et al. give a complete decision procedure for checking feasibility of a given abstract state, but left open the question of how to handle transformers in the most-precise way. Our methodology solves this problem: we can use the quantifier-free update formulas given by Hesse [6] to build best transformers for this abstraction. For example, to compute the abstract transformer for the addition/removal of an edge we would: (1) extend the vocabulary with a constant capturing the current target of the edge; (2) replace each abstract state with the set of states that provide all possible interpretations to the predicates involving the new constant; (3) use the Rakamaric et al. decision procedure to remove the infeasible abstract states; (4) for the remaining states, evaluate Hesse's update formulas to get the successor states.

2 Overview

This section is an informal overview of the methodology presented in the paper. We use a simple Java procedure that reverses a singly-linked list specified in Fig. 1 as a running example. We will run reverse on a cyclic singly-linked list. We use a graphical representation of logical structures to depict a store as a graph.

```
Node reverse(Node x){
  [0] Node y = null;
  [1] while (x != null){
  [2]    Node t = x.next;
  [3]    x.next = y;
  [4]    y = x; x = t; }
  [5] return y;  }
```

Fig. 1. The running example.

Fig. 2(a) is an example of a singly linked list with a cycle. Memory cells are represented by the individuals of the structures (the nodes in the graph). Program variables are represented by constants (the text inside the nodes). Pointer fields in a memory cell are represented by binary relations (the edges of the graph, annotated with the relation name). In this case, the `next` field of the list nodes is represented by the n relation, which is a total function. We can add to the structure auxiliary relations defined using FO(TC) (First Order Logic with Transitive Closure) formulas over the core relations. For example, in Fig. 2(b) we use a unary relation $r_{x,n}$ (written below the nodes) to indicate the existence of a path from the node pointed to by x (defined by $r_{x,n}(v) \overset{\text{def}}{=} n^*(x, v)$). The unary relation c_n states that the node is on a cycle of `next` fields (defined by $c_n(v) \overset{\text{def}}{=} n^+(v, v)$).

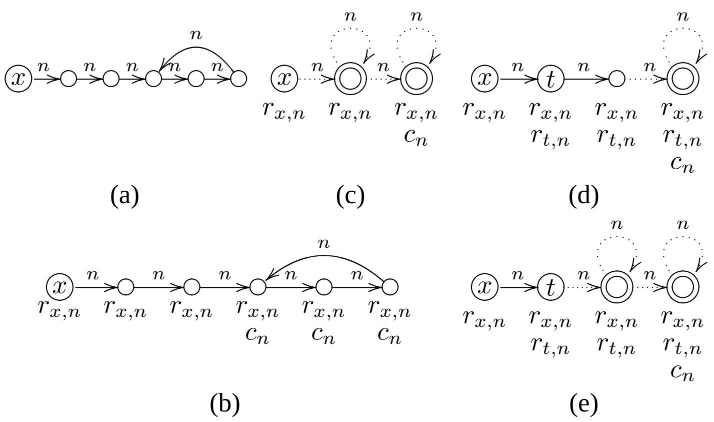

Fig. 2. (a) A concrete structure that represents a singly-linked list with a loop, which is pointed to by x and consists of 6 nodes. (b) The same singly-linked list, this time with auxiliary information. (c) Abstraction of singly-linked lists with loops. (d) & (e) The result of computing the best abstract transformer for the operation `t=x.next` on (c). Note there is also always a concrete node, **null**, with a self-loop (for n) and no other edges. We do not draw this to save space.

In abstract interpretation, we wish to represent a large (possibly infinite) set of stores using a finite set of structures; here this is done by collapsing nodes together into "summary nodes" (drawn as double circles). We use three-valued logic with an additional $\frac{1}{2}$ truth value (for binary relations, this is depicted as a dotted edge) to capture the case in which for some of the nodes represented by the summary node the value is true (1) while for others the value is false (0).[4] Fig. 2(c) shows an abstract structure in which constants are untouched and all the nodes with the same values for unary relations are collapsed together. This type of abstraction is called **canonical abstraction** and is guaranteed to result in structures of bounded size for a given vocabulary. The Embedding Theorem of [3] guarantees that if evaluating formulas (using Kleene semantics) on the abstract structure results in a definite value (i.e., 1 or 0), evaluating the formula on any concrete structure it represents will yield the same value. Kleene semantics can be understood as considering $\frac{1}{2}$ to be $\{0, 1\}$, 0 to be $\{0\}$, and 1 to be $\{1\}$ and evaluating pointwise, e.g., $1 \wedge \frac{1}{2} = \frac{1}{2}$, but $0 \wedge \frac{1}{2} = 0$.

Transformers are given for each operation according to the program's operational semantics. Transformers are specified using **guarded commands** with formulas in FO(TC) called update formulas. For example, for the operation `t=x.next` used in line 2 of Fig. 1, we can use a guard $x \neq null \wedge n(x, x_n)$ to (a) ensure that there is no null-dereference, and (b) bind the value of the `next` field of x to a new (temporary) constant x_n. The update formulas are: $t' := x_n$, $x' := x$, $n'(v_1, v_2) := n(v_1, v_2)$, $c'_n(v) = c_n(v)$, $r'_{x,n}(v) := r_{x,n}(v)$, $r'_{t,n}(v) := n^*(t', v)$. The most precise abstract

[4] For readers familiar with [3], we use tight embedding in this paper. Thus, each summary node represents at least two nodes.

transformer would return a set of abstract structures that captures as tightly as possible (for the abstraction in use) the result of applying the transformer on all the concrete structures represented by the original abstract structure. This kind of abstract transformer is called the best abstract transformer [1] and can be theoretically computed by finding all concrete structures represented by an abstract structure (a.k.a. concretization), computing the transformer on each of them, and abstracting the results. However, because the number of concrete structures represented by an abstract structure is unbounded (and potentially infinite), this is not an algorithm. Fig. 2(d) and Fig. 2(e) show the result for `t=x.next` on the structure in Fig. 2(c). The structure in Fig. 2(d) represents the case in which the list before the cycle is of length 2, and the structure in Fig. 2(e) represents the case it is of length 3 or more. Note that simply evaluating the update formulas on the structure in Fig. 2(c) would not have given us this precise result.

We seek a way to compute the same result as the best transformer without resorting to full concretization. One of the key principles of our methodology is to find a **partial concretization** that 1) is computable, 2) returns a finite set of abstract structures that represents the same concrete structures, and 3) for each of these structures the best abstract transformer can be computed by simply evaluating the update formulas. We call the operation of finding such a partial concretization **Focus** after a similar operation in [3]. Focus replaces each structure with a set of structures, representing the same concrete structures, in which the partitioning of the concrete nodes into summary nodes is more fine-grained. This can be achieved by bifurcating summary nodes into two groups: nodes for which an atomic formula holds, and nodes for which it does not hold. We call such a formula a **focus formula**. For example, Fig. 3(a) and (b) show the result of Focus for the focus formula $n(x, v)$ on the structure in Fig. 2(c). The second and third nodes in the lists of Fig. 3(a) and (b) are the result of bifurcating the second node in Fig. 2(c) according to the focus formula. For the second node, the formula holds, and for the third node the formula does not hold. As we can see, this process can result in multiple structures; Fig. 3(a) corresponds to the case in which the original summary node represents two concrete nodes and in Fig. 3(b) the case in which the summary node represents three or more concrete nodes. We can see that in both cases, the second node has been materialized out of the original summary node.

To automate the Focus operation, we propose an algorithm that can compute the partial concretization for a set of focus formulas: the first phase does not understand the intended meaning of the relations; the second phase applies a "feasibility check" supplied by the developer of the abstraction. An algorithm for feasibility checking should return true iff an abstract structure represents at least one concrete structure. Fig. 3(c) and (d) show structures arising in the Focus process that are infeasible. Structure 3(c) is infeasible because the second node must represent at least two nodes and the first node must have a direct edge to both of them, which contradicts that n is a function. Structure 3(d) is infeasible because the self-loop on the second node means that it must both have a self-loop and not have a self-loop. In §5, we provide algorithms for checking feasibility for several abstractions of commonly used data structures. Note that even if we cannot check feasibility for some abstraction (or have only a sound approximation), the resulting transformer is a sound approximation of the best transformer.

The problem with finding the right focus formulas and using Focus for the transformer given for `t=x.next` is that for the computation of $r'_{t,n}$ we require that the evaluation of $n^*(t', v)$ return precise results — in particular; for any element in the cycle, it should return 1. However, this means that all the edges until the cycle must be 1, which means we need to consider all possible lengths for the segment of the list before the cycle. This is not possible. To solve this problem, we need to somehow limit the update formulas. This leads to our second principle, **monadic-uniform update formulas**.

The update formula for $r'_{t,n}$ can be rewritten as $r'_{t,n}(v) := r_{x,n}(v) \wedge (c_n(x) \vee x \neq v)$. If x is on a cycle, t must be on the same cycle; thus, whatever was reachable from x is now also reachable from t. Otherwise, the only node that was reachable from x and is not reachable from t is x itself. Evaluating this updated transformer on the structures in Fig. 3(a) and (b) results in the structures in Fig. 2(d) and Fig. 2(e). Thus, focusing on $n(x, v)$ was enough. This is not a coincidence. We show that if we limit the update formulas to a certain syntactic class (which we call monadic-uniform), we can automatically find the focus formulas needed for the Focus operation, and the result of Focus is guaranteed to be bounded (a function of the size of the original structure).

The process of finding monadic-uniform update formulas is not trivial, especially when trying to update reachability. Fortunately, we can use existing results from the dynamic descriptive complexity [2, 6] and database [5] communities on maintaining reachability when edges are added or removed. A key step in finding such monadic-uniform update formulas is the addition of auxiliary relations, which together with the other relations can be maintained by monadic-uniform update formulas. In §5, we provide monadic-uniform transformers for the abstractions used for many of the analyses done successfully with TVLA.

Our methodology can be summarized as follows:

1. Find an abstraction that captures the properties you want to verify. Describe it within the framework of parameterized shape analysis of [3].
2. Insure that all update formulas are monadic-uniform, adding extra auxiliary relations as needed.
3. Optionally, develop a feasibility check for abstract structures of this (possibly augmented) vocabulary; or, settle for a sound approximation of the best transformer.

The paper presents the necessary algorithms for binding these ingredients together to compute best abstract transformers.

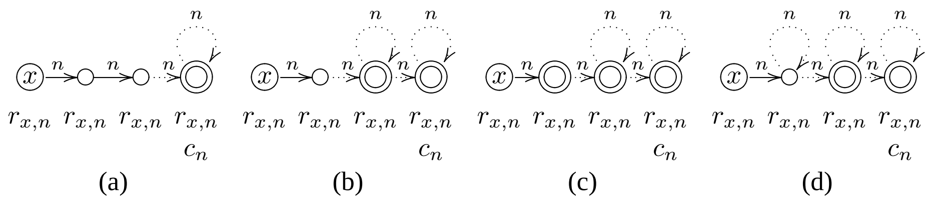

Fig. 3. Some of the structures arising in the process of Focus for the operation `t=x.next` on the structure in Fig. 2(c).

3 Preliminaries

We represent stores as logical structures. This allows us to use logical formulas to define the semantics of statements and abstractions of stores. To simplify the presentation, we describe everything in the context of a specific vocabulary. It should be clear from the description that the formulas are schematic and can be instantiated to the specific program fields and variables.

See [8] for a formal definition of the syntax of FO(TC) formulas. We use the shorthand (when $\varphi_1 \Rightarrow \psi_1, \ldots,$ when $\varphi_k \Rightarrow \psi_k,$ default $\Rightarrow \psi)$ for a sequential case split; i.e., formally it is: $\ldots \vee (\neg\varphi_1 \wedge \ldots \wedge \neg\varphi_{i-1} \wedge \varphi_i \wedge \psi_i) \vee \ldots \vee (\neg\varphi_1 \wedge \ldots \wedge \neg\varphi_k \wedge \psi)$

A 2-valued logical structure is a triple $S = \langle U^S, R^S, C^S \rangle$ of a universe U^S of individuals, a map R^S of relation symbols to truth-valued functions, and a map C^S of constant symbols to individuals. See [8] for a formal definition.

3.1 Programming-Language Statements

Formulas are used to update the store in a standard way as follows:

Definition 1 (Store Updates) *An **update formula** of a relation r of arity k has the form: $r'(v_1, \ldots, v_k) := \varphi_r(v_1, \ldots, v_k)$, where $\varphi_r(v_1, \ldots, v_k)$ is a formula with free variables $v_1, v_2, \ldots v_k$. An **update formula** of a constant c has the form:*
$c' := (\textit{when } \varphi_1 \Rightarrow s_1, \ldots, \textit{when } \varphi_k \Rightarrow s_k, \textit{default } \Rightarrow s_{k+1})$, where the φ_i are closed formulas and the s_i are constant symbols. This is a shorthand for the following formula with one free variable: $\varphi_c(v) \overset{\text{def}}{=} (\ldots, \textit{when } \varphi_i \Rightarrow v = s_i, \ldots, \textit{default } \Rightarrow v = s_{k+1})$ For the special case in which k=0 we simply write $c' := s_1$.

*Every statement st in the programming language is associated with **transformer** τ_{st}, which consists of a **guard formula**, $\textit{guard}_{\tau_{st}}$, and a set of update formulas for each relation and constant symbol in the vocabulary. If the guard formula has free variables, the update formulas can refer to them as constants.*

*Given a 2-valued logical structure, $S = \langle U, R, C \rangle$, the **expansion** of S for τ is the set $\textit{expand}_\tau(S)$ of all the structures $S' = \langle U, R, C' \rangle$ s.t., C' is identical to C except it gives an interpretation to all the free variables of $\textit{guard}_\tau$. We say S' is **expanded** for τ.*

*The application of the transformer τ on a structure $S' \in \textit{expand}_\tau(S)$ is the 2-valued structure $\tau(S') \overset{\text{def}}{=} \langle U, R'', C'' \rangle$, where for every relation symbol r, $R''(r)(\overrightarrow{u}) = [\![\varphi_r(\overrightarrow{u})]\!]^{S'}$, and for every constant symbol c, let $u_c \in U$ be the unique element for which $S', u_c \models \varphi_c$, we have $C''(c) = u_c$. Note that C'' gives an interpretation only to the original constants and not to the free variables of $\textit{guard}_\tau$. The **meaning** of the transformer τ on S is the set $[\![\tau]\!](S) \overset{\text{def}}{=} \{\tau(S') \mid S' \in \textit{expand}(S) \wedge S' \models \textit{guard}_\tau\}$.* $\square$

The free variables in the guard formula allow for the introduction of nondeterminism. These free variables are considered as additional constants by the update formulas. The syntactic form of the update formulas for constants guarantees that for each constant symbol c there is only one u_c for which $S', u_c \models \varphi_c$. Thus, once the free variables have been assigned, the computation of the transformer is deterministic.

st	guard_{st}	**update formulas**
`x = null`	$\mathbf{1}$	$x' := null$
`x = t`	$\mathbf{1}$	$x' := t$
`x = t.sel`	$t \neq null \wedge sel(t, t_{sel})$	$x' := t_{sel}$
`x.sel = y`	$x \neq null$	$\textit{sel}'(v_1, v_2) := (v_1 = x \wedge v_2 = y) \vee (v_1 \neq x \wedge \textit{sel}(v_1, v_2))$
`x == y`	$x = y$	

Table 1. Relation-update formulas that define the semantics of statements that manipulate pointers and pointer-valued fields.

For simplicity, we do not support operations that change the universe. However, because we allow infinite universes, we can easily model the allocation and deallocation of individuals using a designated relation that holds only for allocated individuals (or, if the operational semantics allows, by using a free list).

Table 1 lists the transformers that define the operational semantics of the five kinds of Java-like statements. Here x, t, and y are constants that denote the target of pointer variables `x`, `t`, and `y`, respectively. *sel* is a binary relation that models the pointer field `sel`. We do not specify update-formulas for relations and constants with unchanged values. The guard formulas for statements that access `sel` ensure that no null-dereference has occurred. In case of a field traversal, the guard formula also selects the target of the field using the free variable t_{sel}. Note that program conditions are simply modeled by guard formulas.

Integrity Constraints. We allow restriction of the potential stores that may arise in the program by a finite set of closed formulas called **integrity constraints** and denoted by Σ. We assume that the meaning of every transformer τ **maintains the integrity constraints**, i.e., if $S \models \Sigma$, $S' \in [\![\tau]\!]^S$ a 2-valued structure, then $S' \models \Sigma$.

In the case of pointer fields, we require that every field be a total function. Thus, in particular, the pointer field(s) of $null$ points to $null$.

Auxiliary Information. The most interesting integrity constraints occur as a result of extra relations whose values are derived from other relations. Formally, an **auxiliary** relation r of arity k is defined via a defining formula φ_r with k free variables. This results in the integrity constraint $\forall v_1, \ldots, v_k : r(v_1, \ldots, v_k) \iff \varphi_r$. Thus, every statement must maintain this invariant. Auxiliary information allows us to reduce the complexity of update formulas. Furthermore, it is often the information maintained by auxiliary relations that enables us to compute best abstract transformers.

§2 introduced two types of auxiliary relations, $r_{x,n}$ for reachability from a program variable, and c_n for cyclicity. The interaction between them is used to define a monadic-uniform update formula for traversal of an edge.

3.2 Monadic-Uniform Updates

In this section, we restrict the way the semantics of statements are allowed to be defined to use only formulas of a certain syntactic class. The new stores can differ from the original store in many values but the change should be uniform in the sense defined below. We begin by defining atomic formulas that are essentially unary.

Definition 2 *An atomic formula is **monadic** if it is of the form* $r(c_1, \ldots, c_i, v, c_{i+1}, \ldots, c_{k-1})$ *where* r *is* k*-ary relation and* $c_1, \ldots, c_{k-1}$ *are constant symbols. An* FO(TC) *formula* φ *is **monadic** if all of the atomic formulas appearing in* φ *are monadic or ground.* □

The following formulas are monadic: $r(v, c)$, $v = c$, $r(v)$, $\forall v.r(v, c)$. The following formulas have variables in more than one position, and thus are not monadic: $r(v, v)$, $r(v_1, v_2)$, $v_1 = v_2$. Note that although $r(v, v)$ uses a single variable, it is not monadic.

Next, we define monadic update formulas, which are a restricted case of update formulas in which a tuple is classified by monadic formulas, and for each class, the value of an existing relation is copied.

Definition 3 (Monadic-Uniform Updates) *A **monadic-uniform formula*** $\varphi(v_1, \ldots, v_k)$ *is syntactically equivalent to* $(\ldots, when\ \varphi_i \Rightarrow \psi_i, \ldots, default\ \Rightarrow \psi_l)$ *where the* φ_i *are monadic* FO(TC) *formulas with free variables* $v_1, v_2, \ldots v_k$*, and the* ψ_i *are restricted to either* $\mathbf{1}$*,* $\mathbf{0}$*, or a literal with distinct variables.*

*A **monadic-uniform transformer** is a transformer in which all the update formulas and the guard formula are monadic uniform.* □

All the transformers of Table 1 are constructed to be monadic-uniform transformers (see §5). Monadic-uniform formulas disallow direct interaction between non-monadic relations, e.g., $r(v_1, v_2) \wedge q(v_1, v_2)$ is not monadic-uniform. $r(v, v)$ is not monadic-uniform because it is equivalent to $r(v_1, v_2) \wedge v_1 = v_2$ and captures the interaction between r and equality.

3.3 Canonical Abstraction

In this section, we use 3-valued logic to conservatively represent sets of stores. Formally, we define a lattice of static information where lattice elements are sets of 3-valued structures. A 3-valued structure is similar to a 2-valued structure, except R^S maps to 3-valued truth functions, i.e., whose range is $\{0, 1, \frac{1}{2}\}$. See [8] for a formal definition. We say that the values 0 and 1 are definite values and that $\frac{1}{2}$ is an indefinite value, and define a partial (information) order on truth values as follows $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = \frac{1}{2}$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

Definition 4 *A tight embedding function is a surjective function $f \colon U^S \to U^{S'}$ such that, for every $c \in C$, $C^{S'}(c) = f(C^S(c))$ and for every relation $r \in R$ of arity k, $R^{S'}(r)(u'_1, \ldots, u'_k) = \bigsqcup_{f(u_i)=u'_i, 1 \leq i \leq k} R^S(r)(u_1, \ldots, u_k)$. We say that $S' = f(S)$ and that S' is a* **tight embedding** *of S.* [5]

When the embedding function maps more than one node to some node u, we say that u is a **summary node***. Otherwise, we call the node a* **concrete node***. For summary nodes, $[\![u = u]\!]^{S'} = \frac{1}{2}$. Note that if $C^{S'}(c) = u$ and u is a summary node, only one of the nodes mapped to u equals c, not all of them.*

Canonical embedding*, denoted by β, is the embedding obtained by using unary relation symbols to distinguish between individuals, i.e., two concrete individuals $u_1, u_2 \in U^S$ are mapped to the same individual if and only if they agree on the values of unary relation symbols. For each constant, c, there is an implied unary relation, P_c, true just of c.* $\Box$

According to the **embedding theorem** [3], every formula with a definite value in a structure has the same value in all of the embedded concrete structures.

Canonical abstraction allows us to define the set of stores represented by a 3-valued structure.

Definition 5 *For a 3-valued structure S, $\gamma(S)$ denotes the set of 2-valued structures that S represents, i.e., $\gamma(S) = \{S^\natural \models \Sigma \mid \beta(S^\natural) = S\}$. We say that a structure S is* ***feasible*** *if $\gamma(S) \neq \emptyset$.* $\Box$

The complexity of checking feasibility of a structure comes from the need to satisfy the integrity constraints and because of interactions between auxiliary relations and core relations.

4 Methodology for Developing Computable Transformers

A shape-analysis problem is characterized by a triple of the class of allowed structures, the initial abstraction, and the set of possible atomic operations.

The running example (see Fig. 1) is an instance of the following shape-analysis problem: The class of allowed structures is (possibly cyclic) singly-linked lists. The initial abstraction tracks: pointed to by a program variable (by representing program variables as logical constants), the `next` field (by maintaining a binary relation n),

[5] From now on, whenever we refer to embedding, we mean tight embedding and use the term tight embedding only for emphasis.

reachability from program variables (by unary relations of the form $r_{x,n}(v)$, which indicate that v is reachable from program variable x using the `next` field), and cyclicity (by a unary relation $c_n(v)$, which indicates that v is part of a cycle).

The first step in developing computable best transformers for a shape-analysis problem is to find monadic-uniform transformers for all the operations required. A key step in finding such update formulas is the introduction of additional auxiliary relations that, together with the original relations, can be maintained in a monadic-uniform way.

The main difficulty in maintaining the relations used in the shape-analysis problem for the running example is the maintenance of reachability. Fortunately, we can use (with a small modification to make it monadic-uniform) the DynQF update formulas for transitive closure given by Hesse in [6]. We introduce three auxiliary binary relations. The relation $p_n(v_1, v_2)$ maintains the reflexive transitive closure of the n relation (i.e., existence of a path between v_1 and v_2 using the `next` field). The relation $cut_n(v_1, v_2)$ holds for exactly one edge in each cycle (enforced using appropriate integrity constraints). The relation $pc_n(v_1, v_2)$ (called PathCut by Hesse) maintains the reflexive transitive closure of the un-cut edges.Together, these relations allow us to create monadic-uniform transformers for all the needed operations (see [6] and §5 for more details).

Imperative programs lead to monadic-uniform transformers because they can only change information directly pointed to by variables. The difficulty comes from relations such as reachability in which a local update can cause widespread change. We take advantage of the specific structure of the graphs in each case to build a monadic-uniform transformer for them.

The final step in our methodology is to develop an algorithm for checking the feasibility of an abstract structure of the chosen vocabulary. Here we need to take into account the integrity constraints, including the set of allowed structures and the meaning for all the auxiliary relations.

In §5, we show that, to check feasibility of an abstract structure that can arise in the shape-analysis problem defined above, we can compute a candidate concrete structure s.t. the abstract structure is feasible iff the concrete structure is consistent (i.e., satisfies the integrity constraints) and its β is the original structure. The size of the candidate structure is linear in the size of the original abstract structure. Thus, we can check its feasibility in time polynomial in the size of the original abstract structure.

The rest of the section describes how to compute best transformers for a given shape-analysis problem that has monadic-uniform transformers and a decidable feasibility-checking problem. Proofs can be found in [8].

First, we define the concept of a **focused** structure for a monadic-uniform transformer. For such structures and transformers, the transformer preserves embedding (see Lem. 7).

Definition 6 *We say that S is* ***focused*** *for a τ (denoted by focused$_\tau(S)$) when (1) S is expanded for τ, (2) all the monadic atomic formulas that appear in any update formula of τ or in guard$_\tau$, evaluate to definite truth values in S, and (3) all the constants interpreted by C^S are mapped to concrete nodes.*

We define β_τ to be a canonical embedding function that honors all new constants and monadic atomic formulas appearing in transformer τ. γ_τ is defined analogously to γ but in relation to β_τ. □

The structures in Fig. 3(a) and (b) are focused for `t = x.next` if we map x_n to any concrete node (only when x_n is mapped to the second node of the list will the guard formula hold). For Fig. 2(c), when trying to interpret x_n in a way that will satisfy the guard formula, the only node worth considering is the second node of the list. There are two reasons why such a structure is not focused. First, the second node is a summary node, thus a constant cannot be mapped to it. Second, $n(x, x_n)$, which appears in the guard formula, evaluates to $\frac{1}{2}$. Note that the fact that the structures in Fig. 3(a) and (b) are focused does not mean that all the update formulas evaluate to definite values for all the nodes, e.g., the n relation has several indefinite tuples in resulting structure Fig. 2(e).

For structures that are focused for a transformer τ, we use the canonical embedding function β_τ, and when referring to the feasibility of a focused structure, we mean non-emptiness of γ_τ.

Lemma 7 *Let τ be a monadic-uniform transformer, S be a structure s.t. focused$_\tau(S)$ holds, C be a concrete structure, and f be an embedding function s.t. $f(C) = S$. The following properties hold: (1) $f(\tau(C)) = \tau(S)$, (2) $[\![\mathit{guard}_\tau]\!]^C = [\![\mathit{guard}_\tau]\!]^S$, (3) for every unary relation r and node u we have $[\![r(u)]\!]^{\tau(C)} = [\![r(f(u))]\!]^{\tau(S)}$, and (4) for every constant c, $\tau(S)$ maps c to a concrete node.*

When embedding is preserved, all unary relations are definite, and all the constants are mapped to non-summary nodes, β will return the same value for both updated structures. Cor. 8 entails that a monadic-uniform transformer is actually the best transformer for focused abstract structures.

Corollary 8 *Let τ be a monadic-uniform transformer. If focused$_\tau(S)$ and $f(C) = S$ then $\beta(\tau(C)) = \beta(\tau(S))$*

Cor. 8 suggests a way to compute the best abstract transformer: Given an abstract structure, find a set of feasible focused structures that represent the same concrete structures. Def. 9 makes this notion formal.

Definition 9 *focus$_\tau$ is an operation that given a feasible structure S returns a finite set of structures FS s.t. $\bigcup_{S' \in \gamma(S)} \mathit{expand}_\tau(S') = \bigcup_{F \in FS} \gamma_\tau(F)$ and for every $F \in FS$, F is feasible and focused$_\tau(F)$.* □

We now sketch the algorithm that computes focus$_\tau$. The algorithm systematically replaces each $\frac{1}{2}$ value for monadic formulas by 0 or 1, duplicating structures as necessary. There may be a large but bounded number of such structures. Each candidate structure is checked for feasibility and discarded if infeasible.

Algorithm 10 Given τ, S, compute focus$_\tau(S)$.

```
0.  FS = FS_orig = expand_τ(S)                         // the current set of structures
    MA = the monadic atomic formulas of τ, including the new constants
1.  for each A(v) from MA and F from FS do {
2.      for each node $b \in U^F$ s.t. $[\![A(b)]\!]^F = \frac{1}{2}$ do {   // b must be a summary node
3.          Remove F from FS and replace by $F_{u_1 u_2} : u_j \in \{s, c\}$
                s.t. b is split into $b_0, b_1$, $[\![A(b_i)]\!]^{F_{u_1 u_2}} = i$, and,
                $b_i$ is a summary node in $F_{u_1 u_2}$ iff $u_j = s$. } }
4.  for each structure F, new tuple created, $\bar{t}$, and relation R s.t. $[\![R(\bar{t})]\!]^F = \frac{1}{2}$,
        add structures $F_i : i \in \{0, 1\}$ s.t. $[\![R(\bar{t})]\!]^{F_i} = i$ and $\beta(F_i) \in FS_{orig}$
5.  for each structure F, if $\gamma_t(F) = \emptyset$, remove F from FS
6.  return(FS)
```

Focus can yield a double-exponential number of structures. The maximum number of individuals in a single structure can be exponential in the number of predicates and the number of possible structures is exponential in the number of nodes. From our experience with TVLA, the first blowup — the maximal number of individuals — rarely happens in practice. However, in contrast to TVLA, the use of tight embedding suggests that the second blowup may indeed occur in practice. We are working on ways to remedy the situation, e.g., by moving to non-tight embedding (see [3]).

From the correctness of Alg. 10, our main theorem follows:

Theorem 11. *If S is feasible then we can automatically compute the best transformer:* $bt_\tau(S) \equiv \big\{\beta(\tau(S')) \;\big|\; S' \in \mathit{focus}_\tau(S) \wedge [\![\mathit{guard}_\tau]\!]^{S'} = 1\big\}$

Note that if there is no feasibility check, the methodology still guarantees that we obtain a best transformer, but with respect to a γ that does not force the concrete structures to adhere to the integrity constraints. However, when using this γ, the abstraction is not likely to be strong enough to establish the properties that we desire.

5 Applications

Structures	Vocabulary	Feasibility
Acyclic SLL	p_n, n, PVar	Direct
Acyclic SLL	$r_{x,n}$, n, PVar, Colors	Direct
Cyclic SLL	p_n, pc_n, n, PVar	Direct
Cyclic SLL	$r_{x,n}$, $rc_{x,n}$, n, PVar, Colors	Direct
DLL	p_f, p_b, $c_{f,b}$, $c_{b,f}$, PVar, Colors	Direct/Open
Ordered SLL	$r_{x,n}$, $rc_{x,n}$, n, dle, PVar, $inOrd_{n,dle}$, $inROrd_{n,dle}$	Open
Trees	p, l, r, PVar	Direct
Trees	p, l, r, PVar, Colors	MSO
NUC	p, l, r, $s_{x,y}$, PVar	Direct
NUC	p, l, r, $s_{x,y}$, PVar, Colors	MSO
Shared Trees	p, l, r, PVar	Open

Table 2. Summary of the shape-analysis problems and their feasibility-check status.

This section describes several applications of the methodology described in §4 for computing transformers for different shape-analysis problems. For each problem, we specify the class of allowed structures, the relations we maintain, and, when known, an algorithm for checking feasibility. Further details can be found in [8].

Table 2 summarizes the different shape-analysis problems described in this section and the type of feasibility checks we have for them. For all of these problems, we show monadic-uniform transformers for field manipulations. SLL/DLL stands for Singly/Doubly Linked Lists, and NUC for No Undirected Cycles. PVar stands for Program Variables. A description of each class of structures and the meaning of each relation is given in the appropriate subsection below. Note that for every vocabulary we require a new feasibility-checking algorithm.

Dong and Su [5] show how to update reachability in a general acyclic graph using first-order logic. However, their formulas are not monadic-uniform and it is unclear whether it is possible to make them monadic-uniform.

Relation	Update Formula
`x = y.next`	
guard	$n(y, y_n) \wedge y \neq null \wedge (x = null \vee \bigvee_{z \neq x} r_{z,n}(x))$
x'	y_n
$r'_{x,n}(v)$	$r_{y,n}(v) \wedge y \neq v$
`x.next = null`	
guard	$n(x, x_n) \wedge x \neq null \wedge (x_n = null \vee \bigvee_z (r_{z,n}(x_n) \wedge \neg r_{z,n}(x)))$
$n'(v_1, v_2)$	(when $v_1 = x \Rightarrow v_2 = null$, default $\Rightarrow n(v_1, v_2)$)
$p'_n(v_1, v_2)$	$p_n(v_1, v_2) \wedge \neg(p_n(v_1, x) \wedge p_n(x_n, v_2))$
$r'_{z,n}(v)$	$r_{z,n}(v) \wedge \neg(r_{z,n}(x) \wedge r_{x,n}(v) \wedge x \neq v)$
`x.next = y`	
guard	$x \neq null \wedge \neg r_{y,n}(x) \wedge n(x, null)$
$n'(v_1, v_2)$	(when $v_1 = x \Rightarrow v_2 = y$, default $\Rightarrow n(v_1, v_2)$)
$p'_n(v_1, v_2)$	$p_n(v_1, v_2) \vee (p_n(v_1, x) \wedge p_n(y, v_2))$
$r'_{z,n}(v)$	$r_{z,n}(v) \vee (r_{z,n}(x) \wedge r_{y,n}(v))$

Table 3. Monadic-uniform transformers for acyclic singly-linked lists.

Direct means there is a direct algorithm to check feasibility of an abstract structure. MSO means we can reduce the feasibility check to a satisfiability check of an MSO formula on trees. Open means we are still working on checking feasibility for this problem. We believe that checking feasibility is decidable for all of these problems.

Singly-Linked Lists. The first class of allowed structures we examine is acyclic singly linked lists. The vocabulary includes constants that represent program variables, a functional binary relation n that represents the next field, a unary relation $r_{x,n}$ for each program variable x that represents reachability from x (a.k.a., unary reachability), and a binary relation p_n (path of n) that represents reachability between any two elements. The guard formulas are used to detect null dereferences or the formation of garbage or cycles. Monadic-uniform update formulas can be easily written for all the needed operations.

Table 3 lists the transformers for the field-manipulating operations. Update formulas for unchanged relations are omitted. The update formulas for reachability follow the ones described in [6]. For traversal of a field, we use the free variable y_n of the guard formula to capture the target of the `next` field for y (x_n is used similarly in the removal of an edge).

To check feasibility of a focused abstract structure, we build a single candidate concrete structure s.t. the original structure is feasible iff it is the result of applying β on the candidate structure and the candidate structure satisfies the integrity constraints.

Algorithm 12 *(Checking Feasibility)*
Replace every summary node with two concrete nodes connected by an edge, all incoming edges to the summary node go to the first concrete node, all outgoing edges from the summary nodes start from the second node. Each edge in the abstract structure is translated into a single edge in the concrete structure. We then simply compute β on this structure and return true if it equals the original structure and satisfies the integrity constraints (i.e., n is a total function).

Cyclicity. To handle cyclicity, we use the ideas from [6], which allow for quantifier-free update of reachability in singly-linked lists. The update of [6] is based on the addition of a binary relation, called PathCut, as an auxiliary relation. For every cycle, we call the last edge added to the cycle (i.e., the edge that closed the cycle) a **cut edge**. PathCut indicates reachability over n minus the cut edges. When the cycle is broken, its cut edge is readded to PathCut. The update formula suggested by [6] for removal of an edge is not monadic-uniform. Fortunately, we can easily rewrite that formula to be monadic-uniform.

To analyze programs that manipulate cyclic singly-linked lists, we use a vocabulary similar to that of acyclic singly-linked lists. The additional relations needed to allow updates to be monadic-uniform (and ease feasibility checking) are: cut_n is a binary relation representing the cut edges, pc_n is a binary relation representing PathCut, $rc_{x,n}(v)$ is a unary relation indicating v is reachable from program variable x using pc_n, and $c_n(v)$ is unary relation indicating that v is on a cycle. The resulting abstraction is similar in the distinctions it makes to that of [9]. Because cut_n is needed only to update itself, and the feasibility check can recover the cut edges from pc_n, we can remove cut_n and still compute the best transformer.

We use the DynQF updates by [6] as a basis for monadic-uniform update formulas.

Feasibility checking can be done using the same ideas as for acyclic lists with the necessary changes to support the cut edges.

Trees. To analyze trees using monadic-uniform transformers, we use the following vocabulary: constants represent program variables; two functional binary relations l and r represent the `left` and `right` fields respectively; two new constants x_l and x_r for each program variable x indicate the target of its left and right fields, respectively; a binary relation p represents reachability (existence of a path) between any two elements (using any fields); unary relation $r_{x,sel}$ for each program variable x represents reachability from the sel field of x. The guard formulas verify that each operation maintains treeness.

The key to updating reachability in this case is the observation that between every two nodes there is at most one path. Thus, the paths that should be removed when removing an edge from x to x_l are exactly the ones that would have been added if this edge had been added.

We can either check feasibility by reduction to satisfiability of an MSO formula (similar to the $\hat{\gamma}$ of [10]) on trees or we can check it directly (with lower complexity) by building a single candidate concrete structure in a way similar to singly-linked lists.

No Undirected Cycles. In [11], we introduced a class of structures whose underlying undirected graphs are acyclic (a.k.a. No Undirected Cycles). There we show an abstraction for handling this class of structures and algorithms for computing best abstract transformers for this abstraction. Structures with No Undirected Cycles are acyclic and have the interesting property that each pair of program variables can meet only once (i.e., there is a single shared node reachable from both variables s.t. none of the nodes pointing to that node are reachable from both variables). Furthermore, between any two nodes there is at most one path.

We now define an abstraction similar to [11] and apply our methodology. The vocabulary used for trees in extended with the following constants: For each pair of distinct program variables x and y, we add $s_{x,y}$, which is the unique node in which x and y meet and create sharing (or $null$ if no such node exists).These are used in the guard formulas to detect formation of undirected cycles. We also maintain unary reachability from these constants. We can write a monadic-uniform guard formula using transitive closure that detects the formation of undirected cycles. We can check feasibility of such structures using methods similar to the ones using for trees.

Shared Trees. Shared trees are graphs in which between any two nodes there is at most one (possibly empty) path. A way to visualize shared trees is that from every node looking down the graph you see a tree. Shared trees arise in applicative data structures (e.g., see [12, 13]) and in operating systems and databases performing shadow paging (e.g., see [14]).

We use the same vocabulary as in the case of trees. Updating reachability for this class of structures is done in the same way as in trees, because between any two nodes there is at most one path. Detecting when the shared-trees property has been violated is done by a guard formula when adding an edge. Again, the formula is monadic-uniform but not quantifier-free.

We are working on checking feasibility for shared trees in this vocabulary and believe it is decidable. Because shared trees have unbounded tree width, a direct translation into satisfiability of an MSO formula will not yield decidability.

Uninterpreted Unary Relations. Sets and boolean fields can be added to any of the above shape-analysis problems by introducing uninterpreted unary relations (a.k.a. colors). We allow addition and removal of an element from a set, query for existence of an element in a set, and selection of an arbitrary element from a set. The additional update formulas needed are trivial. Selection is done by using a guard formula with a free variable. The difficulty in checking feasibility when adding colors to a vocabulary, in contrast to the original feasibility-checking problem, comes from the fact that the colors can make distinctions that the original abstraction could not. The binary relations between the now-separate nodes need to be taken into account.

Checking feasibility for singly-linked lists can be done by first checking feasibility ignoring the colors, and then reducing the feasibility for each segment of the list to the Directed Chinese Postman Problem [15], which can be solved in polynomial time. Checking feasibility for trees and structures with No Undirected Cycles, can be done by reduction to MSO.

Other cases. The relations required for analyzing doubly linked lists and ordered lists can also be maintained using monadic-uniform transformers.

We do not have a general feasibility check for any structure over the vocabulary of doubly-linked lists. However, we do know how to check feasibility for all the structures arising in most programs that manipulate doubly-linked lists (e.g., all the example programs of TVLA) because all such structures are only ever small perturbations of well-formed doubly linked lists.

6 Related Work

Specialized Shape Analyzers. Developing specialized shape analysis for commonly used data structures is an active line of research [16, 9, 11, 17]. We are encouraged by the fact that we are able to express all of the above-cited work using our methodology. Moreover, our methodology supports shared trees and the addition of arbitrary colors, which are beyond the scope of existing methods. It should be noted that our current algorithms are more costly. In particular, the ad-hoc algorithm in [11] runs in time essentially linear in the output, which is hard to beat. In the future, we plan to reduce the costs of creating the transformers by: (i) focusing only the necessary parts, (ii) developing more efficient focus algorithms, and (iii) using incrementality to reduce the cost of feasibility checks.

The TVLA System. The results in this paper are inspired by the TVLA system. The TVLA system does not require that update formulas be monadic-uniform. It also allows arbitrary classes of graphs to be used. Also, [18] includes an algorithm for automatically generating update formulae for auxiliary information, which is fully integrated into the system. (§5.4.1 of [19] describes the application of that machinery for an abstraction similar to the one described for cyclic singly-linked lists.) However, the TVLA system does not guarantee that the transformers are the best. Moreover, the system can issue a runtime exception in certain cases when an operation may lead to an infinite number of structures. In this paper, we build specialized shape analyses that can handle many of cases for which TVLA was used. For most of these cases, we can now compute the best abstract transformer. In the future, it may be possible to combine methods like the ones in [18] with our method. For example, there may be a way to generate monadic-uniform update formulas in certain cases.

The focus operation in TVLA differs from the one in this paper in several key aspects including: (i) it requires the user to specify which formulas to focus on, and (ii) it may yield an infinite number of structures. In contrast, in this paper we show that for every monadic-uniform update, there is a computable set of focused structures that lead to best transformers. Our results also shed light on the cases when the updates in TVLA are precise.

Procedures and Libraries. In this paper, we focused on handling programs without procedures and libraries. It is possible to handle procedures and libraries by tabulation of input/output relations between abstract values (e.g., see [20, 21]). It may be also possible to handle specific libraries by allowing monadic-uniform specifications of auxiliary relations that describe an abstraction of the effect on the client module.

Employing Theorem Provers and Decision Procedures. Theorem provers and decision procedures can be employed to prove properties of programs that manipulate the heap (e.g., see [22–25, 7]). Moreover, they can be used to fully automate the process of generating transformers (e.g., see [26–28, 10]).

Results from dynamic descriptive complexity and the methodology of this paper improve the aforementioned results in various ways. For instance, in contrast to the method of Lahiri and Qadeer [24], which requires user intervention, our method handles programs that manipulate cyclic lists in a totally automatic way.

In essence, the introduction of transformers that use only monadic-uniform update formulas can be seen as a way to replace a characterization of *mutations* of data structures with a characterization in terms of *invariants*. That is, two-vocabulary structures (which describe the state before and after the transition) are a natural way to express mutations, whereas standard one-vocabulary structures express invariants. In some cases, the switch from two-vocabulary to one-vocabulary structures results in an order-of-magnitude complexity improvement. In other cases, where decision procedures are not known for—or known not to exist for—two-vocabulary structures, the reduction to one-vocabulary structures restores the possibility of employing decision procedures:

– With two-vocabulary structures, it is easy to see that monadic second-order logic is undecidable even on linked lists. (The intuitive reason is that two functions, plus a few unary relations, can be used to encode a grid.) However, monadic second-order logic on trees is decidable [29], and thus can be used to perform the feasibility checks on one-vocabulary structures that are needed when our method is employed.
– Rakamaric et al. [7] gave a complete decision procedure for checking feasibility of a given (one-vocabulary) abstract state, but left open the question of how to handle transformers in the most-precise way. Our methodology solves this problem: the DynQF updates for singly linked lists of Hesse [6] can be used to recast the problematic transformers using only one-vocabulary formulas, and hence the best transformer is computable as explained in §1.

References

1. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. (1979)
2. Immerman, N.: Descriptive Complexity. Springer-Verlag (1999)
3. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst. (2002)
4. Loginov, A., Reps, T., Sagiv, M.: Abstraction refinement via inductive learning. In: Proc. Computer-Aided Verif. (2005)
5. Dong, G., Su, J.: Incremental and decremental evaluation of transitive closure by first-order queries. Inf. & Comput. **120** (1995) 101–106
6. Hesse, W.: Dynamic Computational Complexity. PhD thesis, Department of Computer Science, UMass, Amherst (2003)
7. Rakamaric, Z., Bingham, J., Hu, A.: A better logic and decision procedure for predicate abstraction of heap-manipulating programs. Tech. Rep. TR-2006-02, Dept. of Comp. Sci., Univ. of BC, Canada (2006)
8. Lev-Ami, T., Sagiv, M., Immerman, N., Reps, T.: Constructing specialized shape analyses for uniform change. Technical Report TR-2006-11-01, Tel-Aviv Univ. (2006) http://www.cs.tau.ac.il/∼tla/2006/papers/TR-2006-11-01.pdf.
9. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: VMCAI. (2005) 181–198
10. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: TACAS. (2004) 530–545
11. Lev-Ami, T., Immerman, N., Sagiv, M.: Abstraction for shape analysis with fast and precise transformers. In: CAV. (2006) 533–546
12. Myers, E.: Efficient applicative data types. In: POPL. (1984) 66–75
13. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
14. Brown, A.L.: Persistent Object Stores. Univ. of St Andrews (1989)
15. Edmonds, J., Johnson, E.L.: Matching, Euler tours and the chinese postman. Mathematical Programming **5** (1973) 88–124
16. Hendren, L.: Parallelizing Programs with Recursive Data Structures. PhD thesis, Cornell Univ., Ithaca, NY (1990)
17. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. (2006) 287–302
18. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: ESOP. (2003)
19. Loginov, A.: Refinement-based program verification via three-valued-logic analysis. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison (2006)
20. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Formal Descriptions of Programming Concepts. (1978) 237–277
21. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: SAS. (2005) 284–302
22. Nelson, G.: Verifying reachability invariants of linked structures. In: POPL. (1983) 38–47
23. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI. (2001) 221–231
24. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: POPL. (2006)
25. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. In: CADE. (2005) 99–115
26. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN. (2001) 103–122
27. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: SPIN. (2003) 235–239
28. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Proc. VMCAI. (2004)
29. Rabin, M.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. **141** (1969) 1–35