A Better Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs⋆

Technical Report UBC-CS-TR-2006-02

Zvonimir Rakamarić[1], Jesse Bingham[2], and Alan J. Hu[1]

[1] Department of Computer Science, University of British Columbia, Canada
{zrakamar,ajh}@cs.ubc.ca
[2] Intel Corporation, Hillsboro, Oregon, USA
jesse.d.bingham@intel.com

Abstract. *Heap-manipulating programs* (HMP), which manipulate unbounded linked data structures via pointers, are a major frontier for software model checking. In recent work, we proposed a small logic and inference-rule-based decision procedure and demonstrated their potential by verifying, via predicate abstraction, some simple HMPs. In this work, we generalize and improve our previous results to be practically useful: we allow more than a single pointer field, we permit updating the data stored in heap nodes, we add new primitives and inference rules for cyclic structures, and we greatly improve the performance of our implementation. Experimentally, we are able to verify many more HMP examples, including three small container functions from the Linux kernel. On the theoretical front, we prove NP-hardness for a small fragment of our logic, completeness of our inference rules for a large fragment, and soundness for the full logic.

1 Introduction

Software model checking has emerged as a vibrant area of formal verification research. Much of the success of applying model checking to software has come from using *predicate abstraction* [11, 7, 3, 13]. Predicate abstraction, in turn, requires a logic and associated decision procedure to define predicates over the (typically infinite) concrete program state. The logic must be expressive enough to allow useful abstractions, but performance of the decision procedure is also paramount, since most predicate abstraction approaches make numerous queries to the decision procedure.

An important class of programs are those we call *heap-manipulating programs* (HMPs): programs that access and modify linked data structures consisting of an unbounded number of uniform *heap nodes*. HMPs access the heap nodes through a finite number of pointers (which we call *node variables*) and following pointer fields between nodes. To apply predicate abstraction to HMPs and assert many interesting correctness properties, one must be able to express the concept of unbounded *reachability* (a.k.a.

⋆ This work was supported by a research grant from the Natural Sciences and Engineering Research Council of Canada and a University of British Columbia Graduate Fellowship.

transitive closure) between nodes. This is done through a binary operator that takes two node terms x and y, and asserts that the second can be reached from the first by following zero or more links; in our syntax this is written as $f^*(x,y)$ (f is the name of the *link function*). For example, $f^*(f(x),x)$ expresses that x is a node in a cyclic linked list.

Several papers have previously identified the importance of transitive closure for HMPs [22, 23, 4, 14, 2, 16, 17]. Unfortunately, adding support for transitive closure to even simple logics often yields undecidability [14]. Recently, we proposed a very minimalist transitive closure logic for HMPs, to demonstrate a different decision procedure approach [6].[3] Rather than using a small-model theorem and enumerating a super-factorial number of possible models (e.g., [4, 2]), our decision procedure was based on inference rules. This paper expands and generalizes our preliminary result to be practically useful. In particular, we now allow more than a single pointer field and permit updating the data stored in heap nodes. Furthermore, we found that for many interesting properties of cyclic linked structures, we needed to introduce a generalized, ternary transitive closure between operator $\mathsf{btwn}_f(x,y,z)$ (similar to Nelson's [23]). Finally, we created a new implementation, with greatly improved performance. We also report some theoretical results for our new logic and decision procedure: satisfiability is NP-hard even for a small fragment of our logic, our decision procedure is sound and always terminates, and the decision procedure is complete for the fragment of the logic without updates. In practice, completeness was not an issue, as we could verify all examples that we could specify. Overall, our improvements enable us to verify a much larger variety of HMPs, such as three small container functions from the Linux kernel.

The appendix provides proofs of the theorems, additional details regarding the decision procedure, pseudocode for the example programs, and the sets of predicates needed for their verification.

1.1 Related Work

There is an extensive literature on verification of HMPs, most of which are based on static analysis and abstract interpretation, such as the well-known TVLA tool [18]. For space reasons, we focus here on approaches, like ours, based on decision procedures and predicate abstraction. We believe such approaches can be competitive with more established approaches, and offer possible benefits such as integration with more general theorem provers.

Balaban et al. [2] present an approach for shape analysis based on predicate abstraction that is similar to ours. To compute the abstraction, they employ a small model theorem, which is a bottleneck in both computation time and memory. Manevich et al. [20] also employ predicate abstraction and verify a number of HMPs, though they only handle (possibly cyclic) singly-linked lists, and their properties tend to be simpler than ours (see Sect. 5).

McPeak and Necula [21] describe a decision procedure which enables them to reason effectively about local heap properties, and to describe a variety of heap structures. However, they can only approximate reachability between nodes, and when pointer dis-

[3] The published paper has some minor errors, which are corrected in a later technical report [5].

equalities are added, their decision procedure becomes incomplete. Heap locality is also employed by recent approaches based on separation logic [8, 19].

MONA [15] is a well-known decision procedure for monadic second-order logic with non-elementary complexity, in which properties of lists, trees and graphs can be expressed. Ranise and Zarba [24] describe a decidable logic for reasoning about acyclic singly-linked lists with an NP-complete decision problem.

First-order axiomatization of reachability was first proposed by Nelson [23]. Lahiri and Qadeer [16] define two new predicates to express reachability of heap nodes in linked lists. To prove properties of HMPs, they use an incomplete set of first-order axioms over those predicates. Lev-Ami et al. [17] also propose a set of axioms, but it works only for acyclic lists. These approaches harness the generality and expressiveness of more general first-order theorem provers, at a possible sacrifice in performance.

1.2 Verification Approach

To assess the usefulness of our logic and decision procedure for predicate abstraction of HMPs, we need to implement them within a verification tool. We have created a very standard implementation of predicate abstraction [11], in which a set of predicates $\phi_1, \ldots, \phi_k$ over the (typically infinite) concrete program state $\mathscr{C}$ are used as an abstraction function $\alpha : \mathscr{C} \to \mathscr{A}$ from the concrete states to abstract states, where $\mathscr{A}$ is the Boolean space $\{0,1\}^k$. As in recent papers on this topic [2, 16], we assume the predicates ϕ_i are given. Automatic abstraction refinement via predicate discovery is an important area of future work.

Verification requires repeatedly computing the abstract post-image operator:

$$\mathsf{post}(A) \;=\; \left\{\alpha(c') \mid \exists c, c' \in \mathscr{C}.(c, c') \in T \wedge \alpha(c) \in A\right\}$$

where T is the transition relation of the concrete system. We compute post precisely using the approach of Das et al. [7], which repeatedly queries the satisfiability of formulas of the form: $\ell_1 \wedge \cdots \wedge \ell_m$, where each ℓ_j is a literal.[4] These conjunction-of-literals are formed by factoring BDD representations of sets of abstract states and taking weakest preconditions [12] of individual program statements. Using this technique, it is sufficient for the decision procedure to merely check for satisfiability of conjunction-of-literals. We refer the reader to our previous paper [6] for details.

2 Heap-Manipulating Programs

In lieu of a formal presentation of HMPs, we give an example called INIT-CYCLIC in Fig. 1 that captures some of the new, interesting features we support. The program takes a non-nil node *head* that points to a cyclic list and sets the (boolean) data fields of all nodes in the list to $\mathtt{true}$. We denote a data field named d of a node x by $d(x)$, and the node pointed to by a link field named f of node x by $f(x)$. Necessary assumptions are formalized by the **assume** statement on line 2 of the program. Subformulas of the form $f^*(x,y)$ express that node y is reachable from node x by following a sequence of any

[4] A *literal* is a predicate ϕ_i or its negation $\neg\phi_i$.

1: **procedure** INIT-CYCLIC(*head*)
2: **assume** $f^*(head,t) \ \wedge\ f^*(f(head),head) \ \wedge\ curr=f(head)$
$\wedge\ \mathsf{btwn}_f(curr,t,head) \ \wedge\ \neg head=\mathsf{nil} \ \wedge\ true$
3: $d(x) := true$;
4: **while** $\neg curr=head$ **do**
5: $d(curr) := true$;
6: $curr := f(curr)$;
7: **end while**
8: **assert** $f^*(head,t) \ \wedge\ f^*(f(head),head) \ \wedge\ d(t) \ \wedge\ \neg head=\mathsf{nil}$
9: **end procedure**

Fig. 1. A program that sets data fields of all nodes in a cyclic list to true

$$
\begin{array}{rcl}
term & ::= & v \mid f(term) \\
atom & ::= & f^*(term,term) \mid term=term \mid \mathsf{btwn}_f(term,term,term) \mid d(term) \mid b \\
literal & ::= & atom \mid \neg atom
\end{array}
$$

Fig. 2. The syntax of our simple transitive closure logic, with $v \in V$, $d \in D$, $b \in B$, and $f \in F$.

length of f links. The subformulas of the form $\mathsf{btwn}_f(x,y,z)$ express that by following a sequence of f links from node x, we'll reach node y before we reach node z, i.e. node y comes between nodes x and z. Because of cyclicity, btwn_f predicates are the key to successful verification of this program. We will formally define these predicates in Sect. 3. The fact that *head* is reachable from $f(head)$ enforces the cyclicality assumption.

The body of INIT-CYCLIC is straightforward. First, the data field of *head* is set to true on line 3. Then, the loop sets the data fields of all other nodes in the list to true. The specification is expressed by the **assert** statement on line 8, and indicates that whenever line 8 is reached, *head* must point to a cyclic list with data fields of all nodes set to true.

The verification problem we wish to solve can be stated as follows: given an HMP, determine whether it is the case that all executions that satisfy all **assume** statements also satisfy all **assert** statements. Since the number of nodes in the heap is unbounded, HMPs are generally infinite state, thus one cannot directly apply finite-state model checking to this problem without using abstraction.

3 A Simple Transitive Closure Logic

Our logic assumes finite sets of *node* variables V, *data* variables B, *data function* symbols D, and *link function* symbols F. The *term*, *atom*, and *literal* syntactic entities are given in Fig. 2. Literals of the form $x=y$, $\neg x=y$, $f^*(x,y)$, and $\neg f^*(x,y)$ (where x and y are terms) are called *equality*, *disequality*, *reachability*, and *unreachability* literals, respectively. Literals of the form $\mathsf{btwn}_f(x,y,z)$ or its negation are called *between* literals, literals of the form $d(x)$ or $\neg d(x)$, where $d \in D$, are called *data* literals, while those of the form b or $\neg b$ are called simply *data variable* literals.

The structures over which the semantics of our logic is defined are called *heap structures*. A heap structure $H = (N,\Theta)$ involves a set of *nodes* N and a function Θ that interprets each symbol σ in $V \cup B \cup D \cup F$ such that

$$\begin{array}{ll} \Theta(\sigma) \in N & \text{if } \sigma \in V \\ \Theta(\sigma) \in \{\mathsf{true}, \mathsf{false}\} & \text{if } \sigma \in B \\ \Theta(\sigma) \in N \to \{\mathsf{true}, \mathsf{false}\} & \text{if } \sigma \in D \\ \Theta(\sigma) \in N \to N & \text{if } \sigma \in F \end{array}$$

Thus Θ interprets each node variable as a node, each data variable as a boolean value, each data function variable as a function that maps nodes to booleans, and each link function symbol as a mapping from nodes to nodes. Heap structures naturally model a linked data structure of nodes, each node having some finite number of pointers to other nodes and some finite number of boolean-valued fields. The *size* of H is defined to be $|N|$. The variables of V model program variables that point to nodes in the data structure, while the variables of B model program variables of boolean type. Link function symbols F model pointers from nodes to nodes, and data function symbols D model data fields of nodes. Clearly, program variables or node fields of any finite enumerated type can be encoded using the booleans accommodated by our logic.

We extend Θ to Θ^e, which interprets any term or atom in a straightforward way, formally defined here. The interpretation of a term τ is defined inductively by:

$$\Theta^e(\tau) = \begin{cases} \Theta(\tau) & \text{if } \tau \in V \\ \Theta(f)(\Theta^e(\tau')) & \text{if } \tau \text{ has the form } f(\tau') \text{ for some term } \tau' \end{cases}$$

Θ^e interprets atoms as boolean values. An equality atom $\tau_1 = \tau_2$ is interpreted as true by Θ^e iff $\Theta^e(\tau_1) = \Theta^e(\tau_2)$. A data atom is interpreted by defining $\Theta^e(d(\tau)) = \Theta(d)(\Theta^e(\tau))$. A reachability atom $f^*(\tau_1, \tau_2)$ is interpreted as true iff there exists some $n \geq 0$ such that $\Theta(f)^n(\Theta^e(\tau_1)) = \Theta^e(\tau_2)$.[5] A between atom $\mathsf{btwn}_f(\tau_1, \tau_2, \tau_3)$ is interpreted as true iff there exist $n_0, m_0 \geq 0$ such that $\Theta^e(\tau_2) = \Theta(f)^{n_0}(\Theta^e(\tau_1))$, $\Theta^e(\tau_3) = \Theta(f)^{m_0}(\Theta^e(\tau_1))$, $n_0 \leq m_0$, and for all n, m such that $\Theta^e(\tau_2) = \Theta(f)^n(\Theta^e(\tau_1))$, $\Theta^e(\tau_3) = \Theta(f)^m(\Theta^e(\tau_1))$, $n_0 \leq n$ and $m_0 \leq m$. Finally, a literal that is not an atom must be of the form $\neg\phi$ where ϕ is an atom, and we simply define $\Theta^e(\neg\phi) = \neg\Theta^e(\phi)$.

Sticking to the usual notation, given a heap structure $H = (N, \Theta)$ and a literal ϕ, we write $H \vDash \phi$ iff $\Theta^e(\phi) = \mathsf{true}$. For a set of literals Φ, we write $H \vDash \Phi$ iff $H \vDash \phi$ for all $\phi \in \Phi$. Given Φ, if there exists H such that $H \vDash \Phi$, we say that Φ is *satisfiable*. In Sect. 4, we will describe our decision procedure for satisfiability, which has a worst-case exponential running time. Here, we note that the problem it solves is NP-hard, hence a polytime algorithm is unlikely to exist. For proof, please see the appendix.

Theorem 1. *Given a set of literals Φ, the problem of deciding if Φ is satisfiable is NP-hard. This holds even when Φ contains no updates, no* btwn *predicates, no data fields, and only mentions a single link function f.*

3.1 Handling Link and Data Function Updates

To handle program assignments that modify the links in the heap, i.e. modify some $f \in F$, we use a special link function symbol f'. The symbols f and f' respectively

[5] Here, function exponentiation represents iterative application: for a function g and an element x in its domain, $g^0(x) = x$, and $g^n(x) = g(g^{n-1}(x))$ for all $n \geq 1$.

model the link function before and after the assignment. Such an assignment has the general form $f(\tau_1) := \tau_2$, where τ_1 and τ_2 are arbitrary terms. The necessary semantic relationship between f and f' can be expressed using the well-known update operator:[6]

$$\Theta^e(f') = \mathsf{update}(\Theta^e(f), \Theta^e(\tau_1), \Theta^e(\tau_2)) \qquad (1)$$

Rather than support update as an interpreted second order function symbol in the logic, our decision procedure implicitly enforces the constraint (1).

Assignments that modify a data field $d \in D$, are handled similarly by using the primed symbol d'. Such an assignment has the general form $d(\tau) := b$, where τ is a term, and b is a data variable. Lines 3 and 5 of the HMP of Fig. 1 are examples of such assignments. Analogously to (1), the semantic relationship between d and d' is

$$\Theta^e(d') = \mathsf{update}(\Theta^e(d), \Theta^e(\tau), \Theta^e(b)) \qquad (2)$$

Our decision procedure knows to implicitly enforce the constraint (2) when it encounters the symbols d and d'.

4 Decision Procedure

The decision problem we aim to solve with our decision procedure is this: given a finite set of literals Φ, does there exist a heap structure H such that $H \models \Phi$? If there is such an H, then we say that Φ is *satisfiable*, otherwise Φ is *unsatisfiable*. Clearly, any algorithm for this problem can be used to decide the satisfiability of a conjunction-of-literals by simply taking Φ to be the set of its conjuncts.

One approach to this problem would be through a small model theorem, akin to other transitive closure logics [2, 4, 24]. Unfortunately, even a very small "small model" bound can generate impractical memory requirements, because the number of heap structures with n nodes is at least n^n.

Our approach has small memory requirements, and is based on exhaustive application of a set of inference rules (IRs). The IRs attempt to prove unsatisfiability by deriving a *contradiction*, meaning the inference of both an atom ϕ and its negation $\neg\phi$. The basic IRs are presented in Fig. 3. The *antecedents* of IR r are the literals appearing above the line, while the *consequents* are those appearing below the line. We say that an IR r is *applicable* (to Φ) if there are terms appearing in Φ such that when these terms are substituted for the term placeholders of r (i.e. x, y, z, x_1, etc.), *all* of r's antecedents appear in Φ, and *none* of r's consequents appear in Φ.

We now give a brief intuition behind some of the IRs. TRANS1 states that the transitive closure f^* must extend the function f. If there is a cycle of length $k \geq 1$ in f, then it follows that any node y reachable from a node on the cycle must be on the cycle as well; this is formalized by CYCLE_k. The fact that if y and z are both reachable from x, either y lies between x and z, or z lies between x and y is captured by BTW5. The following theorem is proven in the appendix.

[6] If g is a function, a is an element in g's domain, and b is an element in g's codomain, then $\mathsf{update}(g,a,b)$ is defined to be the function $\lambda x.(\text{if } x = a \text{ then } b \text{ else } g(x))$.

$$\frac{}{x=x}\textsc{Ident} \qquad \frac{}{f^*(x,x)}\textsc{Reflex} \qquad \frac{f(x)=y}{f^*(x,y)}\textsc{Trans1}$$

$$\frac{f^*(x,y) \quad f^*(y,z)}{f^*(x,z)}\textsc{Trans2} \qquad \frac{f(x)=y \quad f^*(x,z)}{x=z \quad f^*(y,z)}\textsc{Func}$$

$$\frac{f(x_1)=x_2 \quad f(x_2)=x_3 \quad \cdots \quad f(x_k)=x_1 \quad f^*(x_1,y)}{y=x_1 \quad y=x_2 \quad \cdots \quad y=x_k}\textsc{Cycle}_k$$

$$\frac{f^*(x,y) \quad f^*(y,x) \quad f^*(x,z)}{x=y \quad f^*(z,x)}\textsc{Scc} \qquad \frac{f^*(x,y) \quad f^*(x,z)}{f^*(y,z) \quad f^*(z,y)}\textsc{Total}$$

$$\frac{f(x)=z \quad f(y)=z \quad f^*(x,y) \quad f^*(y,x)}{x=y}\textsc{Share} \qquad \frac{d(x) \quad \neg d(y)}{\neg x=y}\textsc{NotEqNodes}$$

$$\frac{}{\mathsf{btwn}_f(x,x,x)}\textsc{BtwReflex} \qquad \frac{f^*(x,y) \quad f^*(y,z) \quad f(z)=x}{\mathsf{btwn}_f(x,y,z)}\textsc{Btw1}$$

$$\frac{\mathsf{btwn}_f(x,y,z)}{\begin{array}{c} f^*(x,y) \\ f^*(y,z) \end{array}}\textsc{Btw2} \qquad \frac{f(x)=w \quad \mathsf{btwn}_f(x,y,z)}{\mathsf{btwn}_f(w,y,z) \quad x=y}\textsc{Btw3}$$

$$\frac{\mathsf{btwn}_f(x,y,z) \quad \mathsf{btwn}_f(x,z,y)}{y=z}\textsc{Btw4} \qquad \frac{f^*(x,y) \quad f^*(x,z)}{\mathsf{btwn}_f(x,y,z) \quad \mathsf{btwn}_f(x,z,y)}\textsc{Btw5}$$

$$\frac{f^*(x,y) \quad f^*(y,z) \quad f^*(z,x)}{\begin{array}{cc} \mathsf{btwn}_f(x,y,z) & \mathsf{btwn}_f(x,z,y) \\ \mathsf{btwn}_f(y,z,x) & \mathsf{btwn}_f(z,y,x) \\ \mathsf{btwn}_f(z,x,y) & \mathsf{btwn}_f(y,x,z) \end{array} \quad x=y \quad x=z \quad y=z}\textsc{Btw6} \qquad \frac{f^*(x,y)}{\begin{array}{c} \mathsf{btwn}_f(x,x,y) \\ \mathsf{btwn}_f(x,y,y) \end{array}}\textsc{Btw7}$$

$$\frac{\mathsf{btwn}_f(x,y,z) \quad f(x)=z}{y=x \quad y=z}\textsc{Btw8} \qquad \frac{f(z)=w \quad \mathsf{btwn}_f(x,y,w) \quad f^*(x,z)}{\mathsf{btwn}_f(x,y,z) \quad y=w}\textsc{Btw9}$$

$$\frac{\mathsf{btwn}_f(x,y,z) \quad \mathsf{btwn}_f(w,z,y) \quad f^*(x,w)}{f^*(z,w) \quad y=z}\textsc{Btw10} \qquad \frac{\mathsf{btwn}_f(w,x,y) \quad \mathsf{btwn}_f(w,y,z)}{\mathsf{btwn}_f(w,x,z)}\textsc{Btw11}$$

$$\frac{\mathsf{btwn}_f(v,u,x) \quad \mathsf{btwn}_f(v,u,y) \quad \mathsf{btwn}_f(u,x,y)}{\mathsf{btwn}_f(v,x,y)}\textsc{Btw12}$$

Fig. 3. Here x, y, z, etc. range over variables V, $f \in F$ ranges over link fields, and $d \in D$ ranges over data fields. Note that $\textsc{Cycle}_k$ actually defines a separate rule for each $k \geq 1$.

Theorem 2. *The inference rules of Fig. 3 are sound.*

Theorem 2 tells us that if iterative application of the IRs yields a contradiction, then we can conclude that the original set of literals is unsatisfiable.

To prove completeness, as in our previous work [6], we first reduce the problem to sets of literals in a certain normal form, then prove completeness for only normal sets. Because of the new features added to the logic and decision procedure, the definition of the normal form has been revised slightly:

Let $\mathsf{Vars}(\Phi)$ denote the subset of the node variables V appearing in Φ.

Definition 1 (normal) *A set of literals Φ is said to be* normal *if all terms appearing in Φ are variables, except that for each $f \in F$ and $v \in \mathsf{Vars}(\Phi)$ there may exist at most one equality literal of the form $f(v) = u$, where $u \in \mathsf{Vars}(\Phi)$.*

Theorem 3. *There exists a polynomial-time algorithm that transforms any set Φ into a normal set Φ' such that Φ' is satisfiable if and only if Φ is satisfiable.*

Thanks to Theorem 3, our decision procedure can without loss of generality assume that Φ is normal. Let us call a set of literals Φ *consistent* if it does not contain a contradiction, and call Φ *closed* if none of the IRs of Fig. 3 are applicable. Thus, we have our completeness theorem:

Theorem 4. *If Φ is consistent, closed, and normal, then Φ is satisfiable.*

Viewed from a high level, our decision procedure first applies the transformation of Theorem 3, and then repeatedly searches for an applicable IR, applies it (i.e. adds one of its consequents to the set), and recurses. If the procedure ever infers a contradiction, it backtracks to the last branching IR with an unexplored consequent, or returns *unsatisfiable* if there is no such IR. If the procedure reaches a point where there are no applicable IRs and no contradictions, then the inferred set of literals is consistent, closed, and normal. Hence, it may correctly return *satisfiable*. For a formal presentation of the decision procedure, see Appendix B.

Theorem 5. *The decision procedure always terminates.*

4.1 A Decision Procedure Extension

To enforce the update constraints (1) and (2), we add a number of additional IRs to our decision procedure. For each of the IRs of Fig. 3 that mention f, we add an analogous IR with f replaced with f'; these enforce analogous constraints between f'^*, $\mathsf{btwn}_{f'}$, f', and $=$ as are enforced by the unmodified IRs of Fig. 3 between f^*, btwn_f, f, and $=$. To enforce the constraint (1), the eight IRs of Fig. 4 are also included. The IRs introduce a fresh variable w that is forced to be equal to $f(\tau_1)$. Similarly, to enforce the constraint (2), for the IR NOTEQNODES of Fig. 3 we add an analogous IR with d replaced with d'; the four IRs of Fig. 5 are also added.

Theorem 6. *The inference rules of Fig. 4 and Fig. 5 are sound.*

$$\frac{}{\begin{array}{l} f'(\tau_1)=\tau_2 \\ f(\tau_1)=w \end{array}}\ \textsc{Update} \qquad \frac{\mathsf{btwn}_f(x,y,z) \qquad \neg x=z}{\begin{array}{ll} \mathsf{btwn}_{f'}(x,y,z) & \begin{array}{l} \mathsf{btwn}_f(x,\tau_1,z) \\ \neg\tau_1=z \end{array} \end{array}}\ \textsc{UpdBtwn}$$

$$\frac{f(x)=y}{\begin{array}{ll} \begin{array}{l} x=\tau_1 \\ y=w \end{array} & f'(x)=y \end{array}}\ \textsc{UpdFunc1} \qquad \frac{f'(x)=y}{\begin{array}{ll} \begin{array}{l} x=\tau_1 \\ y=\tau_2 \end{array} & f(x)=y \end{array}}\ \textsc{UpdFunc2}$$

$$\frac{f^*(x,y)}{\begin{array}{ll} \begin{array}{l} f'^*(x,\tau_1) \\ f'^*(w,y) \end{array} & f'^*(x,y) \end{array}}\ \textsc{UpdTrans1} \qquad \frac{f'^*(x,y)}{\begin{array}{ll} \begin{array}{l} f^*(x,\tau_1) \\ f^*(\tau_2,y) \end{array} & f^*(x,y) \end{array}}\ \textsc{UpdTrans2}$$

$$\frac{f^*(x,\tau_1) \qquad f'^*(x,y)}{f^*(x,y) \qquad f'^*(\tau_1,y)}\ \textsc{UpdTrans3} \qquad \frac{f'^*(x,\tau_1) \qquad f^*(x,y)}{f'^*(x,y) \qquad f^*(\tau_1,y)}\ \textsc{UpdTrans4}$$

Fig. 4. The update inference rules, which are used to extend our logic to support a second function symbol f' for each $f \in F$, with the implicit constraint $f' = \mathsf{update}(f, \tau_1, \tau_2)$, where τ_1 and τ_2 are variables, and w is a fresh variable used to capture $f(\tau_1)$.

$$\frac{}{\begin{array}{ll} \begin{array}{l} d'(\tau) \\ b \end{array} & \begin{array}{l} \neg d'(\tau) \\ \neg b \end{array} \end{array}}\ \textsc{EqData} \qquad \frac{\neg\tau=x}{\begin{array}{ll} \begin{array}{l} d(x) \\ d'(x) \end{array} & \begin{array}{l} \neg d(x) \\ \neg d'(x) \end{array} \end{array}}\ \textsc{PreserveValue}$$

$$\frac{d(x) \qquad \neg d'(x)}{\tau=x}\ \textsc{EqNodes1} \qquad \frac{\neg d(x) \qquad d'(x)}{\tau=x}\ \textsc{EqNodes2}$$

Fig. 5. The data update inference rules, which are used to extend our logic to support a second data function symbol d' for each $d \in D$, with the implicit constraint $d' = \mathsf{update}(d, \tau, b)$, where $\tau \in V$ and b is a boolean variable.

The proof of this theorem is provided in Appendix A.

We don't have a proof that this extended set of IRs is complete. Fortunately, not having such a theorem *does not* compromise the soundness of verification by predicate abstraction. In practice, in our experiments of Sect. 5, we never found any property violations caused by the extended decision procedure erroneously concluding that a set of literals was satisfiable.

5 Experiments

We have tested our tool on a number of HMP examples and summarized the results in Table 1. We ran the experiments on a 2.6 Ghz Pentium 4 machine. The safety properties we checked (when applicable) at the end of the HMP are roughly:

- *no leaks* (NL) – all nodes reachable from the head of the list at the beginning of the program are also reachable at the end of the program.
- *insertion* (IN) – a distinguished node that is to be inserted into a list is actually reachable from the head of the list, i.e. the insertion "worked".

– *acyclic* (AC) – the final list is acyclic, i.e. nil is reachable from the head of the list.
– *cyclic* (CY) – list is a cyclic singly-linked list, i.e. the head of the list is reachable from its successor.
– *doubly-linked* (DL) – the final list is a doubly-linked list.
– *cyclic doubly-linked* (CD) – the final list is a cyclic doubly-linked list.
– *sorted* (SO) – list is a sorted linked list, i.e. each node's data field is less than or equal to its successor's.
– *data* (DT) – data fields of selected (possibly all) nodes in a list are set to a value.
– *remove elements* (RE) – for examples that remove node(s), this states that the node(s) was (were) actually removed. For the program REMOVE-ELEMENTS, RE also asserts that the data field of all removed elements is false.

Often, the properties one is interested in verifying for HMPs involve universal quantification over the heap nodes. For example, we must often express that for all nodes t, t is reachable from the head of the list. Since our logic doesn't support quantification, we use the trick of introducing a Skolem constant t [9, 2] to represent a universally quantified variable, which is described in more detail in our previous paper [6].

The first nine examples in the table are taken from our recent paper [6]; they perform operations on acyclic singly linked lists — reverse, add elements, remove elements, sort, merge, etc. Therefore, we have been able to verify them and without using the extensions described in this paper. However, compared to our previous results on this examples, it can be seen that the new decision procedure substantially improved the performance.

Other examples from the table involve data field updates, cyclic lists, and doubly-linked lists, and we couldn't handle them using the old decision procedure. However, we have been successful in verifying them using described new features added to our logic and decision procedure. These example programs are the following:

REMOVE-ELEMENTS – removes from a cyclic list elements whose data field is false.
REMOVE-SEGMENT – removes the first contiguous segment of elements whose data field is true from a cyclic singly-linked list. This example is taken from a paper by Manevich et al. [20].
SEARCH-AND-SET – searches for an element with specified data fields in a cyclic singly-linked list, and sets data fields of previous elements to true.
SET-UNION – joins two cyclic lists. This example is taken from a paper by Nelson [23].
CREATE-INSERT, CREATE-INSERT-DATA, CREATE-FREE – create new nodes (*malloc*), initialize their data fields, and insert them nondeterministically into a linked list. Also, remove nodes from a linked list and *free* them.[7]
INIT-LIST, INIT-LIST-VAR, INIT-CYCLIC – initialize data fields of acyclic and cyclic singly-linked lists, and set values of data variables.
SORTED-INSERT-DNODES – inserts an element into a sorted linked list so that sortedness is preserved. Every node in the linked list has an additional pointer to a node that contains a data field which is used for sorting.
REMOVE-DOUBLY – removes an element from an acyclic doubly-linked list.

[7] *malloc* and *free* are modelled as removing and adding nodes to an unreachable infinite cyclic list [25].

program	property	CFG edges	preds	time(s)	ptime(s)	DP calls
LIST-REVERSE	NL	6	8	0.2	1.1	184
LIST-ADD	NL ∧ AC ∧ IN	7	8	0.1	0.8	66
ND-INSERT	NL ∧ AC ∧ IN	5	13	0.5	7.9	259
ND-REMOVE	NL ∧ AC ∧ RE	5	12	0.9	19.3	386
ZIP	NL ∧ AC	20	22	17.3	280.7	9153
SORTED-ZIP	NL ∧ AC ∧ SO ∧ IN	28	22	22.8	249.9	14251
SORTED-INSERT	NL ∧ AC ∧ SO ∧ IN	10	20	13.8	217.6	5990
BUBBLE-SORT	NL ∧ AC	21	18	11.1	204.6	3444
BUBBLE-SORT	NL ∧ AC ∧ SO	21	24	114.9	441.0	31446
REMOVE-ELEMENTS*	NL ∧ CY ∧ RE	15	17	8.8	—	3062
REMOVE-SEGMENT*	CY	17	15	2.2	—	902
SEARCH-AND-SET*	NL ∧ CY ∧ DT	9	16	5.3	—	4892
SET-UNION*	NL ∧ CY ∧ DT ∧ IN	9	21	1.4	—	374
CREATE-INSERT*	NL ∧ AC ∧ IN	9	24	14.8	—	3020
CREATE-INSERT-DATA*	NL ∧ AC ∧ IN	11	27	39.7	—	8710
CREATE-FREE*	NL ∧ AC ∧ IN ∧ RE	19	31	457.4	—	52079
INIT-LIST*	NL ∧ AC ∧ DT	4	9	0.1	—	81
INIT-LIST-VAR*	NL ∧ AC ∧ DT	5	11	0.2	—	244
INIT-CYCLIC*	NL ∧ CY ∧ DT	5	11	0.2	—	200
SORTED-INSERT-DNODES*	NL ∧ AC ∧ SO ∧ IN	10	25	77.9	—	7918
REMOVE-DOUBLY*	NL ∧ DL ∧ RE	10	34	24.3	—	3238
REMOVE-CYCLIC-DOUBLY*	NL ∧ CD ∧ RE	4	27	15.6	—	1695
LINUX-LIST-ADD*	NL ∧ CD ∧ IN	6	25	6.4	—	1240
LINUX-LIST-ADD-TAIL*	NL ∧ CD ∧ IN	6	27	7.3	—	1598
LINUX-LIST-DEL*	NL ∧ CD ∧ RE	6	29	24.7	—	2057

Table 1. Results of verifying HMPs. "property" specifies the verified property; "CFG edges" denotes the number of edges in the control-flow graph of the program; "preds" is the number of predicates required for verification; "time" is the total execution time; "ptime" is the total execution time from our previous paper; "DP calls" is the number of decision procedure queries. Examples marked with * can't be verified without using the new decision procedure extensions.

REMOVE-CYCLIC-DOUBLY – removes an element from a cyclic doubly-linked list. This example is taken from a paper by Lahiri and Qadeer [16].

LINUX-LIST-ADD, LINUX-LIST-ADD-TAIL, LINUX-LIST-DEL – examples from Linux kernel list container which add and remove nodes from a cyclic doubly-linked list.

The data fields in all of our examples are booleans. Appendix C provides pseudocode and lists the required predicates for all examples.

As Table 1 shows, we were successful in verifying interesting properties of many examples in reasonable amounts of time. It is hard to make a good comparison with other tools and approaches because the heap structures the tools are able to handle, their expressiveness, and the amount of required manual effort varies greatly. Here, we make a few comparisons to tools similar to ours:

The BUBBLE-SORT example is from Balaban et al. [2]. Our successful verification of this example highlights the advantage of our inference-rule-based approach against their state-of-the-art small-model-theorem-based approach, which spaced out on this problem [1].

The recent experimental results of Manevich et al. [20] are comparable to ours, in spite of the fact they were executed on a slower machine. For most of their examples, however, they only verify the simple property of no null dereferences (and cyclicity for two examples). We are verifying more complicated properties, for instance NL.

For the examples in common with Lahiri and Qadeer [16], we are mostly significantly faster at verifying the same properties, with speed-ups of roughly 1 to 3 orders of magnitude. It should be noted, however, that we used a slightly faster machine, and also that our data fields are merely booleans, while theirs are the full integers. Furthermore, for the REMOVE-CYCLIC-DOUBLY example we are only two times faster. We suspect the the reason behind this is the usage of skolemization which sometimes requires a large number of predicates to define a cyclic doubly-linked list. A limited support for universally quantified variables we are planning to add would solve this problem.

In addition to these experiments, we ran our decision procedure on a couple of queries generated by field constraint analysis tool Bohne [26] for MONA. Initial results show that we are faster than MONA, but the queries we have are too simple to make a more serious comparison.

6 Future Work and Conclusions

Our experiments demonstrate the effectiveness of our work for verification of heap-manipulating programs. There are many directions for future research, some of which are outlined here.

We have found that even minimal support for universally quantified variables (as in the logic of Balaban et al. [2]) would allow expression of many common heap structure attributes. For example, the current logic cannot assert that two terms x and y point to disjoint linked lists; a single universally quantified variable would allow for this property (see Nelson [22, page 22]). We also found that capturing disjointedness is necessary for verifying that LIST-REVERSE always produces an acyclic list; hence we were unable to verify this property. We believe that our decision procedure can be enhanced to handle this case.

A final expressiveness deficiency, that we see no immediate solution to, is the expression of more involved heap structure properties, in particular trees. Though our logic cannot capture "x points to a tree", we believe that it is possible that an extension could be used to verify simple properties of programs that manipulate trees, for example that there are no memory leaks.

We also plan on investigating how existing techniques for predicate discovery and more advanced predicate abstraction algorithms mesh with our decision procedure. Furthermore, we have initial results showing the possibility of incorporating our decision procedure into a Nelson-Oppen style theorem prover. We believe that by doing that, it would be possible to improve the precision of heap abstraction used by the existing software verification tools that employ theorem provers. We would also like to look into possible ways of extending our decision procedure to generate proofs and interpolants.[8]

[8] Thanks to Ken McMillan for the proof-generation and interpolant suggestion.

References

1. I. Balaban, 2005. personal correspondence.
2. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.
3. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
4. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming (ESOP)*, 1999.
5. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs, 2005. UBC Dept. Comp. Sci. Tech Report TR-2005-19, http://www.cs.ubc.ca/cgi-bin/tr/2005/TR-2005-19.
6. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 207–221, 2006.
7. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Conf. on Computer Aided Verification (CAV)*, 1999.
8. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. To appear in TACAS 2006.
9. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Principles of Programming Languages (POPL)*, pages 191–202, 2002.
10. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
11. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conf. on Computer Aided Verification (CAV)*, 1997.
12. D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Principles of Programming Languages (POPL)*, pages 58–70, 2002.
14. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *Workshop on Computer Science Logic (CSL)*, pages 160–174, 2004.
15. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Conf. on Implementation and Application of Automata (CIAA)*, 2000.
16. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Principles of Programming Languages (POPL)*, pages 115–126, 2006.
17. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conf. on Automated Deduction (CADE)*, 2005.
18. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS'00)*, pages 280–301, 2000.
19. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.
20. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 181–198, 2005.
21. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Conf. on Computer Aided Verification (CAV)*, pages 476–490, 2005.
22. G. Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1979.

23. G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages (POPL)*, pages 38–47, 1983.
24. S. Ranise and C. G. Zarba. A decidable logic for pointer programs manipulating linked lists. Unpublished manuscript, 2005.
25. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming (ESOP)*, pages 380–398, 2003.
26. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.

A Proofs

In this appendix, we prove all theorems except Theorem 5, which is proven in Appendix B.

A.1 Proof of Theorem 1

Our proof of Theorem 1 uses a reduction from the following decision problem, which is known to be NP-complete [10].

Definition 1 (BETWEENNESS). *The decision problem* BETWEENNESS *asks, given a finite set A and a set C of triples (a,b,c) of distinct elements from A, if there exists a one-to-one function $g : A \to \{1,2,...,|A|\}$ such that for each $(a,b,c) \in C$ we have either $g(a) < g(b) < g(c)$ or $g(c) < g(b) < g(a)$.*

Theorem 1. *Given a set of literals Φ, the problem of deciding if Φ is satisfiable is NP-hard. This holds even when Φ contains no updates, no* `btwn` *predicates, no data fields, and only mentions a single link function f.*

Proof. We reduce BETWEENNESS to satisfiability of a set of literals Φ adhering to the second sentence of the theorem statement. Let (A,C) be an arbitrary instance of BETWEENNESS.

Given a set of terms T, let $\mathit{distinct}(T)$ be the set of $\binom{|T|}{2}$ literals that enforces that the terms of T are pair-wise unequal, for example

$$\mathit{distinct}(\{x, f(x), y\}) \;=\; \{\neg x = f(x), \neg x = y, \neg f(x) = y\}$$

Similarly, given two sets of terms T_1 and T_2, let $\mathit{distinct}(T_1,T_2)$ be the set of $|T_1||T_2|$ literals that enforces that no term in T_1 is equal to any term in T_2. Our set of literals Φ will involve a variable h, a variable e for each $e \in A$, and for each triple $t \in C$, two variables y_t and z_t. Now, let Φ be the union of the following five sets of literals, where $n = |A|$:

$$\left\{\neg f^{n-2}(h) = f^{n-1}(h), f^{n-1}(h) = f^{n}(h)\right\} \tag{3}$$

$$\mathit{distinct}(A) \tag{4}$$

$$\{f^*(h,e) \mid e \in A\} \tag{5}$$

$$\bigcup_{(a,b,c)\in C} \left\{f^*(y_{(a,b,c)},b), f^*(b,z_{(a,b,c)}), f^*(h,y_{(a,b,c)}), f^*(h,z_{(a,b,c)})\right\} \tag{6}$$

$$\bigcup_{(a,b,c)\in C} \mathit{distinct}\left(\left\{y_{(a,b,c)}, z_{(a,b,c)}\right\}, A \setminus \{a,c\}\right) \tag{7}$$

(3) says that h is the head of a non-circular list of exactly n nodes $h, f(h), \ldots, f^{n-1}(h)$. (4) and (5) say that the elements of A are associated in a one-to-one correspondence with the nodes in this list. To see this, note that (5) implies that for each $e \in A$ we have $e = f^i(h)$ for some $0 \le i < n$ while (4) enforces that there is no $e' \in A \setminus \{e\}$ such that $e' = f^i(h)$. (6) says that for each triple $(a,b,c) \in C$, the variables $y_{(a,b,c)}$ and $z_{(a,b,c)}$ are

both in the list (and are hence both equal to elements of A), and further $y_{(a,b,c)}$ comes (not necessarily strictly) before b and $z_{(a,b,c)}$ comes (not necessarily strictly) after b in the list. Now taking (7) into account allows us to conclude that $y_{(a,b,c)}$ and $z_{(a,b,c)}$ can each only be equal to a or c (hence the previous before and after relations become strict, since a and c are distinct from b).

With the preceding intuition, it is easy to see that if (A,C) is a positive instance of BETWEENNESS then one can construct a heap structure that satisfies Φ using the function g of Def. 1 to define an interpretation of the variables A as nodes in the linked list. The interpretation of the variables $y_{(a,b,c)}$ and $z_{(a,b,c)}$ is respectively a and c if $g(a) < g(c)$, or respectively c and a otherwise. Conversely, from any satisfying heap structure, one can extract a one-to-one function g satisfying Def. 1 by simply using the total order defined by the linked list. Finally, we note that $|\Phi| = \mathcal{O}(|A|^2 + |A|\,|C|)$, and each literal of Φ has length that is at most linear in $|A|$. It follows that Φ can be constructed in time polynomial in the size of (A,C). The NP-hardness of satisfiability in our logic therefore follows from the NP-completeness of BETWEENNESS.

A.2 Proof of Theorem 2

Our proof of Theorem 2 uses the following notation and lemmas. Let us fix a heap structure (N,Θ), and, in a slight abuse, we identify a term x with its interpretation $\Theta^e(x)$ (which is a node in N) and we also identify the symbol f with $\Theta(f)$. Let $x,y \in N$, then $\delta(x,y)$ denotes the minimum n such that $y = f^n(x)$ if such an n exists, otherwise $\delta(x,y) = \infty$. (Hence $\delta(x,y)$ is simply the graph-theoretic directed distance from x to y in the graph of f).

Lemma 1. *For any nodes x, y, and z, if $\delta(x,y)$ is finite, $\delta(x,z)$ is finite, and $\delta(x,y) \leq \delta(x,z)$ then $\mathsf{btwn}_f(x,y,z)$.*

Proof. Follows trivially from the semantics of the btwn_f operator.

Lemma 2. *If $f(y) = x$, then either $\delta(x,y) = \infty$, or $\delta(x,y)$ is finite and $\delta(x,y) \geq \delta(x,z)$ for all z such that that $\delta(x,z)$ is finite.*

Proof. If $\delta(x,y)$ is finite, then x and y must lie adjacent on a cycle in f; the result follows.

Lemma 3. *If $\delta(x,y) \leq \delta(x,z) < \infty$ and $x \neq y$, then $\delta(y,z) \leq \delta(y,x)$.*

Proof. If $\delta(y,x) = \infty$, then the lemma trivially holds. Otherwise we have $0 < \delta(x,y), \delta(y,x) < \infty$ and thus x, y, and z must occur on a cycle. The lemma follows.

Lemma 4. *If $\mathsf{btwn}_f(x,y,z)$, then $\delta(x,y) + \delta(y,z) = \delta(x,z)$.*

Proof. Since $\mathsf{btwn}_f(x,y,z)$, there exist (graph theoretic) paths in the graph of f from x to y and y to z, and hence from x to z. Since the out-degree of all nodes in the graph of f is 1, it follows that all these paths are unique, and that the path from x to z is the concatenation of the path from x to y and the path from y to z. The lemma follows.

Theorem 2. *The inference rules of Fig. 3 are sound.*

Proof. We argue in turn that each inference rule is sound; those IRs of Fig. 3 not mentioned here were proven sound in [5]. Most cases involve an implicit appeal to Lemma 1.

- NOTEQNODES. The antecedents imply that $d(x) \neq d(y)$, which implies $x \neq y$.
- BTWREFLEX. Trivial.
- BTW1. From the antecedents it follows that $\delta(x,y)$ and $\delta(x,z)$ are both finite. In particular, z is reachable from x, and using Lemma 2 we have $\delta(x,y) \leq \delta(x,z)$ and thus $\mathsf{btwn}_f(x,y,z)$.
- BTW2. Trivial.
- BTW3. Let us suppose the antecedents hold, and $x \neq y$. It follows that $\delta(x,y)$ and $\delta(x,z)$ are both positive and finite, and that $\delta(w,y) = \delta(x,y) - 1$ and $\delta(w,z) = \delta(x,z) - 1$. Since $\delta(x,y) \leq \delta(x,z)$, this implies $\delta(w,y) \leq \delta(w,z)$, and thus $\mathsf{btwn}_f(w,y,z)$.
- BTW4. From the antecedents it follows that $\delta(x,y)$ and $\delta(x,z)$ are both finite. From the first antecedent we have $\delta(x,y) \leq \delta(x,z)$; from the second antecedent we have $\delta(x,y) \geq \delta(x,z)$. Thus $\delta(x,y) = \delta(x,z)$, implying $y = z$.
- BTW5. From the antecedents it follows that $\delta(x,y)$ and $\delta(x,z)$ are both finite. If $\delta(x,y) \leq \delta(x,z)$, then we have $\mathsf{btwn}_f(x,y,z)$. If $\delta(x,y) > \delta(x,z)$, then we have $\mathsf{btwn}_f(x,z,y)$.
- BTW6. From the antecedents it follows that $\delta(a,b)$ is finite for all $a,b \in \{x,y,z\}$. If $x = y$, $x = z$, or $y = z$, then one of the right three consequents holds. Hence we assume that x, y, and z are distinct nodes. Suppose that we have $\delta(x,y) \leq \delta(x,z)$. Then, by Lemma 3, we have $\delta(y,z) \leq \delta(y,x)$, and, again by Lemma 3, we have $\delta(z,x) \leq \delta(z,y)$. We conclude that $\mathsf{btwn}_f(x,y,z)$, $\mathsf{btwn}_f(y,z,x)$, and $\mathsf{btwn}_f(z,x,y)$ all hold. The proof that the second-to-leftmost branch of BTW6 follows from $\delta(x,y) > \delta(x,z)$ is similar.
- BTW7. From the antecedent it follows that $\delta(x,y)$ is finite. Since $\delta(x,x) = 0 \leq \delta(x,y)$, we have $\mathsf{btwn}_f(x,x,y)$; since $\delta(x,y) \leq \delta(x,y)$, we also have $\mathsf{btwn}_f(x,y,y)$.
- BTW8. From the antecedents it follows that $\delta(x,y) \leq \delta(x,z) \leq 1$. If $\delta(x,y) = 0$ then $y = x$, while if $\delta(x,y) = 1$ then $y = z$.
- BTW9. Suppose the antecedents hold, and $y \neq w$ From $\mathsf{btwn}_f(x,y,w)$ it follows that $\delta(x,w)$ is finite and positive, and $\delta(x,y) < \delta(x,w)$. Since $f(z) = w$, we have $\delta(x,z) = \delta(x,w) - 1$. Thus $\delta(x,y) \leq \delta(x,z)$, and hence $\mathsf{btwn}_f(x,y,z)$.
- BTW10. If $y = z$, then the right consequent holds, hence we assume $y \neq z$. From the antecedents and $y \neq z$ it follows that z and y are on an f-cycle C. If w is on C, then the left consequent holds and we are done. Otherwise, let $k \geq 0$ be minimal such that $f^k(x)$ is in C. Since $\delta(x,w) < \infty$, we must have $\delta(x,w) < k$, else w would be in C. It follows that $\delta(w,y) = \delta(x,y) - \delta(x,w)$ and that $\delta(w,z) = \delta(x,z) - \delta(x,w)$, thus $\delta(w,y) < \delta(w,z)$, since $\delta(x,y) < \delta(x,z)$, which contradicts the antecedent $\mathsf{btwn}_f(w,z,y)$.
- BTW11. From the antecedents it follows that $\delta(w,x) \leq \delta(w,y) \leq \delta(w,z) < \infty$. Thus, $\delta(w,x) \leq \delta(w,z) < \infty$, implying $\mathsf{btwn}_f(w,x,z)$.
- BTW12. From the antecedent $\mathsf{btwn}_f(u,x,y)$, it follows that $\delta(u,x) \leq \delta(u,y) < \infty$. When we add $\delta(v,u)$ to this inequality, we get $\delta(v,u) + \delta(u,x) \leq \delta(v,u) + \delta(u,y) < \infty$. From the antecedents $\mathsf{btwn}_f(v,u,x)$ and $\mathsf{btwn}_f(v,u,y)$, it follows that $\delta(v,u) + \delta(u,x) = \delta(v,x)$ and $\delta(v,u) + \delta(u,y) = \delta(v,y)$, respectively, by Lemma 4. Therefore, we conclude that $\delta(v,x) \leq \delta(v,y) < \infty$, and thus $\mathsf{btwn}_f(v,x,y)$ is implied.

A.3 Proof of Theorem 3

Theorem 3. *There exists a polynomial-time algorithm that transforms any set Φ into a normal set Φ' such that Φ' is satisfiable if and only if Φ is satisfiable.*

Proof. Our transformation algorithm has two variables Φ_0 and Φ_1 of type "set of literal", such that initially we have $\Phi_0 = \Phi$ and $\Phi_1 = \emptyset$. Now, while there exists mention of a term of the form $f(v)$ (where $v \in V$ and $f \in F$) in Φ_0, create a fresh variable v_{fresh}, replace all occurrences of $f(v)$ in Φ_0 with v_{fresh}, and add the literal $f(v) = v_{fresh}$ to Φ_1. Once we have no terms of the form $f(v)$ in Φ_0, let $\Phi_2 = \Phi_0 \cup \Phi_1$. Now, for each equality of the form $v = u$ (where v and u are variables) in Φ_2, replace all occurrences of u in Φ_2 with v, and remove the equality; Let Φ' be the set obtained by exhaustively applying this reduction. Clearly Φ' satisfies Def. 1, is satisfiable if and only if Φ is, and is constructed in polynomial time.

A.4 Proof of Theorem 4

In order to prove Theorem 4, we will demonstrate how, given a consistent, closed, and normal set of literals Φ, one can construct a heap structure H^{Φ} such that $H^{\Phi} \models \Phi$. For the remainder of this section, let us fix a consistent, closed, and normal set Φ, let $V = \mathsf{Vars}(\Phi)$, and let F, D, and B be respectively the set of link fields, data fields, and data variables mentioned in Φ. The set of nodes of H^{Φ} will be V. For each $f \in F$, we define the relation $f^{*\Phi} \subseteq V \times V$ such that $(u,v) \in f^{*\Phi}$ iff $f^*(u,v) \in \Phi$. For each $v \in V$ and $f \in F$, let us define $\leq_f^v \subseteq V \times V$ as follows: $u \leq_f^v w$ iff $\mathsf{btwn}_f(v,u,w) \in \Phi$. Let $R_f(v) = \{u \mid f^*(v,u) \in \Phi\}$.

Lemma 5. *For all $v \in V$ and $f \in F$, $\leq_f^v$ is a total order on $R_f(v)$.*

Proof. $\leq_f^v$ is clearly reflexive , since BTWREFLEX is not enabled. Now suppose $x \leq_f^v y$ and $y \leq_f^v z$. Then $\mathsf{btwn}_f(v,x,y)$ and $\mathsf{btwn}_f(v,y,z)$ are both in Φ, and thus so too is $\mathsf{btwn}_f(v,x,z)$, since BTW11 is disabled. Therefore $x \leq_f^v z$, and $\leq_f^v$ is transitive. Now suppose $x \leq_f^v y$ and $y \leq_f^v x$. Then $\mathsf{btwn}_f(v,x,y)$ and $\mathsf{btwn}_f(v,y,x)$ are both in Φ, which, since BTW4 is disabled, implies that $x = y \in \Phi$, which implies that $x = y$ (i.e. x and y are the same symbol in V) since Φ is normal. Thus $\leq_f^v$ is antisymmetric. Finally, suppose $x,y \in R_f(v)$. Then $f^*(v,x)$ and $f^*(v,y)$ are in Φ, and thus, since BTW5 is disabled, either $x \leq_f^v y$ or $y \leq_f^v x$.

Lemma 6. *For all $f \in F$ and $v \in V$, the minimal element of $\leq_f^v$ is v.*

Proof. Suppose, on the contrary, that there exists some symbol w different from v such that $w \leq_f^v v$ and hence $\mathsf{btwn}_f(v,w,v) \in \Phi$. Since BTW7 is disabled, we also have $\mathsf{btwn}_f(v,w,w) \in \Phi$. But since BTW4 is disabled, we also have $v = w \in \Phi$, which contradicts Φ being normal.

Now that we have Lemma 5, the following function is well-defined.

Definition 2 **(η_f).** *For each $f \in F$, define the function $\eta_f : V \to V$ such that $\eta_f(v) = v$ if $R_f(v) = \{v\}$, otherwise $\eta_f(v)$ is the $\leq_f^v$-minimal element of $R_f(v) \setminus \{v\}$.*

Definition 3 **(f-basin).** *For $f \in F$ and $v \in V$, if $R_f(v) = R_f(\eta_f(v))$ we say that v is an* f-basin node *and we call $R_f(v)$ a* f-basin.

Lemma 7. *For all $f \in F$ and all $v,u \in V$, if $f(v)=u \in \Phi$, then $\eta_f(v) = u$.*

Proof. Suppose $f(v)=u \in \Phi$. Since TRANS1 is disabled, we have $f^*(v,u) \in \Phi$, and hence $u \in R_f(v)$. Now if v and u are the same symbol, say v, then there cannot exist some other symbol $w \in V$ such that $f^*(v,w) \in \Phi$, since CYCLE$_1$ is disabled and Φ is normal. Thus $R_f(v) = \{v\}$, and from Def. 2, $\eta_f(v) = v$. On the other hand, if v and u are distinct symbols, we claim that u is the $\leq_f^v$-minimal element of $R_f(v) \setminus \{v\}$. Suppose, on the contrary, there exists $y \in R_f(v) \setminus v,u$ such that $y \leq_f^v u$. Than $\mathsf{btwn}_f(v,y,u) \in \Phi$, and we already have $f(v)=u \in \Phi$. However, this contradicts the facts that BTW8 is disabled and Φ is normal. We conclude that $\eta_f(v) = u$.

Lemma 8. *For all $f \in F$, suppose x, y, and z distinct elements of V in the same SCC of $f^{*\Phi}$, and $y \leq_f^x z$. Then $z \leq_f^y x$ and $x \leq_f^z y$.*

Proof. Since Φ is normal and x, y, and z are distinct symbols, we have that none of $x=y$, $x=z$, or $y=z$ are in Φ. Now from our supposition, $\mathsf{btwn}_f(x,y,z), f^*(x,y), f^*(y,z), f^*(z,x) \in \Phi$. Since BTW6 is disabled, either we also have $\mathsf{btwn}_f(y,z,x), \mathsf{btwn}_f(z,x,y) \in \Phi$, and our lemma holds, or we have $\mathsf{btwn}_f(x,z,y), \mathsf{btwn}_f(z,y,x), \mathsf{btwn}_f(y,x,z) \in \Phi$. The latter case yields a contradiction, however, since having both $\mathsf{btwn}_f(x,y,z), \mathsf{btwn}_f(x,z,y) \in \Phi$ and BTW4 disabled implies that $z=y \in \Phi$, which contradicts the first sentence of this proof.

Lemma 9. *For all $f \in F$, $f^{*\Phi}$ is reflexive and transitive.*

Proof. $f^{*\Phi}$ is clearly reflexive and transitive since IDENT and TRANS2 are disabled.

Lemma 10. *For all $f \in F$ and $v \in V$ we have that $R_f(\eta_f(v)) \subseteq R_f(v)$.*

Proof. Let $\eta_f(v) = u$. From Def. 2, there exists $w \in V$ such that $\mathsf{btwn}_f(v,u,w) \in \Phi$, and since BTW2 is disabled, we have $(v,u) \in f^{*\Phi}$. Therefore, by Lemma 9 we are done.

Lemma 11. *For all $f \in F$ and $v \in V$ and let $u = \eta_f(v)$. Then either*

- *v is an f-basin node, and $\leq_f^u$ is identical to $\leq_f^v$ except v is made the maximal, or*
- *$R_f(u) = R_f(v) \setminus \{v\}$ and $\leq_f^u$ is $\leq_f^v$ restricted to $R_f(u)$.*

Proof. If u and v are the same symbol v, then we must have $R_f(v) = \{v\}$ from Def. 2 and $\leq_f^v = \{(v,v)\}$; thus the first bullet holds trivially. Thus we assume for the remainder of the proof that u and v are distinct symbols. We case-split on whether or not $(u,v) \in f^{*\Phi}$.

Case: $(u,v) \in f^{*\Phi}$**.** Then $R_f(u) \supseteq R_f(v)$ from Lemma 9, and thus, by Lemma 10, $R_f(u) = R_f(v)$. Now, let x and y be elements of $R_f(u) \setminus \{v\}$. We wish to show that $x \leq_f^u y$ implies $x \leq_f^v y$. Let us assume that $x \leq_f^u y$. If x and y are the same symbol, x, then the facts that $(u,x),(v,x) \in f^{*\Phi}$ and BTW7 is disabled imply that $x \leq_f^u x$ and $x \leq_f^v x$. Hence, we assume that x and y are distinct symbols. From Lemma 5 exactly one of $x \leq_f^v y$ or $y \leq_f^v x$ holds. In the former case, we are done. In the latter case, we have $x \leq_f^y v$ by Lemma 8. Also, from the definition of η_f, we have that $u \leq_f^v y$, and, again by Lemma 8, we have $v \leq_f^y u$. Since $x \leq_f^y v$, $v \leq_f^y u$, and BTW11 is disabled, we have $x \leq_f^y u$. Finally, by another application of Lemma 8, we conclude $y \leq_f^u x$, which contradicts our assumption $x \leq_f^u y$ and the facts that x and y are distinct and $\leq_f^u$ is a total order.

It remains to show that $x \leq_f^u v$ for all $x \in R_f(u)$. If x is v we are done; else if x is u we are done by Lemma 6. Hence, we assume that x is distinct from v and u. We have that $u \leq_f^v x$ and thus, by Lemma 8, $x \leq_f^u v$. Hence v is the maximal element of $\leq_f^u$.

Case: $(u,v) \notin f^{*\Phi}$**.** Then clearly $v \notin R_f(u)$, and from Lemma 10, $R_f(u) \subseteq R_f(v) \setminus \{v\}$. Conversely, choose $w \in R_f(v) \setminus \{v\}$. Then, from Def. 2, the fact that $\eta_f(v) = u$, and Lemma 5 we have that $u \leq_f^v w$ and hence $\mathsf{btwn}_f(v,u,w) \in \Phi$. Now, since BTW2 is disabled, this implies that $f^*(u,w) \in \Phi$ and thus $w \in R_f(u)$. Therefore, $R_f(u) = R_f(v) \setminus \{v\}$.

Now we argue that $\leq_f^u$ is $\leq_f^v$ restricted to $R_f(u)$. Let $x,y \in R_f(u)$ be such that $x \leq_f^u y$. As in the previous case, we may assume that x and y are distinct symbols. Thus $\mathsf{btwn}_f(u,x,y) \in \Phi$. Since u is the $\leq_f^v$-minimal element of $R_f(v) \setminus \{v\}$, we also have $\mathsf{btwn}_f(v,u,x) \in \Phi$ and $\mathsf{btwn}_f(v,u,y) \in \Phi$. It follows that $\mathsf{btwn}_f(v,x,y) \in \Phi$, since BTW12 is disabled, and thus $x \leq_f^v y$.

Definition 4 **(**$seq_f(v)$**).** *Given $v \in V$ and $f \in F$, let $seq_f(v)$ be the infinite sequence over $R_f(v)$ defined inductively as follows.*

- *If v is an f-basin node, then $seq_f(v) = s^\omega$, where s is the sequence of length $|R_f(v)|$ wherein all elements of $R_f(v)$ are listed according to the total order $\leq_f^v$.*
- *Otherwise, $seq_f(v) = v \cdot seq_f(\eta_f(v))$*

Lemma 12. *For all $f \in F$ and $v \in V$ we have that $seq_f(v) = v \cdot seq_f(\eta_f(v))$*

Proof. If v is an f-basin node, the result follows from Lemmas 6 and 11 and Def. 4. Otherwise, the Lemma follows trivially from Def. 4.

Lemma 13. *For all $f \in F$ and $v \in V$ and $n \geq 0$, the nth element of $seq_f(v)$ is $\eta_f^n(v)$.*

Proof. By induction using Lemma 12.

Lemma 14. *For all $f \in F$ and $v \in V$, the symbols appearing in $seq_f(v)$ are precisely $R_f(v)$.*

Proof. Let $i \geq 0$ be the position of the first f-basin node in $seq_f(v)$. Note that such a node must exist, else, by Lemma 11, we would have that $R_f(v), R_f(\eta_f(v)), R_f(\eta_f^2(v)), \ldots$ would be an infinite sequence of finite sets, each being a proper subset of the previous. We complete the proof by induction on i. If $i = 0$ then v is an f-basin node, the lemma follows trivially by Def. 4. Now assume $i > 0$ and the lemma holds for for all $w \in V$ with first f-basin node of $seq_f(w)$ begin at position $i-1$. From Lemma 12, $seq_f(v) = v \cdot seq_f(\eta_f(v))$, thus the first f-basin node of $seq_f(\eta_f(v))$ is at position $i-1$. Therefore the symbols appearing in $seq_f(\eta_f(v))$ are precisely $R_f(\eta_f(v))$. Now from Lemma 11 we have that $R_f(\eta_f(v)) = R_f(v) \setminus \{v\}$; therefore, since the symbols appearing in $seq_f(v)$ are v along with those appearing in $seq_f(\eta_f(v))$, we are done.

Lemma 15. *For all $f \in F$, $f^{*\Phi}$ is the reflexive transitive closure of η_f.*

Proof. From Lemma 9 we have that $f^{*\Phi}$ is reflexive and transitive. Thus, letting η_f^* denote the reflexive transitive closure of η_f, we must show that $\eta_f^* = f^{*\Phi}$. Suppose $(v,u) \in f^{*\Phi}$; then $u \in R_f(v)$, and hence by Lemma 14, u appears in $seq_f(v)$. Let n be the position of an occurrence of u in $seq_f(v)$. By Lemma 13 we have that $u = \eta_f^n(v)$, thus $(v,u) \in \eta_f^*$. Conversely, suppose that there exists $n \geq 0$ such that $\eta_f^n(v) = u$. Using a simple induction along with Lemma 10 and the fact that $w \in R_f(w)$ for all $w \in V$, one can show that $(v,u) \in f^{*\Phi}$.

Lemma 16. *For all $f \in F$ and $v \in V$, the prefix of $seq_f(v)$ of length $\left|R_f(v)\right|$ is the elements of $R_f(v)$ ordered according to $\leq_f^v$.*

Proof. As in the proof of Lemma 14, let $i \geq 0$ be the position of the first f-basin node in $seq_f(v)$. We proceed by induction on i. If $i = 0$, then v is an f-basin node and the result follows from Def. 4. Now, assume the statement is true for sequences with first f-basin node at position $i-1$ for some $i > 0$. Let $u = \eta_f(v)$. From Lemma 12 we have that $seq_f(v) = v \cdot seq_f(u)$, and thus, from our inductive assumption, the prefix of $seq_f(u)$ of length of length $\left|R_f(u)\right|$ is the elements of $R_f(u)$ ordered according to $\leq_f^u$. From Lemmas 6 and 11 and the fact that v is not an f-basin node, it follows that $R_f(v) = R_f(u) \cup \{v\}$. Therefore, the first $\left|R_f(v)\right|$ elements of $R_f(v)$ are $R_f(u) \cup \{v\} = R_f(v)$. Furthermore, by Lemmas 6 and 11, we have the ordering requirement on this prefix as well.

Lemma 17. *For all $f \in F$ and $u,v,w \in V$, $\mathsf{btwn}_f(u,v,w) \in \Phi$ iff the minimal n and m where $v = \eta_f^n(u)$ and $w = \eta_f^m(u)$ are such that $n \leq m$.*

Proof. ($\Rightarrow$) Since BTW2 is disabled, we have $v,w \in R_f(u)$ and thus, by Lemma 16, v and w appear in the first $\left|R_f(u)\right|$ elements of $seq_f(u)$. Also, since $\mathsf{btwn}_f(u,v,w) \in \Phi$, we have that $v \leq_f^u w$. Finally, using Lemmas 13 and 16, we have that the minimal n and m where $v = \eta_f^n(u)$ and $w = \eta_f^m(u)$ exist, and are such that $n \leq m$.

($\Leftarrow$) Suppose the minimal n and m where $v = \eta_f^n(u)$ and $w = \eta_f^m(u)$ exist and are such that $n \leq m$. Thus, by Lemmas 13 and 14, $v,w \in R_f(u)$. Also, by Lemma 16, we must have $v \leq_f^u w$, therefore, by the definition of $\leq_f^u$ we have $\mathsf{btwn}_f(u,v,w) \in \Phi$.

Definition 5. *Let $H^\Phi = (V, \Theta^\Phi)$ be defined such that*

- *Θ^Φ is the identity on V*
- *For each $f \in F$, let f be interpreted by Θ^Φ as η_f.*
- *For each $d \in D$ and $v \in V$, let*

$$\Theta^\Phi(d)(v) \;=\; \begin{cases} \mathsf{true} & \textit{if } d(v) \in \Phi \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

- *For each $b \in B$, let*

$$\Theta^\Phi(b) \;=\; \begin{cases} \mathsf{true} & \textit{if } b \in \Phi \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

Theorem 4. *If Φ is consistent, closed, and normal, then Φ is satisfiable.*

Proof. We argue that $H^{\Phi} \models \Phi$. Let ϕ be a positive literal of Φ; we show that $H^{\Phi} \models \phi$, case-splitting on the type of ϕ. Here v, u, and w range over V; $f \in F$, $d \in D$, and $b \in B$. Also, only those literals of forms allowed in normal sets (see Def. 1) need be considered.

- $v = v$: clearly satisfied by any heap structure.
- $f(v) = u$: From Lemma 7 we have that $\eta_f(v) = u$.
- $d(v)$: satisfied by H^{Φ} from Def. 5.
- b : satisfied by H^{Φ} from Def. 5.
- $f^*(v,u)$: From Lemma 15 we have that (v,u) is in the reflexive and transition closure of η_f.
- $\mathsf{btwn}_f(v,u,w)$: satisfied by H^{Φ} by Lemma 17.

Let $\neg\phi$ be a negative literal of Φ; we show that $H^{\Phi} \not\models \phi$, case-splitting on the type of ϕ.

- $v = u$: v and u must be distinct symbols, else Φ would contain a contradiction (since IDENT is disabled). Clearly $H^{\Phi} \not\models v = u$, since $v \neq u$ and Θ^{Φ} is the identity on V.
- $d(v)$: not satisfied by H^{Φ} from Def. 5.
- b : not satisfied by H^{Φ} from Def. 5.
- $f^*(v,u)$: Since Φ is consistent, $f^*(v,u) \notin \Phi$ and thus, by Lemma 15, (v,u) is not in the reflexive and transitive closure of η_f.
- $\mathsf{btwn}_f(v,u,w)$: not satisfied by H^{Φ} by Lemma 17.

This completes the proof.

A.5 Proof of Theorem 6

Theorem 6. *The inference rules of Fig. 4 and Fig. 5 are sound.*

Proof. All IRs of Fig. 4 except for UPDBTWN were proven to be sound previously [5]. We use the same abuse of notation used in the proof of Theorem 2.

- UPDBTWN. We employ the notation $\delta(\cdot,\cdot)$ used in the proof of Theorem 2, with the additional notation $\delta'(x,y)$ to denote the distance in f' from x to y. From the antecedents, it follows that $\delta(x,y)$ and $\delta(x,z)$ are both finite, and also that $\delta(x,z) > 0$. Now suppose $\delta(x,\tau_1) \geq \delta(x,z)$. Then it follows that $\delta'(x,z) = \delta(x,z)$ and $\delta'(x,y) = \delta(x,y)$, and thus $\delta'(x,y) \leq \delta(x,z)$ and $\mathsf{btwn}_{f'}(x,y,z)$. On the other hand, if $\delta(x,\tau_1) < \delta(x,z)$, then it follows that $\tau_1 \neq z$ and $\mathsf{btwn}_f(x,\tau_1,z)$.
- EQDATA. This IR ensures that after the update the value of a data function $d'(\tau)$ is equal to b. This is clearly sound, because from the definition $d' = \mathsf{update}(d,\tau,b)$ of d' it can be seen that $d'(\tau)$ has to be equal to b.
- PRESERVEVALUE preserves values of data function of nodes which are not equal to τ and therefore cannot be influenced by the update. It is clearly sound.
- EQNODES1. Trivial, since under the constraint $d' = \mathsf{update}(d,\tau,b)$, we have that $d(x) \neq d'(x)$ implies that $x = \tau$.
- EQNODES2. Analogous to EQNODES1.

B Formalization of Decision Procedure

The procedure takes a normal set of literals Φ; this restriction does not lose us any generality thanks to Lemma 18. The procedure is given in Fig. 6. The notation $\Phi[v_i/v_j]$ on line 8 represents the set obtained by replacing all occurrences of v_j in literals of Φ with v_i. The following three lemmas and a theorem demonstrate the correctness of our algorithm.

Note that the proofs of these lemmas and the theorem assume that the literals are from the logic of Fig. 2; they do not apply to the extended logic of Sect. 3.1. The proofs of Lemmas 18, 19, and Theorem 5 can easily be generalized to deal with the extension. However, we have not yet been able to prove Lemma 20 for the extended decision procedure.

Lemma 18. *If invoked with a normal set Φ, Φ' will be normal if the recursive call of line 12 is reached.*

Proof. If Φ is normal, then any applicable IR r will have all its free terms x, y, z, x_1, etc. instantiated as variables of Φ. Inspection of all IRs reveals that if the free terms are instantiated with variables, then any consequent will either be an equality between variables, disequality between variables, a set of reachability literals involving two/three variables, or a set of between literals involving three/four variables. In the first case, Φ' is assigned by line 8, and clearly performing the substitution preserves normality. In all other cases, Φ' is assigned by line 10, and ϕ is a disequality literal, a set of reachability literals involving two/three variables, or a set of between literals involving three/four variables. Addition of such literals also preserves normality.

Lemma 19. *If* DECIDE*(Φ) returns* UNSAT *then Φ is unsatisfiable.*

Proof. (Sketch) If UNSAT is returned on line 3, then Φ contains a contradiction and is obviously unsatisfiable. If UNSAT is returned on line 16, addition of all consequents of an applicable IR yielded UNSAT from the recursive calls. The proof thus depends on the Theorem 2, which states that the IRs of Fig. 3 are sound.

Lemma 20. *If* DECIDE*(Φ) returns* SAT *then Φ is satisfiable.*

Proof. (Sketch) If SAT is returned, then by applying a sequence of IRs to Φ the algorithm reached a point in which SAT was returned by line 18. Let $\hat{\Phi}$ be the set of literals that caused line 18 to be reached. Then, $\hat{\Phi}$ is obtained from Φ by adding disequality, reachability, and between literals, and doing variable substitutions. Furthermore, $\hat{\Phi}$ is consistent, closed, and, by Lemma 18, normal. Thus, by Theorem 4, $\hat{\Phi}$ is satisfiable, which implies the satisfiability of Φ also.

Theorem 5. *The decision procedure always terminates.*

Proof. Follows from the fact that none of the IRs create new terms, and there are only a finite number of possibly literals that one could add given a fixed set of terms. Also, the variable substitutions can only reduce the number of terms.

```
function Decide(Φ)
    if Φ contains a contradiction then
        return UNSAT
    end if
    if there exists an IR r applicable to Φ then
        for each consequent φ of r do
            if φ is an equality literal of the form v_i = v_j then
                Φ' := Φ[v_i/v_j]
            else
                Φ' := Φ ∪ {φ}
            end if
            if Decide(Φ') = SAT then
                return SAT
            end if
        end for
        return UNSAT
    else
        return SAT
    end if
end function
```

Fig. 6. The decision procedure DECIDE, which requires Φ to be normal.

C Pseudocode of the Examples

```
procedure LIST-REVERSE(x)
    assume ¬x = nil ∧ f*(x, nil) ∧ f*(x, t) ∧ ¬t = nil ∧ y = nil
    while ¬x = nil do
        temp := f(x);
        f(x) := y;
        y := x;
        x := temp;
    end while
    assert f*(y, t)
end procedure
```

Fig. 7. LIST-REVERSE. Predicates used to verify the example: $x = \mathsf{nil}$, $f^*(x, \mathsf{nil})$, $f^*(x,t)$, $t = \mathsf{nil}$, $y = \mathsf{nil}$, $f^*(y,t)$, $f^*(temp,t)$, $f(x) = temp$.

```
procedure LIST-ADD(head, item)
    assume ¬f*(head, item) ∧ f*(head, nil) ∧ f*(head, t) ∧ f(item) = nil ∧ p = head
    if head = nil then
        head := p;
    else
        while ¬f(p) = nil do
            p := f(p);
        end while
        f(p) := item;
    end if
    assert f*(head, item) ∧ f*(head, nil) ∧ f*(head, t)
end procedure
```

Fig. 8. LIST-ADD. Predicates used to verify the example: $f^*(head, item)$, $f^*(head, \mathsf{nil})$, $f^*(head, t)$, $f(item) = \mathsf{nil}$, $p = head$, $head = \mathsf{nil}$, $f(p) = \mathsf{nil}$, $f^*(head, p)$.

procedure ND-INSERT(*head*, *item*)
  assume $\neg f^*(head, item) \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f^*(head, t) \wedge \neg t = \mathsf{nil} \wedge f(item) = \mathsf{nil} \wedge p = head$
  while true **do**
    if $ND \vee f(p) = \mathsf{nil}$ **then**
      $f(item) := f(p);$
      $f(p) := item;$
      break;
    else
      $p := f(p);$
    end if
  end while
  assert $f^*(head, item) \wedge f^*(head, \mathsf{nil}) \wedge f^*(head, t)$
end procedure

Fig. 9. ND-INSERT. Predicates used to verify the example: $f^*(head, item)$, $f^*(head, \mathsf{nil})$, $head = \mathsf{nil}$, $f^*(head, t)$, $t = \mathsf{nil}$, $f(item) = \mathsf{nil}$, $p = head$, $f(p) = \mathsf{nil}$, $f^*(head, p)$, $f^*(item, \mathsf{nil})$, $f^*(item, p)$, $f^*(item, t)$, $f^*(f(p), t)$.

procedure ND-REMOVE(*head*)
  assume $\neg head = \mathsf{nil} \wedge f^*(head, \mathsf{nil}) \wedge f^*(head, t) \wedge \neg t = \mathsf{nil} \wedge p = head \wedge r = f(head)$
  while true **do**
    if $ND \vee f(r) = \mathsf{nil}$ **then**
      $f(p) := f(r);$
      break;
    else
      $p := r;$
      $r := f(r);$
    end if
  end while
  assert $f^*(head, \mathsf{nil}) \wedge (f^*(head, t) \oplus r = t)$
end procedure

Fig. 10. ND-REMOVE. Predicates used to verify the example: $head = \mathsf{nil}$, $f^*(head, \mathsf{nil})$, $f^*(head, t)$, $t = \mathsf{nil}$, $p = head$, $r = f(head)$, $r = t$, $f(r) = \mathsf{nil}$, $f^*(head, p)$, $f^*(p, r)$, $f^*(r, t)$, $f^*(f(p), t)$.

```
procedure Zip(x,y)
    assume f*(x,nil) ∧ f*(y,nil) ∧ (f*(x,t) ∨ f*(y,t)) ∧ z=nil ∧ p=nil ∧ temp=nil
    if x=nil then
        temp := x;
        x := y;
        y := temp;
    end if
    while ¬x=nil do
        if z=nil then
            z := x;
            p := x;
        else
            f(p) := x;
            p := x;
        end if
        x := f(x);
        f(p) := nil;
        if ¬y=nil then
            temp := x;
            x := y;
            y := temp;
        end if
    end while
    assert f*(z,nil) ∧ f*(z,t)
end procedure
```

Fig. 11. Zip. Predicates used to verify the example: $f^*(x,\mathsf{nil})$, $f^*(y,\mathsf{nil})$, $f^*(x,t)$, $f^*(y,t)$, $z{=}\mathsf{nil}$, $p{=}\mathsf{nil}$, $temp{=}\mathsf{nil}$, $f^*(z,\mathsf{nil})$, $f^*(z,t)$, $x{=}\mathsf{nil}$, $y{=}\mathsf{nil}$, $f^*(temp,t)$, $p{=}x$, $f^*(p,\mathsf{nil})$, $f^*(p,t)$, $f^*(z,x)$, $f^*(z,p)$, $p{=}t$, $f^*(y,p)$, $f^*(temp,p)$, $f^*(x,p)$, $f(p){=}x$.

```
procedure SORTED-ZIP(x,y)
    assume f*(x,nil) ∧ f*(y,nil) ∧ ¬t = nil ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil) ∧ (f*(x,t) ⊕
        f*(y,t)) ∧ merge = nil ∧ temp = nil
    while ¬x = nil ∧ ¬y = nil do
        if d(x) < d(y) then
            if ¬temp = nil then
                f(temp) := x;
            else
                merge := x;
            end if
            temp := x;
            x := f(x);
        else
            if ¬temp = nil then
                f(temp) := y;
            else
                merge := y;
            end if
            temp := y;
            y := f(y);
        end if
    end while
    if ¬x = nil then
        if merge = nil then
            merge := x;
        else
            f(temp) := x;
        end if
    end if
    if ¬y = nil then
        if merge = nil then
            merge := y;
        else
            f(temp) := y;
        end if
    end if
    assert f*(merge,t) ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil)
end procedure
```

Fig. 12. SORTED-ZIP. Predicates used to verify the example: $f^*(x,\mathsf{nil})$, $f^*(y,\mathsf{nil})$, $t=\mathsf{nil}$, $d(t)$, $d(f(t))$, $f(t)=\mathsf{nil}$, $f^*(x,t)$, $f^*(y,t)$, $merge=\mathsf{nil}$, $temp=\mathsf{nil}$, $f^*(merge,t)$, $x=\mathsf{nil}$, $y=\mathsf{nil}$, $d(x)$, $d(y)$, $f^*(merge,temp)$, $f(temp)=x$, $f(temp)=y$, $temp=x$, $temp=y$, $merge=x$, $merge=y$. Comparison between data values is defined as a formula over boolean data predicates. For instance, $d(x) \leq d(y)$ is defined as $\neg(d(x) \wedge \neg d(y))$.

```
procedure SORTED-INSERT(head, item)
    assume ¬f*(head, item) ∧ f*(head, nil) ∧ ¬head = nil ∧ (f*(head, t) ⊕ item = t) ∧ ¬t =
        nil ∧ f(item) = nil ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil) ∧ curr = head ∧ succ = f(head)
    while ¬succ = nil ∧ d(item) > d(succ) do
        curr := succ;
        succ := f(curr);
    end while
    if d(head) > d(item) then
        f(item) := head;
        head := item;
    else
        f(item) := succ;
        f(curr) := item;
    end if
    assert f*(head, nil) ∧ f*(head, t) ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil)
end procedure
```

Fig. 13. SORTED-INSERT. Predicates used to verify the example: $f^*(head, item)$, $f^*(head, \mathsf{nil})$, $head = \mathsf{nil}$, $f^*(head, t)$, $item = t$, $t = \mathsf{nil}$, $f(item) = \mathsf{nil}$, $d(t)$, $d(f(t))$, $f(t) = \mathsf{nil}$, $curr = head$, $succ = f(head)$, $succ = \mathsf{nil}$, $d(item)$, $d(head)$, $d(succ)$, $f^*(head, curr)$, $f(item) = succ$, $f(curr) = succ$, $f(item) = curr$.
Comparison between data values is defined as a formula over boolean data predicates. For instance, $d(x) \leq d(y)$ is defined as $\neg(d(x) \wedge \neg d(y))$.

procedure BUBBLE-SORT(x)
    assume $f^*(x,\mathsf{nil}) \wedge f^*(x,t) \wedge \neg t = \mathsf{nil} \wedge y = x \wedge yn = f(y) \wedge prev = \mathsf{nil} \wedge last = \mathsf{nil}$
    while $\neg last = f(x)$ **do**
        while $\neg yn = last$ **do**
            if $d(y) > d(yn)$ **then**
                $f(y) := f(yn)$;
                $f(yn) := y$;
                if $prev = \mathsf{nil}$ **then**
                    $x := yn$;
                else
                    $f(prev) := yn$;
                end if
                $prev := yn$;
                $yn := f(y)$;
            else
                $prev := y$;
                $y := yn$;
                $yn := f(yn)$;
            end if
        end while
        $prev := \mathsf{nil}$;
        $last := y$;
        $y := x$;
        $yn := f(x)$;
    end while
    assert $f^*(x,\mathsf{nil}) \wedge f^*(x,t) \wedge (d(t) \leq d(f(t)) \vee f(t) = \mathsf{nil})$
end procedure

Fig. 14. BUBBLE-SORT. Predicates used to verify the example: $f^*(x,\mathsf{nil})$, $f^*(x,t)$, $t = \mathsf{nil}$, $y = x$, $yn = f(y)$, $prev = \mathsf{nil}$, $last = \mathsf{nil}$, $d(t)$, $d(f(t))$, $f(t) = \mathsf{nil}$, $f(x) = last$, $yn = last$, $d(y)$, $d(yn)$, $f^*(yn,t)$, $f^*(last,t)$, $f^*(x,prev)$, $f(yn) = y$, $f(prev) = y$, $t = y$, $f(yn) = f(y)$, $f^*(x,yn)$, $d(last)$, $prev = y$. Comparison between data values is defined as a formula over boolean data predicates. For instance, $d(x) \leq d(y)$ is defined as $\neg(d(x) \wedge \neg d(y))$.

procedure REMOVE-ELEMENTS(x)
    assume $\neg x = \mathsf{nil} \wedge f^*(f(x),x) \wedge f^*(x,t) \wedge \mathsf{btwn}_f(curr,t,x) \wedge prev = x \wedge curr = f(x)$
    while $\neg curr = x$ **do**
        if $d(curr) = \mathsf{false}$ **then**
            $curr := f(curr);$
            $f(prev) := curr;$
        else
            $prev := curr;$
            $curr := f(curr);$
        end if
    end while
    if $d(x) = \mathsf{false}$ **then**
        if $\neg prev = x$ **then**
            $x := f(x);$
            $f(prev) := x;$
        else
            $x := \mathsf{nil};$
        end if
    end if
    assert $f^*(f(x),x) \wedge (\quad (f^*(x,t) \wedge d(t) = \mathsf{true})$
                        $\vee \; (\neg f^*(x,t) \wedge d(t) = \mathsf{false}))$
end procedure

Fig. 15. REMOVE-ELEMENTS. Predicates used to verify the example: $f^*(f(x),x)$, $curr = x$, $prev = x$, $d(curr)$, $d(x)$, $f^*(x,t)$, $d(t)$, $x = \mathsf{nil}$, $curr = f(x)$, $\mathsf{btwn}_f(curr,t,x)$, $f(prev) = curr$, $f(f(prev)) = curr$, $f^*(f(x),prev)$, $prev = curr$, $f^*(t,prev)$, $f(prev) = t$, $f(curr) = x$.

```
procedure REMOVE-SEGMENT(x)
    assume ¬x = nil ∧ f*(f(x), x) ∧ temp = nil ∧ y = x ∧ z = nil
    while ¬temp = x do
        if d(y) = false then
            temp := f(y);
            y := temp;
        else
            break;
        end if
    end while
    z := y;
    while ¬z = x do
        if d(z) = true then
            temp := f(z);
            z := temp;
        else
            break;
        end if
    end while
    if ¬y = z then
        f(y) := nil;
        f(y) := z;
    end if
    assert f*(f(x), x)
end procedure
```

Fig. 16. REMOVE-SEGMENT. Predicates used to verify the example: $f^*(f(x),x)$, $x = \mathsf{nil}$, $temp = \mathsf{nil}$, $y = x$, $z = \mathsf{nil}$, $x = temp$, $z = x$, $z = y$, $d(y)$, $d(z)$, $f^*(z,x)$, $f^*(z,y)$, $\mathsf{btwn}_f(y,z,x)$, $\mathsf{btwn}_f(y,temp,x)$, $f^*(y,x)$.

procedure SEARCH-AND-SET($x, data_1, data_2$)
  assume $\neg x = \mathsf{nil} \wedge f^*(f(x),x) \wedge f(x) = curr \wedge f^*(x,t) \wedge \mathsf{btwn}_f(curr,t,x) \wedge true$
  $d_1(x) := true$;
  $d_2(x) := true$;
  while $\neg curr = x$ **do**
    if $d_1(curr) = data_1 \wedge d_2(curr) = data_2$ **then**
      break;
    else
      $d_1(curr) := true$;
      $d_2(curr) := true$;
      $curr := f(curr)$;
    end if
  end while
  assert $f^*(f(x),x) \wedge f^*(x,t) \wedge \neg x = \mathsf{nil} \wedge true$
  assert $(x = curr \wedge d_1(t) \wedge d_2(t))$
    $\vee\ (\ \neg x = curr \wedge d_1(curr) = data_1 \wedge d_2(curr) = data_2$
      $\wedge\ (\ (\mathsf{btwn}_f(x,t,curr) \wedge \neg t = curr \wedge d_1(t) \wedge d_2(t))$
        $\vee (\mathsf{btwn}_f(x,t,curr) \wedge t = curr)$
        $\vee \neg \mathsf{btwn}_f(x,t,curr)))$
end procedure

Fig. 17. SEARCH-AND-SET. Predicates used to verify the example: $f^*(f(x),x)$, $f(x) = curr$, $d_1(curr)$, $d_2(curr)$, $data_1$, $data_2$, $x = curr$, $f^*(x,t)$, $x = \mathsf{nil}$, $\mathsf{btwn}_f(curr,t,x)$, $\mathsf{btwn}_f(x,t,curr)$, $d_1(t)$, $d_2(t)$, $true$, $curr = t$, $x = t$.

procedure SET-UNION(a, b)
  assume $f(a) = curr \wedge f^*(f(a),a) \wedge f^*(f(b),b)$
  assume $f^*(a,t) \wedge \neg f^*(b,t) \wedge \neg d(t)$
  assume $\neg f^*(a,s) \wedge f^*(b,s) \wedge d(s)$
  assume $\neg d(a) \wedge d(b)$
  assume $\mathsf{btwn}_f(f(a),t,a) \wedge \mathsf{btwn}_f(f(b),s,b)$
  assume $\neg t = \mathsf{nil} \wedge \neg s = \mathsf{nil}$
  $tmpd := d(b)$;
  $d(a) := tmpd$;
  while $\neg curr = a$ **do**
    $d(curr) := tmpd$;
    $curr := f(curr)$;
  end while
  $tmp := f(a)$;
  $f(a) := f(b)$;
  $f(b) := tmp$;
  assert $f^*(f(b),b) \wedge f^*(b,t) \wedge d(t) \wedge f^*(b,s) \wedge d(s) \wedge d(b) \wedge \neg t = \mathsf{nil} \wedge \neg s = \mathsf{nil}$
end procedure

Fig. 18. SET-UNION. Predicates used to verify the example: $a = curr$, $f(a) = curr$, $f^*(f(a),a)$, $f^*(f(b),b)$, $f^*(f(a),t)$, $f^*(f(b),t)$, $d(t)$, $f^*(f(a),s)$, $f^*(f(b),s)$, $d(s)$, $d(a)$, $d(b)$, $\mathsf{btwn}_f(f(a),t,a)$, $\mathsf{btwn}_f(f(b),s,b)$, $t = \mathsf{nil}$, $s = \mathsf{nil}$, $tmpd$, $d(curr)$, $\mathsf{btwn}_f(f(a),s,b)$, $\mathsf{btwn}_f(tmp,t,a)$, $\mathsf{btwn}_f(curr,t,a)$.

1: **procedure** CREATE-INSERT(*head*)
2: **assume** $p = head \wedge \neg t = \mathsf{nil} \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f^*(f(malloc), malloc) \wedge \neg f(malloc) = pmalloc \wedge item = \mathsf{nil} \wedge f(pmalloc) = malloc \wedge \neg malloc = \mathsf{nil} \wedge f^*(malloc, pmalloc)$
3: **assume** $(f^*(head,t) \wedge \neg f^*(malloc,t)) \ \vee \ (\neg f^*(head,t) \wedge f^*(malloc,t))$
4: **assume** $\neg f(malloc) = pmalloc$;
5: $item := malloc$;
6: $malloc := f(malloc)$;
7: $f(pmalloc) := malloc$;
8: **while** true **do**
9: **if** $ND \vee f(p) = \mathsf{nil}$ **then**
10: $f(item) := f(p)$;
11: $f(p) := item$;
12: **break**;
13: **else**
14: $p := f(p)$;
15: **end if**
16: **end while**
17: **assert** $\neg t = \mathsf{nil} \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f^*(f(malloc), malloc) \wedge \neg item = \mathsf{nil} \wedge f^*(head, item) \wedge f(pmalloc) = malloc \wedge \neg malloc = \mathsf{nil} \wedge f^*(malloc, pmalloc)$
18: **assert** $(f^*(head,t) \wedge \neg f^*(malloc,t)) \ \vee \ (\neg f^*(head,t) \wedge f^*(malloc,t))$
19: **end procedure**

Fig. 19. CREATE-INSERT. Predicates used to verify the example: $f^*(head,t)$, $\mathsf{nil} = f(p)$, $p = head$, $t = \mathsf{nil}$, $f^*(head,\mathsf{nil})$, $head = \mathsf{nil}$, $f^*(f(malloc), malloc)$, $f(malloc) = pmalloc$, $f^*(malloc,t)$, $item = \mathsf{nil}$, $f^*(head, item)$, $f(pmalloc) = malloc$, $malloc = \mathsf{nil}$, $f^*(malloc, pmalloc)$, $f^*(head, p)$, $f^*(item, \mathsf{nil})$, $f^*(item, p)$, $f^*(item, t)$, $f^*(f(p), t)$ $item = t$, $f(pmalloc) = item$, $f(f(pmalloc)) = malloc$, $f^*(malloc, item)$, $\mathsf{nil} = f(item)$.
Lines 4-7 model a *malloc* statement by removing a node from an infinite cyclic list which represents unallocated nodes.

1: **procedure** CREATE-INSERT-DATA(*head*)
2: **assume** $p = head \land \neg t = \mathsf{nil} \land f^*(head, \mathsf{nil}) \land \neg head = \mathsf{nil} \land f^*(f(malloc), malloc) \land \neg f(malloc) = pmalloc \land item = \mathsf{nil} \land f(pmalloc) = malloc \land \neg malloc = \mathsf{nil} \land f^*(malloc, pmalloc)$
3: **assume** $(f^*(head,t) \land \neg f^*(malloc,t) \land \neg(d(head) \oplus d(t)))$ $\lor$ $(\neg f^*(head,t) \land f^*(malloc,t))$
4: **assume** $\neg f(malloc) = pmalloc$;
5: $item := malloc$;
6: $malloc := f(malloc)$;
7: $f(pmalloc) := malloc$;
8: $tmpd := d(head)$;
9: $d(item) := tmpd$;
10: **while** true **do**
11: **if** $ND \lor f(p) = \mathsf{nil}$ **then**
12: $f(item) := f(p)$;
13: $f(p) := item$;
14: **break**;
15: **else**
16: $p := f(p)$;
17: **end if**
18: **end while**
19: **assert** $\neg t = \mathsf{nil} \land f^*(head, \mathsf{nil}) \land \neg head = \mathsf{nil} \land f^*(f(malloc), malloc) \land \neg item = \mathsf{nil} \land f^*(head, item) \land f(pmalloc) = malloc \land \neg malloc = \mathsf{nil} \land f^*(malloc, pmalloc)$
20: **assert** $(f^*(head,t) \land \neg f^*(malloc,t) \land \neg(d(head) \oplus d(t)))$ $\lor$ $(\neg f^*(head,t) \land f^*(malloc,t))$
21: **end procedure**

Fig. 20. CREATE-INSERT-DATA. Predicates used to verify the example: $f^*(head,t)$, $\mathsf{nil} = f(p)$, $p = head$, $t = \mathsf{nil}$, $f^*(head, \mathsf{nil})$, $head = \mathsf{nil}$, $f^*(f(malloc), malloc)$, $f(malloc) = pmalloc$, $f^*(malloc,t)$, $item = \mathsf{nil}$, $f^*(head, item)$, $f(pmalloc) = malloc$, $malloc = \mathsf{nil}$, $f^*(malloc, pmalloc)$, $d(head)$, $d(t)$, $f^*(item, p)$, $f^*(item,t)$, $f^*(f(p),t)$ $item = t$, $f(pmalloc) = item$, $f(f(pmalloc)) = malloc$, $f^*(malloc, item)$, $\mathsf{nil} = f(item)$, $f^*(head, p)$, $f^*(item, \mathsf{nil})$, *tmpd*.
Lines 4-7 model a *malloc* statement by removing a node from an infinite cyclic list which represents unallocated nodes.

1: **procedure** CREATE-FREE(*head*)
2: **assume** $p = head \wedge \neg t = \mathsf{nil} \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f^*(f(malloc), malloc) \wedge \neg f(malloc) = pmalloc \wedge item = \mathsf{nil} \wedge f(pmalloc) = malloc \wedge \neg malloc = \mathsf{nil} \wedge f^*(malloc, pmalloc)$
3: **assume** $(f^*(head,t) \wedge \neg f^*(malloc,t)) \;\vee\; (\neg f^*(head,t) \wedge f^*(malloc,t))$
4: **assume** $\neg f(malloc) = pmalloc$;
5: $item := malloc$;
6: $malloc := f(malloc)$;
7: $f(pmalloc) := malloc$;
8: **while** true **do**
9: **if** $ND \vee f(p) = \mathsf{nil}$ **then**
10: $f(item) := f(p)$;
11: $f(p) := item$;
12: **break**;
13: **else**
14: $p := f(p)$;
15: **end if**
16: **end while**
17: $p := head$;
18: $r := f(head)$;
19: **while** true **do**
20: **if** $ND \vee f(r) = \mathsf{nil}$ **then**
21: $f(p) := f(r)$;
22: $f(pmalloc) := r$;
23: $f(r) := malloc$;
24: $malloc := r$;
25: **break**;
26: **else**
27: $p := r$;
28: $r := f(r)$;
29: **end if**
30: **end while**
31: **assert** $\neg t = \mathsf{nil} \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f^*(f(malloc), malloc) \wedge \neg item = \mathsf{nil} \wedge f(pmalloc) = malloc \wedge \neg malloc = \mathsf{nil} \wedge f^*(malloc, pmalloc) \wedge \neg f^*(head, r)$
32: **assert** $(f^*(head,t) \wedge \neg f^*(malloc,t) \wedge \neg r = t \wedge (f^*(head, item) \oplus r = item)) \;\vee\; (\neg f^*(head,t) \wedge f^*(malloc,t) \wedge (f^*(head, item) \oplus r = item))$
33: **end procedure**

Fig. 21. CREATE-FREE. Predicates used to verify the example: $f^*(head,t)$, $\mathsf{nil} = f(p)$, $p = head$, $t = \mathsf{nil}$, $f^*(head, \mathsf{nil})$, $head = \mathsf{nil}$, $f^*(f(malloc), malloc)$, $f(malloc) = pmalloc$, $f^*(malloc,t)$, $item = \mathsf{nil}$, $f^*(head, item)$, $f(pmalloc) = malloc$, $malloc = \mathsf{nil}$, $f^*(malloc, pmalloc)$, $\mathsf{nil} = f(r)$, $r = t$, $r = item$, $f^*(head,r)$, $f^*(head,p)$, $f^*(item, \mathsf{nil})$, $f^*(item,p)$, $f^*(malloc, item)$, $f^*(item,t)$, $f^*(f(p),t)$ $item = t$, $f(pmalloc) = item$, $f(f(pmalloc)) = malloc$, $\mathsf{nil} = f(item)$, $f(pmalloc) = r$, $f^*(f(p),r)$ $f^*(f(p), f(r))$.

Lines 4-7 model a *malloc* statement by removing a node from an infinite cyclic list which represents unallocated nodes. Lines 22-24 model a *free* statement by returning a node to the infinite cyclic list of unallocated nodes.

procedure INIT-LIST(x)
  assume $f^*(x,t) \wedge f^*(x,\mathsf{nil}) \wedge curr = x \wedge \neg t = \mathsf{nil} \wedge true$
  while $\neg curr = \mathsf{nil}$ **do**
    $d(curr) := true$;
    $curr := f(curr)$;
  end while
  assert $f^*(x,t) \wedge f^*(x,\mathsf{nil}) \wedge d(t) \wedge true \wedge \neg t = \mathsf{nil}$
end procedure

Fig. 22. INIT-LIST. Predicates used to verify the example: $curr = \mathsf{nil}$, $curr = x$, $f^*(x,t)$, $f^*(x,\mathsf{nil})$, $d(t)$, $true$, $t = \mathsf{nil}$, $f^*(curr,t)$, $f^*(t,curr)$.

procedure INIT-LIST-VAR(x, tmp)
  assume $f^*(x,t) \wedge f^*(x,\mathsf{nil}) \wedge curr = x \wedge \neg t = \mathsf{nil} \wedge true$
  while $\neg curr = \mathsf{nil}$ **do**
    $d(curr) := true$;
    $curr := f(curr)$;
  end while
  $tmp := d(x)$;
  assert $f^*(x,t) \wedge f^*(x,\mathsf{nil}) \wedge d(t) \wedge true \wedge tmp \wedge \neg t = \mathsf{nil}$
end procedure

Fig. 23. INIT-LIST-VAR. Predicates used to verify the example: $curr = \mathsf{nil}$, $curr = x$, $f^*(x,t)$, $f^*(x,\mathsf{nil})$, $d(t)$, $true$, $t = \mathsf{nil}$, tmp, $f^*(curr,t)$, $f^*(t,curr)$, $d(x)$.
Parameter tmp is a boolean variable.

procedure INIT-CYCLIC(x)
  assume $f^*(x,t) \wedge f^*(f(x),x) \wedge curr = f(x) \wedge \mathsf{btwn}_f(curr,t,x) \wedge \neg x = \mathsf{nil} \wedge true$
  $d(x) := true$;
  while $\neg curr = x$ **do**
    $d(curr) := true$;
    $curr := f(curr)$;
  end while
  assert $f^*(x,t) \wedge f^*(f(x),x) \wedge d(t) \wedge true \wedge \neg x = \mathsf{nil}$
end procedure

Fig. 24. INIT-CYCLIC. Predicates used to verify the example: $curr = x$, $curr = f(x)$, $f^*(x,t)$, $f^*(f(x),x)$, $d(t)$, $true$, $\mathsf{btwn}_f(curr,t,x)$, $x = \mathsf{nil}$, $t = x$, $\mathsf{btwn}_f(x,t,curr)$, $f^*(t,curr)$.

procedure SORTED-INSERT-DNODES($head, item$)
  assume $\neg f^*(head, item) \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil}$
  assume $f^*(head, t) \oplus item = t$
  assume $\neg t = \mathsf{nil} \wedge f(item) = \mathsf{nil}$
  assume $d(t) \leq d(f(t)) \vee f(t) = \mathsf{nil}$
  assume $curr = head \wedge succ = f(head) \wedge f(g(t)) = \mathsf{nil} \wedge g(g(t)) = \mathsf{nil} \wedge f(g(item)) = \mathsf{nil} \wedge$ $g(g(item)) = \mathsf{nil} \wedge g(t) = s$
  while $\neg succ = \mathsf{nil} \wedge d(g(item)) > d(g(succ))$ **do**
    $curr := succ$;
    $succ := f(curr)$;
  end while
  if $d(g(head)) > d(g(item))$ **then**
    $f(item) := head$;
    $head := item$;
  else
    $f(item) := succ$;
    $f(curr) := item$;
  end if
  assert $f^*(head, \mathsf{nil}) \wedge f^*(head, t) \wedge (d(t) \leq d(f(t)) \vee f(t) = \mathsf{nil}) \wedge g(t) = s$
end procedure

Fig. 25. SORTED-INSERT-DNODES. Predicates used to verify the example: $f^*(head, t)$, $succ = \mathsf{nil}$, $d(g(item))$, $d(g(succ))$, $d(g(t))$, $d(g(f(t)))$, $t = \mathsf{nil}$, $f(t) = \mathsf{nil}$, $d(g(head))$, $item = t$, $curr = head$, $f^*(head, item)$, $succ = f(head)$, $f^*(head, \mathsf{nil})$, $f(item) = \mathsf{nil}$, $head = \mathsf{nil}$, $f(g(t)) = \mathsf{nil}$, $g(g(t)) = \mathsf{nil}$, $f(g(item)) = \mathsf{nil}$, $g(g(item)) = \mathsf{nil}$, $g(t) = s$, $f(item) = succ$, $f(curr) = succ$, $f(item) = curr$, $f^*(head, curr)$.
Comparison between data values is defined as a formula over boolean data predicates. For instance, $d(x) \leq d(y)$ is defined as $\neg(d(x) \wedge \neg d(y))$.

```
procedure REMOVE-DOUBLY(head, tail, node)
    assume next*(head, tail) ∧ prev*(tail, head) ∧ nil = next(tail) ∧ nil = prev(head)
    assume next*(head, t) ∧ prev*(tail, t)
    assume next*(head, node) ∧ prev*(tail, node)
    assume next*(head, prev(node)) ∧ prev*(tail, next(node))
    assume ¬node = nil ∧ ¬t = nil
    assume (    head = t
                ∧ (head = t ⊕ t = f(g(t)))
                ∧ (tail = t ⊕ t = g(f(t))))
           ∨ (  ¬head = t ∧ (head = t ⊕ t = f(g(t)))
                ∧ (tail = t ⊕ t = g(f(t)))
                ∧ (head = node ⊕ node = f(g(node)))
                ∧ (tail = node ⊕ node = g(f(node))))
    if prev(node) = nil then
        head := next(node);
    else
        temp := prev(node);
        next(temp) := next(node);
    end if
    if next(node) = nil then
        tail := prev(node);
    else
        temp := next(node);
        prev(temp) := prev(node);
    end if
    assert next*(head, tail) ∧ prev*(tail, head) ∧ nil = next(tail) ∧ nil = prev(head)
    assert ¬next*(head, node) ∧ ¬prev*(tail, node)
    assert ¬node = nil ∧ ¬t = nil
    assert (node = t ∧ ¬next*(head, t) ∧ ¬prev*(head, t) ∧ ¬head = t ∧ ¬tail = t)
        ∨ (   ¬node = t ∧ next*(head, t) ∧ prev*(tail, t)
              ∧ (head = t ⊕ t = f(g(t)))
              ∧ (tail = t ⊕ t = g(f(t))))
end procedure
```

Fig. 26. REMOVE-DOUBLY. Predicates used to verify the example: $\mathsf{nil} = prev(node)$, $\mathsf{nil} = next(node)$,
$next^*(head, tail)$, $prev^*(tail, head)$,
$\mathsf{nil} = next(tail)$, $\mathsf{nil} = prev(head)$,
$next^*(head, t)$, $prev^*(tail, t)$,
$head = t$, $tail = t$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, node)$, $prev^*(tail, node)$,
$head = node$, $tail = node$,
$node = prev(next(node))$, $node = next(prev(node))$,
$next^*(head, prev(node))$, $prev^*(tail, next(node))$,
$\mathsf{nil} = prev(next(node))$, $\mathsf{nil} = next(prev(node))$,
$node = \mathsf{nil}$, $node = t$, $t = \mathsf{nil}$, $head = next(node)$, $head = temp$, $temp = prev(node)$, $next^*(head, temp)$, $temp = t$, $temp = next(prev(node))$, $next(node) = next(prev(node))$, $prev(temp) = prev(node)$, $prev(node) = t$.

procedure REMOVE-CYCLIC-DOUBLY($head, entry$)

  assume $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assume $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assume $next^*(head, t) \wedge prev^*(head, t)$
  assume $next^*(head, prev(t)) \wedge prev^*(head, next(t))$
  assume $next^*(head, entry) \wedge prev^*(head, entry)$
  assume $next^*(head, prev(entry)) \wedge prev^*(head, next(entry))$
  assume $t = prev(next(t)) \wedge t = next(prev(t))$
  assume $head = prev(next(head)) \wedge head = next(prev(head))$
  assume $entry = prev(next(entry)) \wedge entry = next(prev(entry))$
  assume $\neg entry = head$
  $p := prev(entry);$
  $n := next(entry);$
  $prev(n) := p;$
  $next(p) := n;$
  assert $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assert $\neg next^*(head, entry) \wedge \neg prev^*(head, entry)$
  assert $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assert $head = prev(next(head)) \wedge head = next(prev(head))$
  assert $\neg entry = prev(next(entry)) \wedge \neg entry = next(prev(entry))$
  assert $\neg entry = head$
  assert $(\quad next^*(head, t) \wedge prev^*(head, t) \wedge t = prev(next(t))$
    $\wedge\ t = next(prev(t)) \wedge \neg entry = t \wedge next^*(head, prev(t)) \wedge prev^*(head, next(t)))$
   $\vee\ (\quad \neg next^*(head, t) \wedge \neg prev^*(head, t)$
    $\wedge\ \neg t = prev(next(t)) \wedge \neg t = next(prev(t)) \wedge entry = t)$

end procedure

Fig. 27. REMOVE-CYCLIC-DOUBLY. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, entry)$, $prev^*(head, entry)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$head = prev(next(head))$, $head = next(prev(head))$,
$entry = prev(next(entry))$, $entry = next(prev(entry))$,
$next^*(head, prev(entry))$, $prev^*(head, next(entry))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = entry$, $entry = head$, $next(entry) = n$, $prev(entry) = p$, $n = head$, $prev(n) = p$, $n = t$, $next(head) = entry$, $p = t$.

procedure LINUX-LIST-ADD($head, new$)
  assume $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assume $\neg next^*(head, new) \wedge \neg prev^*(head, new)$
  assume $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assume $next(new) = \mathsf{nil} \wedge prev(new) = \mathsf{nil}$
  assume $head = prev(next(head)) \wedge head = next(prev(head))$
  assume $(\quad next^*(head,t) \wedge prev^*(head,t) \wedge t = prev(next(t)) \wedge t = next(prev(t))$
    $\wedge\ \neg t = new \wedge next^*(head, prev(t)) \wedge prev^*(head, next(t)))$
    $\vee\ (\quad \neg next^*(head,t) \wedge \neg prev^*(head,t) \wedge \neg t = prev(next(t)) \wedge \neg t = next(prev(t))$
    $\wedge\ t = new \wedge \neg next^*(head, prev(t)) \wedge \neg prev^*(head, next(t)))$
  $p := head;$
  $n := next(head);$
  $prev(n) := new;$
  $next(new) := n;$
  $prev(new) := p;$
  $next(p) := new;$
  assert $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assert $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assert $next^*(head,t) \wedge prev^*(head,t)$
  assert $next^*(head, prev(t)) \wedge prev^*(head, next(t))$
  assert $next^*(head, new) \wedge prev^*(head, new)$
  assert $t = prev(next(t)) \wedge t = next(prev(t))$
  assert $\neg next(new) = \mathsf{nil} \wedge \neg prev(new) = \mathsf{nil}$
  assert $head = prev(next(head)) \wedge head = next(prev(head))$
end procedure

Fig. 28. LINUX-LIST-ADD. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head,t)$, $prev^*(head,t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, new)$, $prev^*(head, new)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$next(new) = \mathsf{nil}$, $prev(new) = \mathsf{nil}$,
$head = prev(next(head))$, $head = next(prev(head))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = new$, $p = head$, $next(new) = n$, $prev(n) = new$, $n = t$, $head = t$, $prev(new) = head$, $n = head$,
$n = next(head)$.

procedure LINUX-LIST-ADD-TAIL($head, new$)

  assume $next^*(next(head), head) \wedge prev^*(prev(head), head)$

  assume $prev^*(head, next(head)) \wedge next^*(head, prev(head))$

  assume $\neg next^*(head, new) \wedge \neg prev^*(head, new)$

  assume $next(new) = \mathsf{nil} \wedge prev(new) = \mathsf{nil}$

  assume $head = prev(next(head)) \wedge head = next(prev(head))$

  assume $(\quad next^*(head, t) \wedge prev^*(head, t) \wedge t = prev(next(t)) \wedge t = next(prev(t))$
  $\quad\quad \wedge\ \neg t = new \wedge next^*(head, prev(t)) \wedge prev^*(head, next(t)))$
  $\quad \vee \quad (\quad \neg next^*(head, t) \wedge \neg prev^*(head, t) \wedge \neg t = prev(next(t)) \wedge \neg t = next(prev(t))$
  $\quad\quad \wedge\ t = new \wedge \neg next^*(head, prev(t)) \wedge \neg prev^*(head, next(t)))$

  $p := prev(head);$

  $n := head;$

  $prev(n) := new;$

  $next(new) := n;$

  $prev(new) := p;$

  $next(p) := new;$

  assert $next^*(next(head), head) \wedge prev^*(prev(head), head)$

  assert $prev^*(head, next(head)) \wedge next^*(head, prev(head))$

  assert $next^*(head, t) \wedge prev^*(head, t)$

  assert $next^*(head, prev(t)) \wedge prev^*(head, next(t))$

  assert $next^*(head, new) \wedge prev^*(head, new)$

  assert $t = prev(next(t)) \wedge t = next(prev(t))$

  assert $\neg next(new) = \mathsf{nil} \wedge \neg prev(new) = \mathsf{nil}$

  assert $head = prev(next(head)) \wedge head = next(prev(head))$

end procedure

Fig. 29. LINUX-LIST-ADD-TAIL. Predicates used to verify the example: $next^*(next(head), head)$, $prev^*(prev(head), head)$, $next^*(head, t)$, $prev^*(head, t)$, $t = prev(next(t))$, $t = next(prev(t))$, $next^*(head, new)$, $prev^*(head, new)$, $prev^*(head, next(head))$, $next^*(head, prev(head))$, $next(new) = \mathsf{nil}$, $prev(new) = \mathsf{nil}$, $head = prev(next(head))$, $head = next(prev(head))$, $next^*(head, prev(t))$, $prev^*(head, next(t))$, $t = new$, $p = t$, $prev(new) = p$, $next^*(head, p)$, $next(p) = head$, $new = prev(head)$, $prev^*(p, next(head))$, $n = head$, $prev(new) = head$, $prev^*(p, next(t))$, $n = t$.

procedure LINUX-LIST-DEL($head, entry$)
  assume $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assume $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assume $next^*(head, t) \wedge prev^*(head, t)$
  assume $next^*(head, prev(t)) \wedge prev^*(head, next(t))$
  assume $next^*(head, entry) \wedge prev^*(head, entry)$
  assume $next^*(head, prev(entry)) \wedge prev^*(head, next(entry))$
  assume $t = prev(next(t)) \wedge t = next(prev(t))$
  assume $head = prev(next(head)) \wedge head = next(prev(head))$
  assume $entry = prev(next(entry)) \wedge entry = next(prev(entry))$
  assume $\neg next(entry) = \mathsf{nil} \wedge \neg prev(entry) = \mathsf{nil}$
  assume $\neg entry = head$
  $p := prev(entry)$;
  $n := next(entry)$;
  $prev(n) := p$;
  $next(p) := n$;
  $next(entry) := \mathsf{nil}$;
  $prev(entry) := \mathsf{nil}$;
  assert $next^*(next(head), head) \wedge prev^*(prev(head), head)$
  assert $prev^*(head, next(head)) \wedge next^*(head, prev(head))$
  assert $\neg next^*(head, entry) \wedge \neg prev^*(head, entry)$
  assert $\neg next^*(head, prev(entry)) \wedge \neg prev^*(head, next(entry))$
  assert $next(entry) = \mathsf{nil} \wedge prev(entry) = \mathsf{nil}$
  assert $head = prev(next(head)) \wedge head = next(prev(head))$
  assert $\neg entry = prev(next(entry)) \wedge \neg entry = next(prev(entry))$
  assert $\neg entry = head$
  assert $(\quad next^*(head, t) \wedge prev^*(head, t) \wedge t = prev(next(t)) \wedge t = next(prev(t))$
    $\wedge\ \neg entry = t \wedge next^*(head, prev(t)) \wedge prev^*(head, next(t)))$
   $\vee\ (\quad \neg next^*(head, t) \wedge \neg prev^*(head, t) \wedge \neg t = prev(next(t)) \wedge \neg t = next(prev(t))$
    $\wedge\ entry = t \wedge \neg next^*(head, prev(t)) \wedge \neg prev^*(head, next(t)))$
end procedure

Fig. 30. LINUX-LIST-DEL. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, entry)$, $prev^*(head, entry)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$next(entry) = \mathsf{nil}$, $prev(entry) = \mathsf{nil}$,
$head = prev(next(head))$, $head = next(prev(head))$,
$entry = prev(next(entry))$, $entry = next(prev(entry))$,
$next^*(head, prev(entry))$, $prev^*(head, next(entry))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = entry$, $entry = head$, $next(entry) = n$, $prev(entry) = p$, $n = head$, $prev(n) = p$, $n = t$, $next(head) = entry$, $p = t$.