Shape Analysis of Single-Parent Heaps*

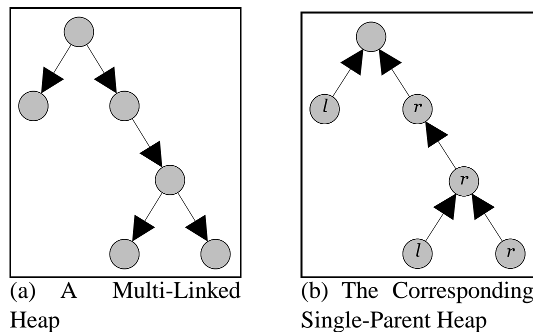
Ittai Balaban[1], Amir Pnueli[1 2], and Lenore D. Zuck[3]

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

Abstract. We define the class of *single-parent heap systems*, which rely on a singly-linked heap in order to model destructive updates on tree structures. This encoding has the advantage of relying on a relatively simple theory of linked lists in order to support abstraction computation. To facilitate the application of this encoding, we provide a program transformation that, given a program operating on a multi-linked heap without sharing, transforms it into one over a single-parent heap. It is then possible to apply shape analysis by predicate and ranking abstraction as in [2]. The technique has been successfully applied on examples with trees of fixed arity (balancing of and insertion into a binary sort tree).

1 Introduction

In [2] we propose a framework for shape analysis of singly-linked graphs based on a small model property of a restricted class of first order assertions with transitive closure. Extending this framework to allow for heaps with multiple links per node entails extending the assertional language and proving a stronger small model property. At this point, it is not clear whether such a language extension is decidable (see [10, 11] for relevant results).



(a) A Multi-Linked Heap

(b) The Corresponding Single-Parent Heap

Fig. 1. Multi-Linked to Single-Parent Heap Transformation

* This research was supported in part by ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

This paper deals with verification of programs that perform destructive updates of heaps consisting only of trees of bounded or unbounded arity, to which we refer as *multi-linked* heaps. We bypass the need to handle trees directly by transforming heaps consisting of multiple trees into structures consisting of singly-linked lists (possibly with shared suffixes). This is accomplished by "reversing" the parent-to-child edges of the trees populating the heap, as well as associating scalar data with nodes. We refer to the transformed heap as a *single-parent* heap. Fig. 1(a) and Fig. 1(b) together demonstrate the transformation of a multi-linked heap that consists of a binary tree to its single-parent counterpart. In the latter graph, edges are directed from children to parents, and each child is annotated with boolean information denoting whether it is a left or right child.

Verification of temporal properties of multi-linked heap systems can be performed as follows: Given a multi-linked system and a temporal property, the system and property are (automatically) transformed into their single-parent counterparts. Then, a counter-example-guided predicate- (and possibly ranking-) abstraction refinement method ([2, 3]) is applied. If a counter-example (on the transformed system) is produced, it is automatically mapped into a counter-example of the original (multi-linked) system.

The rest of this paper is organized as follows: After we discuss related work, we present the formal model in Section 2 and define predicate abstraction thereof. Section 3 defines systems over single-parent heaps and Section 4 describes their model reduction. Section 5 defines systems over multi-linked heaps, and Section 6 shows how to transform them to single-parent heap systems. We conclude in Section 7.

Related Work

Numerous frameworks have been suggested for analyzing singly-linked heaps, e.g., [15, 18, 6, 9, 7], all assuming that programs access heap cells solely by reachability from variables. This effectively disallows backward traversal, a necessary feature when reducing trees to singly-linked structures.

The correspondence between tree structures and singly-linked structures is the basis of the proof of decidability of first-order logic with one function symbol in [8]. More generally, the observation that complex data structures with regular properties can be reduced to simpler structures has been utilized in [14, 12, 16, 19]. However, it is not always straightforward to apply, and, to our knowledge, has not been applied in the context of predicate abstraction. Several assumptions that hold true in analysis of "conventional" programs over singly-linked heaps (e.g., C- or Pascal-programs), cannot be relied upon when reducing trees to lists. For example, the number of roots of the heap is no longer bounded by the number of program variables.

The use of path compression in heaps to prove small model properties of logics of linked structures, has been used before, e.g., in [5] and more recently in [2, 20]. Our work on parameterized systems relies on a small model theorem for checking inductiveness of assertions. The small model property there is similar to the one here with respect to *stratified data*. However, with respect to unstratified data (such as graphs), the work on parameterized systems suggests using logical instantiation as a heuristic (see, e.g., [1]), whereas here completeness is achieved using graph-theoretic methods.

2 The Formal Framework

In this section we present our computational model, as well as the method of predicate abstraction.

2.1 Fair Discrete Systems

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

- V — A set of *system variables*. A *state* of $\mathcal{D}$ 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 a $\mathcal{D}$-successor state $s' \in \Sigma$. We assume that every state has a ρ-successor.
- $\mathcal{J}$ — 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 *run* of an FDS $\mathcal{D}$ is a possibly infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is a $\mathcal{D}$-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 $s_{\ell+1}[v]$.

A *computation* of $\mathcal{D}$ is an infinite run that satisfies

- *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.

We say that a temporal property φ is *valid over* $\mathcal{D}$, denoted by $\mathcal{D} \models \varphi$, if for every computation σ of $\mathcal{D}$, $\sigma \models \varphi$. We are interested in *safety* properties, of the form $\square\, p$, and *progress* properties, of the form $\square\,(p \rightarrow \diamondsuit\, q)$, where p and q are state assertions. Since our methodology for verifying safety properties can be easily extended to verification of progress properties (along the lines of [3]), we restrict here to the former. Yet, we include the fairness requirements here for the sake of completeness, while they are only necessary when dealing with progress.

2.2 Predicate Abstraction

The material here is a summary of [13] and [2]. We fix an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ whose set of states is Σ. We consider a set of *abstract variables* $V_A = \{u_1, \ldots, u_n\}$ that range over finite domains. An *abstract state* is an interpretation that assigns to each variable u_i a value in the domain of u_i. We denote by Σ_A the (finite) set of all abstract states. An *abstraction mapping* is presented by a set of equalities

$$\alpha_{\mathcal{E}} : \quad u_1 = \mathcal{E}_1(V),\ \ldots,\ u_n = \mathcal{E}_n(V),$$

where each $\mathcal{E}_i$ is an expression over V ranging over the domain of u_i. The abstraction $\alpha_{\mathcal{E}}$ induces a semantic mapping $\alpha_{\mathcal{E}} : \Sigma \mapsto \Sigma_A$, from the states of $\mathcal{D}$ to the set of abstract states.

Usually, most of the abstract variables are boolean, and then the corresponding expressions $\mathcal{E}_i$ are predicates over V, which is why this type of abstraction is referred to as *predicate abstraction*. The abstraction mapping $\alpha_{\mathcal{E}}$ can be expressed succinctly by $V_A = \mathcal{E}(V)$.

Throughout the rest of the paper, when there is no ambiguity, we shall refer to $\alpha_{\mathcal{E}}$ simply as α. For an assertion $p(V)$, we define its α-abstraction (with some overloading of notation) by $\alpha(p): \ \exists V.(V_A = \mathcal{E}(V) \ \wedge \ p(V))$.

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 S is in $\|\alpha(p)\|$ iff *there exists* some concrete p-state that is abstracted into S. A bi-assertion $\beta(V, V')$ is abstracted by:

$$\alpha^2(p): \ \exists V, V'.(V_A = \mathcal{E}(V) \ \wedge \ V'_A = \mathcal{E}(V') \ \wedge \ \beta(V, V'))$$

See [2] for a discussion justifying the use of over-approximating abstractions in this setting. The abstraction of $\mathcal{D}$ by α is the system

$$\mathcal{D}^\alpha \ = \ \langle V_A, \alpha(\Theta), \alpha^2(\rho), \bigcup_{J \in \mathcal{J}} \alpha(J), \bigcup_{(p,q) \in \mathcal{C}} \langle \alpha(p), \alpha(q) \rangle \, \rangle$$

The soundness of predicate abstraction is derived from [13]:

Theorem 1. *For a system $\mathcal{D}$, abstraction α, and a temporal formula ψ:*

$$\mathcal{D}^\alpha \models \psi^\alpha \qquad \Longrightarrow \qquad \mathcal{D} \models \psi$$

3 Single-Parent Heaps

A *single-parent heap system* is an extension of the model of *finite heap systems* (FHS) of [2] specialized for representing trees. Such a system is parameterized by a positive integer h, which is the heap size. Some auxiliary arrays may be used to specify more complex structures (e.g., ordered trees). However, each node u has a single link to which we refer as its "parent," and denote it by $parent(u)$.

Example 1 (Tree Insertion). For example, we present in Fig. 2 a program that inserts a node into a binary sort tree rooted at a node r. If the data contained in node n is not already contained in the tree, then n is inserted as a new leaf. Otherwise the tree is not modified. Obviously, n is to be an isolated node, i.e., to be both a root and a leaf. To allow for the presentation of a sorted binary tree, we use an array ct (child-type) such that $ct[u]$ equals *left* or *right* if node u is, respectively, the left or right child of its parent. We also require that any two children of the same parent must have different child-types. In Subsection 3.2 we expand on the notion of sibling order. One may wish to show, for example, that program TREE-INSERT satisfies the following for every x:

$$\begin{aligned} \textit{no-loss} &: \textit{parent}^*(x,r) \rightarrow \Box\ \textit{parent}^*(x,r) \\ \textit{no-gain} &: x \neq n\ \wedge\ \neg\textit{parent}^*(x,r) \rightarrow \Box\ \neg\textit{parent}^*(x,r) \end{aligned}$$

```
r, t   : [1..h]                         init t = r
n      : [0..h]                         init n > 0
parent : array [0..h] of [0..h]         init parent[n] = parent[r] = 0 ∧ parent[0] = 0 ∧
                                             ∀u . parent[u] ≠ n
ct     : array [0..h] of {left, right}  init ∀i ≠ j . parent[i] = parent[j] ≠ 0 → ct[i] ≠ ct[j]
data   : array [0..h] of [1..k]
done   : bool                           init done = FALSE
  1 : while ¬done do
        2 : if data[n] = data[t] then
              3 : done := TRUE
        4 : elseif data[n] < data[t] then
              5 : if ∀j.parent[j] ≠ t ∨ ct[j] ≠ left then
                    6 : (parent[n], ct[n]) := (t, left)
                    7 : done := TRUE
                  else
                    8 : t := ε j . parent[j] = t ∧ ct[j] = left
        9 : elseif ∀j.parent[j] ≠ t ∨ ct[j] ≠ right then
              10 : (parent[n], ct[n]) := (t, right)
              11 : done := TRUE
            else
              12 : t := ε j . parent[j] = t ∧ ct[j] = right
  13 :
```

Fig. 2. Program TREE-INSERT inserts a new node n into a binary sort tree rooted at node r

The ϵ-expressions, $\epsilon\ j\ .\ cond$ in lines 8 and 12 denote "choose any node j that satisfies *cond*." For both statements in this program, it is easy to see that there is exactly one node j that meets *cond*. However, this is not always the case, and then such an assignment is interpreted non-deterministically. We also allow for universal tests, as those in lines 5 and 9, that test for existence of a particular node's left or right child. ▟

3.1 Unordered Single-Parent Heaps

We now formally define the class of single-parent heap systems. Let $h > 0$ be the *heap size*. We allow the following data types:

bool Variables whose values are boolean. With no loss of generality, we assume that all *finite domain* (unparameterized) values are encoded as **bool**s;

$$
\begin{array}{l}
\mathit{error} \wedge \mathit{error}' \wedge \mathit{presEx}(\mathit{error}) \\
\quad \vee \\
\neg \mathit{error} \wedge \\
\left[\begin{array}{l}
\quad\ \pi = 1 \wedge \neg \mathit{done} \wedge \pi' = 2 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 1 \wedge \mathit{done} \wedge \pi' = 13 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 2 \wedge \mathit{data}[t] = \mathit{data}[n] \wedge \pi' = 3 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 2 \wedge \mathit{data}[t] \neq \mathit{data}[n] \wedge \pi' = 4 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 3 \wedge \pi' = 1 \wedge \mathit{done}' \wedge \mathit{presEx}(\pi, \mathit{done}) \\
\vee\ \pi = 4 \wedge \mathit{data}[n] < \mathit{data}[t] \wedge \pi' = 5 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 4 \wedge \mathit{data}[t] \leq \mathit{data}[n] \wedge \pi' = 9 \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 5 \wedge \pi' = 6 \wedge (\forall j.\mathit{parent}[j] \neq t \vee \mathit{ct}[j] \neq \mathit{left}) \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 5 \wedge \pi' = 8 \wedge (\exists j.\mathit{parent}[j] = t \wedge \mathit{ct}[j] = \mathit{left}) \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 6 \wedge n = 0 \wedge \mathit{error}' \wedge \mathit{presEx}(\mathit{error}) \\
\vee\ \pi = 6 \wedge \pi' = 7 \wedge n \neq 0 \wedge \mathit{parent}'[n] = t \wedge \mathit{ct}'[n] = \mathit{left} \wedge \mathit{presEx}(\pi, \mathit{parent}[n], \mathit{ct}[n]) \\
\vee\ \pi = 7 \wedge \pi' = 1 \wedge \mathit{done}' \wedge \mathit{presEx}(\pi, \mathit{done}) \\
\vee\ \pi = 8 \wedge \pi' = 1 \wedge (\exists j\ .\ \mathit{parent}[j] = t \wedge \mathit{ct}[j] = \mathit{left} \wedge t' = j) \wedge \mathit{presEx}(\pi, t) \\
\vee\ \pi = 9 \wedge \pi' = 10 \wedge (\forall j.\mathit{parent}[j] \neq t \vee \mathit{ct}[j] \neq \mathit{right}) \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 9 \wedge \pi' = 12 \wedge (\exists j.\mathit{parent}[j] = t \vee \mathit{ct}[j] = \mathit{right}) \wedge \mathit{presEx}(\pi) \\
\vee\ \pi = 10 \wedge n = 0 \wedge \mathit{error}' \wedge \mathit{presEx}(\mathit{error}) \\
\vee\ \pi = 10 \wedge \pi' = 11 \wedge n \neq 0 \wedge \mathit{parent}'[n] = t \wedge \mathit{ct}'[n] = \mathit{right} \wedge \mathit{presEx}(\pi, \mathit{parent}[n], \mathit{ct}[n]) \\
\vee\ \pi = 11 \wedge \pi' = 1 \wedge \mathit{done}' \wedge \mathit{presEx}(\pi, \mathit{done}) \\
\vee\ \pi = 12 \wedge \pi' = 1 \wedge (\exists j\ .\ \mathit{parent}[j] = t \wedge \mathit{ct}[j] = \mathit{right} \wedge t' = j) \wedge \mathit{presEx}(\pi, t) \\
\vee\ \pi = 13 \wedge \pi' = 13 \wedge \mathit{presEx}(\pi)
\end{array}\right]
\end{array}
$$

Fig. 3. The transition relation of TREE-INSERT. The variable π represents the program counter.

index Variables whose value is in the range $[0..h]$;
index → bool arrays (**bool** arrays) that map heap elements to boolean values (such as ct above, where the value *left* is mapped to FALSE, and *right* to TRUE);
index → index arrays (**index** arrays), that describe the heap structure. We allow at most two **index** arrays, which we usually denote by $parent$ and $parent'$.

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 FALSE and TRUE, and the **index** constant 0. In order to have all functions in the model total, we define both **bool** and **index** arrays as having the domain **index**. A well-formed program should never assign a value to $Z[0]$ for any (**bool** or **index**) array Z. On the other hand, unless stated otherwise, all quantifications are taken over the range $[1..h]$.

We refer to **index** elements as *node*s. If in state s, the **index** variable x has the value ℓ, then we say that in s, *x points to the node ℓ*. An **index** *term* is the constant 0, an **index** variable, or an expression $Z[y]$, where Z is an **index** array and y is an **index** variable.

Atomic formulae are defined as follows:

- If x is a boolean variable, B is a **bool** array, and y is an **index** variable, then x and $B[y]$ are atomic formulae.
- 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.

We find it convenient to include "preservation statements" for each transition, that describe the variables that are not changed by the transition. There are two types of such statements:

1. Assertions of the form *pres*$(\{v_1, \ldots, v_k\}) = \bigwedge_{i=1}^{k} v'_i = v_i$ where all v_i's are scalar (**bool** or **index**) variables;
2. Assertions of the form *pres*$_H(\{a_1, \ldots, a_k\}) = \bigwedge_{i=1}^{k} \forall h \notin H \,.\, a'_i[h] = a_i[h]$ where all a_i's are arrays and H is a (possible empty) set of **index** variables. Such an assertion denotes that all but finitely many (usually a none or a single) entries of arrays indexed by certain nodes remain intact.

Note that preservation formulae are at most universal. We abuse notation and use the expression $presEx(v_1, \ldots, v_k)$ to denote the preservation of all variables, excluding the terms $v_1, \ldots, v_k$, which are either variables or array terms of the form $A[x]$.

Fig. 3 presents the transition relation associated with the program of Fig. 2. The implied encoding introduces an additional **bool** variable *error* which is set to TRUE whenever there is an attempt to assign a value to $A[0]$, for some array A. Consequently, the transitions corresponding to statements 6 and 10 set *error* to TRUE if $n = 0$, which is tested before assigning values to $parent[n]$ and to $ct[n]$.

A *restricted A-assertion* is either one of the following forms: $\forall y \,.\, Z[y] \neq u$, $\forall y \,.\, Z[y] \neq u \;\vee\; B[y]$, $\forall y \,.\, Z[y] \neq u \;\vee\; \neg B[y]$, *pres*$_H(Z)$, and *pres*$_H(B)$, where Z is an **index** array and B is a **bool** array, and H is a (possibly empty) set of **index** variables. A *restricted EA-assertion* is a formula of the form $\exists \vec{x} \,.\, \psi(\vec{u}, \vec{x})$, where $\vec{x}$ is a list of **index** variables, and $\psi(\vec{u}, \vec{x})$ is a boolean combination of atomic formulae and restricted A-assertions, where restricted A-assertions appear under positive polarity. Note that in restricted EA-assertions, universally quantified variables may *not* occur in tcf's. As the initial condition Θ we allow restricted EA-assertions, and in the transition relation ρ and fairness requirements we only allow restricted EA-assertions without tcf's. Properties of systems are restricted EA-assertions. Abstraction predicates are boolean combinations of atomic formulae and non-preservation universal formulae. This last restriction ensures that the language of abstraction predicates is closed under negation, an assumption needed during abstraction computation.

Note that restricted EA-assertions are more expressive than restricted A-assertions of [2] in that they allow, by means of existential and universal quantification, for traversal of trees in both directions.

3.2 Ordered Single-Parent Heaps

We now formalize the notion of order among siblings, as seen in Example 1. An *ordered* single-parent heap system is one that includes a distinguished ct : **index** $\rightarrow [1..k]$ array, for some constant k, that denotes for each heap node its place among its siblings. This allows the subtrees of a given root node to be distinguished by their ct order. We now extend the assertional language with a new type of atomic formula: For each $i \in [1..k]$, the formula $i\text{-}subtree_Z(x_1, x_2)$ denotes that x_1 is in the ith subtree of x_2, where x_1 and x_2 are **index** variables and Z is an **index** array. This is formally expressed by the formula

$$i\text{-}subtree_Z(x_1, x_2) : \; \exists u \,.\, Z[u] = x_2 \;\wedge\; ct[u] = i \;\wedge\; Z^*(x_1, u)$$

We support these predicates explicitly rather than as derived forms because, due to the transitive closure over a quantified variable, they would otherwise be outside of the

assertional language allowed for abstraction predicates. Throughout the paper, when the **index** array Z is apparent from the context, we use the short form i-*subtree*(x_1, x_2). For example, in the context of program TREE-INSERT of Example 1, the predicates *left-subtree* and *right-subtree* denote the left and right subtree relations among nodes of the $parent$ array, whereas *left-subtree*$'$ and *right-subtree*$'$ denote subtree relations among nodes of the $parent'$ array.

4 Computing Symbolic Abstractions of Single-Parent Heaps

We show how to symbolically compute the abstraction of a single-parent heap system by extending the methodology of [2]. That methodology is based on a small model property establishing that satisfiability of a restricted assertion is checkable on a small instantiation of a system. The main effort here is dealing with the extensions to the assertional language introduced for single-parent heap systems. For simplicity, it is assumed that all scalar values are represented by multiple boolean values.

Assume a vocabulary $\mathcal{V}$ of typed variables, as well as the *primed* version of said variables. Furthermore, assume that there is a single unprimed **index** array in $\mathcal{V}$ as well as a single primed one. These will be denoted throughout the rest of this section by $parent$ and $parent'$, respectively. A *model* M of *size* $h+1$ for $\mathcal{V}$ consists of:

- A positive integer $h > 0$;
- For each boolean variable $b \in \mathcal{V}$, a boolean value $M[b] \in \{\text{FALSE}, \text{TRUE}\}$. It is required that $M[\text{FALSE}] = \text{FALSE}$ and $M[\text{TRUE}] = \text{TRUE}$;
- For each **index** variable $x \in \mathcal{V}$, a value $M[x] \in [0..h]$. It is required that $M[0] = 0$;
- For each **bool** array $B \in \mathcal{V}$, a function $M[B]\colon [0..h] \rightarrow \{\text{FALSE}, \text{TRUE}\}$;
- For each **index** array $Z \in \{parent, parent'\}$, a function $M[Z]\colon [0..h] \rightarrow [0..h]$.

Let φ be a restricted EA-assertion, which we fix for this section. We require that if a term of the form $parent'[u]$ occurs in φ where u is a free or existentially quantified variable in φ, then φ also contains the preservation formula associated with $parent$. Note that this requirement is satisfied by any reasonable φ — assertions that contain primed variables occur only in proofs for abstraction computation (rather than in properties of systems), and are generated automatically by the proof system. In such cases, the assertion generated includes also the transition relation, which includes all preservation formulae. We include this requirement explicitly since the proof of the small model theorem depends on it.

Given a model M, one can evaluate the formula φ over the model M. The model M is a *satisfying model* for φ, if φ evaluates to TRUE in M, i.e., if $M \models \varphi$. An **index** term $t \in \{u, Z[u]\}$ in φ, where u is an existentially quantified or a free variable, is called a *free term*. Let $\mathcal{T}_\varphi$ denote the minimal set consisting of the following:

- The term 0 and all free terms in φ;
- For every array $Z \in \mathcal{V}$, if $Z[u] \in \mathcal{T}_\varphi$ then $u \in \mathcal{T}_\varphi$;
- For every **bool** array $B \in \mathcal{V}$, if $B[u] \in \varphi$, then if B is unprimed, $parent[u] \in \mathcal{T}_\varphi$, and if B is primed, $parent'[u] \in \mathcal{T}_\varphi$;
- If $parent'[u] \in \mathcal{T}_\varphi$ then $parent[u] \in \mathcal{T}_\varphi$ (this is similar to *history closure* of [2]).

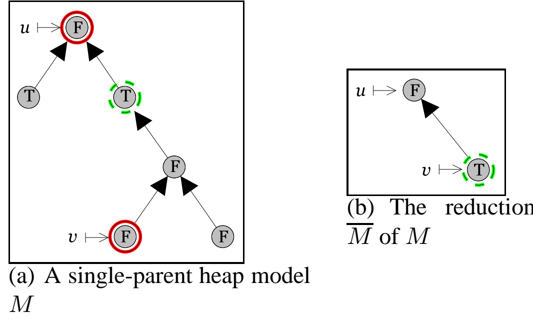
Let M be a model that satisfies φ with size greater then $|\mathcal{T}_\varphi|+1$ as follows: Let N be the set of $[0..h]$ values that M assigns to free terms in $\mathcal{T}_\varphi$. Assume that $N = \{n_0, \ldots, n_m\}$ where $0 = n_0 < \cdots < n_m$. Obviously, $m \leq |\mathcal{T}_\varphi|$. Define a mapping $\gamma \colon N \to [0..m]$ such that $\gamma(u) = i$ iff $M[u] = n_i$ (Recall that $M[\mathcal{T}_\varphi] = N$, so that γ is onto).

We now define the model $\overline{M}$. We start with its size and the interpretation of the scalars: $\overline{M}[h] = m+1$; For each **bool** variable b, $\overline{M}[b] = M[b]$; For each term $u \in \mathcal{T}_\varphi$ $\overline{M}[u] = \gamma(u)$.

Let $Z \in \{parent, parent'\}$ be an **index** array, and let $j \in [0..m]$. Consider the Z-chain in M $\alpha\colon n_j = u_0, \ldots$ such that for every $i \geq 1$, $M[Z](u_{i-1}) = M[u_i]$. If there is some $i \geq 1$ such that $u_i \in N$, then let k be the minimal such i. We then say that u_{k-1} is the M *representative of* Z *for* j and define $\overline{M}[Z](j) = \gamma(u_k)$. If no such i exists, then $\overline{M}[Z](j) = m+1$.

As for the interpretation of $\overline{M}$ over **bool** arrays, we distinguish between the case of unprimed and primed arrays. For an unprimed (resp. primed) **bool** array B, for every $j \in [0..m]$, if the M representative of $parent$ (resp. $parent'$) is defined and equals v, then let $\overline{M}[B](j) = M[B](v)$. Otherwise, $\overline{M}[B](j) = M[B](n_j)$. As for $\overline{M}[B](m+1)$, let $d \in [0..h]$ be the minimal such that $M[d] \notin N$. Then $\overline{M}[B](m+1)$ is defined to be $M[B](d)$.

Example 2 (Model Reduction).



(a) A single-parent heap model M

(b) The reduction $\overline{M}$ of M

Fig. 4. Model Reduction

Let $parent$ and $data$ be **index** and **bool** arrays respectively, and let φ be the assertion:

$$\varphi\colon\ \exists u, v \,.\, u \neq v \ \wedge\ \forall y \,.\, (parent[y] \neq u \ \vee\ data[y])$$

Since there are no free variables in φ, and since no array term refers to the uth or v^{th} element, it follows that $\mathcal{T}_\varphi$ consists only of the **index** terms u and v. Let M be a model of φ of size 7, as shown in Fig. 4(a). The interpretations by M of terms in $\mathcal{T}_\varphi$ are the highlighted nodes. Each node y is annotated with the value $M[data](y)$ (e.g., the node pointed to by u has data value of FALSE). $\overline{M}$, which is the reduction of M with respect to $\mathcal{T}_\varphi$, is given in Fig. 4(b). The M representative of $parent$ for $M[v]$ is given by the node highlighted by a dashed line in Fig. 4(a). As shown here, the node pointed to by v in $\overline{M}$ takes on the properties of this representative node.

Theorem 2. *If $M \models \varphi$ then φ is satisfiable by a model of size at most $|\mathcal{T}_\varphi| + 1$.*

Before we prove the theorem, we make the following observation:

Observation 1 *For every $n_i, n_j \in N$ and every* **index** *array Z, the following all hold:*

1. *if $M(Z)[n_i] = n_j$ then $\overline{M}(Z)[i] = j$, and if $\overline{M}(Z)[i] = j$ then $M \models Z^*(i, j)$;*
2. *$M \not\models Z^*(n_i, n_\ell)$, for any $n_\ell \in N$, iff $\overline{M}(Z)[i] = m+1$.*
3. *$M \models Z^*(n_i, n_j)$ iff $\overline{M} \models Z^*(i, j)$;*
4. *If $B'[u]$ occurs in φ for some $u \in \mathcal{T}_\varphi$ and a* **bool** *array $B \in \mathcal{V}$, then u, $parent[u]$, and $parent'[u]$ are all in $\mathcal{T}_\varphi$.*

Proof. (1), (2), and (4) follow immediately from the construction. As for (3), in one direction assume that $M \models Z^*(n_i, n_j)$. Thus, there exists a Z-chain $\alpha\colon n_i = v_0, \ldots, v_k = n_j$ in M. Remove all the non-N nodes from α, and let $v_{i_0}, \ldots, v_{i_n}$ be the remaining nodes. From the definition of $\overline{M}(Z)$ it follows that for every $\ell = 1, \ldots, n$, $\overline{M}(Z)[\gamma(v_{i_{j-1}})] = \gamma(v_{i_j})$. Thus, $\overline{M} \models Z^*(\gamma(v_0), \gamma(v_k) = Z^*(i, j)$. In the other direction assume that $\overline{M} \models Z^*(i, j)$. Since $n_i \in N$, $i \neq m+1$. Therefore, there exists a Z chain $\alpha\colon i = u_0, u_1, \ldots, u_k = j$ in $\overline{M}$ such that for every $\ell = 1, \ldots, \ell$, $\overline{M}(Z)(u_{\ell-1}) = u_\ell$. From part(1) it now follows that $M \models Z^*(n_i, n_j)$. □

We now return to the proof of Theorem 2

Proof. Assume that φ is satisfiable. Recall that φ is a restricted EA-assertion, i.e., φ is for the form $\varphi\colon \exists \vec{x}.\psi(\vec{u}, \vec{x})$, where $\vec{x}$ and $\vec{u}$ are disjoint lists of **index** variables, and ψ is a boolean combination of atomic formulae and restricted A-assertions. A model satisfies φ is it model can be augmented by an interpretation of $\vec{x}$ such that the augmented model satisfies $\psi(\vec{x}, \vec{u})$. Let M be such an augmented model, and let $\overline{M}$ be its reduction with respect to $\mathcal{T}_\psi = \mathcal{T}_\varphi$. To prove the theorem, we need to show that $\overline{M} \models \psi$ if $M \models \psi$.

Assume therefore that $M \models \psi$. To show that $\overline{M} \models \psi$, it suffices to show that (1) every atomic formula p is true in $\overline{M}$ iff it is true in M, and (2) every restricted A-assertion p that is satisfied in M is also satisfied in $\overline{M}$. (Recall that restricted A-assertions only may appear in ψ only under positive polarity.)

For the first case, let p be an atomic sub-formula of ψ. We distinguish between the following cases:

p **is a tcf formula.** The claim follows immediately from Observation 1 (part 3).

p **is of the form** $i\text{-}subtree_Z(x_1, x_2)$**.** Z, x_1, and x_2 are assumed to be an **index** array and **index** variables, respectively. In this case, we are dealing with an ordered heap as defined in Subsection 3.2, and assume the presence of an array ct : **index** $\rightarrow [1..k]$, with $i \in [1..k]$. In one direction, assume that $M \models p$. Expanding the definition of p to $\exists u\ .Z[u] = x_2 \ \wedge\ ct[u] = i \ \wedge\ Z^*(x_1, u)$, we conclude that $M \models Z^*(x_1, x_2)$.
We first identify the Z-chain from x_1 to x_2 in M, i.e. the node sequence $M[x_1] = u_1, \ldots, u_\ell, u_{\ell+1} = M[x_2]$ such that $M[Z](u_j) = u_{j+1}$, for every $j = 1, \ldots, \ell$. Let n_j be the node u_a, for the maximal $a \in [1..\ell]$, such that $n_j \in N$. Then u_ℓ is the M representative of Z for n_j. Since $M[Z](u_\ell) = u_{\ell+1} = M[x_2]$, it must be the case that $M[ct](u_\ell) = i$. By construction, $\overline{M}[ct](j) = M[ct](u_\ell) = i$, and

$\overline{M}[Z](j) = \gamma(M[Z](u_\ell)) = \gamma(M[x_2])$. Furthermore, from Observation 1 (part 3) we conclude that node j is Z-reachable from node $\overline{M}[x_1]$ in $\overline{M}$. Thus, x_1 is in the ith subtree of x_2 in $\overline{M}$, i.e., $\overline{M} \models \exists u\ .Z[u] = x_2\ \wedge\ ct[u] = i\ \wedge\ Z^*(x_1, u)$, and the claim holds.

In the other direction, assume that $\overline{M} \models p$. Let $\overline{M}[x_1] = j < m{+}1$ and $\overline{M}[x_2] = \ell < m{+}1$. The claim is proven by considering the Z-chain in $\overline{M}$ from j to ℓ and, based on the definition of $\overline{M}$, constructing a corresponding Z-chain in M from $M[x_1] = n_j$ to $M[x_2] = n_\ell$ in which n_j is in the ith subtree of n_ℓ.

p is a bool variable. The claim follows trivially from the construction of $\overline{M}$.

p is of the form $B[u]$ for an index variable u and a bool array B. It then follows that $parent[u]$ or $parent'[u]$ is in $\mathcal{T}_\varphi$, according to whether B is unprimed or primed, and then it follows from the construction that $\overline{M}[B](u) = M[B](u)$.

p is of the form $t_1 = t_2$ for index terms t_1 and t_2. Since $t_1, t_2 \in \mathcal{T}_\varphi$, it follows from the construction that $M \models t_1 = t_2$ iff $\overline{M} \models t_1 = t_2$.

For the second case, let p be a universal formula. We distinguish between two cases. The first is when p is in one of the forms: $\forall y.Z[y] \neq u$, $\forall y.Z[y] \neq u\ \vee\ B[y]$, or $\forall y.Z[y] \neq u \vee \neg B[y]$. We show here the second case; The other two are similar. Recall that u must be in $\mathcal{T}_\varphi$, and assume that $M(u) = n_j$. Assume, by way of contradiction, that $M \models \forall y.Z[y] \neq u\ \vee\ B[y]$ and for some $i \in [0..m{+}1]$, $\overline{M} \models Z[i] = j\ \wedge\ \neg B[i]$. If $i = m{+}1$, then obviously $M(Z)[i] = m{+}1$, and thus $\overline{M} \not\models Z[i] = u$. Hence, $i \neq m{+}1$. From Observation 1 it follows that $Z[n_i] \neq n_j$. Thus, there exists a Z-representative $v \neq n_i$ for i in M. From the construction it follows that $\overline{M}(Z)[i] = \gamma(M(Z)[v])$ and that $\overline{M}(B)[i] = M(B)[v]$. From the assumption that $\overline{M} \models \neg B[i]$, it follows that $\neg M(B)[v]$, and from the assumption that $\overline{M} \models p$ it then follows that $M(Z)[v] \neq n_j$, contradicting the assumption that $\overline{M}(Z)[i] = j$.

It remains to show the claim for the case that p is a preservation formula. We distinguish between the following cases:

p is a preservation formula of a index array. Hence, p is of the form $\forall y.Z'[y] = Z[y] \vee \bigvee_{i=j}^{n}(y = y_i)$, where $y_1, \ldots, y_n$ are **index** variables in $\mathcal{T}_\varphi$ and Z is **index** array. Denote by Y the set $\{y_1, \ldots, y_n\}$ and by $\gamma(Y)$ the set $\{\gamma(y_1), \ldots, \gamma(y_k)\}$. Thus p can be re-written as $\forall y.Z'[y] = Z[y]\ \vee\ y \in Y$. Assume that $M \models p$. We have to show that $\overline{M} \models p$, i.e., that $\overline{M} \models \forall i \in [0..m{+}1].Z'[i] = Z[i]\ \vee\ i \in \gamma(Y)$. Assume, by way of contradiction, that for some $i \in [0..m{+}1]$, $\overline{M} \models Z'[i] \neq Z[i]\ \wedge\ i \notin \gamma(Y)$. We show that $\overline{M}(Z)[i] = \overline{M}(Z')[i]$, contradicting the assumption. Since $\overline{M}(Z)[m{+}1] = \overline{M}(Z')[m{+}1] = m{+}1$, it follows that $i \neq m{+}1$. Consider the Z-chain $n_i = u_0, u_1, \ldots$ and the Z'-chain $n_i = v_0, v_1, \ldots$ in M. Since $i \neq \gamma(Y)$, it follows, from the assumption that $M \models p$, that $M \models Z[u_0] = Z[v_0]$, hence $v_1 = u_1$. Proceeding like this, we obtain that either

1. For all $j \geq 0$, $u_j = v_j$, or
2. For some $m \geq 1$, $u_m = v_m \in Y$, and for all $j = 0, \ldots, m-1$, $u_j = v_j \notin Y$.

In the first case we obtain that $\overline{M}(Z')[i] = \overline{M}(Z)[i]$. In the second case,since $u_m = v_m \in Y \in \mathcal{T}_\varphi$, we obtain that i has the same Z-representative in M, and thus $\overline{M}(Z)[i] = \overline{M}(Z')[i]$. (Note that this Z-representative is either $u_j = v_j$ for some $j < m$, or $u_m = j_m$. The claim follows in either case.)

ψ **is a preservation formula of a bool array.** Following the notation of the previous part, assume p is of the form $\forall y.B'[y] = B[y] \ \vee \ y \in Y$ where Y is a set of **index** variables in $\mathcal{T}_\varphi$. Assume that $M \models p$, and that $\overline{M} \not\models p$, i.e., for some $i \in [0..m{+}1]$, $\overline{M} \models B'[i] \neq B[i] \ \wedge \ i \notin \gamma(Y)$. Since $\overline{M}(B)[m{+}1] = M(B)[d]$, $\overline{M}(B')[d] = M(B')[d]$, and $d \notin Y$, it follows that $i \neq m{+}1$.

This case is handled similarly to the previous case, considering the Z-chain $n_i = u_0, \ldots$ and Z'-chain $n_i = v_0, \ldots$ in M, and conclude that $\overline{M}(B')[i] = \overline{M}(B)[i]$. The only difference is in the inductive step: Let $k \geq 0$, and assume that for all $j \leq k$, $u_j = v_j$ and $u_j \notin Y$. If $M(Z')[v_k] = M(Z)[v_k]$, then obviously $v_{k+1} = u_{k+1}$. Otherwise, $M(Z')[v_k] \neq M(Z)[v_k]$. From Observation 1, part (4), it follows that $v_k, u_{k+1}, v_{k+1} \in \mathcal{T}_\varphi$. It thus follows that i has the same Z and Z representative in M (which is either v_0, v_j for some $j < k$, or v_k) and therefore $\overline{M}(B)[i] = \overline{M}(B')[i]$. □

The discussion below is similar to the one in [2]; see details there. For a restricted EA-assertion φ and a positive integer $h_0 > 0$, define the h_0-*bounded* version of φ, denoted by $\lfloor \varphi \rfloor_{h_0}$, to be the conjunction $\varphi \wedge \forall y \,.\, y \leq h_0$. Theorem 2 can be interpreted as stating that φ is satisfiable iff $\lfloor \varphi \rfloor_{|\mathcal{T}_\varphi|}$ is satisfiable.

We next extend the small model theorem to the computation of abstraction of systems. Consider an abstraction α, where 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 $\alpha(\varphi)(V_A) = \bigvee_{i \in sat(\varphi)} (V_A = U_i)$.

Consider the assertion $\psi_0 : U_i = \mathcal{E}_\alpha(V) \wedge \varphi(V)$. Let $h_0 = |\mathcal{T}_{\psi_0}|$. Our reinterpretation of Theorem 2 states that ψ_0 is satisfiable iff $\lfloor \psi_0 \rfloor_{h_0}$ is satisfiable. Therefore, $sat(\lfloor \varphi \rfloor_{h_0}) = sat(\varphi)$. Thus, $\alpha(\varphi)(V_A) \leftrightarrow \alpha(\lfloor \varphi \rfloor_{h_0})(V_A)$. This can be extended to abstraction of assertions that refer to primed variables. Recall that the abstraction of such an assertion involves a double application of the abstraction mapping, an unprimed version and a primed version. Assume that $\varphi(V, V')$ is such an assertion, and consider $\psi_1 : (U_i = \mathcal{E}_A(V)) \wedge (U_j = \mathcal{E}_A(V')) \ \wedge \ \varphi(V, V')$. Let $h_1 = |\mathcal{T}_{\psi_1}|$. By the same reasoning, we have $\alpha(\varphi)(V_A, V'_A) \leftrightarrow \alpha(\lfloor \varphi \rfloor_{h_1}(V_A, V'_A))$.

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 $\mathcal{T}_\psi$, for every abstraction formula ψ necessary for computing the abstraction of all the components of S. Then we have the following theorem:

Theorem 3. *Let S be a single-parent heap system, 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 $\lfloor S \rfloor^\alpha_{h_0}$.*

As a consequence, in order to compute the abstract system S^α, we can instantiate the system S to a heap of size h_0, and use propositional methods, e.g., BDD-techniques[4], to compute the abstract system $\lfloor S \rfloor^\alpha_{h_0}$. Note that h_0 is linear in the number of system

[4] In our experiments we use TLV ([17]).

variables. This process is fully automatic once the predicate base is given. The exact manner by which predicates themselves are derived (e.g., by user input or as part of a refinement loop) is orthogonal to the method presented here.

5 Multi-Linked Heap Systems

In this section we define *multi-linked heap systems* with a bounded out-degree on nodes. A multi-linked heap is represented similar to a single-parent heap, only, instead of having a single **index** array, we allow for some $k > 1$ **index** arrays, each describing one of the links a node may have. We denote these arrays by $link_1, \ldots, link_k$. Thus, each $link_i$ is an array $[0..h] \rightarrow [0..h]$. We are mainly interested in *non-sharing heaps*, defined as follows:

Definition 1. *A* non-sharing heap *is one that satisfies the following requirements:*

1. *For every $i = 1, \ldots, k$, $link_i[0] = 0$.*
2. *For every* **bool** *array B, $\neg B[0]$.*
3. *No two distinct positive nodes may share a common positive child. This requirement can be formalized as*

$$\forall j, \ell \in [1..h], i, r \in [1..k] \,.\, (j \neq \ell) \wedge (link_i[j] = link_r[\ell]) \;\rightarrow\; link_i[j] = 0$$

4. *No two distinct links of a positive node may point to the same positive child. This can be formalized as*

$$\forall j \in [1..h], s, t \in [1..k] \,.\, (s \neq t) \wedge (link_s[j] = link_t[j]) \;\rightarrow\; link_s[j] = 0$$

▟

We refer to the conjunction of the requirements in Definition 1 by the formula *no_sharing*. A state violating one of these three requirements is called a *sharing state*.

A multi-linked system is called *sharing-free* if none of its computations ever reaches a sharing state, nor does a computation ever attempt to assign a value to $A[0]$ for some array A.

Let $\mathcal{D}\colon \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be a k-bounded multi-linked heap system. Fig. 5 describes a BNF-like syntax of the assertions used in describing $\mathcal{D}$. There, `Ivar` denotes an unprimed **index** variable, `Iarr` denotes an unprimed **index** array, `Bvar` denotes an unprimed **bool** variable, and `Barr` denotes an unprimed **bool** array. The expression $reach(x, y)$ abbreviates $(x, y) \in (\bigcup_{i=1}^{k} link_i)^*$, and the expression $cycle(x)$ abbreviates $(x, x) \in (\bigcup_{i=1}^{k} link_i)^+$. The **Preservation** assertion is just like in the single-parent case and we require that if `Assign` appears in τ, then the **Preservation** assertion that is conjoined with it includes preservation of all variables that don't appear in the left-hand-side of any clause of `Assign`.

For example, consider a binary tree, which is a multi-linked heap with bound 2 and no sharing. Each of *left* and *right* is a $link$. Program TREE-INSERT in Fig. 6 is the standard algorithm for inserting a new node, n, into a sorted binary tree rooted at r.

$$
\begin{aligned}
\texttt{MCond1} ::= {} & \textsc{True} \mid \texttt{Bvar} \mid \texttt{Barr[Ivar]} \mid \texttt{Ivar} = \texttt{Ivar} \mid \texttt{Ivar} = 0 \mid \\
& \texttt{Iarr[Ivar]} = \texttt{Ivar} \mid \texttt{Iarr[Ivar]} = 0 \mid \\
& \texttt{MCond1} \vee \texttt{MCond1} \mid \neg\texttt{MCond1} \\
\texttt{MCond2} ::= {} & \texttt{MCond1} \mid \mathit{reach}(\texttt{Ivar},\texttt{Ivar}) \mid \mathit{cycle}(\texttt{Ivar}) \mid \\
& \neg\texttt{MCond2} \mid \texttt{MCond2} \vee \texttt{MCond2} \\
\texttt{Assign} ::= {} & \epsilon \mid \texttt{Bvar}' \mid \neg\texttt{Bvar}' \mid \texttt{Barr}'\texttt{[Ivar]} \mid \neg\texttt{Barr}'\texttt{[Ivar]} \mid \\
& \texttt{Bvar}' = \texttt{Bvar} \mid \texttt{Ivar}' = 0 \mid \texttt{Ivar}' = \texttt{Ivar} \mid \\
& \texttt{Iarr}'\texttt{[Ivar]} = \texttt{Ivar} \mid \texttt{Iarr}'\texttt{[Ivar]} = 0 \mid \texttt{Assign} \wedge \texttt{Assign} \\
\Theta ::= {} & \texttt{MCond2} \ \wedge \mathit{no_sharing} \\
\rho ::= {} & \textsc{True} \mid \texttt{MCond1} \ \wedge \texttt{Assign} \ \wedge \ \textbf{Preservation} \mid \rho \ \vee \ \rho \\
\mathcal{J} ::= {} & \emptyset \mid \mathcal{J} \cup \{ \texttt{MCond1} \} \\
\mathcal{C} ::= {} & \emptyset \mid \mathcal{C} \cup \{ (\texttt{MCond1}, \texttt{MCond1}) \}
\end{aligned}
$$

Fig. 5. Grammar for Assertions for Multi-Linked Systems

6 Reducing Multi-Linked into Single-Parent Heaps

We now show how to transform multi-linked heap systems into ordered single-parent heap systems.

6.1 The Transformation

Let $\mathcal{D}_m : \langle \mathcal{V}_m, \Theta_m, \rho_m, \mathcal{J}_m, \mathcal{C}_m \rangle$ be a k-bounded multi-linked heap system. Thus, $\mathcal{V}_m$ includes the **index** arrays $link_1, \ldots, link_k$. We transform $\mathcal{D}_m$ into a single-parent heap system $\mathcal{D}_s : \langle \mathcal{V}_s, \Theta_s, \rho_s, \mathcal{J}_s, \mathcal{C}_s \rangle$ as follows:

The set of variables $\mathcal{V}_s$ consists of the following:

1. $\mathcal{V}_m \setminus \{link_1, \ldots, link_k\}$, i.e., we remove from $\mathcal{V}_m$ all the $link$ arrays;
2. An **index** array $parent\colon [0..h] \mapsto [0..h]$ that does not appear in $\mathcal{V}_m$;
3. A **bool** array $ct\colon [0..h] \mapsto [0..k]$ that does not appear in $\mathcal{V}_m$ (recall our convention that "**bool**" can be any finite-domain type);
4. A new **bool** variable *error*; *error* is set when $\mathcal{D}_m$ contains an erroneous transition such as one that introduces sharing in the heap, or attempts to assign values to $A[0]$ for some array A.

Intuitively, we replace the **index** $link$ arrays with a single **index** $parent$ array that reverses the direction of the links, and assign to $ct[i]$ (*child type*) the "birth order" of i in the heap. The variable *error* is boolean and is set when $\mathcal{D}_m$ cannot be transformed into a singe-parent system. This is caused by either an assignment to $A[0]$ or by a violation of the non-sharing requirements. When such an error occurs, *error* is raised, and remains so, i.e., ρ_s implies $\mathit{error} \rightarrow \mathit{error}'$.

Definition 2. *A single-parent state is said to be* well formed *if the parent of 0 is itself, all the* **bool** *arrays* $\mathcal{B} \subset \mathcal{V}_s$ *associate 0 with the value* FALSE*, and no parent has two*

```
left, right : array [0..h] of [0..h] init no_sharing
data        : array [0..h] of bool
r, n        : [1..h]                  init ¬reach(r, n) ∧ ¬cycle(r) ∧
                                           left[n] = 0 ∧ right[n] = 0
t           : [0..h]                  init t = r
done        : bool                    init done = FALSE
  1 :  while ¬done do
           2 : if data[n] = data[t] then
                   3 : done := TRUE
           4 : elseif data[n] < data[t] then
                   5 : if left[t] = 0 then
                           6 : left[t] := n
                           7 : done := TRUE
                       else
                           8 : t := left[t]
           9 : elseif right[t] = 0 then
                   10 : right[t] := n
                   11 : done := TRUE
               else
                   12 : t := right[t]
  13 :
```

Fig. 6. Program TREE-INSERT of Fig. 2, adapted to the encoding of trees as multi-linked heaps.

distinct children with the same birth order, i.e.,

$$\texttt{wf}:\ parent[0] = 0 \ \wedge\ \bigwedge_{B\in\mathcal{B}}(\neg B[0]) \wedge$$
$$\forall i \neq j\,.\,(parent[i] = parent[j] \neq 0 \ \rightarrow\ ct[i] \neq ct[j])$$

▟

To transform ρ_m, $\mathcal{J}_m$, and $\mathcal{C}_m$ into their $\mathcal{D}_s$ counterparts, it suffices to transform M-assertions over $\mathcal{V}_m \cup \mathcal{V}'_m$ into restricted EA-assertions over $\mathcal{V}_s \cup \mathcal{V}'_s$. To transform Θ_m, which is of the form *no_sharing* $\wedge\ \varphi$, where φ is an `MCond2`, into Θ_s, we take the conjunction of `wf` and the transformation of φ. It thus remains to transform M-assertions. Recall that ρ_m is a disjunction of clauses (see Section 5), each one of the form

$$\varphi\ \wedge\ \tau\ \wedge\ presEx(\mathcal{V}_m - \{V\})$$

where $V \subseteq \mathcal{V}_m$, φ is an `MCond` over $\mathcal{V}_m$, and τ is an `Assign` statement of the form $\bigwedge_{v\in V} v' = E_v(\mathcal{V}_m)$ (where E_v is some expression). When we transform such a ρ_m-disjunct, we sometimes obtain several disjuncts. We assume that each has its obvious $presEx$ assertions over $\mathcal{V}_s$. At times, for simplicity of representation, we do not express the transformation directly in DNF. Yet, in those cases, the DNF form is straightforward.

It thus remains to show how to transform M-assertions into restricted EA-assertions. This is done by induction on the M-assertions, where we ignore the preservation part (which, as discussed above, is defined by the transition relation for both $\mathcal{D}_m$ and $\mathcal{D}_s$.)

Let ψ be an M-assertion. In the following cases, ψ remains unchanged in the transformation:

1. ψ contains no reference to **index** variables and arrays;

2. ψ is of the form $x_1 = x_2$ where x_1 and x_2 are both primed, or both unprimed, **index** variables;
3. ψ is of the form $x_1 = x_2$ where x_1 is a primed, and x_2 is an unprimed, **index** variable;
4. ψ is of the form $x = 0$ where x is a (either primed or unprimed) **index** variable;
5. ψ is of the form $B[x]$, where B is an unprimed **bool** array.

The other cases are treated below. We now denote primed variables explicitly, e.g., x_1 refers to an unprimed variable, and x'_1 refers to a primed variable:

1. An assertion of the form $link_j[x_2] = x_1$ is transformed into

$$\begin{aligned} & (x_2 = 0 \ \wedge \ x_1 = 0) \\ \vee \ & (x_2 \neq 0 \ \wedge \ x_1 = 0 \ \wedge \ \forall z \, . \, (parent[z] \neq x_2 \vee ct[z] \neq j)) \\ \vee \ & (x_2 \neq 0 \ \wedge \ x_1 \neq 0 \ \wedge \ parent[x_1] = x_2 \ \wedge \ ct[x_1] = j) \end{aligned}$$

   In the case that $x_2 \neq 0$ and $x_1 = 0$, x_2 should have no j^{th} child. If $x_2 \neq 0$ and $x_1 \neq 0$, then x_1 should have x_2 as a parent and the child type of x_1 should be j.
2. A transitive closure formula $reach(x_1, x_2)$ is transformed into

$$(x_1 \neq 0 \ \wedge \ x_2 \neq 0 \ \wedge \ parent^*(x_2, x_1)) \ \vee \ (x_2 = 0)$$

   The first disjunct deals with the case where x_1 and x_2 are both non-0 nodes, and then the reachability direction is reversed, reflecting reversal of heap edges in the transformation to a single-parent heap. The second disjunct deals with the case that $x_2 = 0$, and then, since $k > 0$, there is a path from any node into 0.
3. A transitive closure formula $cycle(x)$, where x is an **index** variable, is transformed into $parent^*(parent[x], x)$.
4. An assertion of the form $x'_1 = link_j[x_2]$ is transformed into:

$$\begin{aligned} (x_2 = 0 \wedge x'_1 = 0) \ & \vee (x_2 \neq 0 \ \wedge \ x'_1 = 0 \wedge \forall y \, . \, (parent[y] \neq x_2 \vee ct[y] \neq j)) \\ & \vee (x_2 \neq 0 \wedge \exists y \, . \, (parent[y] = x_2 \wedge ct[y] = j \wedge x'_1 = y) \end{aligned}$$

   In case $x_2 = 0$, this transition sets x_1 to 0 since we assume that in non-sharing states $link_j[0] = 0$ for every $j = 1, \ldots, k$. Otherwise, if x_2 has no j^{th} child, then x_1 is set to 0. Otherwise, there exists a y which is the jth child of x_2, and then x_1 is set to y.
5. An assertion of the form $B'[x]$, where B is an unprimed **bool** array, is transformed differently based on its polarity. If it appears under *positive* polarity, it is transformed into:

$$(x = 0 \ \wedge \ \textit{error}') \ \vee \ (x \neq 0 \ \wedge \ B'[x])$$

   The error condition reflects an attempt to assign TRUE to $B[0]$. If the assertion $B'[x]$ appears under *negative* polarity, then no erroneous assignment is possible, and the assertion remains unchanged by the transformation.

6. An assertion of the form $link'_j[x_1] = x_2$ is transformed into:

$$\begin{array}{l} \mathit{Err} \wedge \mathit{error}' \quad \vee \\ \quad \neg \mathit{Err} \\ \wedge\ (x_2 = 0\ \vee\ (x_2 \neq 0 \wedge \mathit{parent}'[x_2] = x_1\ \wedge\ ct'[x_2] = j)) \\ \wedge\ \begin{pmatrix} \quad \forall z \,.\, (\mathit{parent}[z] \neq x_1\ \vee\ ct[z] \neq j) \\ \vee\ \exists z \,.\, (\mathit{parent}[z] = x_1\ \wedge\ ct[z] = j\ \wedge\ (z = x_2 \vee \mathit{parent}'[z] = 0)) \end{pmatrix} \end{array}$$

Where *Err* is defined by:

$$(x_1 = 0 \wedge x_2 \neq 0) \vee (x_2 \neq 0 \wedge \mathit{parent}[x_2] \neq 0 \wedge (\mathit{parent}[x_2] \neq x_1 \vee ct[x_2] \neq j))$$

I.e., the assignment may cause an error by either attempting to assign a nonzero value to $link_j[0]$, or by introducing sharing (when x_2 either has a parent that is not x_1, or is x_1's i^{th} child for some $i \neq j$).
When there is no error, x_2 should become the jth child of x_1 unless it is 0, which is expressed by the first conjunct of the non-error case; in addition, any node that was the jth child of x_1 before the transition should become "orphaned," which is expressed by the second conjunct of the non-error case.

The following observation follows trivially from the construction above:

Observation 2 *The transformation of an* M*-assertion is a restricted EA-assertion.*

Having defined the system transformation, we can now demonstrate the complete verification process of the tree insertion program.

Example 3 (Verification of TREE-INSERT*).*
We wish to verify that the multi-linked tree insertion program given in Fig. 6 satisfies the following specification:

$$\begin{array}{rl} \mathit{no\text{-}loss}: & \forall x \,.\, \mathit{reach}(r,x) \rightarrow \Box\, \mathit{reach}(r,x) \\ \mathit{no\text{-}gain}: & \forall x \,.\, x \neq n\ \wedge\ \neg\mathit{reach}(r,x) \rightarrow \Box\, \neg\mathit{reach}(r,x) \\ \mathit{insertion}: & (\forall u \,.\, \mathit{reach}(r,u) \rightarrow \mathit{data}[u] \neq \mathit{data}[n]) \rightarrow \Box\, \mathit{at_}13 \rightarrow \mathit{reach}(r,n) \end{array}$$

We begin by eliminating the universal quantifiers in the *no-loss* and *no-gain* properties by introducing a *skolem constant* x. This is done by augmenting the program with a variable with an undetermined initial value that stays constant throughout a computation. This is a purely syntactic transformation.

As for the *insertion* property, unfortunately the abstraction computation method of Section 4 disallows any occurrence of $reach$ predicates under universal quantification. Therefore, we heuristically *instantiate* the universal variable u to derive the following (stronger) property:

$$\mathit{insertion}: \left(\textstyle\bigwedge_{u \in \{r,n,t\}} \mathit{reach}(r,u) \rightarrow \mathit{data}[u] \neq \mathit{data}[n]\right) \rightarrow \Box\, \mathit{at_}13 \rightarrow \mathit{reach}(r,n)$$

We proceed by applying the system transformation, resulting in the single-parent heap system[5] shown in Fig. 7. We now apply predicate abstraction. We use the predicate

[5] Note that this automatically-derived version is less optimal than the manually-constructed single-parent system given in Fig. 2.

$$
\begin{array}{rl}
\Theta\!: & parent[0]=0 \;\wedge\; \forall i \neq j \,.\, (parent[i]=parent[j] \neq 0 \rightarrow ct[i] \neq ct[j]) \;\wedge \\
& \neg parent^*(n,r) \;\wedge\; \forall i \,.\, (parent[i] \neq n) \;\wedge\; t=r \;\wedge\; \neg parent^*(parent[r],r) \;\wedge\; \neg done \\
\rho\!: & error \;\wedge\; error' \;\wedge\; presEx(error) \\
& \quad\vee \\
& \neg error \;\wedge \\
& \left[\begin{array}{l}
\quad \pi=1 \;\wedge\; \neg done \;\wedge\; \pi'=2 \;\wedge\; presEx(\pi) \\
\vee\; \pi=1 \;\wedge\; done \;\wedge\; \pi'=13 \;\wedge\; presEx(\pi) \\
\vee\; \pi=2 \;\wedge\; data[t]=data[n] \;\wedge\; \pi'=3 \;\wedge\; presEx(\pi) \\
\vee\; \pi=2 \;\wedge\; data[t] \neq data[n] \;\wedge\; \pi'=4 \;\wedge\; presEx(\pi) \\
\vee\; \pi=3 \;\wedge\; \pi'=1 \;\wedge\; done' \;\wedge\; presEx(\pi, done) \\
\vee\; \pi=4 \;\wedge\; t \neq 0 \;\wedge\; data[n] < data[t] \;\wedge\; \pi'=5 \;\wedge\; presEx(\pi) \\
\vee\; \pi=4 \;\wedge\; (t=0 \;\vee\; data[t] \leq data[n]) \;\wedge\; \pi'=9 \;\wedge\; presEx(\pi) \\
\vee\; try(5, left) \;\vee\; try(9, right) \\
\vee\; \pi=13 \;\wedge\; \pi'=13 \;\wedge\; presEx(\pi)
\end{array}\right] \\
\\
try(link, \pi_0)\!: & \left[\begin{array}{l}
\quad \pi=\pi_0 \;\wedge\; \pi'=\pi_0+1 \;\wedge\; presEx(\pi) \;\wedge \\
\quad (t=0 \;\vee\; (t \neq 0 \;\wedge\; \forall j \,.\, parent[j] \neq t \;\vee\; ct[j] \neq left)) \\
\vee\; \pi=\pi_0 \;\wedge\; \pi'=\pi_0+3 \;\wedge\; t \neq 0 \;\wedge\; presEx(\pi) \\
\quad (\exists j \,.\, parent[j]=t \;\wedge\; ct[j]=left) \\
\vee\; \pi=\pi_0+1 \;\wedge\; error' \;\wedge\; presEx(error) \;\wedge \\
\quad (t=0 \;\vee\; (t \neq 0 \;\wedge\; parent[n] \neq 0 \;\wedge\; (parent[n] \neq t \;\vee\; ct[n] \neq left))) \;\wedge \\
\vee\; \pi=\pi_0+1 \;\wedge\; \pi'=\pi_0+2 \;\wedge\; t \neq 0 \;\wedge \\
\quad (parent[n]=0 \;\vee\; (parent[n]=t \;\wedge\; ct[n]=left)) \;\wedge \\
\quad parent'[n]=t \;\wedge\; ct'[n]=left \;\wedge\; presEx(\pi, parent[n], ct[n]) \;\wedge \\
\quad \left(\begin{array}{l}
\quad \forall j \,.\, (parent[j] \neq t \;\vee\; ct[j] \neq left) \\
\vee\; \exists j \,.\, (parent[j]=t \;\wedge\; ct[j]=left \;\wedge\; (j=n \;\vee\; parent'[j]=0))
\end{array}\right) \wedge \\
\vee\; \pi=\pi_0+2 \;\wedge\; \pi'=1 \;\wedge\; done' \;\wedge\; presEx(\pi, done) \\
\vee\; \pi=\pi_0+3 \;\wedge\; \pi'=1 \;\wedge\; presEx(\pi, t) \;\wedge \\
\quad \left(\begin{array}{l}
\quad t=0 \;\wedge\; t'=0 \\
\vee\; t \neq 0 \;\wedge\; t'=0 \;\wedge\; \forall j \,.\, (parent[j] \neq t \;\vee\; ct[j] \neq left) \\
\vee\; t \neq 0 \;\wedge\; \exists j \,.\, (parent[j]=t \;\wedge\; ct[j]=left \;\wedge\; t'=j)
\end{array}\right)
\end{array}\right]
\end{array}
$$

Fig. 7. The single-parent program resulting from transformation of the tree insertion program of Fig. 6

base given by the following set of assertions:

$$
\mathcal{P}: \left\{\begin{array}{l}
p_1 : \forall j \,.\, parent[j] \neq n, \\
p_2 : \textit{left-subtree}(n,r), \\
p_3 : \textit{right-subtree}(n,r), \\
p_4 : parent^*(t,r), \\
p_5 : \exists j \,.\, parent[j]=t, \\
p_6 : data[t]=data[n], \\
p_7 : parent^*(x,r)
\end{array}\right\}
$$

Note that the predicate p_1 is in fact an inductive invariant, a fact that can be decided (without the use of abstraction) by directly applying Theorem 2 to check validity of the verification conditions

$$
\begin{array}{l}
\text{I1. } \Theta \rightarrow p_1 \\
\text{I2. } p_1 \;\wedge\; \rho \rightarrow p_1'
\end{array}
$$

Having decided the invariance of p_1, it is possible to optimize the abstraction computation by removing p_1 from the predicate base, and by constraining the concrete state space to p_1-states only. ▟

In the following section we establish the soundness of the transformation.

6.2 Correctness of Transformation

In order for the above transformation to fit into the verification process proposed in Section 1, we have to show that the result of the verification, as carried out on the transformed system and property, holds with respect to the untransformed counterparts. Such a result is provided by Theorem 4 below. To show that the abstraction computation method of Section 4 is sound with respect to a transformed program and property, we use Observation 2 and Theorem 5 below. For simplicity of presentation, in this section we do not take into account fairness requirements. However, it is straightforward to extend the results, i.e., show that the heap transformation preserves satisfiability of justice requirements, and that the computation transformation preserves compassion.

Let $\mathcal{D}_m : \langle \mathcal{V}_m, \Theta_m, \rho_m, \mathcal{J}_m, \mathcal{C}_s \rangle$ be a k-bounded multi-linked heap system over the set of variables $\mathcal{V}_m$, with $k > 1$, and let $\mathcal{D}_s : \langle \mathcal{V}_s, \Theta_s, \rho_s, \mathcal{J}_s, \mathcal{C}_s \rangle$ be its transformation into a single-parent heap system. The transformation into a single-parent heap system induces a mapping $\mathcal{S} \colon \Sigma_m \to \Sigma_s$. The mapping $\mathcal{S}$ is formally defined below.

Definition 3. *Let $\mathcal{S}$ be a mapping from Σ_m into Σ_s, such that for every $s_m \in \Sigma_m$, if $s_s = \mathcal{S}(s_m)$, then the following all hold:*

1. *For every* **bool** *variable $v \in \mathcal{V}_m$, $s_s[v] = s_m[v]$;*
2. *For every* **bool** *array $B \in \mathcal{V}_m$ and $x \in [0..h]$, $s_s[B](x) = s_m[B](x)$;*
3. *For every* **index** *variable $x \in \mathcal{V}_m$, $s_s[x] = s_m[x]$*
4. *$s_s[parent](0) = 0$ and $s_s[ct](0) = 1$.*
5. *Let $y \in [1..h]$. If for all $z \in [1..h]$ and $i \in [1..k]$, $s_m[link_i](z) \neq y$, then $s_s[parent](y) = 0$ and $s_s[ct](y) = 1$. Otherwise, $s_s[parent](y) = x$ and $s_s[ct](y) = j$ where (x, j) is the lexicographically minimal pair in $\{(z, i) : z \in [1..h],\ i \in [1..k],$ and $s_m[link_i](z) = y\}$.*
6. $s_m[error] = \begin{cases} \text{FALSE}, & \text{if } s_m \models no_sharing \\ \text{TRUE}, & \text{otherwise} \end{cases}$

∎

We first make the following observation regarding $\mathcal{S}$:

Observation 3 *The inverse $\mathcal{S}^{-1}$ is well defined for any well formed non-error state $s_s \in \Sigma_s$. That is, if $s_s \models$ `wf` $\wedge\ \neg error$ then there exists a state $s_m \in \Sigma_k$ such that $\mathcal{S}(s_m) = s_s$.*

Lemma 1. *Let $s_m \in \Sigma_m$, and let $s_s = \mathcal{S}(s_m)$. Then $s_m \models no_sharing \iff s_s \models$ `wf` $\wedge\ \neg error$.*

Proof. The reverse direction holds trivially. We now assume that $s_m \models no_sharing$, and show that s_s satisfies `wf`, i.e.,

$$\begin{array}{l} \neg error \ \wedge\ parent[0] = 0 \ \wedge\ \bigwedge_{B \in \mathcal{B}} (\neg B[0]) \ \wedge \\ \forall i \neq j \,.\, (parent[i] = parent[j] \neq 0 \ \rightarrow\ ct[i] \neq ct[j]) \end{array}$$

where $\mathcal{B} \subset \mathcal{V}_s$ is the set of **bool** arrays of $\mathcal{D}_s$. $s_s[error] = \text{FALSE}$, $s_s[parent](0) = 0$, and $s_s[B](0) = \text{FALSE}$, for all $B \in \mathcal{B}$, all follow from the definition of $\mathcal{S}$. The universal condition follows from two properties:

– The links in a multi-linked heap are functional, i.e., for every $i \in [1..k]$, every node has at most one $link_i$-child.
– From Item 5 of the definition of $\mathcal{S}$, we have that for any nodes u and v, and $i \in [1..k]$, we have $s_s[parent](u) = v$ and $s_s[ct](u) = i$ iff $s_m[link_i](v) = u$. □

Lemma 2. *Let $s_m \in \Sigma_m$ be a state that satisfies the no_sharing constraint, and let $s_s = \mathcal{S}(s_m)$. Let φ_m be a boolean combination of* M*-atomic formulae over $\mathcal{D}_m$, and let φ_s be its transformation into an assertion over $\mathcal{D}_s$. Then: $s_m \models \varphi_m \iff s_s \models \varphi_s$*

Proof. The claim follows immediately from Lemma 1 for the case that φ_m is an M-atomic non-$reach$ and non-*cycle* formula. For the other cases, we distinguish between:

φ_m **is of the form** $reach(x_1, x_2)$. Then, φ_s is of the from

$$(x_1 \neq 0 \ \wedge \ x_2 \neq 0 \ \wedge \ parent^*(x_2, x_1)) \ \vee \ (x_2 = 0)$$

From the definition of $\mathcal{S}$ it follows that $s_s[x_1] = s_m[x_1]$ and $s_s[x_2] = s_m[x_2]$. In one direction, assume that $s_m \models \varphi_m$. If $s_m[x_2] = 0$, then obviously $s_s \models \varphi_s$. Otherwise, assume that $s_m[x_2] \neq 0$. Hence, for some $n \geq 1$ there exist nodes $s_m[x_1] = u_1, \ldots, u_n = s_m[x_2]$ such that for every $i = 1, \ldots, n$, there exists some $j_i \in [1..k]$ such that $s_m \models link_{j_i}[u_i] = u_{i+1}$, and $s_m[u_i] \neq 0$. Since $\mathcal{D}_m \models$ *no_sharing*, it follows that for every $i = 1, \ldots, n-1$, $s_s[parent](u_{i+1}) = u_i$. Thus, $s_s \models parent^*(u_n, u_1)$. Thus $s_s \models \varphi_s$.
In the other direction, assume that $s_s \models \varphi_s$. If $s_s[x_1] = 0$, then $s_s[x_2] = 0$, and then $s_m \models \varphi_m$ trivially follows. Assume therefore that $s_s[x_1] \neq 0$. If $s_s[x_2] \neq 0$, an argument, similar to the one used for this case in the other direction, shows that $s_m \models \varphi_m$. If $s_s[x_2] = 0$, then let $u \neq 0$ be such that there is a $s_s[parent]$-path from u to $s_s[x_1]$, and for some $i \in [1..k]$, and for every y either $s_s[parent](y) \neq u$ or $M_k[ct](y) \neq i$. Thus, $s_m[link_i](u) \neq y$ for every y. It thus follows that $s_m[link_i](u) = 0$. Similar arguments to the previous direction show that there is a $(\bigcup_{i=1}^{k} link_i)$-path from $s_m[x_1]$ to u. We can therefore conclude that $s_m \models reach^*(x_1, x_2)$.

φ_m **is of the form** $cycle(x)$. This case is similar to the previous case. □

Since the initial condition of $\mathcal{D}_m$ is not a restricted A-assertion, it needs to be dealt with separately:

Lemma 3. *Let $s_m \in \Sigma_m$ such that $s_m \models$ no_sharing. Let $s_s = \mathcal{S}(s_m)$. Then: $s_m \models \Theta_m \iff s_s \models \Theta_s$*

Proof. As a consequence of the grammar in Fig. 5, Θ_m is of the form $\psi \wedge$ *no_sharing* where ψ is a boolean combination of M-atomic formulae. Section 6 defines Θ_s as $\psi_s \wedge$ `wf`, where ψ_s is the transformation of ψ by the rules of Section 6 and `wf` is given in Definition 2. From Lemma 2 we have that if $s_m \models$ *no_sharing*, then $s_m \models \psi$ iff $s_s \models \psi_s$. From Definition 3 we have $s_s \models \neg error$, and from Lemma 1 we have $s_m \models$ *no_sharing* iff $s_s \models \neg error \wedge$ `wf`. Thus $s_m \models \Theta_m$ iff $s_s \models \Theta_s$. □

We now extend Lemma 2 to show that transformation of the transition relation preserves the mapping $\mathcal{S}$:

Lemma 4. *Let $s_m \in \Sigma_m$ and $s_s = \mathcal{S}(s_m)$, such that $s_m \models$ no_sharing. Then for any state $s'_m \in \Sigma_m$, $\mathcal{S}(s'_m)$ is a ρ_s-successor of s_s if s'_m is a ρ_m-successor of s_m. Furthermore, if $s'_m \models$ no_sharing, then the reverse direction holds as well.*

Proof. Let $s'_m \in \Sigma_m$ be a state such that $s'_m \models$ *no_sharing*. Since ρ_m is a disjunction of clauses, Let $\varphi(\mathcal{V}_m) \wedge \tau(\mathcal{V}_m, \mathcal{V}'_m) \wedge \mathit{preserve}(\mathcal{V}_m, \mathcal{V}'_m)$ be one such arbitrary clause. Then the transformed clause is given by $\varphi_s(\mathcal{V}_s) \wedge \tau_s(\mathcal{V}_s, \mathcal{V}'_s)$, where $\varphi_s(\mathcal{V}_s)$ is the transformation of $\varphi(\mathcal{V}_m)$ and $\tau_s(\mathcal{V}_s, \mathcal{V}'_s)$ is the transformation of $\tau(\mathcal{V}_m, \mathcal{V}'_m)$ (recall that the preservation conjunct, present in the original clause, is discarded by the transformation, and that τ_s encapsulates variable preservation clauses).

From Lemma 2 and Lemma 3 we have $s_m \models \varphi(\mathcal{V}_m)$ iff $s_s \models \varphi_s(\mathcal{V}_s)$. Let $s'_s = \mathcal{S}(s'_m)$. It is left to show that $(s_m, s'_m) \models \tau(\mathcal{V}_m, \mathcal{V}'_m) \wedge \mathit{preserve}(\mathcal{V}_m, \mathcal{V}'_m)$ iff $(s_s, s'_s) \models \tau_s(\mathcal{V}_s, \mathcal{V}'_s)$. Since τ is a conjunction of `Assign` formulas, we show that for each type of atomic `Assign` formula $\psi(\mathcal{V}_m, \mathcal{V}'_m)$ and its transformation $\psi_s(\mathcal{V}_s, \mathcal{V}'_s)$, $(s_m, s'_m) \models \psi(\mathcal{V}_m, \mathcal{V}'_m) \implies (s_s, s'_s) \models \psi_s(\mathcal{V}_s, \mathcal{V}'_s)$, and if $s'_m \models$ *no_sharing* then the reverse direction holds as well.

ψ **has the form** $x'_1 = t_2$ where t_2 is either an **index** variable or 0. In this case the claim holds trivially for both directions.

ψ **has the form** $B'[x]$ or $\neg B'[x]$, where B is a **bool** array and x is an **index** variable. In the case of $\neg B'[x]$, the claim follows trivially. In the case of $B'[x]$, ψ_s is the formula $(x = 0 \wedge \mathit{error}') \vee (x \neq 0 \wedge B'[x])$.

1. $s'_m \models$ *no_sharing*. Then $s'_m \models \neg B[0]$, and $s'_s \models \neg \mathit{error}$. If $(s_m, s'_m) \models B'[x]$, then x cannot be 0 in s_m, nor in s_s. From $\mathcal{S}$ we have $(s_s, s'_s) \models x \neq 0 \wedge B'[x]$. Otherwise, if $(s_s, s'_s) \models \psi_s$, then the claim follows from the definition of $\mathcal{S}$ and the fact that *error* is FALSE in s'_s.
2. $s'_m \not\models$ *no_sharing*. Then $s'_s \models$ *error*. If $s_m[x] = 0$, then from the definition of $\mathcal{S}$ we have $(s_s, s'_s) \models x = 0 \wedge \mathit{error}'$. Thus $(s_m, s'_m) \models \psi \implies (s_s, s'_s) \models \psi_s$. Otherwise, $s_m[x] = s_s[x] \neq 0$. Since, by definition of $\mathcal{S}$, $s'_m[B](s_m[x]) = s'_s[B](s_s[x])$, then $(s_m, s'_m) \models x \neq 0 \wedge B'[x]$ iff $(s_s, s'_s) \models x \neq 0 \wedge B'[x]$.

ψ **has the form** $x'_1 = \mathit{link}_j[x_2]$. We focus on the nontrivial case that $s_m[x_2] \neq 0$ and $s'_m[x'_1] \neq 0$. First assume that x_2 is a leaf, i.e., $s_m[\mathit{link}_j](s_m[x_2]) = 0$. In this case $s'_m[x_1] = 0$, and by definition of $\mathcal{S}$, $s'_s[x_1] = 0$. From the assumption, we have $s_m \models \mathit{link}_j[x_1] = 0$. Then by Lemma 2, $s_s \models \forall y \,.\, (\mathit{parent}[y] \neq x_2 \ \vee\ \mathit{ct}[y] \neq j)$. Otherwise, assume that x_2 is not a leaf, i.e., $s_m[\mathit{link}_j](s_m[x_2]) \neq 0$. Then by definition of $\mathcal{S}$, there exists a node $u \neq 0$ such that $s'_m[x_1] = u$ and $s'_m[\mathit{link}_j](s_m[x_2]) = u$. Then by definition of $\mathcal{S}$, $s_s[\mathit{parent}](u) = s_s[x_2]$, $s_s[\mathit{ct}](u) = j$, and $s'_s[x_1] = u$. Thus $(s_s, s'_s) \models \exists y \,.\, (\mathit{parent}[y] = x_2 \ \wedge\ \mathit{ct}[y] = j \ \wedge\ x'_1 = y)$. In the reverse direction, if s_m and s'_m both satisfy the *no_sharing* constraint, then the claim follows trivially from the definition of $\mathcal{S}$.

ψ **has the form** $link'_j[x_1] = x_2$. Then ψ_s is the formula

$$Err \wedge error' \qquad (1)$$

$$\vee$$

$$\left[\begin{array}{l} \quad \neg Err \\ \wedge \ (x_2 = 0 \ \vee \ (x_2 \neq 0 \ \wedge \ parent'[x_2] = x_1 \ \wedge \ ct'[x_2] = j)) \\ \wedge \ \left(\begin{array}{l} \quad \forall z \,.\, (parent[z] \neq x_1 \ \vee \ ct[z] \neq j) \\ \vee \ \exists z \,.\, (parent[z] = x_1 \ \wedge \ ct[z] = j \ \wedge \\ \qquad (z = x_2 \vee parent'[z] = 0)) \end{array}\right) \end{array}\right] \qquad (2)$$

First assume $(s_m, s'_m) \models \psi$. Let $u_1 = s_m[x_1]$ and $u_2 = s_m[x_2]$. We consider two cases:

1. Node u_2 has multiple parents in s'_m, one of which must be u_1. In this case, we have $s'_m \models$ *no_sharing*. Furthermore, by definition of $\mathcal{S}$, we have $s'_s[error] =$ TRUE and $s_s \models Err$. Thus $(s_s, s'_s) \models \psi_s$.
2. Node u_2 has a single parent in s'_m, which must be u_1. In this case it must be the case that $s_s \models \neg Err$. We now show that (s_s, s'_s) satisfies the other two conjuncts in disjunct (2) of ψ_s. The conjunct $(x_2 = 0 \ \vee \ (x_2 \neq 0 \ \wedge \ parent'[x_2] = x_1 \ \wedge \ ct'[x_2] = j))$ follows from the definition of $\mathcal{S}$. As for the third conjunct, consider first the case that u_1 has no j-child in s_m. Then by definition of $\mathcal{S}$, $s_s \models \forall z \,.\, parent[z] \neq x_1 \ \vee \ ct[z] \neq j$. Otherwise, there exists a node z that is the j-child of u_1 in s_m. If z is not u_2, then it is no longer the j-child of u_1 in s'_m. It follows from the definition of $\mathcal{S}$ that $(s_s, s'_s) \models \psi_s$.

It is left to show the reverse direction, under the assumption that $s'_m \models$ *no_sharing*. It follows that $s'_s[error] =$ FALSE. Thus, it must be the case that (s_s, s'_s) satisfies disjunct (2) of ψ_s. Let $u_1 = s_s[x_1]$ and $u_2 = s_s[x_2]$. From the definition of $\mathcal{S}$ and the conjunct $(x_2 = 0 \ \vee \ (x_2 \neq 0 \ \wedge \ parent'[x_2] = x_1 \ \wedge \ ct'[x_2] = j))$ we conclude that if $u_2 \neq 0$, then u_2 is a j-child of u_1 in s'_m. If $u_2 = 0$, then from the third conjunct we conclude that u_1 has no child in s'_m. Therefore, $(s_m, s'_m) \models \psi$. □

Corollary 1. *Let $\mu : s^0_m, s^1_m, \ldots$ be a (finite or infinite) sequence of states that consists only of non-sharing states. Then μ is a run of $\mathcal{D}_m$ iff $\mathcal{S}(\mu) : \mathcal{S}(s^0_m), \mathcal{S}(s^1_m) \ldots$ is a run of $\mathcal{D}_s$ without error states.*

Proof. The proof is by induction on the run length. At the base case, from Lemma 3 we have that $\mathcal{S}(s^0_m) \models \Theta_s$ iff $s^0_m \models \Theta_m$. Since Θ_m is defined to include the conjunct *no_sharing*, then s^0_m satisfies the non-sharing constraint, and by definition of $\mathcal{S}$ we have $\mathcal{S}(s^0_m) \models \neg error$.

For the inductive step, let $s^0_m, \ldots, s^n_m$ be a run of $\mathcal{D}_m$ that is without sharing, and let $\mathcal{S}(s^0_m), \ldots, \mathcal{S}(s^n_m)$ be a run of $\mathcal{D}_s$ that is without error states. By Lemma 4 and the definition of $\mathcal{S}$, a $\mathcal{D}_m$-state s^{n+1}_m without sharing is a ρ_m-successor of s^n_m iff $\mathcal{S}(s^{n+1}_m)$ is a ρ_s-successor of s_s such that $\mathcal{S}(s^{n+1}_m)[error] =$ FALSE. □

From Lemma 2, Corollary 1, and Observation 3 we can now prove:

Theorem 4 (Soundness). *Assume that for every reachable $\mathcal{D}_m$-state $s_m \in \Sigma_m$, $s \models$ no_sharing. Let φ_m be a temporal property over* M*-restricted A-assertions over $\mathcal{V}_m$, and let φ_s be φ_m, where every assertion over $\mathcal{V}_m$ is replaced with its transformation into a restricted EA-assertion over $\mathcal{V}_s$. Then: $\mathcal{D}_s \models \varphi_s \iff \mathcal{D}_m \models \varphi_m$*

While Theorem 4 shows that validity of temporal formulae carries from multi-linked systems into single-parent ones only when the former satisfy non-sharing, we prove that if the latter never reaches an error state, then the former never violates non-sharing:

Theorem 5 (Non-sharing). *If $\mathcal{D}_s \models \square \neg$error then $\mathcal{D}_m \models \square$ no_sharing.*

Proof. Assume that $\mathcal{D}_m$ has a computation with a prefix $s_m^0, \ldots, s_m^n$, where for any $0 \leq i < n$, $s_m^i \models$ *no_sharing* and $s_m^n \not\models$ *no_sharing*. Following Corollary 1, the sequence $\mathcal{S}(s_m^0), \ldots, \mathcal{S}(s_m^{n-1})$ is an error-free run of $\mathcal{D}_s$. From Lemma 4, $\mathcal{S}(s_m^n)$ is a successor in $\mathcal{D}_s$ of $\mathcal{S}(s_m^{n-1})$. From the definition of $\mathcal{S}$ we have $\mathcal{S}(s_m^n) \models$ *error*. □

Thus, to verify $\mathcal{D}_m \models \varphi_m$, one would initially perform a "sanity check" by verifying $\mathcal{D}_s \models \square \neg$*error*. If this is successful, then the process outlined in Section 1 can be carried out. Theorem 4 guarantees not only that correctness of $\mathcal{D}_s$ implies correctness of $\mathcal{D}_m$, but also that a counterexample over $\mathcal{D}_s$ is mappable back into $\mathcal{D}_m$.

7 Conclusion

We describe a transformation from programs that perform destructive updates over multi-linked heaps without sharing into single-parent heaps that is based on the idea of simulating a tree (or forest) by a set of converging lists. We then apply an abstraction-based verification framework to automatically verify properties of systems over multi-linked heaps.

We applied our technique to verify properties of insertion into AVL trees. We are currently implementing more benchmarks, including an implementation of 2-3 trees. We are also extending the transformation to allow for unbounded out-degrees in the multi-linked heap, and to heaps whose "backbone" is single-parent, which would allow us to model algorithms that "flip" heap edges (a surprisingly useful feature). In the longer term, we would like to investigate how to use multi-linked heap systems as the basis for further structure simulation (e.g., as in [19, 12]).

Acknowledgement: We would like to thank Viktor Kuncak and Greta Yorsh for their insight regarding structure simulation. We also would like to thank the anonymous reviewers for their constructive comments.

References

1. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13rd Intl. Conference on Computer Aided Verification*, pages 221–234. LNCS 2102, 2001.
2. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In R. Cousot, editor, *Proc. of the 6th Int. Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lect. Notes in Comp. Sci.*, pages 164–180. Springer, 2005.

3. I. Balaban, A. Pnueli, and L. D. Zuck. Modular ranking abstraction. To appear in International Journal of Foundations of Computer Science (IJFCS), 2007. See *http://www.cs.nyu.edu/acsys/pubs/permanent/ranking-companion-pre.pdf*.
4. T. Ball and R. B. Jones, editors. *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*. Springer, 2006.
5. M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In S. D. Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 1999.
6. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In Ball and Jones [4], pages 386–400.
7. J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2006.
8. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives of Mathematical Logic. Springer-Verlag, 1997. Second printing (Universitext) 2001.
9. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In Ball and Jones [4], pages 517–531.
10. E. Grädel, M. Otto, and E. Rosen. Undecidability results on two-variable logics. In R. Reischuk and M. Morvan, editors, *STACS*, volume 1200 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 1997.
11. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In J. Marcinkowski and A. Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2004.
12. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *Proc.* 16th *Intl. Conference on Computer Aided Verification*, Lect. Notes in Comp. Sci., pages 281–294. Springer-Verlag, 2004.
13. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
14. N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc.* 20th *ACM Symp. Princ. of Prog. Lang.*, pages 196–205, New York, NY, USA, 1993. ACM Press.
15. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *Proc. of the* 6th *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lect. Notes in Comp. Sci.*, pages 181–198. Springer, 2005.
16. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.
17. 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.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
19. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. In *Proc. of the* 7th *Int. Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.
20. G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In L. Aceto and A. Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006.