Shape Analysis by Predicate Abstraction*

Ittai Balaban[1], Amir Pnueli[1], and Lenore D. Zuck[2]

[1] New York University, New York
{balaban,amir}@cs.nyu.edu
[2] University of Illinois at Chicago
lenore@cs.uic.edu

Abstract. The paper presents an approach for shape analysis based on predicate abstraction. Using a predicate base that involves reachability relations between program variables pointing into the heap, we are able to analyze functional properties of programs with destructive heap updates, such as list reversal and various in-place list sorts. The approach allows verification of both safety and liveness properties. The abstraction we use does not require any abstract representation of the heap nodes (e.g. abstract shapes), only reachability relations between the program variables.

The computation of the abstract transition relation is precise and automatic yet does not require the use of a theorem prover. Instead, we use a small model theorem to identify a truncated (small) finite-state version of the program whose abstraction is identical to the abstraction of the unbounded-heap version of the same program. The abstraction of the finite-state version is then computed by BDD techniques.

For proving liveness properties, we augment the original system by a well-founded ranking function, which is abstracted together with the system. Well-foundedness is then abstracted into strong fairness (compassion). We show that, for a restricted class of programs that still includes many interesting cases, the small model theorem can be applied to this joint abstraction.

Independently of the application to shape-analysis examples, we demonstrate the utility of the ranking abstraction method and its advantages over the direct use of ranking functions in a deductive verification of the same property.

1 Introduction

The goal of *shape analysis* is to analyze properties of programs that perform destructive updating on dynamically allocated storage (heaps) [11]. Programs manipulating heap structures can be viewed as *parameterized* in the number of heap nodes, or, alternatively, the memory size.

This paper presents an approach for shape analysis based on *predicate abstraction* that allows for analyses of functional properties such as safety and liveness. The abstraction used does *not* require any abstract representation of the heap nodes (e.g. abstract shapes), but rather, requires only reachability relations between the program variables.

* This research was supported in part by NSF grant CCR-0205571 and ONR grant N00014-99-1-0131

R. Cousot (Ed.): VMCAI 2005, LNCS 3385, pp. 164–180, 2005.

States are abstracted using a predicate base that contains reachability relations among program variables pointing into the heap. The computation of the abstract states and transition relation is precise and automatic and does not require the use of a theorem prover. Rather, we use a small model theorem to identify a truncated (small) finite-state version of the program whose abstraction is identical to the abstraction of the unbounded-heap version of the same program. The abstraction of the finite-state version is then computed by BDD techniques.

For proving liveness properties, we augment the original system by a well-founded ranking function, which is then abstracted together with the system. Well-foundedness is abstracted into strong fairness (compassion). We show that, for a restricted class of programs (that still includes numerous interesting cases), the small model theorem can be applied to this joint abstraction.

We demonstrate the power of the ranking abstraction method and its advantages over direct use of ranking functions in a deductive verification of the same property, independent of its application to shape-analysis examples.

The method is illustrated on two examples, both using (singly) linked lists: List reversal and in-place sort. We show how various predicate abstractions can be used to establish various safety properties, and how, for each program, one of the abstractions can be augmented with a progress monitor to establish termination.

The paper is organized as follows. Section 2 describes the formal model of *fair transitions systems* and their finite heap version, *finite heap systems*. Section 3 has an overview of finitary abstraction and predicate abstraction. Section 4 deals with the symbolic computation of abstractions. It states and proves the small model property, and describes how to apply it to obtain abstract finite heap systems. Section 5 deals with proving liveness of heap systems. Sections 2–5 use a list reversal program as a running example. Section 6 presents a more involved example of a nested loop bubble sort, and shows its formal verification using the new method.

Related Work

The work in [16] presents a parametric framework for shape analysis that deals with the specification language of the shape analysis framework and the construction of the shape analyzer from the specification. A 2-value logic is used to represent concrete stores, and a 3-valued logic is used to represent abstract stores. Properties are specified by first-order formulae with transitive closure; these also describe the transitions of the system. The shape analyzer computes a fixed point of the set of equations that are generated from the analysis specification. The systems considered in [16] are more general than ours, e.g., we allow at most one "next pointer" for each node. Due to the restricted systems and properties we consider, we do not have to abstract the heap structure itself, and therefore our computation of the transition relation is precise. Moreover, their work does not handle liveness properties.

In [7], Dams and Namjoshi study shape analysis using predicate abstraction and model checking. Starting with shape predicates and a property, the method iteratively computes weakest preconditions to find more predicates and constructs abstract programs that are then model checked. As in the [16] framework, the abstraction computed in not precise. Some manual intervention is required to apply widening-like techniques and guide the system into convergence. This work, too, does not handle liveness.

There are several works studying logics for shape analysis. E.g., [5] present a decidable logic for reasoning about heap structures. No treatment of liveness is described.

Some related but less relevant works are [9, 8] that study concurrent garbage collection using predicate abstraction, [10] that study loop invariants using predicate abstraction, and [13] that calculates weakest preconditions for reachability. All these works do not apply shape analysis or use shape predicates.

2 The Formal Framework

In this section we present our computation model.

2.1 Fair Discrete Systems

As our computational model, we take a *fair discrete system* (FDS) $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- V — A set of *system variables*. A *state* of S provides a type-consistent interpretation of the variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.
- Θ — The *initial condition*: An assertion (state formula) characterizing the initial states.
- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements (assertions); A computation must include infinitely many states satisfying each of the justice requirements.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q \rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

For an assertion ψ, we say that $s \in \Sigma$ is a ψ-state if $s \models \psi$.

A *computation* of an FDS S is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, ...$, the state $s_{\ell+1}$ is an S-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1} \rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.
- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Compassion* – for every $\langle p, q \rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

2.2 Finite Heap Systems

To allow the automatic computation of abstractions, we place further restrictions on the systems we study, leading to the model of *finite heap systems* (FHS), that is essentially the model of bounded discrete systems of [2] specialized to the case of heap programs.

For brevity, we describe here a simplified two-type model; the extension for the general multi-type case is straightforward.

We allow the following data types parameterized by the positive integer h, intended to specify the heap size:

1. **bool**: boolean and finite-range scalars; With no loss of generality, we assume that all finite domain values are encoded as booleans.
2. **index**: $[0..h]$;
3. Arrays of the types **index** $\mapsto$ **bool** (**bool** array) and **index** $\mapsto$ **index** (**index** array).

We assume a signature of variables of all of these types. Constants are introduced as variables with reserved names. Thus, we admit the boolean constants **0** and **1**, and the **index** constant nil. An additional reserved-name variable is H : **index** whose value is always h.

We often refer to an element of type **index** as a *node*. If the interpretation of an **index** variable x in a state s is ℓ, then we say that in s, x *points to the node* ℓ. An **index** *term* is an **index** variable or an expression $Z[y]$, where Z is an **index** array and y is an **index** variable.

Atomic formulas are defined as follows:

- If x is a boolean variable, B is a **index** $\mapsto$ **bool** array, and y is an **index** variable, then x and $B[y]$ are atomic formulas.
- If t_1 and t_2 are **index** terms, then $t_1 = t_2$ is an atomic formula.
- A *Transitive closure* formula (*tcf*) of the form $Z^*(x_1, x_2)$, denoting that x_2 is Z-reachable from x_1, where x_1 and x_2 are **index** variables and Z is an **index** array.

A *restricted A-assertion* is a formula of the form $\forall \vec{y}.\psi(\vec{x}, \vec{y})$, where $\vec{y}$ is a list of **index** variables that do not include nil, and $\psi(\vec{x}, \vec{y})$ is a boolean combination of atomic formulas such that the only atomic formulas referring to a universally quantified y are of the forms $B[y]$, $y = u$, or $Z_1[y] = Z_2[y]$ under positive polarity. In particular, note that in restricted A-assertions, universally quantified variables may *not* occur in tcf's. As the initial condition Θ, the transition relation ρ, as well as the fairness requirements, we only allow restricted A-assertions.

The definition of restricted A-assertions allows for programs that manipulate heap elements strictly via a constant set of *reference* variables, which is in accordance with most programming languages. The set of operations that are allowed is however greatly restricted. For example, arithmetic operations are not allowed. While the present definition doesn't allow inequalities, it is not hard to extend it to support them.

Example 1 (List Reversal). Consider program LIST-REVERSAL in Fig. 1, which is a simple list reversal program. The array Nxt describes the pointer structure. We ignore the actual data values, but they can easily be added as **bool** type variables.

Fig. 2 describes the FHS corresponding to program LIST-REVERSAL. The expression *pres*(V_1) is an abbreviation for $\bigwedge_{v \in V_1}(v' = v)$, i.e., *pres*$(V_1)$ means that all the variables in V_1 are not changed by the transition. The expression *pres-array*(Nxt, U) is an abbreviation for $\forall u \in$ **index**$.u \notin U \rightarrow (Nxt'[u] = Nxt[u])$. Note that all the clauses in Fig. 2 are restricted assertions. The justice requirement states that as long as the program has not terminated, its execution continues.

$$
\begin{array}{l}
\quad H \;\; : \textbf{integer where } H = h \\
\quad x, y \;\; : [0..h] \textbf{ init } y = nil \\
\quad Nxt \;\; : \textbf{array } [0..h] \textbf{ of } [0..h] \\
\left[\begin{array}{l}
1 : \textbf{while } x \neq nil \textbf{ do} \\
2 : \quad\; (x, y, Nxt[x]) := (Nxt[x], x, y) \\
\quad \textbf{end} \\
3 :
\end{array}\right]
\end{array}
$$

Fig. 1. Program LIST-REVERSAL

$$
\begin{array}{ll}
V : & \left\{\begin{array}{ll}
H : & \textbf{integer} \\
x, y : & [0..h] \\
Nxt : & \textbf{array } [0..h] \textbf{ of } [0..h] \\
\pi : & [1..3]
\end{array}\right. \\
\Theta : & H = h \;\wedge\; \pi = 1 \;\wedge\; y = nil \\
\rho : & \left[\begin{array}{l}
\quad \pi = 1 \;\wedge\; x = nil \;\wedge\; \pi' = 3 \;\wedge\; \mathit{pres}(\{H, x, y\}) \;\wedge\; \mathit{pres\text{-}array}(Nxt, \emptyset) \\
\vee \;\; \pi = 1 \;\wedge\; x \neq nil \;\wedge\; \pi' = 2 \;\wedge\; \mathit{pres}(\{H, x, y\}) \;\wedge\; \mathit{pres\text{-}array}(Nxt, \emptyset) \\
\vee \;\; \pi = 2 \;\wedge\; x' = Nxt[x] \;\wedge\; y' = x \;\wedge\; Nxt'[x] = y \;\wedge\; \pi' = 1 \;\wedge \\
\quad\; \mathit{pres}(\{H\}) \;\wedge\; \mathit{pres\text{-}array}(Nxt, \{x\}) \\
\vee \;\; \pi = 3 \;\wedge\; \pi' = 3 \;\wedge\; \mathit{pres}(\{H, x, y, \pi\}) \;\wedge\; \mathit{pres\text{-}array}(Nxt, \emptyset)
\end{array}\right] \\
\mathcal{J} : & \{\pi \neq 1, \pi \neq 2\} \\
\mathcal{C} : & \emptyset
\end{array}
$$

Fig. 2. FHS for Program LIST-REVERSAL

3 Abstraction

We fix an FHS $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ whose set of states is Σ for this section.

3.1 Finitary Abstraction

The material here is an overview of (a somewhat simplified version of) [12]. See there for details.

An *abstraction* is a mapping $\alpha \colon \Sigma \to \Sigma_A$ for some set Σ_A of *abstract states*. The abstraction α is *finitary* if the set of abstract states Σ_A is finite. We focus on abstractions that can be represented by a set of equations of the form $u_i = E_i(V)$, $i = 1, \ldots, n$, where the E_i's are assertions over the concrete variables (V) and $\{u_1, \ldots, u_n\}$ is the set of *abstract variables*, denoted by V_A. Alternatively, such α can be expressed by:

$$V_A = \mathcal{E}_\alpha(V)$$

For an assertion $p(V)$, we define its abstraction by:

$$\alpha(p) \colon \;\; \exists V. (V_A = \mathcal{E}_A(V) \;\wedge\; p(V))$$

The semantics of $\alpha(p)$ is $\|\alpha(p)\| = \{\alpha(s) \mid s \in \|p\|\}$. Note that $\|\alpha(p)\|$ is, in general, an over-approximation – an abstract state is in $\|\alpha(p)\|$ iff *there exists* some concrete

p-state that is abstracted into it. An assertion $p(V, V')$ over both primed and unprimed variables is abstracted by:

$$\alpha(p): \quad \exists V, V'.(V_A = \mathcal{E}_A(V) \wedge V'_A = \mathcal{E}_A(V') \ \wedge \ p(V, V'))$$

The assertion p is said to be *precise with respect to the abstraction* α if $\|p\| = \alpha^{-1}(\|\alpha(p)\|)$, i.e., if two concrete states are abstracted into the same abstract state, they are either both p-states, or they are both $\neg p$-states. For a temporal formula ψ in positive normal form (where negation is applied only to state assertions), ψ^α is the formula obtained by replacing every maximal state sub-formula p in ψ by $\alpha(p)$. The formula ψ is said to be *precise with respect to* α if each of its maximal state sub-formulas are precise with respect to α.

In all cases discussed in this paper, the formulae are precise with respect to the relevant abstractions. Hence, we can restrict to the over-approximation semantics.

The α*-abstracted version of* S is the system

$$S^\alpha \ = \ \langle V_A, \alpha(\Theta), \alpha(\rho), \bigcup_{J \in \mathcal{J}} \alpha(J), \ \bigcup_{(p,q) \in \mathcal{C}} (\alpha(p), \alpha(q)) \rangle$$

From [12] we derive the soundness of finitary abstraction:

Theorem 1. *For a system S, abstraction α, and a positive normal form temporal formula ψ:*

$$S^\alpha \models \psi^\alpha \qquad \Longrightarrow \qquad S \models \psi$$

Thus, if an abstract system satisfies an abstract property, then the concrete system satisfies the concrete property.

3.2 Predicate Abstraction

Predicate abstraction is an instance of finitary abstraction where the abstract variables are boolean. Following [15], an initial predicate abstraction is chosen as follows: Let $\mathcal{P}$ be the (finite) set of atomic state formulas occurring in ρ, Θ, $\mathcal{J}$, $\mathcal{C}$ and the concrete formula ψ that refer to non-control and non-primed variables. Then the abstraction α is the set of equations $\{B_p = p : p \in \mathcal{P}\}$. The formula ψ^α is then checked over S^α producing either a confirmation that $S^\alpha \models \psi^a$ or a counterexample. In the former case, the process terminates concluding that $S \models \psi$. Else, the counterexample produced is concreticized and checked whether it is indeed a feasible S-trace. If so, the process terminates concluding that $S \not\models \psi$. Otherwise, the concrete trace implies a refinement α' of α under which the abstract error trace is infeasible. The process repeats (with a') until it succeeds – ψ is proven to be valid or invalid – or the refinement reaches a fixpoint, in which case the process fails. See [6, 3, 4] for discussion of the iterated abstraction refinement method.

We close this section by demonstrating the process of predicate abstraction on program LIST-REVERSAL. In the next section we show how to automatically compute the abstraction.

Example 2 (List Reversal Abstraction). Consider program LIST-REVERSAL of Example 1. One of the safety properties one wishes to prove is that no elements are removed from the list, i.e., that every element initially reachable from x is reachable from y upon termination. This property can be expressed by:

$$\forall t.(\pi = 1 \wedge t \neq nil \wedge Nxt^*(x,t)) \rightarrow \Box(\pi = 3 \rightarrow Nxt^*(y,t)) \tag{1}$$

We augment the program with a generic variable t, which is a variable whose initial value is unconstrained and remains fixed henceforth. Then validity of Formula (1) reduces to the validity of:

$$(\pi = 1 \wedge t \neq nil \wedge Nxt^*(x,t)) \rightarrow \Box(\pi = 3 \rightarrow Nxt^*(y,t)) \tag{2}$$

Following the above discussion, to prove the safety property of Formula (2), the set $\mathcal{P}$ consists of $x = nil$, $t = nil$, $Nxt^*(x,t)$, and $Nxt^*(y,t)$, which we denote as the abstract variables x_nil, t_nil, r_xt, and r_yt respectively.

The abstract program is ABSTRACT-LIST-REVERSAL, shown in Fig. 3, and the abstract property corresponding to Formula (2) is:

$$\psi^\alpha : (\Pi = 1 \wedge \neg t\_nil \wedge r\_xt) \rightarrow \Box(\Pi = 3 \rightarrow r\_yt)$$

where Π is the program counter of the abstract program.

x_nil, t_nil, r_xt, r_yt : **bool**
init $x\_nil = t\_nil = \mathbf{0}, r\_xt = \mathbf{1}, r\_yt = t\_nil$

1 : **while** $\neg x\_nil$ **do**
2 :
$(r\_xt, r\_yt) :=$ **case**
$\neg r\_xt \wedge \neg r\_yt$: $(\mathbf{0},\mathbf{0})$
$\neg r\_xt \wedge r\_yt$: $\{(\mathbf{0},\mathbf{1}),(\mathbf{1},\mathbf{1})\}$
otherwise : $\{(\mathbf{0},\mathbf{1}),(\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{1})\}$
esac
$x\_nil :=$ **if** r_xt **then 0 else** $\{\mathbf{0},\mathbf{1}\}$
end
3 :

Fig. 3. Program ABSTRACT-LIST-REVERSAL

It is now left to check whether $S^\alpha \models \psi^\alpha$, which can be done, e.g., using a model checker. Here, the initial abstraction is precise enough, and program ABSTRACT-LIST-REVERSAL satisfies ψ^α. In Section 6 we present a more challenging example requiring several iterations of refinement.

4 Symbolic Computation of Abstractions

This section describes a methodology for symbolically computing an abstraction of an FHS. The methodology is based on a small model property, that establishes that satisfiability of a restricted assertion can be checked on small instantiation of a system.

Let $\mathcal{V}$ be a *vocabulary* of typed variables, whose types are taken from the restricted type system allowed in an FHS. A *model* M for $\mathcal{V}$ consists of the following elements:

- A positive integer $h > 0$.
- For each boolean variable $b \in \mathcal{V}$, a boolean value $M[b] \in \{\mathbf{0}, \mathbf{1}\}$. It is required that $M[\mathbf{0}] = \mathbf{0}$ and $M[\mathbf{1}] = \mathbf{1}$.
- For each **index** variable $x \in \mathcal{V}$, a natural value $M[x] \in [0..h]$. It is required that $M[nil] = 0$ and $M[H] = h$.
- For each boolean array $B \in \mathcal{V}$, a boolean function $M[B] : [0..h] \mapsto \{\mathbf{0}, \mathbf{1}\}$.
- For each **index** array $Z \in \mathcal{V}$, a function $M[Z] : [0..h] \mapsto [0..h]$.

We define the *size* of model M to be $h + 1$. Let $\varphi = \forall \vec{y}.\psi(\vec{x}, \vec{y})$ be a restricted A-assertion, where $\vec{x}$ is the set of free variables appearing in φ. For a given $\vec{x}$-model M, we can evaluate the formula φ over the model M. Model M is called a *satisfying model* for φ if $M \models \varphi$. An **index** term $t \in \{x, Z[x]\}$ is called a *free term* in φ. Let $\mathcal{T}_\varphi$ denote the set consisting of the term nil and all free terms which occur in formula φ.

A model M is called a *Z-uniform model* (*uniform model* for short), if for every $k \in [0..h]$ and every **index** arrays Z_1 and Z_2 such that $M[Z_1](k) = k_1$ and $M[Z_2](k) = k_2$ for $k_1 \neq k_2$, then k and at least one of k_1 or k_2 are M-interpretations of a free term belonging to $\mathcal{T}_\varphi$. A restricted A-assertion is called a Z-uniform assertion (uniform assertion for short) if all its models are Z-uniform. For example, assertion ρ of Fig. 2 is uniform where Z_1 and Z_2 are the arrays Nxt and Nxt'. From now on, we will restrict our attention to uniform assertions and their models. This restriction is justified since in all programs we are studying here, every pointer that is being updated is assigned a value of a variable or a free term, e.g., $Nxt'[x] = y$ or $Nxt'[y] = Nxt[yn]$ (though the value of the pointer before the assignment is not necessarily pointed to by any variable).

The following theorem states that if φ has a satisfying model, then it has a small satisfying model. The theorem is a variant of a similar one stated originally in [14].

Theorem 2 (Small model property). *Let $\varphi : \forall \vec{y}.\psi$ be a uniform restricted A-assertion and $\mathcal{T}$ be a set of free terms containing $\mathcal{T}_\varphi$. Then φ has a satisfying model iff it has a satisfying model of size not exceeding $|\mathcal{T}| + 1$.*

Proof. Let M be a satisfying model of size exceeding $|\mathcal{T}| + 1$. We will show that M can be reduced to a smaller satisfying model $\overline{M}$ whose size does not exceed $|\mathcal{T}| + 1$.

Let $0 = n_0 < \cdots < n_m$ be all the distinct values that model M assigns to the terms in $\mathcal{T}$. Obviously, $m < |\mathcal{T}|$. Let d be the minimal value in $[0..h]$ which is different from each of the n_i's. Define a mapping $\gamma\colon [0..h] \to [0..m]$ as follows:

$$\gamma(u) = \begin{cases} i & \text{if } u = n_i \\ m{+}1 & \text{otherwise} \end{cases}$$

We define the model $\overline{M}$ as follows:

- $\overline{h} = m{+}1$.
- $\overline{M}[x] = M[x]$ for each boolean variable $x \in \mathcal{T}$.
- $\overline{M}[u] = \gamma(M[u])$, for each free **index** variable $u \in \mathcal{T}$.

- $\overline{M}[B] = \lambda i.$**if** $i \leq m$ **then** $M[B](n_i)$ **else** $M[B](d)$, for each boolean array $B \in \mathcal{T}$.
- Finally consider an **index** array $Z \in \mathcal{T}$. We let $\overline{M}[Z](m{+}1) = m{+}1$. For $i \leq m$ let $v = M[Z](n_i)$. If some $n \in \{n_0, \ldots, n_m\}$ is Z-reachable from v in M, let n_j, $j \leq m$, be the "Z-closest" to v, and then $\overline{M}[Z](i) = j$. Otherwise, $\overline{M}[Z](i) = m{+}1$.

Concerning the last clause in the definition, note that if n_j is "Z_1-closest" to v then, due to uniformity, it is also the "Z_2-closest" to v, for every Z_2.

It remains to show that $\overline{M} \models \varphi$ under the assumption that $M \models \varphi$. The proof of this claim is presented in Appendix A. □

For example, consider a formula φ and a set $\mathcal{T} = \{nil, v_1, v_2, v_3\}$ that includes $\mathcal{T}_\varphi$. Let M be a uniform model with $h = 7;\ M[v_1] = 1; M[v_2] = 3, M[v_3] = 5, M[Nxt] = [6, 6, 7, 5, 5, 5, 7]$. Then, according to the construction, $\overline{h} = 4; \overline{M}[nil] = 0; \overline{M}[v_1] = 1; \overline{M}[v_2] = 2; \overline{M}[v_3] = 3; \overline{M}[Nxt] = [3, 4, 3, 4]$.

Given a restricted A-assertion φ and a positive integer h_0, we define the h_0-*bounded* version of φ, denoted $\lfloor \varphi \rfloor_{h_0}$, to be the conjunction $\varphi \wedge (H \leq h_0)$. Theorem 2 can be interpreted as stating that φ is satisfiable iff $\lfloor \varphi \rfloor_{|\mathcal{T}|}$ is satisfiable.

Next, we would like to extend the small model theory to the computation of abstractions. Consider first the case of a restricted A-assertion φ which only refers to unprimed variables. As explained in Subsection 3.1, the abstraction of φ is given by $\alpha(\varphi) = \exists V(V_{_A} = \mathcal{E}_{_A}(V) \wedge \varphi(V))$. Assume that the set of (finitely many combinations of) values of the abstract system variables $V_{_A}$ is $\{U_1, \ldots, U_k\}$. Let *sat*(φ) be the subset of indices $i \in [1..k]$, such that $U_i = \mathcal{E}_\alpha(V) \wedge \varphi(V)$ is satisfiable. Then, it is obvious that the abstraction $\alpha(\varphi)$ can be expanded into

$$\alpha(\varphi)(V_{_A}) = \bigvee_{i \in \mathit{sat}(\varphi)} (V_{_A} = U_i) \tag{3}$$

Next, let us consider the abstraction of $\lfloor \varphi \rfloor_{|\mathcal{T}|}$, where $\mathcal{T}$ consists of all free terms in φ and $\mathcal{E}_\alpha(V)$ and the variable H, i.e. all the free terms in the assertion $U_i = \mathcal{E}_\alpha(V) \wedge \varphi(V) \wedge (H \leq h_0)$. Our reinterpretation of Theorem 2 implies that *sat*$(\lfloor \varphi \rfloor_{|\mathcal{T}|}) =$ *sat*(φ) which leads to the following theorem:

Theorem 3. *Let φ be an assertion which only refers to unprimed variables, $\alpha : V_{_A} = \mathcal{E}_{_A}(V)$ be an abstraction mapping, $\mathcal{T}$ be the set of free terms in the formula $(U_i = \mathcal{E}_A(V)) \wedge \varphi(V) \wedge (H \leq h_0)$, and $h_0 = |\mathcal{T}|$. Then*

$$\alpha(\varphi)(V_{_A}) \quad \sim \quad \alpha(\lfloor \varphi \rfloor_{h_0})(V_{_A})$$

Theorem 3 deals with assertions that do not refer to primed variables. It can be extended to the abstraction of an assertion such as the transition relation ρ. Recall that the abstraction of such an assertion involves a double application of the abstraction mapping, an unprimed version and a primed version. Thus, we need to consider the set of free terms in the formula $(U_i = \mathcal{E}_{_A}(V)) \wedge U_j = \mathcal{E}_{_A}(V') \ \wedge \ \rho(V, V')$ plus the variable H.

Next we generalize these results to entire systems. For an FHS $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ and positive integer h_0, we define the h_0-bounded version of S, denoted $\lfloor S \rfloor_{h_0}$, as

$\langle V \cup \{H\}, \lfloor\rho\rfloor_{h_0}, \lfloor\mathcal{J}\rfloor_{h_0}, \lfloor\mathcal{C}\rfloor_{h_0}\rangle$, where $\lfloor\mathcal{J}\rfloor_{h_0} = \{\lfloor J\rfloor_{h_0} \mid J \in \mathcal{J}\}$ and $\lfloor\mathcal{C}\rfloor_{h_0} = \{(\lfloor p\rfloor_{h_0}, \lfloor q\rfloor_{h_0}) \mid (p,q) \in \mathcal{C}\}$. Let h_0 be the maximum size of the sets of free terms for all the abstraction formulas necessary for computing the abstraction of all the components of S. Then we have the following theorem:

Theorem 4. *Let S be an* FHS*, α be an abstraction mapping, and h_0 the maximal size of the relevant sets of free terms as described above. Then the abstract system S^α is equivalent to the abstract system $\lfloor S\rfloor^\alpha_{h_0}$.*

We use TLV [1] to compute the abstract system $\lfloor S\rfloor^\alpha_{h_0}$. The only manual step in the process is the choice of the state predicates. As discussed in Section 3, the initial choice is usually straightforward. One of the attractive advantages of using a model checker for the abstraction is that it can be invisible – thus, the abstraction, and checking of the (abstract) property over it, can be done completely automatically, and the user need not see the abstract program, giving rise to the *method of invisible abstraction*. However, because of the need for refinement, the user may actually prefer to view the abstract program.

Example 3. Consider again program LIST-REVERSAL of Example 1. In Example 2 (of Section 3) we described its abstraction, which was manually derived. In order to obtain an automatic abstraction for the system whose set of free terms is $\mathcal{T} = \{nil, H, x, y, t, x', y', Nxt'[x]\}$, we bounded the system by $h_0 = 8$.

We compute the abstraction in TLV by initially preparing an input file describing the concrete truncated system. We then use TLV's capabilities for dynamically constructing and updating a model to construct the abstract system by separately computing the abstraction of the concrete initial condition, transition relation, and fairness requirements.

Having computed the abstract system, we check the safety property ψ^α, which, of course, holds. All code is in *http://www.cs.nyu.edu/acsys/shape-analysis*.

5 Liveness

5.1 Transition Abstraction

State abstraction often does not suffice to verify liveness properties and needs to be augmented with *transition abstraction*. Let $(\mathcal{D}, \succ)$ be a partially ordered well founded domain, and assume a *ranking function* $\delta\colon \Sigma \to \mathcal{D}$. Define a function *decrease* by:

$$decrease \;=\; \begin{cases} 1 & \delta \succ \delta' \\ 0 & \delta = \delta' \\ -1 & \text{otherwise} \end{cases}$$

Transition abstraction can be incorporated into a system by (synchronously) composing the system with a *progress monitor* [12], shown in Fig. 4. The compassion requirement corresponds to the well-foundedness of $(\mathcal{D}, \succ)$: the ranking cannot decrease infinitely many times without increasing infinitely many times. To incorporate this in a state abstraction α, we add the defining equation $dec_A = dec$ to α.

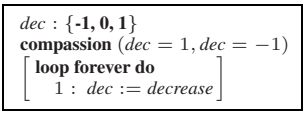

Fig. 4. Progress Monitor $M(\delta)$ for a Ranking δ

Example 4 (List Reversal Termination). Consider program LIST-REVERSAL and the termination property $\diamondsuit(\pi = 3)$. The loop condition $x \neq nil$ in line 1 implies that the set of nodes starting with x is a measure of progress. This suggests the ranking $\delta = \{i \mid Nxt^*(x, i)\}$ over the well founded domain $(2^{\mathbb{N}}, \supset)$. That is, the rank of a state is the set of all nodes which are currently reachable from x. As the computation progresses, this set loses more and more of its members until it becomes empty. Using a sufficiently precise state abstraction, one can model check that the abstract property $\diamondsuit(\Pi = 3)$ indeed holds over the program.

Just like the case of predicate abstraction, we lose nothing (except efficiency) by adding potentially redundant rankings. The main advantage here over direct use of ranking functions within deductive verification is that one may contribute as many elementary ranking functions as one wishes. Assuming a finitary abstraction, it is then left to the model-checker to sort out their interaction and relevance. To illustrate this, consider the program NESTED-LOOPS in Fig. 5. The statements $x := ?, y := ?$ in lines 0 and 2 denote assignments of a random natural to x and y. Due to this unbounded non-determinism, a deductive termination proof of this program needs to use a ranking function ranging over lexicographic triplets, whose core is $(\pi = 0, x, y)$. With augmentation, however, one need only provide the rankings $\delta_1 : x$ and $\delta_2 : y$.

5.2 Computing the Augmented Abstraction

We aim to apply symbolic abstraction computation of Section 4 to systems augmented with progress monitors. However, since progress monitors are not limited to restricted A-assertions, such systems are not necessarily FHS's. Thus, for any ranking function δ, one must show that Theorem 4 is applicable to such an extended form of FHS's. Since all assertions in the definition of an augmented system, with the exception of the transition relation, are restricted A-assertions, we need only consider the augmented transition relation $\rho \wedge \rho_\delta$, where ρ is the unaugmented transition relation and ρ_δ is defined as $dec' = decrease$. Let $\mathcal{T}$ be a set consisting of all free terms in the assertions $\rho \wedge \rho_\delta$,

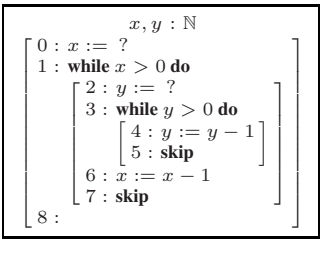

Fig. 5. Program NESTED-LOOPS

$\mathcal{E}_\alpha(V)$, and $\mathcal{E}_\alpha(V')$, as well as the variable H. Then Theorem 4 holds if it is the case that

$$sat(\lfloor \rho \ \wedge \ \rho_\delta \rfloor_{|\mathcal{T}|}) = sat(\rho \ \wedge \ \rho_\delta) \tag{4}$$

Since proving Formula (4) for an arbitrary ranking is potentially a significant manual effort, we specifically consider the following commonly used ranking functions over the well founded domain $(2^{\mathbb{N}}, \supset)$:

$$\delta_1(x) = \{i \mid Nxt^*(x,i)\} \tag{5}$$

$$\delta_2(x,y) = \{i \mid Nxt^*(x,i) \ \wedge \ Nxt^*(i,y)\} \tag{6}$$

In the above, x, y are **index** variables, and Nxt is an **index** array. Ranking δ_1 is used to measure the progress of a forward moving pointer x, while ranking δ_2 is used to measure the progress of pointers x and y toward each other. Throughout the rest of this section we assume that the variables x and y appearing in δ_1 or δ_2 are free terms in the unaugmented transition relation.

In order to extend the small model property to cover transition relations of the form ρ_δ we impose stronger conditions on the set of terms $\mathcal{T}$. A term set $\mathcal{T}$ is said to be *history closed* if for every term of the form $Nxt[x]$, $Nxt'[x] \in \mathcal{T}$ only if $Nxt[x] \in \mathcal{T}$. From now on, we restrict to history-closed term sets. Note that history closure implies a stronger notion of uniformity as follows: For any model M and nodes k, k_1, k_2, if $M[Nxt](k) = k_1 \neq k_2 = M[Nxt'](k)$, then all of k, k_1, k_2 are pointed to by terms in $\mathcal{T}$.

The following theorem, whose proof is in Appendix B, establishes the soundness of our method for proving liveness for the two ranking functions we consider.

Theorem 5. *Let S be an unaugmented* FHS *with transition relation ρ, δ_i be a ranking with $i \in \{1,2\}$, M be a uniform model satisfying $\rho \ \wedge \ \rho_\delta$, $\mathcal{T}$ be a history-closed term set containing the variable H and the free* **index** *terms in the assertions $\rho \wedge \rho_\delta$, $\mathcal{E}_\alpha(V)$, and $\mathcal{E}_\alpha(V')$, and $\overline{M}$ be the appropriate reduced model of size $h_0 = |\mathcal{T}|$.*

Then $\overline{M} \models \rho_{\delta_i}$ only if $M \models \rho_{\delta_i}$.

Example 5 (List Reversal Termination, concluded). In Example 4 we propose the ranking δ_1 to verify termination of program LIST-REVERSAL. From the Theorem 5 it follows that there is a small model property for the augmented program. The bound of the truncated system, according to Theorem 4, is

$$h_0 = |\mathcal{T}| = |\{H, nil, x, y, x', y', Nxt'[x], Nxt[x]\}| = 8$$

We have computed the abstraction, and proved termination of LIST-REVERSAL using TLV.

6 Bubble Sort

We present our experience in verifying a bubble sort algorithm on acyclic, singly-linked lists. The program is given in Fig. 6. The requirement of acyclicity is expressed in the initial condition $Nxt^*(x, nil)$ on the array Nxt. In Subsection 6.1 we summarize the proof of some safety properties. In Subsection 6.2 we discuss issues of computational efficiency, and in Subsection 6.3 we present a ranking abstraction for proving termination.

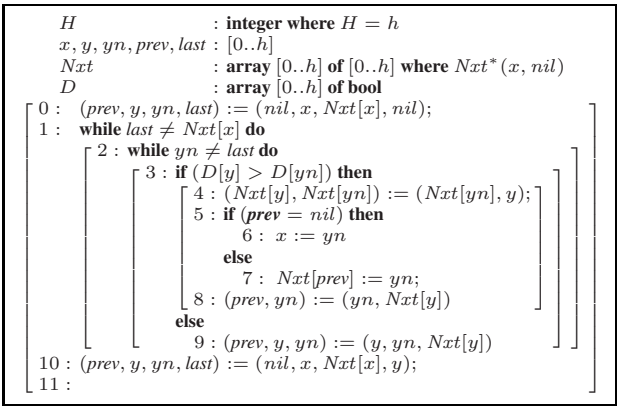

Fig. 6. Program BUBBLE SORT

6.1 Safety

Two safety properties of interest are preservation and sortedness, expressed as follows:

$$\forall t.(\pi = nil \ \wedge \ t \neq \mathit{nil} \ \wedge \ Nxt^*(x,t)) \rightarrow \Box(Nxt^*(x,t)) \tag{7}$$

$$\forall t, s.(\pi = 11 \ \wedge \ Nxt^*(x,t) \ \wedge \ Nxt^*(t,s)) \Rightarrow D[t] \leq D[s] \tag{8}$$

As in Example 2 we augment the program with a generic variable for each universal variable. The initial abstraction consists of predicates collected from atomic formulas in properties (7) and (8) and from conditions in the program. These predicates are

$$\mathit{last} = Nxt[x],\ yn = \mathit{last},\ D[y] > D[yn],\ \mathit{prev} = nil,\ t = nil,$$
$$Nxt^*(x, nil),\ Nxt^*(x,t),\ Nxt^*(t,s),\ D[t] \leq D[s]$$

This abstraction is too coarse for either property, requiring several iterations of refinement. Since we presently have no heuristic for refinement, new predicates must be derived manually from concretized counterexamples. In shape analysis typical candidates for refinement are reachability properties among program variables that are not expressible in the current abstraction. For example, the initial abstraction cannot express any nontrivial relation among the variables x, last, y, yn, and *prev*. Indeed, our final abstraction includes, among others, the predicates $Nxt^*(x, \mathit{prev})$ and $Nxt^*(yn, \mathit{last})$. In the case of *prev, y*, and *yn*, it is sufficient to use 1-step reachability, which is more efficiently computed. Hence we have the predicates $Nxt[\mathit{prev}] = y$ and $Nxt[y] = yn$.

6.2 Optimizing the Computation

When abstracting BUBBLE SORT, one difficulty, in terms of time and memory, is in computing the BDD representation of the abstraction mapping. This becomes apparent as the abstraction is refined with new graph reachability predicates. Naturally, computing the abstract program is also a major bottleneck.

One optimization technique used is to compute a series of increasingly more refined (and complex) abstractions $\alpha_1, \ldots, \alpha_n$, with α_n being the desired abstraction.

For each $i = 1, \ldots, n-1$, we abstract the program using α_i and compute the set of abstract reachable states. Let φ_i be the concretization of this set, which represents the strongest invariant expressible by the predicates in α_i. We then proceed to compute the abstraction according to α_{i+1}, while using the invariant φ_i to limit the state space. This technique has been invaluable in limiting state explosion, almost doubling the size of models we have been able to handle.

6.3 Liveness

Proving termination of BUBBLE SORT is more challenging than that of LIST-REVERSAL due to the nested loop. While a deductive framework would require constructing a global ranking function, the current framework requires only to identify individual rankings of each loop. Therefore we examine both loops independently, specifically their exit conditions.

The outer loop condition ($\mathit{last} \neq \mathit{Nxt}[x]$) implies that "nearness" of *last* to x is a measure of progress. We conjecture that after initialization, subsequent assignments advance *last* "backward" toward x. This suggests the ranking δ_2 defined in Subsection 5.2. As for the inner loop, it iterates while $yn \neq \mathit{last}$. We conjecture that yn generally progresses "forward" toward the list tail. This suggests the ranking δ_1 from Subsection 5.2.

We use δ_1 and δ_2 as a ranking augmentation, as well as a version of state abstraction described in Subsection 6.1 that omits predicates related to generic variables.

7 Conclusion

We have shown an approach for combining augmentation and predicate abstraction with model-checking, for the purpose of performing shape analysis without explicit representation of heap shapes. Using a small model property as a theoretical basis, we are able to use the model-checker in a role traditionally relegated to external decision procedures. Consequently, the complete process, from abstraction to verification, is automatic and fully encapsulated in the model-checker. We have shown successful application of the method to two programs that perform destructive heap updates – a list reversal algorithm and a bubble sort algorithm on linked lists.

In the immediate future we plan to focus on optimization of the abstraction computation. One such direction is to integrate with a SAT-solver. Another natural direction is to generalize the model from singly-linked structures to trees and finite DAG's.

Acknowledgement

We would like to thank Scott Stoller who suggested to us that small model properties can be used for shape analysis.

References

1. A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.

2. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In CAV'01, pages 221–234. LNCS 2102, 2001.
3. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, 2002.
4. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.
5. M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, 1999.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
7. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 310–324. Springer-Verlag, 2003.
8. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 51. IEEE Computer Society, 2001.
9. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 160–171. Springer-Verlag, 1999.
10. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202. ACM Press, 2002.
11. N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
12. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
13. G. Nelson. Verifying Reachability Invariants of Linked Structures. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, pages 38–47, 1983.
14. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
16. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

A Proof of Claim in Theorem 2

To complete the proof of Theorem 2, we show that, with the given construction of $\overline{M}$, $\overline{M} \models \varphi$ under the assumption that $M \models \varphi$.

To interpret the formula φ over $\overline{M}$, we consider an arbitrary assignment $\overline{\eta}$ to the quantified variables $\vec{y}$ which assigns to each variable y a value $\overline{\eta}[y] \in [0..m{+}1]$. For compatibility, we pick an assignment η, which assigns an M-value to variable y, given by $\eta(y) = \textbf{if } \overline{\eta}(y) = i \textbf{ then } n_i \textbf{ else } d$. It remains to prove that $(\overline{M}, \overline{\eta}) \models \psi$ under the assumption that $(M, \eta) \models \psi$. For simplicity, we denote by M_η the joint interpretation (M, η) which interprets all quantified variables according to η and all other terms according to M. Similarly, let $\overline{M}_\eta$ denote the joint interpretation $(\overline{M}, \overline{\eta})$.

We list below several properties of the extended model $\overline{M}_\eta$

P1. For every boolean variable b, $\overline{M}_\eta[b] = M_\eta[b]$.
P2. For every boolean array B and variable $u \in \vec{x} \cup \vec{y}$, $\overline{M}_\eta[B[u]] = M_\eta[B[u]]$.
P3. If t is a free term, then $M_\eta[t] \in \{n_0, \ldots, n_m\}$ and $\overline{M}_\eta[t] = i$ iff $M_\eta[t] = n_i$.
P4. If t is a non-free term, such that $M_\eta[t] = n_i$ for some $i \leq m$, then $\overline{M}_\eta[t] = i$.
P5. If x_1 and x_2 are free variables then $\overline{M}_\eta \models Z^*(x_1, x_2)$ iff $M_\eta \models Z^*(x_1, x_2)$

Properties **P1–P3** are direct consequences of the definition of $\overline{M}_\eta$. Let us consider Property **P4**. For the case that $t = y$, then $M_\eta[y] = n_i$ iff $\eta[y] = n_i$ iff $\overline{\eta}[y] = i$ iff $\overline{M}_\eta[y] = i$. The other case is that $t = Z[y]$. By considering separately the cases that $M_\eta[y] = n_j$ and $M_\eta[y] = d$, we can show that $\overline{M}_\eta[Z[y]] = i$.

Property **P5** follows from the definition of $\overline{M}[Z]$ and the fact that $\overline{M}[Z](m{+}1) = m{+}1$, so that no spurious Z-chains through $m{+}1$ are generated by $\overline{M}$.

To prove $\overline{M} \models \varphi$, it is sufficient to show that each free atomic formula (i.e., a formula not referring to any of the $\vec{y}$ variables) is true in $\overline{M}_\eta$ iff it is true in M_η and, for each non-free atomic formula p, if $M_\eta \models p$ then $\overline{M}_\eta \models p$. The relaxation in the requirements about non-free atomic formulas stems from the fact that they always appear under positive polarity in φ. We consider in turn each type of an atomic formula p that may occur in ψ.

For the case that p is a boolean variable b or a boolean term $B[u]$, the claim follows from properties **P1**, **P2**.

Next, consider the case that p is the formula $t_1 = t_2$, where t_1 and t_2 are free **index** terms. According to Property **P3**, the values of t_1 and t_2 are equal in $\overline{M}_\eta$ iff they are equal in M_η.

Turning to the case that p is the formula $y = u$, where y is a quantified variable, the correspondence between the assignments $\overline{\eta}$ and η, guarantee that this equality holds in $\overline{M}_\eta$ iff it holds in M_η.

Finally, let us consider the non-free atomic formula $Z_1[y] = Z_2[y]$, and the case that $M_\eta \models Z_1[y] = Z_2[y]$. For the case that $M_\eta[y] = d$, the equality holds in $\overline{M}_\eta$ since $\overline{M}_\eta[Z_1](m{+}1) = \overline{M}_\eta[Z_2](m{+}1) = m{+}1$. Otherwise, let $M_\eta[y] = n_i$, and let $n = M_\eta[Z_1](n_i) = M_\eta[Z_2](n_i)$. If $n = n_j$ then $\overline{M}_\eta[Z_1(y)] = \overline{M}_\eta[Z_2(y)] = j$. Otherwise, $\overline{M}_\eta[Z_1(y)]$ and $\overline{M}_\eta[Z_2(y)]$ are both equal to j, where n_j is the closest n_k which is Z_1-reachable (equivalently, Z_2-reachable) from n, if there exist one. If no such n_k is Z_1-reachable from n, then $\overline{M}_\eta[Z_1(y)] = \overline{M}_\eta[Z_2(y)] = m + 1$.

The case of atomic formulas of the form $Z^*(x_1, x_2)$ follows from Property **P5**. □

B Proof of Theorem 5

Theorem 5 claims that $M \models \rho_{\delta_i}$ implies $\overline{M} \models \rho_{\delta_i}$, where ρ_{δ_i} is defined as $dec' = decrease$. We prove the claim for a ranking δ_1 of the form $\delta_1(x) = \{i \mid Nxt^*(x, i)\}$ specified in equation (5). The case of δ_2 is justified by similar arguments.

The evaluation of δ_1 in M, written $M[\delta_1]$, is the set $\{i \mid M[Nxt^*](M[x], i)\}$, i.e, the set of all M-nodes which are reachable from $M[x]$ by $M[Nxt]$-links. The evaluation of δ_1 in $\overline{M}$ and of δ_1' in M and $\overline{M}$ are defined similarly.

First note the following property of terms in $\mathcal{T}$: It follows directly from Property **P5** of Theorem 2 that, for any term t in $\mathcal{T}$ and $\delta \in \{\delta_1, \delta_1'\}$, $M[t] \in M[\delta]$ iff $\overline{M}[t] \in \overline{M}[\delta]$.

To prove the claim it is enough to show that both properties $\delta_1 \supset \delta_1'$ and $\delta_1 = \delta_1'$ are satisfied by M iff they are satisfied by $\overline{M}$. First assume $M \models \delta_1 \supset \delta_1'$. It is easy to show that $\delta_1 \supseteq \delta_1'$ is satisfied in $\overline{M}$. This is true since by construction, any node $i \in [0 \ldots N]$ is pointed to in $\overline{M}$ by a term in $\mathcal{T}$, and membership in δ_1, δ_1' is preserved for such terms.

It is left to show that $\delta_1 \neq \delta_1'$ is satisfied in $\overline{M}$. We do this by identifying a term in $\mathcal{T}$ that M interprets as a node in $M[\delta_1] - M[\delta_1']$. Such a term must point to a node in $\overline{M}$ that is a member of $\overline{M}[\delta_1] - \overline{M}[\delta_1']$. To perform this identification, let ℓ be a node in $M[\delta_1] - M[\delta_1']$. Let $M[x] = r_1, \ldots, r_q = \ell$ denote the shortest Nxt-path in M from the node $M[x]$ to ℓ, i.e., for $i = 1, \ldots, q{-}1$, $M[Nxt](r_i) = r_{i+1}$. Let j be the maximal index in $[1..q]$ such that $r_j \in \{n_0, \ldots, n_m\}$, i.e., r_j is the M-image of some term $t \in \mathcal{T}$. If $r_j \notin M[\delta_1']$, our identification is complete.

Assume therefore that $r_j \in M[\delta_1']$. According to our construction, there exists an $M[Nxt]$-chain connecting r_j to ℓ, proceeding along $r_{j+1}, r_{j+2}, \ldots, \ell$. Consider the chain of $M[Nxt']$-links starting from r_j. At one of the intermediate nodes: $r_j, \ldots, \ell$, the $M[Nxt]$-chain and the $M[Nxt']$-chain must diverge, otherwise ℓ would also belong to $M[\delta_1']$. Assume that the two chains diverge at r_k, for some $j \leq k < q$. Then, according to strong uniformity (implied by history closure), $r_{k+1} \in \{n_0, \ldots, n_m\}$, contradicting the assumed maximality of j.

In the other direction, assume that $\overline{M}$ satisfies $\delta_1 \supset \delta_1'$. We first show that M satisfies $\delta_1 \supseteq \delta_1'$. Let n be a node in $M[\delta_1']$, and consider a Nxt'-path from $M[x']$ to n in M. Let m be the ancestor nearest to n that is pointed to by a term in $\mathcal{T}$. From Theorem 2 it follows that $m \in M[\delta_1]$. The fact $n \in M[\delta_1]$ follows by induction on path length from m to n and by uniformity of M and $\overline{M}$. Therefore $M[\delta_1] \supseteq M[\delta_1']$. We now show that M satisfies $\delta_1 \supset \delta_1'$. Let j be a node such that $j \in \overline{M}[\delta_1] - \overline{M}[\delta_1']$. By construction, j is pointed to in $\overline{M}$ by a term t or $j = m{+}1$. In the first case, t points to a node n_j in M, such that $n_j \in M[\delta_1] - M[\delta_1']$, and we are done. In the latter case, from construction we have $\overline{M}[Nxt](m{+}1) = \overline{M}[Nxt'](m{+}1) = m{+}1$. Therefore, if $m{+}1$ is not Nxt'-reachable from $\overline{M}[x']$, there must exist a node i in $\overline{M}[\delta_1] - \overline{M}[\delta_1']$ such that $\overline{M}[Nxt](i) \neq \overline{M}[Nxt'](i)$. By uniformity, i must be pointed to in $\overline{M}$ by a term in $\mathcal{T}$. From Theorem 2 there exists a corresponding node in M.

It is left to show that $M \models (\delta_1 = \delta_1')$ iff $\overline{M} \models (\delta_1 = \delta_1')$. This is done by similar arguments.

The case of δ_2, while not presented here, is shown by generalization: While δ_1 involves nodes reachable from a single distinguished pointer x, δ_2 involves nodes on a path between x and a pointer y. Thus, given node ℓ satisfying some combination of properties of membership in δ_2, δ_2', we identify a node satisfying the same properties, that is also pointed to by a term in $\mathcal{T}$. Here, however, we consider not only distant ancestors of ℓ on the path from x, but also distant successors on the path to y. □