Inductive Reasoning for Shape Invariants

Lilia Georgieva[1] and Patrick Maier[2]

[1] School of Math. and Comp. Sciences, Heriot-Watt University, Edinburgh
http://www.macs.hw.ac.uk/~lilia/
[2] LFCS, School of Informatics, University of Edinburgh
http://homepages.inf.ed.ac.uk/pmaier/

Abstract. Automatic verification of imperative programs that destructively manipulate heap data structures is challenging. In this paper we propose an approach for verifying that such programs do not corrupt their data structures. We specify heap data structures such as lists, arrays of lists, and trees inductively as solutions of logic programs. We use off-the-shelf first-order theorem provers to reason about these specifications.

1 Introduction

In this paper we show how to reason effectively about pointer programs using automatic first-order theorem provers. Common approaches to such reasoning rely on transitive-closure to express reachability in linked data structures. However, first-order theorem provers cannot handle transitive closure accurately, because there is no finite first-order axiomatisation. Instead, various approximations have been proposed for proving (non-)reachability in linked data structures. Yet, there is no universal scheme for approximating transitive closure — the choice of approximation depends on the data structure and on the type of (non-)reachability problem at hand.

We propose a different approach to reasoning about pointer programs. Instead of reasoning about reachability, we reason about the *extension* of heap data structures, i. e., about sets of heap cells. This is sufficient to express many (non-)reachability problems, e. g., "x is reachable from the head of the list", or "the lists pointed to by x and y are separate".

We define common data structures, including acyclic lists, cyclic lists, sorted lists, and binary trees, as logic programs. More precisely, the logic programs define *shape types*, i. e., monadic predicates capturing the extension (set of heap cells) of the given data structure. These logic programs, if viewed as universal first-order theories, have many models, some of which will contain junk, i. e., unreachable heap cells; see Fig 3, showing a shape type for a singly linked list containing junk. The issue of junk models can be avoided if we confine ourselves to least models, i. e., to inductive reasoning. We approximate this inductive least model reasoning by first-order verification conditions. The main contributions of this paper are the following:

– We present logic programs defining a number of common shape types (Section 2). The programs are carefully chosen to harness the power of automatic resolution-based theorem provers (in our case study SPASS [29]) and SMT solvers with heuristic-driven quantifier instantiation (in our case study Yices [6]).

– We describe a methodology to verify that pointer programs maintain shape invariants (Section 3), which may express properties like "a data structure is a sorted doubly-linked list". The method relies on user-provided code annotations and on verification condition generation.

2 Modelling Data Structures

2.1 Logical Heap Model

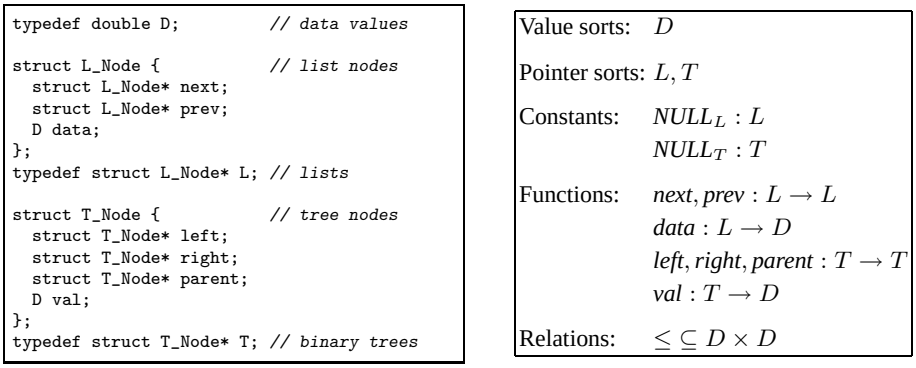
We work in the framework of *many-sorted first-order logic with equality*, assuming familiarity with the basic syntactic and semantic concepts.

Notation. A *signature* Σ declares finite sets of sorts $\Sigma_{\mathcal{S}}$, function symbols $\Sigma_{\mathcal{F}}$ and relation symbols $\Sigma_{\mathcal{R}}$. Function symbols f and relation symbols R have associated arities, usually written as $R \subseteq T_1 \times \cdots \times T_m$ resp. $f : T_1 \times \cdots \times T_n \to T_0$ (or $f : T$ if f is a constant). Given signatures Σ and Δ, their union $\Sigma\Delta$ is a signature. We call Δ an *extension* of Σ if $\Delta = \Sigma\Delta$; we call the extension *relational* if additionally $\Delta_{\mathcal{S}} = \Sigma_{\mathcal{S}}$ and $\Delta_{\mathcal{F}} = \Sigma_{\mathcal{F}}$.

A *Σ-algebra* $\mathbf{A}$ interprets sorts $T \in \Sigma_{\mathcal{S}}$ as carriers $T^{\mathbf{A}}$, function symbols $f \in \Sigma_{\mathcal{F}}$ as functions $f^{\mathbf{A}}$, and relation symbols $R \in \Sigma_{\mathcal{R}}$ as relations $R^{\mathbf{A}}$. We call $\mathbf{B}$ a *Δ-extension* (or simply *extension*) of $\mathbf{A}$ if Δ is an extension of Σ and $\mathbf{B}$ is a Δ-algebra whose Σ-reduct $\mathbf{B}|_\Sigma$ is $\mathbf{A}$.

First-order formulas over Σ are constructed by the usual logical connectives (including the equality predicate $=$). We write $\mathbf{A}, \alpha \models \phi$ to denote that formula ϕ is true in Σ-algebra $\mathbf{A}$ under variable assignment α; we may drop α if ϕ is closed.

Logical model of program state. We consider programs in a subset of the programming language C. With regard to the heap, these programs may allocate and free records (`structs` in C terminology) on the heap, and they may dereference and update pointers to these records. However, they may not perform address arithmetic, pointer type casts, or use variant records (`unions` in C terminology). Under these restrictions, any given program state (i. e., heap plus values of program variables) can be viewed a many-sorted Σ-algebra $\mathbf{A}$ in the following way. (i) The C types are viewed as sorts. There are two classes of sorts: *value* sorts corresponding the C base types (`int`, `float`, etc.) and *pointer* sorts corresponding to C record types. (ii) The elements of the carrier $T^{\mathbf{A}}$ of a value sort T are the values of the C type `T`. $\mathbf{A}$ interprets the standard functions (like addition) and relations (like order) on value sorts as intended. (iii) The elements of the carrier $T^{\mathbf{A}}$ of a pointer sort T are the addresses of records of the C type `T` in the given heap, plus the special address NULL_T, which represents `NULL` pointers of type `T`. (iv) A field `f` of type `T'` in a record of type `T` corresponds to a unary function symbol $f : T \to T'$. Its interpretation $f^{\mathbf{A}}$ maps addresses in $T^{\mathbf{A}}$ to elements of $T'^{\mathbf{A}}$ (i. e., to values or addresses, depending on whether T' is a value or pointer sort). (v) To capture the values of program variables, we extend the signature Σ with constants, one per program variable. For example, the program variable `x` of type `T` is represented by the logical constant x of sort T; the value of `x` is the interpretation $x^{\mathbf{A}}$ of x in $\mathbf{A}$.



```
typedef double D;          // data values

struct L_Node {            // list nodes
  struct L_Node* next;
  struct L_Node* prev;
  D data;
};
typedef struct L_Node* L; // lists

struct T_Node {            // tree nodes
  struct T_Node* left;
  struct T_Node* right;
  struct T_Node* parent;
  D val;
};
typedef struct T_Node* T; // binary trees
```

Value sorts:	D
Pointer sorts:	L, T
Constants:	$\mathit{NULL}_L : L$
	$\mathit{NULL}_T : T$
Functions:	$\mathit{next}, \mathit{prev} : L \to L$
	$\mathit{data} : L \to D$
	$\mathit{left}, \mathit{right}, \mathit{parent} : T \to T$
	$\mathit{val} : T \to D$
Relations:	$\leq \subseteq D \times D$

Fig. 1. Data type declaration in C (left) and corresponding signature Σ (right).

Figure 1 shows a sample C data type declaration and the corresponding signature Σ. As an aside, note that the unary functions in our heap model are total, unlike models of separation logic where the heap is represented by partial functions. A Σ-algebra $\mathbf{A}$ may thus contain junk, i. e., unreachable cells pointing to whatever they like. This does not matter as we will restrict our attention to the well-behaved clusters of the heap cells that are cut out by the shape types presented in the next section, and ignore all the rest.[3]

2.2 Shape Types as Logic Programs

Logic programs. Let Σ and Δ be signatures such that $\Sigma\Delta$ is a relational extension of Σ. A clause (over $\Sigma\Delta$) is called Δ-*Horn* (resp. *definite* Δ-*Horn*) if it contains at most one (resp. exactly one) positive Δ-literal. A $\langle\Sigma, \Delta\rangle$-*LP* is a finite set of Δ-Horn clauses over $\Sigma\Delta$.

Given a Σ-algebra $\mathbf{A}$, we call a $(\Sigma\Delta)$-extension $\mathbf{B}$ of $\mathbf{A}$ an $\mathbf{A}$-*model* of $\mathcal{P}$ if $\mathbf{B} \models \mathcal{P}$; we call $\mathcal{P}$ $\mathbf{A}$-*satisfiable* if it has an $\mathbf{A}$-model. Note that for some $\mathbf{A}$, $\mathcal{P}$ may not be $\mathbf{A}$-satisfiable (because $\mathcal{P}$ may contain non-definite Δ-Horn clauses). However, a standard argument (Proposition 1) shows that if $\mathcal{P}$ is $\mathbf{A}$-satisfiable then it has a least $\mathbf{A}$-model $\mathbf{B_0}$ (in the sense that $R^{\mathbf{B_0}} \subseteq R^{\mathbf{B}}$ for all $R \in \Delta_{\mathcal{R}}$ and all $\mathbf{A}$-models $\mathbf{B}$ of $\mathcal{P}$); we denote $\mathbf{B_0}$ by $\mathrm{lm}(\mathcal{P}, \mathbf{A})$. Thus, $\mathcal{P}$ may be viewed as a transformer taking a Σ-algebra and computing its least $(\Sigma\Delta)$-extension consistent with $\mathcal{P}$ (if it exists).

Proposition 1. *Let $\mathcal{P}$ be a $\langle\Sigma, \Delta\rangle$-LP and $\mathbf{A}$ a Σ-algebra. If $\mathcal{P}$ is $\mathbf{A}$-satisfiable then it has a least $\mathbf{A}$-model.*

Shape types. Informally, a *shape type* is a unary predicate on the heap, characterising the collection of heap cells that form a particular data structure (e. g., a sorted list). Its purpose is twofold: It serves to enforce integrity constraints (like sortedness) on the data structure, and it provides a handle to specify invariants (like separation of two lists).

[3] This is very much what a programmer does, who is also not concerned about the contents of unreachable memory locations.

In the remainder of this section, we will define a number of shape types by $\langle \Sigma, \Delta \rangle$-LPs $\mathcal{P}$, where Σ is the signature of the heap (cf. Section 2.1), which $\Sigma\Delta$ extends with a unary predicate S. Given a heap $\mathbf{A}$, the least model $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ may be viewed as an *annotated heap*, tagging heap cells (as belonging to the interpretation of S) which form the data structure specified by $\mathcal{P}$. We note that the existence of least models will be guaranteed by Proposition 1 because the LPs will be evidently $\mathbf{A}$-satisfiable in all intended heaps $\mathbf{A}$ (i. e., in all $\mathbf{A}$ where the data structure in question is not corrupt).

Shape types of segments of linked lists. Figure 2 presents $\langle \Sigma, \Delta \rangle$-LPs defining the shape types of various list segments (singly- or doubly-linked, sorted or not). The programs are parameterised by input and output signatures Σ and Δ (the latter declaring only the single symbol S).

The simplest LP $\mathcal{P}_{\text{List}}$ defines the shape type S of unsorted singly-linked list segments. The input signature comprises the sort of list cells L, the *next*-pointer function, the pointer p to the head, and the pointer q to the tail of the list beyond the segment. The clauses express (in order of appearance) that (i) the first cell of the tail (pointed to by q) does not belong to S, (ii) the head of the segment belongs to S unless p points to the tail, (iii) no cell in S points to the head, (iv) S is closed under following *next*-pointers up to q, and (v) each cell in S is pointed to by at most one cell in S (i. e., no sharing in S). Figure 3 (top) shows two models of $\mathcal{P}_{\text{List}}$. The first one is the intended least model, where the interpretation of S really is the set of cells forming the list segment from p to q, whereas in the second model the interpretation of S contains some *junk*, i. e., cells that are unreachable from the head.

The LP $\mathcal{P}_{\text{DList}}$ defines the shape type S of doubly-linked list segments from head cell p to last cell r (where the cells ahead of p and behind r are s and q, respectively), see the picture in Figure 3 (middle). The program defines S as both, a *next*-linked list segment from p to q, and a *prev*-linked list segment from r to s. Moreover, it demands that p belongs to S iff r does, and that p and r are the only cells in S that may point to s and q, respectively. Finally, the last two clauses force the *next*-pointers inside S to be converses of the *prev*-pointers, and vice versa.

The LP $\mathcal{P}_{\text{SList}}$ defines the shape type S of sorted, singly-linked list segments. Its parameter list extends that of $\mathcal{P}_{\text{List}}$ by the data sort D, the total ordering $\leq$ on D, and the *data* field. It adds one more clause to $\mathcal{P}_{\text{List}}$, for comparing the data values of adjacent elements in the list segment.

Finally, the LP $\mathcal{P}_{\text{SDList}}$ combines the LPs $\mathcal{P}_{\text{DList}}$ and $\mathcal{P}_{\text{SList}}$, defining the shape type S of sorted, doubly-linked list segments.

Shape types of cyclic lists. The LP $\mathcal{P}_{\text{CList}}$ in Figure 2 defines the shape type S of cyclic singly-linked lists. Its input signature comprises the sort of list cells L, the *next*-pointer function, and the pointer p into the cyclic list. The clauses express (in order of appearance) that: (i) `NULL` does not belong to S, (ii) the cell pointed to by p belongs to S unless p points to `NULL`, (iii) S is closed under following *next*-pointers, and (iv) each cell in S is pointed to by at most one cell in S (i. e., no sharing in S).
The LP $\mathcal{P}_{\text{CDList}}$ defines the shape type S of doubly-linked cyclic lists by extending $\mathcal{P}_{\text{CList}}$ with a clause forcing *next* and *prev* inside S to be converses. We remark that the LPs for

LP for singly-linked list segments

$\mathcal{P}_{\text{List}}[L; \mathit{next} : L \to L, p, q : L; S \subseteq L]$
$= \{\neg\, S(q),$
$\quad S(p) \vee p = q,$
$\quad \forall x{:}L\ .\ S(p) \wedge S(x) \Rightarrow \mathit{next}(x) \neq p,$
$\quad \forall x{:}L\ .\ S(x) \wedge \mathit{next}(x) \neq q \Rightarrow S(\mathit{next}(x)),$
$\quad \forall x, y, z{:}L\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y\}$

LP for doubly-linked list segments

$\mathcal{P}_{\text{DList}}[L; \mathit{next}, \mathit{prev} : L \to L, p, q, r, s : L; S \subseteq L]$
$= \mathcal{P}_{\text{List}}[L; \mathit{next}, p, q; S] \cup \mathcal{P}_{\text{List}}[L; \mathit{prev}, r, s; S]$
$\cup\ \{S(r) \Rightarrow S(p),\ \ S(p) \Rightarrow S(r),$
$\quad \forall x{:}L\ .\ S(x) \wedge \neg\, S(\mathit{prev}(x)) \Rightarrow x = p \wedge \mathit{prev}(x) = s,$
$\quad \forall x{:}L\ .\ S(x) \wedge \neg\, S(\mathit{next}(x)) \Rightarrow x = r \wedge \mathit{next}(x) = q,$
$\quad \forall x{:}L\ .\ S(x) \wedge S(\mathit{next}(x)) \Rightarrow \mathit{prev}(\mathit{next}(x)) = x,$
$\quad \forall y{:}L\ .\ S(y) \wedge S(\mathit{prev}(y)) \Rightarrow \mathit{next}(\mathit{prev}(y)) = y\}$

LP for singly-linked sorted list segments

$\mathcal{P}_{\text{SList}}[D, L; \mathit{next} : L \to L, \mathit{data} : L \to D, p, q : L, \leq\, \subseteq D \times D; S \subseteq L]$
$= \mathcal{P}_{\text{List}}[L; \mathit{next}, p, q; S]$
$\cup\ \{\forall x{:}L\ .\ S(x) \wedge S(\mathit{next}(x)) \Rightarrow \mathit{data}(x) \leq \mathit{data}(\mathit{next}(x))\}$

LP for doubly-linked sorted list segments

$\mathcal{P}_{\text{SDList}}[D, L; \mathit{next}, \mathit{prev} : L \to L, \mathit{data} : L \to D, p, q, r, s : L, \leq\, \subseteq D \times D; S \subseteq L]$
$= \mathcal{P}_{\text{DList}}[L; \mathit{next}, \mathit{prev}, p, q, r, s; S] \cup \mathcal{P}_{\text{SList}}[D, L; \mathit{next}, \mathit{data}, p, q, \leq; S]$

LP for singly-linked cyclic lists

$\mathcal{P}_{\text{CList}}[L; \mathit{next} : L \to L, p, \mathit{NULL}_L : L; S \subseteq L]$
$= \{\neg\, S(\mathit{NULL}_L),$
$\quad S(p) \vee p = \mathit{NULL}_L,$
$\quad \forall x{:}L\ .\ S(x) \Rightarrow S(\mathit{next}(x)),$
$\quad \forall x, y, z{:}L\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y\}$

LP for doubly-linked cyclic lists

$\mathcal{P}_{\text{CDList}}[L; \mathit{next}, \mathit{prev} : L \to L, p, \mathit{NULL}_L : L; S \subseteq L]$
$= \mathcal{P}_{\text{CList}}[L; \mathit{next}, p, \mathit{NULL}_L; S] \cup \{\forall x{:}L\ .\ S(x) \Rightarrow \mathit{prev}(\mathit{next}(x)) = x \wedge \mathit{next}(\mathit{prev}(x)) = x\}$

LP for arrays of singly-linked NULL-terminated lists

$\mathcal{P}_{\text{ListArray}}[I, L; \mathit{next} : L \to L, a : I \to L, \mathit{NULL}_L : L; S \subseteq I \times L]$
$= \{\forall i{:}I\ .\ \neg\, S(i, \mathit{NULL}_L),$
$\quad \forall i{:}I\ .\ S(i, a(i)) \vee a(i) = \mathit{NULL}_L,$
$\quad \forall i{:}I\ \forall x{:}L\ .\ S(i, a(i)) \wedge S(i, x) \Rightarrow \mathit{next}(x) \neq a(i),$
$\quad \forall i{:}I\ \forall x{:}L\ .\ S(i, x) \wedge \mathit{next}(x) \neq \mathit{NULL}_L \Rightarrow S(i, \mathit{next}(x)),$
$\quad \forall i{:}I\ \forall x, y, z{:}L\ .\ S(i, x) \wedge S(i, y) \wedge S(i, z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y,$
$\quad \forall i, j{:}I\ \forall x{:}L\ .\ S(i, x) \wedge S(j, x) \Rightarrow i = j\}$

LP for arrays of singly-linked cyclic lists

$\mathcal{P}_{\text{CListArray}}[I, L; \mathit{next} : L \to L, a : I \to L, \mathit{NULL}_L : L; S \subseteq I \times L]$
$= \{\forall i{:}I\ .\ \neg\, S(i, \mathit{NULL}_L),$
$\quad \forall i{:}I\ .\ S(i, a(i)) \vee a(i) = \mathit{NULL}_L,$
$\quad \forall i{:}I\ \forall x{:}L\ .\ S(i, x) \Rightarrow S(i, \mathit{next}(x)),$
$\quad \forall i{:}I\ \forall x, y, z{:}L\ .\ S(i, x) \wedge S(i, y) \wedge S(i, z) \wedge \mathit{next}(x) = z \wedge \mathit{next}(y) = z \Rightarrow x = y,$
$\quad \forall i, j{:}I\ \forall x{:}L\ .\ S(i, x) \wedge S(j, x) \Rightarrow i = j\}$

LP for binary trees

$\mathcal{P}_{\text{Tree}}[T; \mathit{left}, \mathit{right} : T \to T, r, \mathit{NULL}_T : T; S \subseteq T]$
$= \{\neg\, S(\mathit{NULL}_T),$
$\quad S(r) \vee r = \mathit{NULL}_T,$
$\quad \forall x{:}T\ .\ S(r) \wedge S(x) \Rightarrow (\mathit{left}(x) \neq r \wedge \mathit{right}(x) \neq r),$
$\quad \forall x{:}T\ .\ S(x) \wedge \mathit{left}(x) \neq \mathit{NULL}_T \Rightarrow S(\mathit{left}(x)),$
$\quad \forall x{:}T\ .\ S(x) \wedge \mathit{right}(x) \neq \mathit{NULL}_T \Rightarrow S(\mathit{right}(x)),$
$\quad \forall x, y, z{:}T\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = z \wedge \mathit{left}(y) = z \Rightarrow x = y,$
$\quad \forall x, y, z{:}T\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{right}(x) = z \wedge \mathit{right}(y) = z \Rightarrow x = y,$
$\quad \forall x, y, z{:}T\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = z \wedge \mathit{right}(y) = z \Rightarrow x = y,$
$\quad \forall x, y, z{:}T\ .\ S(x) \wedge S(y) \wedge S(z) \wedge \mathit{left}(x) = y \wedge \mathit{right}(x) = z \Rightarrow y \neq z\}$

LP for binary trees with parent pointers

$\mathcal{P}_{\text{PTree}}[T; \mathit{left}, \mathit{right}, \mathit{parent} : T \to T, r, s, \mathit{NULL}_T : T; S \subseteq T]$
$= \mathcal{P}_{\text{Tree}}[T; \mathit{left}, \mathit{right}, r, \mathit{NULL}_T; S]$
$\cup\ \{s \neq \mathit{NULL}_T \Rightarrow S(r),$
$\quad S(r) \Rightarrow s = \mathit{parent}(r),$
$\quad \neg\, S(s),$
$\quad \forall x{:}T\ .\ S(x) \wedge S(\mathit{left}(x)) \Rightarrow \mathit{parent}(\mathit{left}(x)) = x,$
$\quad \forall x{:}T\ .\ S(x) \wedge S(\mathit{right}(x)) \Rightarrow \mathit{parent}(\mathit{right}(x)) = x,$
$\quad \forall y{:}T\ .\ S(y) \wedge S(\mathit{parent}(y)) \Rightarrow (\mathit{left}(\mathit{parent}(y)) = y \vee \mathit{right}(\mathit{parent}(y)) = y)\}$

Fig. 2. LPs defining shape types of list segments, cyclic lists, arrays of lists, binary trees.

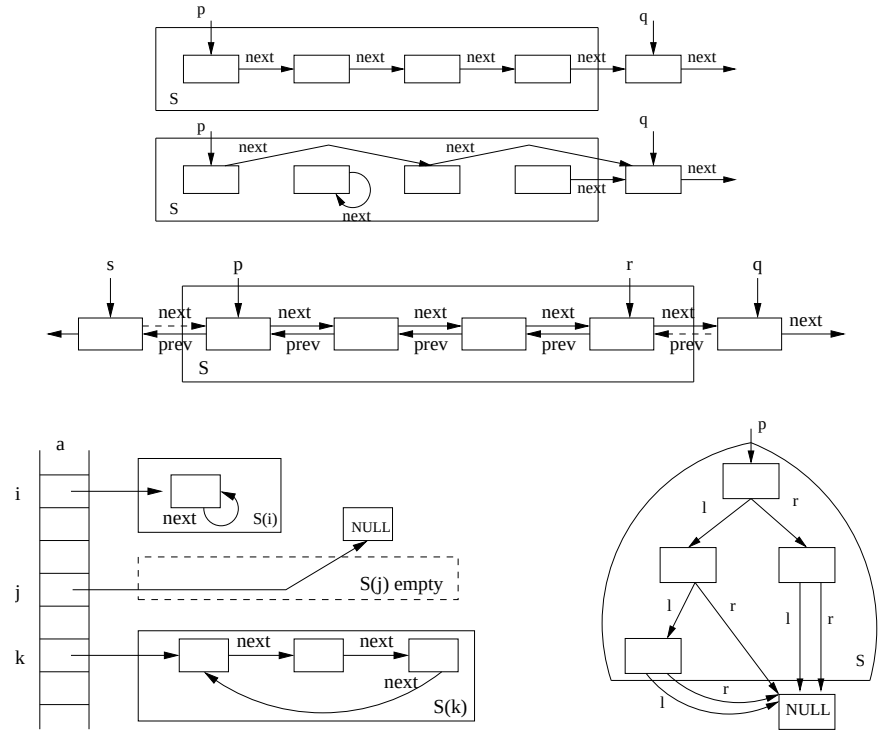


Fig. 3. From top to bottom: shape types S of singly-linked list segments (intended least model and model with unreachable junk), doubly-linked list segments, array of cyclic lists, and binary trees.

cyclic lists are more elegant than the corresponding LPs for singly- and doubly-linked list segments.

Shape types of arrays of lists. The LP $\mathcal{P}_{\text{CListArray}}$ in Figure 2 defines the shape type S of arrays of singly-linked cyclic lists, see Figure 3 (bottom left) for a graphical depiction. The input signature comprises the array index sort I, the list cell sort L, the *next*-pointer function, and the function a mapping array indices to pointers into the lists. The shape type S is a binary relation between array indices and list cells. Note that we model arrays as functions from index type to element type[4], ignoring array bounds. In the light of this, the shape type S may be viewed as an array of sets of list cells rather than as a binary relation.

The first four clauses of $\mathcal{P}_{\text{CListArray}}$ state that for each index i, the unary relation $S(i,\cdot)$ — S with fixed first argument i — is the shape type of a cyclic singly-linked list pointed to by $a(i)$; note how these four clauses correspond to the clauses of $\mathcal{P}_{\text{CList}}$.

[4] Our model assumes that arrays do not live in the heap.

The last clause states that the shape types $S(i,\cdot)$ and $S(j,\cdot)$ must be disjoint for distinct indices i and j.

The LP $\mathcal{P}_{\text{ListArray}}$ defines the shape type S of arrays of singly-linked NULL-terminated lists in a similar way. Its first five clauses force each shape type $S(i,\cdot)$ to be a singly-linked list segment from $a(i)$ to NULL_L, and the last clause forces disjointness of distinct shape types $S(i,\cdot)$ and $S(j,\cdot)$.

Shape types of binary trees. The LP $\mathcal{P}_{\text{Tree}}$ defines the shape type S of plain binary trees, see the picture in Figure 3 (bottom right). The input signature comprises the sort of tree nodes T, the *left*- and *right*-pointer functions and the pointer r to the root. The clauses borrow heavily from the LP $\mathcal{P}_{\text{List}}$ for singly-linked list segments (with q replaced by NULL_T) and express that: (1) NULL does not belong to S, (2) the root belongs to S unless r points to NULL, (3) no node in S points to the root, (4-5) S is closed under following *left*- and *right*-pointers up to NULL, and (6-9) there is no sharing in S because each node in S is pointed to by at most one node in S (clauses 6-8) and has distinct *left*- and *right*-successors (clause 9).

The LP $\mathcal{P}_{\text{PTree}}$ defines the shape type S of binary trees with parent pointers by extending $\mathcal{P}_{\text{Tree}}$. The additional clauses express that (1-2) s is the parent of the root r unless r is not in S, in which case s must be NULL, (3) s does not belong to S, and (4-6) the *parent*-pointers are converse to the union of the *left*- and *right*-pointers.

3 Verifying Pointer Programs

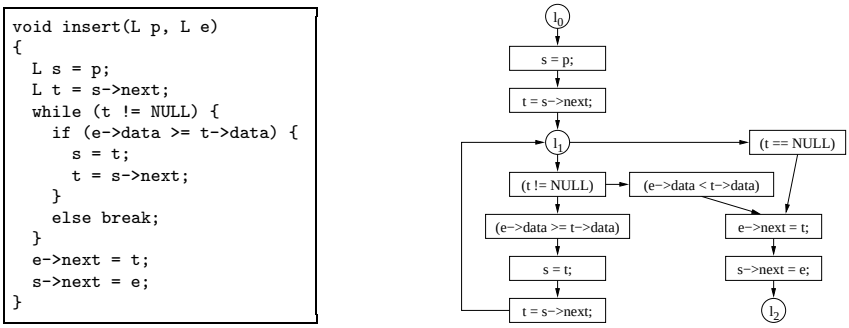
We aim to verify imperative programs that manipulate dynamic data structures on the heap. Given the code of a C function plus specifications of its input and output (and possibly of loop invariants), we want to verify that the program maintains certain *shape invariants*, e. g., that the sorted list being updated remains a list and sorted.

Notation. Given a signature Σ, we define the signature Σ' as a copy of Σ where all functions f (except the constants NULL_T) and relations R are replaced by *primed* functions f' resp. relations R'. Given a Σ-formula ϕ (resp. a $\langle\Sigma,\Delta\rangle$-LP $\mathcal{P}$), we write ϕ' (resp. $\mathcal{P}'$) for the Σ-formula (resp. $\langle\Sigma',\Delta'\rangle$-LP) that arises from ϕ (resp. $\mathcal{P}$) by replacing all functions f (except NULL_T) and relations R by f' and R', respectively.

3.1 Verification Problem

We verify C functions by checking verification conditions. To do this, we convert the code of a function into a control flow graph (CFG) and find a set of cut locations, to which we attach shape invariants. Each path between cut locations gives rise to a verification condition (VC), which claims that the path establishes the invariant at its end location, given that the invariant at the start location was assumed.

We will not elaborate on the well-known techniques for translating C code to CFGs and identifying cut locations; the reader may consult Figure 4 for an example. The figure shows the code for inserting an element e into a non-empty sorted list pointed to by p. The CFG has three cut locations l_0 (the entry location) to l_2 (the exit location), with four paths σ_1 to σ_4 between them.



path	path formula π
$\sigma_1 : l_0 \to l_1 =$ `s = p;` `t = s->next;`	$s' = p \wedge$ $t' = \mathit{next}(s') \wedge$ $p' = p \wedge e' = e \wedge \mathit{next}' = \mathit{next} \wedge \mathit{data}' = \mathit{data}$
$\sigma_2 : l_1 \to l_1 =$ `(t != NULL)` `(e->data >= t->data)` `s = t;` `t = s->next;`	$t \neq \mathit{NULL}_L \wedge$ $\mathit{data}(e) \geq \mathit{data}(t) \wedge$ $s' = t \wedge$ $t' = \mathit{next}(s') \wedge$ $p' = p \wedge e' = e \wedge \mathit{next}' = \mathit{next} \wedge \mathit{data}' = \mathit{data}$
$\sigma_3 : l_1 \to l_2 =$ `(t != NULL)` `(e->data < t->data)` `e->next = t;` `s->next = e;`	$\exists \mathit{next}_1 : L \to L$. $t \neq \mathit{NULL}_L \wedge$ $\mathit{data}(e) < \mathit{data}(t) \wedge$ $\mathit{next}_1(e) = t \wedge (\forall x{:}L \,.\, x = e \vee \mathit{next}_1(x) = \mathit{next}(x)) \wedge$ $\mathit{next}'(s) = e \wedge (\forall x{:}L \,.\, x = s \vee \mathit{next}'(x) = \mathit{next}_1(x)) \wedge$ $p' = p \wedge e' = e \wedge s' = s \wedge t' = t \wedge \mathit{data}' = \mathit{data}$
$\sigma_4 : l_1 \to l_2 =$ `(t == NULL)` `e->next = t;` `s->next = e;`	$\exists \mathit{next}_1 : L \to L$. $t = \mathit{NULL}_L \wedge$ $\mathit{next}_1(e) = t \wedge (\forall x{:}L \,.\, x = e \vee \mathit{next}_1(x) = \mathit{next}(x)) \wedge$ $\mathit{next}'(s) = e \wedge (\forall x{:}L \,.\, x = s \vee \mathit{next}'(x) = \mathit{next}_1(x)) \wedge$ $p' = p \wedge e' = e \wedge s' = s \wedge t' = t \wedge \mathit{data}' = \mathit{data}$

loc.	formula ϕ of shape invariant $\langle \mathcal{P}, \phi \rangle$ — see Section 3.1 for LP $\mathcal{P}$
l_0	$S \cap E = \emptyset \wedge S(p) \wedge E(e) \wedge \mathit{next}(e) = \mathit{NULL}_L \wedge \mathit{data}(p) \leq \mathit{data}(e)$
l_1	$S = S_0 \wedge E = E_0 \wedge p = p_0 \wedge e = e_0 \wedge$ $S \cap E = \emptyset \wedge S(p) \wedge E(e) \wedge \mathit{next}(e) = \mathit{NULL}_L \wedge \mathit{data}(p) \leq \mathit{data}(e) \wedge$ $S(s) \wedge \mathit{data}(s) \leq \mathit{data}(e) \wedge \mathit{next}(s) = t \wedge (t = \mathit{NULL}_L \vee S(t))$
l_2	$S = S_0 \uplus E_0 \wedge p = p_0 \wedge e = e_0$

path	shape effect ε
σ_1	$S' = S \wedge E' = E$
σ_2	$S' = S \wedge E' = E$
σ_3	$S' = S \uplus E$
σ_4	$S' = S \uplus E$

Fig. 4. Insert an element into a non-empty list sorted in ascending order: C code, control flow graph, paths through the CFG, shape invariants, and shape effects.

State signature. Associated with a C function is a *state signature* Σ, the signature of the Σ-algebras serving as logical models of program state, see Section 2.1. In the following, Σ always refers to a fixed state signature.

In the example of Figure 4, Σ declares the value sort D and the pointer sort L, the constants $p, e, s, t, \mathit{NULL}_L : L$, the functions $\mathit{data} : L \to D$ and $\mathit{next} : L \to L$ and the order relation $\leq \subseteq D \times D$.

Path formulas. A path σ through the CFG is a sequence consisting of variable assignments `x = e;` array updates `a[i] = e;` heap updates `x->f = e;` and conditions `(c)` where `e` is an expression (an R-value in C terminology) and `c` is a conditional

expression.[5] The translation of such paths to first-order logic is well-known and will not be detailed here; the reader is referred to Figure 4 for examples. The *path formula* π resulting from translation of a path σ is a $(\Sigma\Sigma')$-formula, where the signatures Σ and Σ' belong to the state at the start and end locations of σ, respectively. Two things to note. First, part of each path formula is an explicit "frame condition" stating which variables and pointer fields do not change; note the use of second-order equalities like $\mathit{next}' = \mathit{next}$ as short-hands for more complex first-order expressions. Second, path formulae, like the ones for paths σ_3 and σ_4, may start with a string of second-order $\exists$-quantifiers to project away intermediate state; these quantifiers will always be eliminable by Skolemisation.

Shape invariants. A *Δ-shape invariant* is a pair $\langle \mathcal{P}, \phi \rangle$, where $\mathcal{P}$ is a $\langle \Sigma, \Delta \rangle$-LP and ϕ is a $(\Sigma\Delta)$-formula. Its purpose is to constrain the program state by the relating shape types, which are defined by the LP $\mathcal{P}$, with each other or with program variables.

Shape invariants are associated with cut locations in the CFG. Figure 4 presents the shape invariants for the `insert` function. These shape invariants involve two shape types S and E defined by the LP $\mathcal{P} = \mathcal{P}_{\text{SList}}[D, L; \mathit{next}, \mathit{data}, p, \mathit{NULL}_L, \leq; S] \cup \mathcal{P}_{\text{List}}[L; \mathit{next}, e, \mathit{NULL}_L, \leq; E] \cup \{\forall x{:}L \,.\, \neg\, S(x) \vee \neg\, E(x)\}$. I. e., S is the shape type of `NULL`-terminated sorted lists pointed to by p, E of `NULL`-terminated lists pointed to by e, and both shape types are disjoint. Note that the LP $\mathcal{P}$ is common to all three invariants. The shape invariant at l_0, for instance, stipulates that the lists S and E are disjoint and non-empty (because they contain their heads p and e), E is of length 1 (because the `next`-pointer of its head e is `NULL`), and the data at p is less than or equal to the data at e. Note the use of set-relational expressions like $S \cap E = \emptyset$ as short-hand for more complex first-order expressions. The shape invariant at l_2 stipulates that the list S is the sum of the start lists S_0 and E_0[6] and that the program variables p and e retain their start values p_0 and e_0, respectively. This, together with sortedness of S, which is enforced by the LP defining S, is a statement of functional correctness of `insert`.

Verification condition. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, and $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k. To prove correctness of σ, we must show that every execution establishes the *post* shape invariant $\langle \mathcal{Q}, \psi \rangle$ at the end, provided that the *pre* shape invariant $\langle \mathcal{P}, \phi \rangle$ held at the start. This translates to the following verification condition.

$$\forall(\Sigma\Sigma')\text{-algebra } \mathbf{A} : \mathrm{lm}(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi \implies \mathrm{lm}(\mathcal{Q}', \mathbf{A}) \models \psi' \qquad \text{(VC)}$$

Note that the antecedent of (VC) tacitly depends on the existence of $\mathrm{lm}(\mathcal{P}, \mathbf{A})$, and the succedent tacitly states that the existence of $\mathrm{lm}(\mathcal{Q}', \mathbf{A})$ follows from the antecedent.

[5] To keep the presentation simple, we ignore dynamic memory allocation and function calls. Both could be handled: memory allocation through tracking the set of allocated heap cells, function calls through extra cut locations before and after call sites.

[6] The use of subscript 0 indicates values of program variables or shape types at the initial location l_0. Strictly speaking, a shape invariant is not just constraining the program state at location ℓ, but the relation between the initial state and the state at location ℓ.

The trouble with (VC) is that it requires reasoning in least models, i. e., inductive reasoning. The next section presents our methodology to rephrase the inductive condition (VC) in first-order logic.

3.2 Approximating Inductive Reasoning

The obvious problem with using first-order provers for reasoning about shape types is their ignorance of least models. For instance, a VC on a path σ may be invalid in first-order logic because there is a counter model which picks the least interpretation for shape type S at the start of σ but the greatest interpretation for S at the end.

To deal with this problem, we weaken the VC by speculatively assuming a *shape effect* relating shape types at the start and end of σ. Often, the weakened VC becomes provable in first-order logic. However, we still have to justify the assumed shape effect. We do so by proving two further first-order VCs, which together imply that the shape effect is an inductive consequence of the LPs defining the shape types.

Shape effects. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, and $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k. A *shape effect* for σ is a $(\Sigma\Delta\Sigma'\Lambda')$-formula ε which is *back-and-forth total*, that is,

- $\forall(\Sigma\Delta\Sigma')$-algebra $\mathbf{A}$: $\mathbf{A} \models \mathcal{P} \cup \{\pi\} \implies \exists(\Sigma\Delta\Sigma'\Lambda')$-extension $\mathbf{B}$: $\mathbf{B} \models \varepsilon$, and
- $\forall(\Sigma\Sigma'\Lambda')$-algebra $\mathbf{C}$: $\mathbf{C} \models \mathcal{Q}' \cup \{\pi\} \implies \exists(\Sigma\Delta\Sigma'\Lambda')$-extension $\mathbf{D}$: $\mathbf{D} \models \varepsilon$.

The purpose of a shape effect ε is to relate the shape types at the start of path σ with those at the end. A convenient way to specify simple shape effects is to write set-relational expressions like $S' = S \uplus E$ (cf. Figure 4) as short-hands for more complex quantified expressions. This style also makes it easy to check the totality requirement. For example, back-and-forth totality of the shape effect $S' = S \uplus E$ for σ_3 holds because (1) every $(\Sigma\Delta\Sigma')$-algebra which interprets S and E disjointly (which is enforced by the LP $\mathcal{P}$) has a $(\Sigma\Delta\Sigma'\Lambda')$-extension which interprets S' as the sum of S and E, and (2) every $(\Sigma\Sigma'\Lambda')$-algebra (which interprets S') has a $(\Sigma\Delta\Sigma'\Lambda')$-extension which interprets S and E disjointly such that their sum is S'. Note that in this particular case, totality is independent of the path formula π.

Notation. Given a Σ-algebra $\mathbf{A}$, one may need to compare one $\mathbf{A}$-model of the $\langle \Sigma, \Delta \rangle$-LP $\mathcal{P}$ to another one. Logically, this can be done by fusing the two models, which requires duplicating all symbols that are not shared, i. e., all relations in Δ.

We define the signature $\hat{\Delta}$ as a copy of Δ where all relations R are replaced by *capped* relations $\hat{R}$. Given a Δ-shape invariant $\langle \mathcal{P}, \phi \rangle$, we write $\langle \hat{\mathcal{P}}, \hat{\phi} \rangle$ for the $\hat{\Delta}$-shape invariant that arises from $\langle \mathcal{P}, \phi \rangle$ by replacing all relations R in Δ with $\hat{R}$. Given a shape effect ε (as defined above) for a path σ, we write $\hat{\varepsilon}$ for the $(\Sigma\hat{\Delta}\Sigma'\hat{\Lambda}')$-formula ε that arises from ε by replacing all relations R in $\Delta\Lambda$ with R'; note that $\hat{\varepsilon}$ is also a shape effect for σ.

Verification conditions. Given a path σ from ℓ to k, let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k, and ε the shape effect for σ. To prove the inductive condition (VC) it suffices to prove the following three first-order conditions.

$$\mathcal{P} \cup \mathcal{Q}' \cup \{\pi, \varepsilon, \phi\} \models \psi' \tag{VC1}$$

$$\mathcal{P} \cup \{\pi, \varepsilon, \phi\} \models \mathcal{Q}' \tag{VC2}$$

$$\begin{gathered} \mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\textstyle\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S', \pi, \varepsilon, \hat{\varepsilon}\} \models \\ \mathcal{P} \cup \hat{\mathcal{P}} \cup \{\textstyle\bigwedge_{S \in \Delta_\mathcal{R}} \hat{S} \subseteq S, \bigvee_{S \in \Delta_\mathcal{R}} \hat{S} \neq S\} \end{gathered} \tag{VC3}$$

(VC1) and (VC2) together state preservation of shape invariants, subject to the (yet unjustified) assumption of a shape effect ε. (VC2) can be read as a model transformation: Given any model of $\mathcal{P}$ that satisfies the path formula and the pre shape invariant, the shape effect ε will produce models of $\mathcal{Q}'$. Finally, (VC3) implies that ε preserves minimal models. It can be seen as a reverse model transformation which preserves order: Given any two models of $\mathcal{Q}'$ such that both satisfy the path formula and one is strictly contained in the other, the shape effects ε and $\hat{\varepsilon}$ will produce two models of $\mathcal{P}$ such that one is strictly contained in the other.

Soundness. The following theorem proves that the conditions (VC1) – (VC3) together imply that a shape effect is an inductive consequence (i. e., entailed in the least model) of the LPs defining the shape types. Soundness of the first-order verification conditions w. r. t. to the inductive condition (VC) is an easy corollary.

Theorem 2. *Let σ be a path from ℓ to k. Let π be the path formula for σ, $\langle \mathcal{P}, \phi \rangle$ the Δ-shape invariant at ℓ, $\langle \mathcal{Q}, \psi \rangle$ the Λ-shape invariant at k, and ε the shape effect for σ. Assume that* (VC2) *and* (VC3) *hold. For all $(\Sigma\Sigma')$-algebras $\mathbf{A}$, if lm$(\mathcal{P}, \mathbf{A})$ exists and lm$(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi$ then lm$(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ exists and lm$(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \models \varepsilon$.*

Proof. Towards a contradiction assume there is a $(\Sigma\Sigma')$-algebra $\mathbf{A}$ such that $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ and $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ exist and $\mathrm{lm}(\mathcal{P}, \mathbf{A}) \models \pi \wedge \phi$, but $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \not\models \varepsilon$. As ε is total, the $(\Sigma\Delta\Sigma')$-model $\mathrm{lm}(\mathcal{P}, \mathbf{A})$ of π extends to a $(\Sigma\Delta\Sigma'\Lambda')$-model $\mathbf{B}$ of ε. Thus, $\mathbf{B} \models \mathcal{P} \cup \{\pi, \varepsilon, \phi\}$, which by (VC2) implies $\mathbf{B} \models \mathcal{Q}'$. Note that $\mathbf{B}$ is not an extension of the $(\Sigma\Sigma'\Lambda')$-model $\mathrm{lm}(\mathcal{Q}', \mathbf{A})$, for if it were then $\mathbf{B} = \mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A})$ and hence $\mathrm{lm}(\mathcal{P} \cup \mathcal{Q}', \mathbf{A}) \models \varepsilon$, which would contradict our assumption. Thus $\mathbf{B}|_{\Sigma\Sigma'\Lambda'}$ is a non-least, hence non-minimal, $\mathbf{A}$-model of $\mathcal{Q}'$, which implies that $\mathbf{B}$ has a $(\Sigma\Delta\Sigma'\Lambda'\hat{\Lambda}')$-extension $\mathbf{C}$ such that $\mathbf{C} \models \mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S'\}$. As the shape effect $\hat{\varepsilon}$ is total, the $(\Sigma\Sigma'\hat{\Lambda}')$-model $\mathbf{C}|_{\Sigma\Sigma'\hat{\Lambda}'}$ of π extends to a $(\Sigma\hat{\Delta}\Sigma'\hat{\Lambda}')$-model $\mathbf{D}$ of $\hat{\varepsilon}$. Hence, the $(\Sigma\Delta\hat{\Delta}\Sigma'\Lambda'\hat{\Lambda}')$-algebra $\mathbf{E}$, which extends both $\mathbf{C}$ and $\mathbf{D}$, is a model of $\mathcal{Q}' \cup \hat{\mathcal{Q}}' \cup \{\bigwedge_{S \in \Lambda_\mathcal{R}} \hat{S}' \subseteq S', \bigvee_{S \in \Lambda_\mathcal{R}} \hat{S}' \neq S', \pi, \varepsilon, \hat{\varepsilon}\}$. By (VC3), this implies $\mathbf{E} \models \mathcal{P} \cup \hat{\mathcal{P}} \cup \{\bigwedge_{S \in \Delta_\mathcal{R}} \hat{S} \subseteq S, \bigvee_{S \in \Delta_\mathcal{R}} \hat{S} \neq S\}$, i. e., $\mathbf{E}$ is an extension of a non-minimal $\mathbf{A}$-model of $\mathcal{P}$, contradictory to $\mathbf{E}|_{\Sigma\Delta\Sigma'} = \mathbf{C}|_{\Sigma\Delta\Sigma'} = \mathbf{B}|_{\Sigma\Delta\Sigma'} = \mathrm{lm}(\mathcal{P}, \mathbf{A})$. □

Corollary 3. *If* (VC1) – (VC3) *hold then* (VC) *holds.*

3.3 Experiments

We have used our methodology to successfully verify a number of simple heap-manipulating algorithms, including the sorted list insert function from Figure 4. Other examples include functions for inserting and deleting elements into binary trees, and functions for moving elements between ring buffers organised in an array. All functions were manually annotated with shape invariants and effects. Due to lack of space, we do not report on these experiments in detail.

We have run our experiments on the theorem provers SPASS and Yices. Both provers succeeded to prove all verification conditions. The typical run-time per VC was below 10 seconds for SPASS, below 1 second for Yices. More experiments are necessary to determine whether our methodology scales to more complex code.

We remark on the somewhat surprising fact that Yices succeeded on all VCs, despite its incomplete heuristics for quantifier instantiation (which we did not assist using the trigger mechanism). We suspect that a key reason for this is our choice of defining shape types by logic programs, which to a first-order theorem prover are just universally quantified clauses; avoiding existential quantifiers seems to suit Yices well. However, we did observe cases where Yices' instantiation heuristic was sensitive to the formulation of particular clauses (especially the no-sharing clauses for binary trees).

4 Related Work

Efficient theorem provers make first-order logic attractive framework for studying reachability in mutable linked data structures. However, transitive closure, essential for properties of pointer structures presents a challenge because first-order theorem provers cannot handle transitive closure.

Various approaches for program analysis that use first-order logic have been investigated. We next discuss the most prominent.

The logic of interpreted sets and bounded quantification is used for specifying properties of heap manipulating programs [18]. The logic uses first-order logic and is interpreted over a finite partially-ordered set of sorts. It provides a ternary reachability predicate and allows bounded universal quantification over two different kinds of (potentially unbounded) sets. Following this approach first-order SMT solvers, augmented with theories, are used for precise verification of heap-manipulating programs. An alternative framework uses ground logic enriched with ternary predicate [22].

The use of a decidable fragment of first-order logic augmented with arithmetic on scalar field to specify properties of data structures is studied in [20]. In contrast to ours, this approach does not use theories for recursive predicates like reachability, and relies on user provided ghost variables to express properties of data structures.

In [19] a first-order formula, in which transitive closure occurs is simulated by a first-order formula, where transitive closure is encoded by adding a new relation symbol for each binary predicate. This together with inductively defined first-order axioms assures that transitive closure is interpreted correctly. A set of axioms defines the properties of transitive closure inductively. The axioms are not complete over infinite models. If the axioms are such that every finite, acyclic model satisfying them must

interpret the encoding of transitive closure as the reflexive, transitive closure of its interpretation of the transitive binary relation, then the axiom schema is complete [3]. Its incompleteness notwithstanding, the induction schema in [19] allows for automatically proving properties of simple programs using SPASS [29].

Alternative approach for symbolic shape analysis [30] uses the framework of (extensions of) decidable fragments of first-order logic e. g., guarded fixpoint logic [10]. The logic expresses reachability along paths and from a specific point, but not reachability between a pair of program variables [12].

Syntactically defined logics for shape analysis, such as local shape logic $\mathcal{LSL}$ [26] and role logic [15], are closely related to first-order logic. Our approach is applicable to their translation in first-order logic. The logic $\mathcal{LSL}$ [26] is strictly less expressive than the two variable fragment of first order logic with counting. Role logic [15] is variable free logic, which is equally expressive as first-order logic with transitive closure and consequently undecidable. A decidable fragment of role logic is as expressive as the two variable fragment of first-order logic with counting. Role logic is closely related to description logic which we have investigated for symbolic shape analysis [9].

Approaches based on three valued logic, which use over-approximation, have been studied in [13, 32]. The semantics of statements and the query of interest are expressed in three valued logic. Only restricted fragments of the logic are decidable [12].

Prominent verification approaches for analysis of data structures use parameterised abstract domains; these analyses include parametric shape analysis [2] as well as predicate abstraction [1, 11] and generalisations of predicate abstraction [16, 17]. Similarly to our approach, reasoning about reachability in program analysis and verification following parametric shape analysis or generalisations of predicate abstraction, are dependent on the invariants that the program maintains for the specific data structure that it manipulates. An algorithm for inferring loop invariants of programs that manipulate heap-allocated data structures, parameterised by the properties to be verified is implemented in Bohne [31]. Bohne infers universally quantified invariants using symbolic shape analysis based on Boolean heaps [24]. Abstraction predicates can be Boolean-valued state predicates (which are either true or false in a given state) or predicates denoting sets of heap objects in a given state (which are true of a given object in a given state).

An algebraic approach towards analysis of pointer programs in the framework of first-order logic is presented in [21]. The underlying pointer-structures and properties such as reachability and sharing are modelled by binary relations and the properties are calculated by a set of rewrite rules.

Separation logic [23] is distinguished by the use of a spatial form of conjunction $(P * Q)$, which allows the spatial orientation of a data structure to be captured without having to use auxiliary predicates. Least and greatest fixpoint operators can be added to separation logic, so that pre- and post-condition semantics for a while-language can be wholly expressed within the logic [27]. Formalisation of recursively defined properties on inductively (and co-inductively) defined data structures is then achievable in the language. The addition of the recursion operators in separation logic leads to alterations to the standard definition of syntactic substitution and the classic substitution; the reasons are related both to the semantics of stack storage and heap storage as well as to the inclusion of the recursion operators [27]. Inductive shape analysis based on

separation logic using programmer supplied invariant checkers and numerical domain constraints is proposed in [4]. This approach is applicable to more complex data structures defined by counting, namely red-black trees.

In [25] pointer-based data structure of singly-linked lists and a theory of linked lists is defined as a class of structures of many-sorted first-order logic. The theory is expressive and allows for reasoning about cells, indexed collections of cells, and the reachability of a certain cell from another. The theory is developed for linked lists only.

Alternative languages for modelling and reasoning include modal μ-calculus [14, 28], expressive description logics [5], the propositional dynamic logic [8] and temporal logics [7], and rewriting approaches based on first-order logic [21].

5 Conclusion

In this paper we study imperative programs with destructive update of pointer fields. We model shape types, such as linked lists, cyclic lists, and binary trees as least models of logic programs. We approximate the inductive reasoning about least models by first-order reasoning. We demonstrate that the method is effective for simple programs.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213. ACM, 2001.

[2] M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *ESOP'99*, LNCS 1576, pages 2–19. Springer, 1999.

[3] D. Calvanese. Finite model reasoning in description logics. In *KR'96*, pages 292–303. Morgan Kaufman, 1996.

[4] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM, 2008.

[5] G. De Giacomo and M. Lenzerini. Concept language with number restrictions and fixpoints, and its relationship with mu-calculus. In *ECAI'94*, pages 411–415. John Wiley and Sons, 1994.

[6] B. Dutertre and L. De Moura. The YICES SMT solver, 2006. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf.

[7] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B*, pages 995–1072. Elsevier and MIT, 1990.

[8] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[9] L. Georgieva and P. Maier. Description logics for shape analysis. In *SEFM'05*, pages 321–331. IEEE, 2005.

[10] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *LICS'99*, pages 45–54. IEEE, 1999.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'02*, pages 58–70. ACM, 2002.
[12] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL'04*, LNCS 3210, pages 160–174. Springer, 2004.
[13] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV'04*, LNCS 3114, pages 281–294. Springer, 2004.
[14] D. Kozen. Results on the propositional μ-calculus. In *ICALP'82*, LNCS 140, pages 348–359. Springer, 1982.
[15] V. Kuncak and M. C. Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003. Available at http://arxiv.org/abs/cs.PL/0408018.
[16] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, LNCS 3114, pages 135–147. Springer, 2004.
[17] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.
[18] S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182. ACM, 2008.
[19] T. Lev-Ami, N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE'05*, LNCS 3632, pages 99–115. Springer, 2005.
[20] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV'05*, LNCS 3576, pages 476–490. Springer, 2005.
[21] B. Möller. Linked lists calculated. Technical Report 1997-07, Department of Computer Science, University of Augsburg, 1997.
[22] G. Nelson. Verifying reachability invariants of linked structures. In *POPL'83*, pages 38–47. ACM, 1983.
[23] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, LNCS 2142, pages 1–19. Springer, 2001.
[24] A. Podelski and T. Wies. Boolean heaps. In *SAS'05*, LNCS 3672, pages 268–283. Springer, 2005.
[25] S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06*, pages 206–215. IEEE, 2006.
[26] A. Rensink. Canonical graph shapes. In *ESOP'04*, LNCS 2986, pages 401–415. Springer, 2004.
[27] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theor. Comput. Sci.*, 351(2):258–275, 2006.
[28] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Inf. Comput.*, 81(3):249–264, 1989.
[29] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *CADE'02*, LNCS 2392, pages 275–279. Springer, 2002.
[30] T. Wies. Symbolic shape analysis. Master's thesis, Saarland University, Saarbrücken, 2004.
[31] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. C. Rinard. On verifying complex properties using symbolic shape analysis. Technical Report MPI-I-2006-2-1, Max-Planck Institute for Computer Science, 2006. Available at http://arxiv.org/abs/cs.PL/0609104.
[32] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS'04*, LNCS 2988, pages 530–545. Springer, 2004.