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

Jesse Bingham and Zvonimir Rakamarić

Department of Computer Science, University of British Columbia, Canada
jesse.d.bingham@intel.com zrakamar@cs.ubc.ca

Abstract. An important and ubiquitous class of programs are *heap-manipulating programs* (HMP), which manipulate unbounded linked data structures by following pointers and updating links. *Predicate abstraction* has proved to be an invaluable technique in the field of software model checking; this technique relies on an efficient decision procedure for the underlying logic. The expression and proof of many interesting HMP safety properties require *transitive closure* predicates; such predicates express that some node can be reached from another node by following a sequence of (zero or more) links in the data structure. Unfortunately, adding support for transitive closure often yields undecidability, so one must be careful in defining such a logic. Our primary contributions are the definition of a simple transitive closure logic for use in predicate abstraction of HMPs, and a decision procedure for this logic. Through several experimental examples, we demonstrate that our logic is expressive enough to prove interesting properties with predicate abstraction, and that our decision procedure provides us with both a time and space advantage over previous approaches.

1 Introduction

In recent years *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 the use of *predicate abstraction* on the program source [16, 14, 3, 18]. In predicate abstraction, sets of states of the program and program transitions are over-approximated using a finite set of predicates over the program variables. These predicates (or boolean combinations thereof) typically express features of the program under verification such

* This is the second version of UBC Department of Computer Science Technical Report TR-2005-19. The results in the original version, and also those reported in an abridged published version [6], were found to have been produced by a buggy implementation of our approach. The tool used for the experiments in this paper corrects this bug, and also uses a decision procedure implemented in C++ (rather than Perl), and hence is an order of magnitude faster. Unfortunately, fixing this bug revealed to us that, contrary to the erroneous results in the original version and [6], the example program REMOVE-ELEMENTS (which involves a circularly linked list) cannot be handled by our approach. We have concluded that the logic defined herein is not adequate for handling programs that manipulate circularly linked lists. However, we are currently researching a promising solution to this problem as discussed in Sect. 8 of this version. The authors were supported by grants from the Natural Sciences and Research Council of Canada (NSERC). Jesse Bingham has moved to Intel Corporation, Hillsboro, Oregon, U.S.A.

as its conditionals and relevant propositions about its variables. An integral ingredient in predicate abstraction is a decision procedure for the logic of the predicates. Since most approaches involve many queries to this decision procedure, performance is paramount.

An important class of programs are those we call *heap-manipulating programs* (HMPs), which are 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 (that 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. *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 circular linked list.

Several papers have previously identified the importance of transitive closure for HMPs [30, 31, 5, 19, 2, 23]. Unfortunately, adding support for transitive closure to even relatively tame logics often yields undecidability [19]. Our first contribution is a fragment of the decidable logics that we show (through several nontrivial experiments) is still expressive enough to verify properties of interest for HMPs using predicate abstraction. Decidability of our logic follows from a small model theorem, akin to that of Benedikt et al. [5] and Balaban et al. [2], which states that if a set of predicates is satisfiable, then it is satisfiable by a heap structure with some bounded number of nodes. A naive decision procedure can thus enumerate all the (super-factorial but finite) number of structures of size up to this bound. We do not formally state or prove a small model theorem in this paper, rather, our second and most important contribution is an efficient decision procedure for our logic. We show that this procedure, though a worst case exponential time algorithm, solves the vast majority of queries sent to it during predicate abstraction very quickly. The result is an approach that can have large time and memory savings over decision procedures that enumerate all models, even when BDDs are used for this enumeration, as done by Balaban et al. [2].

The paper is organized as follows. Sect. 2 summarizes other work on verification of HMPs. Predicate abstraction and our verification framework (based on previous work), is outlined in Sect. 3. HMPs are introduced in Sect. 4. Sects. 5 and 6 respectively define our transitive closure logic and the decision procedure. We present experimental results in Sect. 7. Sect. 8 draws conclusions and discusses several important extensions to our logic and decision procedure that we believe are possible, but have been left as future work. 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.

2 Related Work

Balaban et al. [2] present an approach for shape analysis based on predicate abstraction that is similar to ours. The logic they use for describing properties of heap structures

has slightly richer expressiveness than the logic we define in this paper.[1] The major difference between the two approaches is the way a program abstraction is computed. To compute the abstraction, they employ a small model theorem, and build BDDs representing all models up to the small model size. This is a bottleneck in both computation time and memory, since these BDDs tend to blow-up. The technique of Kesten and Pnueli [21] for establishing termination employed by Balaban et al. is likely compatible with our work also.

McPeak and Necula [28] specify heap data structures using *local equality axioms*, which constrain only a bounded fragment of the heap around some node. This enables them to describe a variety of shapes and reason about scalar values without abstracting them, while still preserving decidability. However, they can only approximate reachability between nodes (though *un*reachability is precise). When pointer disequalities are added, their decision procedure becomes incomplete. We handle both reachability and disequalities, but we can't describe such a variety of shapes. In addition, we compute an inductive invariant of a program automatically (given an appropriate set of predicates), while they require a user to provide loop invariants, which can be a significant burden.

The *Pointer Assertion Logic Engine* (PALE) [29] specifies heap structures using graph types [22], which are tree-shaped data structures augmented with extra pointers. The authors show that many common heap structures can be defined that way, some of which we cannot express. PALE relies on a decision procedure with non-elementary complexity, so, there are programs that cannot be verified in practice. Furthermore, loop invariants must be provided by the user.

The *Three Valued Logic Analyzer* (TVLA) [32, 25] extends conventional abstract interpretation with a third "uncertain" logic value, and builds so-called *3-valued logical structures* that abstract the reachable states at each program point (a.k.a. *canonical abstraction*). The abstract semantics of program statements are defined by *abstract transformers*, which can be generated by TVLA or user-defined if necessary. We cannot handle all heap structures that TVLA can, however, the abstract invariant we compute is always the most precise w.r.t. the given set of predicates. TVLA does not make such a guarantee, although some work has been done to make TLVA more precise [33]. TVLA is also employed by Manevich et al. [27], who observe that the number of shared nodes in linked lists is bounded and present a novel definition of "uninterrupted list segments". This is used to define predicate and canonical abstractions of potentially circular singly linked lists, and enables them to verify some HMPs that we are not able to verify, though their properties tend to be simpler than ours (see Sect. 7).

Lahiri and Qadeer [23] define two new predicates to express reachability of heap nodes in linked lists. To prove properties of HMPs, they use first-order axioms over those predicates. The given set of axioms is incomplete, and they provide an induction principle that is used to derive additional axioms when necessary. Because of the purely first-order axiomatization, they are able to harness the power of available automated theorem provers; they use UCLID [8] as the underlying inference engine.

Dams and Namjoshi [11] propose an approach based on predicate abstraction and model checking. They abstract a program by iteratively calculating weakest preconditions of shape predicates, and are able to handle second-order shape properties such

[1] Whereas our logic is unquantified, they allow restricted universal quantification.

as reachability, cyclicity, and sharing. The algorithm doesn't use a decision procedure, and as a consequence, new predicates can be generated in every iteration. Hence, the algorithm often has to be manually provided with "approximation hints" to converge.

3 Verification Approach

3.1 Predicate Abstraction

Our approach to verifying heap programs is based on *predicate abstraction* [16], which is an instance of *abstract interpretation* [10]. In the framework of abstract interpretation, a *concrete system* (in our case an HMP) is verified by constructing a finite-state over-approximation of the concrete system called the *abstract system*. Let $\mathscr{C}$ (the *concrete states*) be the set of states of the concrete system. Predicate abstraction employs a finite set of predicates $\phi_1, \ldots, \phi_k$ in some logic that are assertions about concrete states. Corresponding to the predicates respectively are the *abstract boolean variables* $b_1, \ldots, b_k$. The set of *abstract states* $\mathscr{A}$ will be the set of assignments to the abstract boolean variables. The *abstraction function* $\alpha : \mathscr{C} \to \mathscr{A}$ is defined such that $\alpha(c)(b_i) = \mathsf{true}$ if and only if $c \vDash \phi_i$. A set of concrete states C is then abstracted by

$$\alpha(C) = \{\alpha(c) \mid c \in C\}$$

Note that since $\mathscr{A}$ is finite, $\alpha(C)$ is always finite as well. In contrast, $\mathscr{C}$ is often infinite; in our case the infinitude of concrete states arises from the unboundedness of the heap in HMPs.

Let $R \subseteq \mathscr{C}$ be the set of concrete states that are reachable in the concrete system. We wish to verify that a property expressed as a state assertion ψ over the concrete states holds for all members of R, i.e. that the implication $R \to \psi$ holds. Predicate abstraction is used to solve this problem by computing a set $R^\alpha \subseteq \mathscr{A}$ such that $\alpha(R) \subseteq R^\alpha$. Verification succeeds if one can prove that $R^\alpha \to \psi$. A key difference in the various approaches to predicate abstraction is how R^α is computed [16, 14, 12, 15, 2, 11]. This typically involves numerous queries to a decision procedure for the underlying logic and there are tradeoffs between how accurately R^α approximates $\alpha(R)$ and the number and complexity of these queries.

R^α is usually computed as a fixpoint of some approximation of the *abstract post image operator* $\mathsf{post} : 2^{\mathscr{A}} \to 2^{\mathscr{A}}$, defined as follows. Given a set of abstract states A, let

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

where T is the transition relation of the concrete system. $\mathsf{post}(A)$ is thus the set of abstract states representing concrete states that are concrete successors of those states represented by A.

Since predicate abstraction is an incomplete approach, if it fails to verify the property, this can either happen because the concrete systems actually violates the property, or because of the loss of information inherent in the abstraction. Finding the "right" set of predicates for verification to go through can be tricky business. Many works have addressed this issue of *predicate discovery* [13, 4, 18, 11], which falls under the more

general umbrella of *abstraction refinement* [9]. As in recent papers on this topic [2, 23], in our current framework, predicates are added by manual inspection of counterexample behaviors; applying automatic predicate discovery techniques is an important area of future work.

3.2 Computing post

Our tool computes post precisely; the algorithm can be viewed as an improvement over the following naive algorithm. Since post distributes over disjunction,[2] computing $\mathsf{post}(A)$ is reducible to computing $\mathsf{post}(\rho)$ for each cube ρ in some disjunctive normal form decomposition of A. Here, *cube* means a partial boolean assignment to the abstract variables, and represents all abstract states that agree on this subset of the abstract variables.[3] By using a BDD [7] to represent A, we can easily obtain such a decomposition. The naive algorithm cycles through all 2^k abstract states a, and checks if $a \in \mathsf{post}(\rho)$; $\mathsf{post}(\rho)$ is then the BDD representing the disjunction of all such a. Each check of $a \in \mathsf{post}(\rho)$ involves a call to the decision procedure to determine if the following formula is satisfiable:

$$\gamma(\rho) \wedge \mathsf{wp}(\gamma(a)) \tag{1}$$

where γ is the *concretization function*, and wp is the *weakest precondition* operator [17]. Intuitively, γ maps a cube to a logic formula that denotes the set of concrete states represented by the cube. Formally, for a cube μ let $P(\mu)$ (resp. $N(\mu)$) denote the set $\{i \mid \mu(b_i) = \mathsf{true}\}$ (resp. $\{i \mid \mu(b_i) = \mathsf{false}\}$). Then define

$$\gamma(\mu) \quad = \quad \bigwedge_{i \in P(\mu)} \phi_i \ \wedge \ \bigwedge_{i \in N(\mu)} \neg \phi_i$$

The weakest precondition operator wp is a syntactic transformation on logic formulas that depends on the program statement under consideration [17]. For example, for an assignment statement $x := e$, where x is a variable and e is some expression, $\mathsf{wp}(\pi)$ is constructed by syntactically replacing all occurrences of x with e in the formula π.[4] Our approach applies wp at the granularity of individual program statements when performing predicate abstraction.

Das et al.'s computation of post that we employ uses several straightforward optimizations over this naive algorithm [14]. First, if (1) contains a syntactic contradiction, meaning the existence of a predicate and its negation, then clearly the formula is not satisfiable. In such circumstances there is no need to call the decision procedure. When computing $\mathsf{post}(\rho)$, our implementation initially computes a BDD C representing the set of all a that won't yield such a contradiction. Second, rather than enumerating all $a \in C$, we do recursive case-splitting on the abstract variables, which allows for pruning of large portions of C. For example, let μ be the cube that assigns true to b_1 and leaves all other variables unconstrained. Then if $\gamma(\rho) \wedge \mathsf{wp}(\gamma(\mu))$ is unsatisfiable, then so too is $\gamma(\rho) \wedge \mathsf{wp}(\gamma(a))$ for *any* abstract state a that has b_1 equal to true. Hence, our algorithm would only explore those abstract states having b_1 false.

[2] meaning that $\mathsf{post}(A_1 \vee A_2) = \mathsf{post}(A_1) \vee \mathsf{post}(A_2)$

[3] A *partial boolean assignment* maps each variable b_i to an element of $\{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}$.

[4] This only works under the assumption that x cannot be aliased.

```
 1: procedure ND-INSERT(head, item)
 2:     assume ¬f*(head, item) ∧ f*(head, nil) ∧ ¬head = nil ∧ f(item) = nil ∧ p = head
 3:     while true do
 4:         if ND ∨ f(p) = nil then
 5:             f(item) := f(p);
 6:             f(p) := item;
 7:             break
 8:         else
 9:             p := f(p);
10:         end if
11:     end while
12:     assert f*(head, item) ∧ f*(head, nil)
13: end procedure
```

Fig. 1. A program that nondeterministically inserts a node *item* into the list pointed to by *head*. Here *ND* is a boolean value that is nondeterministically true or false.

4 Heap-Manipulating Programs

In our framework, the *heap* consists of an unbounded number of *nodes*. HMPs allow for node variables (pointers), data fields for nodes, a link field f for nodes, and all other variables are modelled (or encoded as) booleans.

In lieu of a formal presentation of HMPs, we give an example called ND-INSERT in Fig. 1 that captures most of the interesting features. This program takes a node *head* and a node *item*, and inserts *item* into the linked list pointed to by *head* at a position selected nondeterministically. *head* is assumed to be non-nil and to point to an acyclic linked list that does not contain *item*. These assumptions are formalized by the **assume** statement on line 2 of the program. In the **assume** statement, and also in the **assert** statement, the subformulas of the form $f^*(x,y)$ express that node y is reachable from node x by following a sequence of f links of any length; we will formally define these predicates in Sect. 5. The fact that nil is reachable from *head* enforces the acyclicality assumption.[5]

The body of ND-INSERT is straightforward; a pointer p walks the list, and *item* is inserted at some point. The loop breaks once the insertion has occurred. The expression *ND* represents a nondeterministic boolean value. *item* is inserted when either $ND =$ true, or the end of the list is reached (detected by the disjunct $f(p) =$ nil on line 4). The specification is expressed by the **assert** statement on line 12, and indicates that whenever line 12 is reached, *head* must point to an acyclic list that contains *item*.

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.

[5] In our logical framework, nil is modelled simply as a node having $f(\mathsf{nil}) = \mathsf{nil}$.

$$
\begin{array}{rcl}
v & \in & V \\
d & \in & D \\
b & \in & B \\
term & ::= & v \mid f(term) \\
atom & ::= & f^*(term, term) \mid term = term \mid d(term) \mid b \\
literal & ::= & atom \mid \neg atom
\end{array}
$$

Fig. 2. The syntax of our simple transitive closure logic.

5 A Simple Transitive Closure Logic

Our logic assumes finite sets of *node* variables V, *boolean* variables B, *data function* variables D, and a single *link function* symbol 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*, *inequality*, *reachability*, and *unreachability* literals, respectively. 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 *boolean 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 finite set of *nodes* N and a function Θ that interprets each symbol σ in $V \cup B \cup D \cup \{f\}$ such that

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

Thus Θ interprets each node variable as a node, each boolean variable as a boolean value, each data function variable as a function that maps nodes to booleans, and the link function f is interpreted as a mapping from nodes to nodes. Heap structures naturally model a linked data structure of nodes, each node having a single pointer to another node 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. 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 the obvious 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

$\Theta(f)^n(\Theta^e(\tau_1)) = \Theta^e(\tau_2)$.[6] Finally, a literal that is not an atom is 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 $\phi \in \Phi$.

6 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 $H \vDash \Phi$? 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 (1) by simply taking Φ to be the set of its conjuncts.

Decidability of this problem follows from a small model theorem enjoyed by our logic, akin to other transitive closure logics [2, 5]. Our small model theorem states that Φ is satisfiable if and only if there exists H of size at most n such that $H \vDash \Phi$, where n is the number of distinct terms mentioned in Φ. Hence, a decision procedure can simply enumerate the finite set of such H, and for each one check if $H \vDash \Phi$. However, since the number of such heap structures is at least n^n, this approach is impractical. Employing BDDs [7] to represent the set of heap structures that satisfy Φ [2] is also memory-intensive; building a BDD for the literal $f^*(x,y)$ over just 8 nodes cannot be done in 2 GB of memory. This stems from the fact that such a BDD must represent the multitude of different paths that could exist between the nodes $\Theta^e(x)$ and $\Theta^e(y)$.

Our approach has relatively small memory requirements, and is based on a set of inference rules (IRs) with the property that Φ is satisfiable if and only if their exhaustive application does not introduce a contradiction. Here *contradiction* means the inference of both an atom ϕ and its negation $\neg\phi$. The IRs are presented in Fig. 3. For an IR r, the *antecedents* of 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 explain each IR of Fig. 3. IDENT states that each node variable is equal to itself, while REFLEX enforces that any node variable is reachable from itself. TRANS1 states that the transitive closure f^* must extend the function f. TRANS2 simply enforces that f^* is transitive. FUNC asserts that if $f(x) = y$ and z is reachable from x, then z must also be reachable from y, unless $x = z$. 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$_k$. Similar to FUNC is SCC, which states that if x and y are distinct and mutually reachable from each other, and z is reachable from x, then x is reachable from z (since x must lie on a cycle of f). TOTAL requires that if y and z are both reachable from another node x, then there must exist some reachability relationship between y and z. The fact that in a cycle of f, no two distinct nodes x and y can have

[6] 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 $n \geq 1$.

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

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

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

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

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

Fig. 3. The set of inference rules. Here x, y, and z range over (not necessarily distinct) terms. In the rules IDENT and REFLEX, v is restricted to be a variable that is already mentioned; this restriction prevents either of these rules from introducing new terms. $\textsc{Cycle}_k$ actually defines a separate rule for each $k \geq 1$.

$f(x)=f(y)$ is captured by SHARE. Given the preceding intuition, it is easy to prove the following.

Theorem 1. *The inference rules of Fig. 3 are sound.*

Theorem 1 tells us that if iterative application of the IRs yields a contradiction, then we can conclude that the original set of literals is unsatisfiable. Conversely, we have proven our IRs to be complete with respect to sets of literals in a certain normal form, and Theorem 2 below states that it is sufficient to restrict attention to such sets. 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*

1. *For each $v_i \in \mathsf{Vars}(\Phi)$, there exists*
   (a) at most one equality literal of the form $f(v_i) = v_j$, where $v_j \in \mathsf{Vars}(\Phi)$, and
   (b) the literal $v_i = v_i$.
   All equality literals of Φ are required to be of one of the forms (a) or (b).
2. *All inequality literals are of the form $\neg v_i = v_j$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
3. *All reachability literals are of the form $f^*(v_i, v_j)$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
4. *All unreachability literals are of the form $\neg f^*(v_i, v_j)$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
5. *There exist no data or boolean variable literals in Φ.*

Theorem 2. *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 2, 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. The following completeness result is the crux of our decision procedure.

Theorem 3. *If Φ is consistent, closed, and normal, then Φ is satisfiable.*

The proof of Theorem 3 is quite technical, and involves reasoning about the dependencies between digraphs of partial functions and the digraphs of their transitive closures. For details, please see Sect. A.1.

Viewed from a high level, our decision procedure first applies the transformation of Theorem 2, and then repeatedly searches for an applicable IR, applies it (i.e. adds a consequent to the set), and recurses. The recursion is necessary for those IRs that *branch*, i.e. have multiple consequents. 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, by Theorem 3, it may correctly return *satisfiable*. For a formal presentation of the decision procedure, see Appendix B. We note that our decision procedure is guaranteed to terminate because none of the IRs introduce new terms.

6.1 An Extension

In order to handle program assignments that mutate the links in the heap, i.e. modify f, we must extend our logic and decision procedure to support simultaneous reference to f and f', which respectively 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. Lines 5 and 6 of the HMP of Fig. 1 are examples of such assignments. The semantic relationship between f and f' can be expressed using the well-known update operator:[7]

$$\Theta^e(f') = \mathsf{update}(\Theta^e(f), \Theta^e(\tau_1), \Theta^e(\tau_2)) \tag{2}$$

Rather than support update as an interpreted second order function symbol in the logic, we add inference rules that implicitly enforce the constraint (2). For each of the eight IRs of Fig. 3 that mention f, we add an analogous IR with f replaced with f'; these enforce analogous constraints between f'^*, f', and $=$ as are enforced by the unmodified IRs of Fig. 3 between f^*, f, and $=$. Furthermore, to enforce the constraint (2), the seven IRs of Fig. 4 are also included. The IRs introduce a fresh variable w that is forced to be equal to $f(\tau_1)$. This allows us to state that $\Theta^e(f) = \mathsf{update}(\Theta^e(f'), \Theta^e(\tau_1), \Theta^e(w))$, and hence the symmetry between the IRs UPDFUNC1 and UPDFUNC2, between UPDTRANS1 and UPDTRANS2, and between UPDTRANS3 and UPDTRANS4. Note that these IRs can introduce new terms, however, given a normal set of literals, the number of new terms is bounded. This implies that the extended decision procedure always terminates.

Theorem 4. *The inference rules of Fig. 4 are sound.*

The proof of this theorem is provided in Appendix A. We have yet to flesh out the details of a proof of a conjecture analogous to Theorem 3 stating that this extended

[7] 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 $\lambda x.(\text{if } x = a \text{ then } b \text{ else } g(x))$.

$$\frac{}{\begin{array}{l} f'(\tau_1)=\tau_2 \\ f(\tau_1)=w \end{array}}\ \textsc{Update}$$

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

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

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

Fig. 4. The update inference rules, which are used to extend our logic to support a second function symbol f', with the implicit constraint $f' = \mathsf{update}(f, \tau_1, \tau_2)$, where τ_1 and τ_2 are fixed but arbitrary terms, and w is a fresh variable used to capture $f(\tau_1)$. Note that the rule UPDATE can introduce literals that violate normalcy (Def. 1) in the case that τ_1 or τ_2 are not variables. However, this can be remedied by the addition of a new variable and equality literal for each sub-term of τ_1 and τ_2.

set of IRs is complete. However, we have empirical support for this conjecture: in conducting our experiments of Sect. 7, we never found any property violations caused by the extended decision procedure erroneously concluding that a set of literals was satisfiable. Of course, not having such a theorem *does not* compromise the soundness of verification by predicate abstraction.

7 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 Pentium 4 2.6 GHz machine. The safety properties we checked (when applicable) at the end of the HMP are:

- *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.
- *sorted* (SO) – list is a sorted linked list, i.e. each node's data field is less than or equal to its successor's.
- *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.

program	property	CFG edges	preds	time (sec)	DP calls
LIST-REVERSE	NL	6	8	0.1	184
LIST-ADD	NL ∧ AC ∧ IN	7	8	0.1	66
ND-INSERT	NL ∧ AC ∧ IN	5	13	0.5	259
ND-REMOVE	NL ∧ AC ∧ RE	5	12	0.9	386
ZIP	NL ∧ AC	20	22	17.8	9153
SORTED-ZIP	NL ∧ SO ∧ IN	28	22	23.4	14251
SORTED-INSERT	NL ∧ AC ∧ SO	10	20	14.2	5990
BUBBLE-SORT	NL ∧ AC	21	18	11.4	3444
BUBBLE-SORT	NL ∧ AC ∧ SO	21	24	119.5	31446

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 average execution time over five runs to prove the properties; "DP calls" is the number of decision procedure queries. The largest memory usage for all these examples was 125 MB.

Often, the properties one is interested in verifying for HMPs involve universal quantification over the heap nodes. For example, to assert the property NL, we must express that for all nodes t, if t is reachable from *head* initially, then t is also reachable from *head* (or some other node) at the end of the program. Since our logic doesn't support quantification, we use the trick of introducing a Skolem constant t [15, 2] to represent a universally quantified variable. Here, t is a new node variable that is initially assumed to satisfy the antecedent of our property, and is otherwise unmodified by the program. For the example program of Fig. 1, we can express NL by conjoining $\neg t = \mathsf{nil} \wedge f^*(head, t)$ to the **assume** statement on line 2, and conjoining $f^*(head, t)$ to the assertion on line 12. Since t can be any non-nil node reachable from *head*, if the assertion is never violated, we have proven NL.

Our example programs are the following:

LIST-REVERSE – a classical HMP example that performs in-place reversal of a linked list.

LIST-ADD – a linked list is traversed, and the end of the list is reached. Then, a node is added to the end of the list.

ND-INSERT – pseudocode for this example is given in Fig. 1.

ND-REMOVE – similar to ND-INSERT, except that instead of inserting a node, a node is nondeterministically chosen and removed from the list.

ZIP – zips two linked lists, shuffling the elements of both list into one. Then, the tail of the longer list is appended to the resulting list. This example is taken from a paper by Jensen et al. [20].

SORTED-ZIP – joins the elements of two sorted lists into one, also sorted. Here the data elements are simply booleans, so "sorted" means that all nodes with false fields come before nodes with true fields.

SORTED-INSERT – inserts a node into a sorted linked list so that sortedness is preserved. This is a modification of the example from a technical report by Lahiri and Qadeer [23].[8]

BUBBLE-SORT – The bubble sort example sorts elements of a linked list using the bubble sort algorithm. It is taken from a paper by Balaban et al. [2]. The data fields on which we sort are again booleans.

Appendix C provides pseudocode and lists the required predicates for these examples.

As Table 1 shows, we were successful in verifying interesting properties of many examples in reasonable amounts of time. Of special note is our verification of sortedness for BUBBLE-SORT. This example is from Balaban et al. [2]; because of the BDD blow-up inherent in their decision procedure, their tool spaced out for the small model bound necessary for sound verification [1]. In contrast, our trading of space for time appears to be quite advantageous here.

A published running time of TVLA on the bubble sort example [24] is over twice as slow as us, but they are using a slower machine. The recent experimental results of Manevich et al. [27] 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 (they also verify cyclicity for two examples). We are verifying more complicated properties, for instance SO. Very recently, Loginov et al. [26] have used TVLA to fully automatically verify the bubblesort example.

For the two examples in common with Lahiri and Qadeer [23],[9] LIST-REVERSE and SORTED-INSERT, we are significantly faster at verifying the same properties, with respective speed-ups of roughly 3 and 2 orders of magnitude. It should be noted, however, that we used a slightly faster machine, and also that for SORTED-INSERT, our data fields are merely booleans, while theirs are the full integers.

8 Future Work and Conclusions

Despite the fact that this work is in its early stages, our experiments demonstrate its effectiveness for verification of heap-manipulating programs. There are many directions for future research, which are outlined here.

We have identified the following issues related to the expressiveness of the simple transitive closure logic presented in this paper:

- This paper only supports a single link function f, yet clearly many heap-manipulating programs involve multiple link fields.
- We have found that even minimal support for universally quantified variables (as in the logic of Balaban et al. [2]) would allow expression of common heap structure attributes. For example, the current logic cannot assert that two terms x and y point

[8] To simplify things, they require that the input list starts with a dummy element whose data field value has to be less than all possible values of that data field. We don't have such requirements in our example, which makes it slightly more complicated.

[9] We were unable to run our tool on four of Lahiri and Qadeer's [23] examples because we have yet to implement support for data field mutations.

to disjoint linked lists; a single universally quantified variable would allow for this property (see Nelson [30, page 22]). We found that capturing disjointedness is necessary for verifying that LIST-REVERSE always produces an acyclic list; hence we were unable to verify this property.

– Though interesting properties of circular linked lists, e.g. "points to a circularly linked list", can be expressed in our logic, we have found that our logic cannot capture shape invariants strong enough to *prove* such properties with predicate abstraction. Hence, none of our experimental programs of Sect. 7 involve circularly linked lists.

  The problem relates to characterizing node ordering in circularly linked lists. Suppose x, y, and z are nodes in such a list; it typically is necessary to express that y does or does not "come between" x and z in the list; our current logic cannot express this. Nelson [31] and Manevich et al. [27] have previously recognized the importance of such properties. We are currently investigating the addition of a predicate that expresses *comes between* to our logic and have had encouraging preliminary results on HMPs with circular lists.

We believe that our decision procedure can be enhanced to handle each of these three cases. 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. Our approach appears to be very promising, despite the fact that we have yet to harness the recent innovations in these areas.

Acknowledgement

We acknowledge our mutual supervisor Alan J. Hu for his support during this project and Ittai Balaban, Shuvendu Lahiri, and Shaz Qadeer for answering our questions; we also thank Shaz for suggesting this research problem to us.

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. T. Ball, A. Podelski, and S.K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.

5. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming (ESOP)*, 1999.
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)*, 2006. To appear.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.
8. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Conf. on Computer Aided Verification (CAV)*, pages 78 – 92, 2002.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symp. on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
11. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 310–323, 2003.
12. S. Das and D. L. Dill. Successive approximation of abstract transition relations,. In *IEEE Symp. on Logic in Computer Science (LICS)*, 2001.
13. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2002.
14. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Conf. on Computer Aided Verification (CAV)*, 1999.
15. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Principles of Programming Languages (POPL)*, pages 191–202, 2002.
16. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conf. on Computer Aided Verification (CAV)*, 1997.
17. D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Principles of Programming Languages (POPL)*, pages 58–70, 2002.
19. 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.
20. J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 226–236, 1997.
21. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
22. N. Klarlund and M. I. Schwartzbach. Graph types. In *Symp. on Principles of Programming Languages (POPL)*, pages 196–205, 1993.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists, 2005. Microsoft Research Tech Report MSR-TR-2005-97.
24. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Intl. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
25. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS'00)*, pages 280–301, 2000.
26. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *Conf. on Computer Aided Verification (CAV)*, pages 519–533, 2005.

27. 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.
28. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Conf. on Computer Aided Verification (CAV)*, pages 476–490, 2005.
29. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.
30. G. Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1979.
31. G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages (POPL)*, pages 38–47, 1983.
32. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Programming Languages and Systems*, 24(3):217–298, 2002.
33. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.

A Proofs

Theorem 1. *The inference rules of Fig. 3 are sound.*

Proof. The rules IDENT, REFLEX, TRANS1, and TRANS2 are clearly sound. For the rest of the proof, in a slight abuse of notation, we will identify terms, literals, and f with their interpretation by some fixed Θ^e. For FUNC, if $y = f(x)$ and $z = f^n(x)$ for some $n \geq 0$, then in the case $n = 0$ we find $x = z$, and in the case $n \geq 1$ we have $z = f^{n-1}(y)$ and hence $f^*(z,y)$ holds. For CYCLE_k, $k \geq 1$, suppose all the antecedents hold; then $y = f^n(x_1)$ for some $n \geq 0$ and thus $y = x_{1+(n \bmod k)}$. For SCC, suppose all the antecedents hold. In the case $x = y$, then one of the consequents holds trivially. In the case $x \neq y$, then x is on a cycle of f, hence $f*(x,z)$ implies $f*(z,x)$. For TOTAL, suppose all the antecedents hold. Then $y = f^n(x)$ and $z = f^m(x)$ for some $n,m \geq 0$. Now if $n \geq m$, then $y = f^{n-m}(z)$, otherwise if $n < m$, then $z = f^{m-n}(y)$. For SHARE, suppose all the antecedents hold, and suppose $x \neq y$. From the third and fourth antecedents, x and y lie on the same cycle of f, and it follows from the first antecedent that z is also on this cycle. If we restrict the domain of f to be this cycle, f must be a permutation, which contradicts $f(x) = f(y)$.

Theorem 2. *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. First, if there exists no contradiction in Φ, we can clearly remove all boolean literals. Thus, without loss of generality, we assume that Φ has no boolean literals. 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 $f(v_i)$ (where $v_i \in V$) in Φ_0, create a fresh variable v_{fresh}, replace all occurrences of $f(v_i)$ in Φ_0 with v_{fresh}, and add the literal $f(v_i) = v_{fresh}$ to Φ_1. Once we have no terms of the form $f(v_i)$ in Φ_0, let

$$\Phi_2 \;=\; \Phi_0 \cup \Phi_1 \cup \{v_i = v_i \mid v_i \in \mathsf{Vars}(\Phi_0 \cup \Phi_1)\}$$

Clearly Φ_2 satisfies conditions 1-4 of Def. 1, is satisfiable if and only if Φ is, and is constructed in polynomial time; it remains to remove data literals from Φ_2. For each $d \in D$, let

$$\Phi_2^d \;=\; \left\{\neg v_i = v_j \mid d(v_i) \in \Phi_2 \wedge \neg d(v_j) \in \Phi_2\right\}$$

Then let

$$\Phi' = \Phi_2 \cup \bigcup_{d \in D} \Phi_2^d$$

It is easy to see that Φ' is satisfiable iff Φ_2 is; furthermore the size of Φ' is at most quadratic in that of Φ_2.

Theorem 4. *The inference rules of Fig. 4 are sound.*

Proof. We use the same abuse of notation used in the proof of Theorem 1. UPDATE is sound since w is a fresh variable. UPDFUNC1 and UPDFUNC2 clearly respect the facts that $f' = \mathsf{update}(f, \tau_1, \tau_2)$ and $f = \mathsf{update}(f', \tau_1, w)$. For UPDTRANS1, suppose

$y = f^n(x)$ for some $n \geq 0$. We case split on whether or not $\tau_1 = f^m(x)$ for some $m < n$. If so, then $y = f^{n-m}(\tau_1) = f^{n-m-1}(w)$, where $n-m-1 \geq 0$, thus $f^*(w,y)$. If $\tau_1 \neq f^m(x)$ for all m such that $0 \leq m < n$, then $f'^j(x) = f^j(x)$ for all j such that $0 \leq j \leq n$, hence $f'^*(x,y)$. UPDTRANS2 is analogous to UPDTRANS1. For UPDTRANS3, suppose that $\tau_1 = f^n(x)$ and $y = f'^m(x)$ for some n and m, and let our choices of n and m be minimal. Now if $m \leq n$ we clearly have $y = f^m(x)$. Otherwise, if $m > n$, then $y = f'^{m-n}(\tau_1)$. UPDTRANS4 is analogous to UPDTRANS3.

A.1 Proof of Theorem 3

In order to prove Theorem 3, we will demonstrate how, given a consistent, closed, and normal set of literals Φ (let us call such a Φ a *CCN set*), one can construct a heap structure H such that $H \vDash \phi$ for all $\phi \in \Phi$. Since Φ does not contain data or boolean literals, defining H boils down to finding a function $f : N \to N$ for some set N (the nodes), as well as an interpretation $I : \mathsf{Vars}(\Phi) \to N$ that is consistent will all literals of Φ. We will take N to simply be $\mathsf{Vars}(\Phi)$, and I will be the identity function. Because Φ is normal, its literals of the form $f(v_i) = v_j$ define a partial function $f_\Phi : \mathsf{Vars}(\Phi) \to \mathsf{Vars}(\Phi)$; the total function f must clearly be an extension of f_Φ.[10] Let $f^*_\Phi \subseteq \mathsf{Vars}(\Phi) \times \mathsf{Vars}(\Phi)$ be such that $(v_i, v_j) \in f^*_\Phi$ iff $f^*(v_i, v_j) \in \Phi$. For H to exist, it turns out that the digraph $G = (\mathsf{Vars}(\Phi), f^*_\Phi)$ must be what we call a *basin* graph, and furthermore G must be compatible (in some sense defined below) with f_Φ.

This section starts by defining basin graphs and compatibility, and proving the key Lemma 1. All this is done without mention of our logic. Then we state and prove Lemma 5, which explains the connection between these notions and CCN sets of literals. Lemmas 1 and 5 are the required ingredients to prove Theorem 3, which concludes this section.

Definition 2 (basin graph) *A* nontrival SCC[11] *is a SCC involving at least* 2 *nodes. A* maximal SCC *is a SCC such that adding any other node will not also be an SCC. A* basin *in a digraph (V,E) is a maximal SCC S such that $(S \times (V \setminus S)) \cap E = \emptyset$. A digraph (V,E) is called a* basin graph *if*

1. *E is reflexive and transitive, and*
2. *All nontrival maximal SCCs of (V,E) are basins, and*
3. *For any $r,v,u \in V$ such that $(r,u) \in E$ and $(r,v) \in E$, we have that either $(u,v) \in E$ or $(v,u) \in E$.*

Let us add some intuition to the notion of a basin graph. In a basin graph, it is impossible to "leave" a nontrivial SCC (item 2 of Def. 2). Also, if any two vertices share a common ancestor, then they must have an edge between them (item 3). A basin

[10] Given sets A and B, a *partial function* $p : A \to B$ is a relation such that for each $a \in A$, either $p(a) \in B$ or $p(a)$ is *undefined*. A *total* function (or simply *function*) is a partial function in which all points are defined. A partial function $q : A \to B$ is an *extension* of another partial function $p : A \to B$ if for for all $a \in A$ either $p(a)$ is undefined or $q(a) = p(a)$.

[11] A SCC stands for *strongly connected component*, which is a set of vertices that are pair-wise reachable from each other.

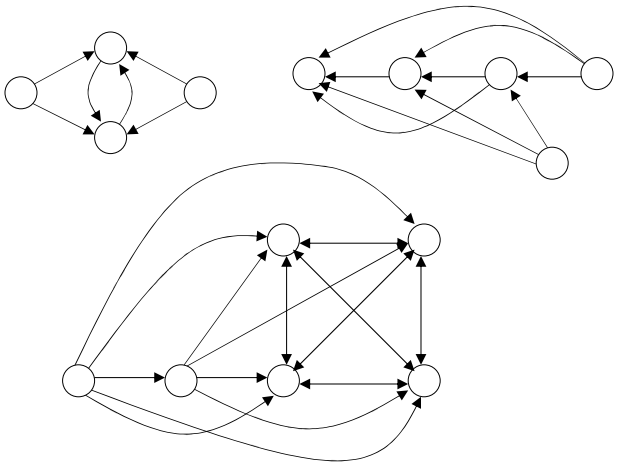

Fig. 5. An example of a basin graph. The self loops on each node are not shown. Note that there are two nontrivial SCCs, one with 2 nodes and one with 4 nodes.

graph may be envisioned as the union of disjoint components. Each component is a basin with zero or more *totally ordered inverted trees incoming* to a basin. By *totally ordered inverted tree* we mean a tree-like structure wherein each vertex has an edge to all of its dependents. By *incoming* we mean that there is an edge from every vertex of the tree to every vertex of the basin. Fig. 5 gives an example of a basin graph.

Let $G = (V,E)$ be a basin graph, and let $p : V \to V$ be a partial function. We wish to determine under what circumstances there exists a total function $f : V \to V$ such that f extends p, and E is the reflexive transitive closure of f. We capture these requirements in the following definition, and then prove that they are necessary and sufficient. For any $v \in V$, let $E(v) = \{u \mid (v,u) \in E\}$.

Definition 3 (compatible) *A basin graph $G = (V,E)$ and a partial function $p : V \to V$ are said to be* compatible *if:*

1. *for all $v \in V$ such that $p(v)$ is defined we have $(v, p(v)) \in E$.*
2. *whenever $p^k(v) = v$ for some $k \geq 1$, the set $\{v, p(v), p^2(v), \ldots, p^{k-1}(v)\}$ is a basin of G.*
3. *for all $v \in V$ such that $p(v)$ is defined we have that $E(v) \setminus \{v\} \subseteq E(p(v))$.*
4. *for any basin B of G and $v \in B$ we have $|\{u \mid p(u) = v\} \cap B| \leq 1$*

Lemma 1. *Let $G = (V,E)$ be a basin graph, and let $p : V \to V$ be a partial function compatible with G. Then there exists a total function $f : V \to V$ such that f is an extension of p, and E is precisely the reflexive and transitive closure of f.*

Proof. By repeatedly applying Lemma 3 below, one can extend p into a total function f that is compatible with G. By Lemma 4 below, E must be the reflexive and transitive closure of f.

Lemma 2. *Let G and p be compatible, and let B be a basin of G. Then $p(B) \subseteq B$, and B is either a cycle of p, or a disjoint union of paths.*

Proof. Suppose there exists $v \in B$ such that $p(v) \notin B$. From item 1 of Def. 3, we have $(v, p(v)) \in E$, contradicting B being a basin. Now suppose B is neither a cycle nor a disjoint union of paths in p. Then either:

- There exists $v \in B$ and $k \geq 1$ such that $p^k(v) = v$ and $\{v, p(v), \ldots, p^{k-1}(v)\} \subsetneq B$. From item 2 of Def. 3, $\{v, p(v), \ldots, p^{k-1}(v)\}$ must be a basin in G, which contradicts B being a basin.
- There exists $v, u, w \in B$ such that $p(v) = p(u) = w$ and $v \neq u$. This contradicts item 4 of Def. 3.

Given a partial function p and some v such that $p(v)$ is undefined, and an element u, let $p^{v \mapsto u}$ be the partial function that is the same as p, except $p^{v \mapsto u}(v) = u$. Thus $p^{v \mapsto u}$ extends p by defining a value for v. Lemma 3 below shows that if a basin graph G and a partial function p are compatible, and $p(v)$ is undefined, then we can always select u such that $p^{v \mapsto u}$ is also compatible with G. The following definition shows how, given G, p, and v, one may select such a u.

Definition 4 (*pickdef*(G, p, v)) *Given compatible $G = (V, E)$ and p, and a node $v \in V$ such that $p(v)$ is undefined, pickdef(G, p, v) is a node $u \in V$ selected as follows.*[12]

- **If** *v is in a basin B of G, then*
  - **If** *$p \downarrow B$ is a single path, then select u to be the start of that path.*[13]
  - **Else** *let u be the first node in any path of $p \downarrow B$ that does not include v*
- **Else** *select u to be a node in $E(v)$ such that $E(v) \setminus E(u) = \{v\}$.*

Lemma 3. *If G and p are compatible, and $p(v)$ is undefined, then taking $u =$ pickdef(G, p, v) we have that G and $p^{v \mapsto u}$ are compatible.*

Proof. First we argue that *pickdef*(G, p, v) always selects *some* vertex, then we argue taking $u =$ *pickdef*(G, p, v) will always satisfy the lemma statement.

If v is in a basin, then Lemma 2 states that $p \downarrow B$ is either a set of paths or a cycle. Def. 4 accommodates only the case of paths, which is sufficient since $p(v)$ being undefined clearly implies that $p \downarrow B$ cannot be a cycle. Now if v is not in a basin, we must argue that that there exists $u \in E(v)$ such that $E(v) \setminus E(u) = \{v\}$. Since v is not in a basin, there exists some w such that $v \in E(v) \setminus E(w)$. If $\{v\} \subsetneq E(v) \setminus E(w)$, then let r be an element of $E(v) \setminus (E(w) \cup \{v\})$. It follows that $\{v\} \subseteq E(v) \setminus E(r) \subsetneq E(v) \setminus E(w)$, and we may repeat this reduction until the first $\subseteq$ becomes a $=$.

Now we argue that G and $p^{v \mapsto u}$ satisfy the four requirements of Def. 3:

[12] Since u is not necessarily unique, we "select" u rather than "define" u

[13] The notation $p \downarrow B$ is the restriction of p to B, defined as follows. Let B' be the subset of V defined by $B' = B \cup p(B)$. Then $p \downarrow B$ is the partial function of the form $B \to B'$ obtained by restricting p to the domain B.

1. It is easy to see that in all cases, u is such that $(v,u) \in E$.
2. The only way there can exist a cycle in $p^{v \mapsto u}$ that is not present in p is when v is in a basin B of G and $p \downarrow B$ is a single path. In this case, the path obviously includes all nodes of B, and as does the cycle that results in $p^{v \mapsto u}$.
3. Clearly $E(v) \setminus \{v\} \subseteq E(u)$ holds if v is in a basin, since in this case u is selected to be in v's basin. Otherwise, Def. 4 selects u such that $u \in E(v)$ and $E(v) \setminus E(u) = \{v\}$. Since G being transitive and $u \in E(v)$ imply $E(u) \subseteq E(v)$, we have $E(v) \setminus \{v\} = E(u)$, which implies requirement 3.
4. Clearly requirement 4 can only possibly be violated if v is in a basin B of G, in which case u is always selected so that there does not exist $w \in B$ such that $p(w) = u$.

Lemma 4. *Let $G = (V,E)$ be a basin graph that is compatible with a total function f. Then E is the reflexive transitive closure of f.*

Proof. Since G is a basin graph, E is reflexive and transitive. Let F be the reflexive transitive closure of f. Clearly from condition 1 of Def. 3 we have $F \subseteq E$; the remainder of the proof is devoted to proving that $E \subseteq F$. Choose $(v,u) \in E$. If $v = u$ then $(v,u) \in F$, since F is reflexive. Otherwise, consider the sequence $\sigma = (v_0, v_1, v_2, \ldots)$, where $v_i = f^i(v)$ and $v_0 = v$; we must show that u appears in σ. Since V is finite, σ must eventually be periodic, hence we may define integers $0 \leq i < j$ such that j is maximal such that $v_0, \ldots, v_{j-1}$ contains no repeated elements, and i is such that $v_i = v_j$. Thus, from condition 2 of Def. 3, $B = \{v_i, v_{i+1}, \ldots, v_{j-1}\}$ is a basin in G. Now suppose u does not appear in σ. Clearly from condition 1 of Def. 3 and the transitivity of E, we have that $(v_\ell, v_r) \in E$ for all $0 \leq \ell < r$. Then both $(v,u) \in E$ and $(v,v_i) \in E$, which imply either $(v_i,u) \in E$ or $(u,v_i) \in E$, by condition 3 of Def. 2. The first case contradicts B being a basin, thus we assume that $(u,v_i) \in E$; this case reaches a contradiction by iteratively calling upon condition 3 of Def. 3. To see this, we note that since $u \neq v_\ell$ for all $\ell \geq 0$, condition 3 of Def. 3 implies that if $u \in E(v_\ell)$ then $u \in E(v_{\ell+1})$. Since $u \in E(v_0)$, by induction we may conclude that $u \in E(v_\ell)$ for all $\ell \geq 0$, and in particular $u \in E(v_i)$. Again, this contradicts B being a basin. Hence u occurs in σ, and $(v,u) \in F$.

We now state Lemma 5, which formalizes the connection between CCN sets, basin graphs, and compatible partial functions.

Lemma 5. *Let Φ be a CCN set of equality and reachability literals. Then $(\mathsf{Vars}(\Phi), f^*_\Phi)$ is a basin graph that is compatible with f_Φ.*

Proof. Since Φ is closed, no IR is applicable. Clearly if REFLEX and TRANS2 are not applicable, then f^*_Φ must be reflexive and transitive, hence condition 1 of Def. 2 is satisfied. If SCC is not applicable, then condition 2 is satisfied. Finally, if TOTAL is not applicable, then condition 3 is satisfied. Therefore $(\mathsf{Vars}(\Phi), f^*_\Phi)$ is a basin graph.

Now, since TRANS1 is not applicable, condition 1 of Def. 3 holds of $(\mathsf{Vars}(\Phi), f^*_\Phi)$ and f_Φ. Since CYCLE$_k$ can not be applied for any $k \geq 1$, condition 2 of Def. 3 holds. Since FUNC cannot be applied, condition 3 of Def. 3 holds. Finally, since SHARE cannot be applied condition 4 of Def. 3 holds. Thus we also have compatibility between $(\mathsf{Vars}(\Phi), f^*_\Phi)$ and f_Φ.

Theorem 3. *If Φ is a CCN set, then Φ is satisfiable.*

Proof. Let $N = \mathsf{Vars}(\Phi)$. By Lemma 5, $(\mathsf{Vars}(\Phi), f^*_\Phi)$ is a basin graph that is compatible with f_Φ. By Lemma 1, there exists a total function f that extends f_Φ and (N, f^*_Φ) is the reflexive and transitive closure of f. Thus f clearly satisfies all equality and reachability literals of Φ. All inequality literals are between (distinct) variables, hence taking the variable interpretation to be the identity preserves all inequalities. Finally, all unreachability literals are also satisfied by f, since Φ is consistent and the transitive closure of f is contained in f^*_Φ.

B Formalization of Decision Procedure

Armed with Theorem 3, our decision procedure is straightforward. The procedure takes a normal set of literals Φ; this restriction does not loose us any generality thanks to Lemma 6. 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 four lemmas demonstrate the correctness of our algorithm.

Note that the proofs of these lemmas assume that the are literals from the logic of Fig. 2; they do not apply to the extended logic of Sect. 6.1. The proofs of Lemmas 6, 7, and 9 can easily be generalized to deal with the extension. However, we have not yet been able to prove Lemma 8 for the extended decision procedure.

Lemma 6. *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, or a reachability literals involving two variables. In the first case, Φ' is assigned by line 8, and clearly performing the substitution preserves normality. In the second case, Φ' is assigned by line 10 and ϕ is a reachability literal between two variables. Addition of such a literal also preserves normality.

Lemma 7. *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 all applicable IR yielded UNSAT from the recursive calls. The proof thus depends on the soundness of the IRs; this can be confirmed by inspection of Fig. 3.

Lemma 8. *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 reachability literals, and doing variable substitutions. Furthermore, $\hat{\Phi}$ is consistent, closed, and, by Lemma 6, normal. Thus, by Theorem 3, $\hat{\Phi}$ is satisfiable, which implies the satisfiability of Φ also.

Lemma 9. DECIDE*(Φ) 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.

```
 1: function DECIDE(Φ)
 2:     if Φ contains a contradiction then
 3:         return UNSAT
 4:     end if
 5:     if there exists an IR r applicable to Φ then
 6:         for each consequent φ of r do
 7:             if φ is an equality literal of the form v_i = v_j then
 8:                 Φ' := Φ[v_i/v_j]
 9:             else
10:                 Φ' := Φ ∪ {φ}
11:             end if
12:             if DECIDE(Φ') = SAT then
13:                 return SAT
14:             end if
15:         end for
16:         return UNSAT
17:     else
18:         return SAT
19:     end if
20: end function
```

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

C Pseudocode of the Examples

```
 1: procedure LIST-REVERSE(x)
 2:     assume ¬x = nil ∧ f*(x, nil) ∧ f*(x,t) ∧ ¬t = nil ∧ y = nil
 3:     while ¬x = nil do
 4:         temp := f(x);
 5:         f(x) := y;
 6:         y := x;
 7:         x := temp;
 8:     end while
 9:     assert f*(y,t)
10: end procedure
```

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

```
 1: procedure LIST-ADD(head, item)
 2:     assume ¬f*(head, item) ∧ f*(head, nil) ∧ f*(head,t) ∧ f(item) = nil ∧ p = head
 3:     if head = nil then
 4:         head := p;
 5:     else
 6:         while ¬f(p) = nil do
 7:             p := f(p);
 8:         end while
 9:         f(p) := item;
10:     end if
11:     assert f*(head, item) ∧ f*(head, nil) ∧ f*(head,t)
12: end procedure
```

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

```
1:  procedure ND-INSERT(head, item)
2:      assume ¬f*(head, item) ∧ f*(head, nil) ∧ ¬head = nil ∧ f*(head, t) ∧ ¬t = nil ∧ f(item) =
            nil ∧ p = head
3:      while true do
4:          if ND ∨ f(p) = nil then
5:              f(item) := f(p);
6:              f(p) := item;
7:              break
8:          else
9:              p := f(p);
10:         end if
11:     end while
12:     assert f*(head, item) ∧ f*(head, nil) ∧ f*(head, t)
13: end procedure
```

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

```
1:  procedure ND-REMOVE(head)
2:      assume ¬head = nil ∧ f*(head, nil) ∧ f*(head, t) ∧ ¬t = nil ∧ p = head ∧ r = f(head)
3:      while true do
4:          if ND ∨ f(r) = nil then
5:              f(p) := f(r);
6:              break
7:          else
8:              p := r;
9:              r := f(r);
10:         end if
11:     end while
12:     assert f*(head, nil) ∧ (f*(head, t) ⊕ r = t)
13: end procedure
```

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

```
 1: procedure ZIP(x,y)
 2:     assume f*(x,nil) ∧ f*(y,nil) ∧ (f*(x,t) ∨ f*(y,t)) ∧ z=nil ∧ p=nil ∧ temp=nil
 3:     if x=nil then
 4:         temp := x;
 5:         x := y;
 6:         y := temp;
 7:     end if
 8:     while ¬x=nil do
 9:         if z=nil then
10:             z := x;
11:             p := x;
12:         else
13:             f(p) := x;
14:             p := x;
15:         end if
16:         x := f(x);
17:         f(p) := nil;
18:         if ¬y=nil then
19:             temp := x;
20:             x := y;
21:             y := temp;
22:         end if
23:     end while
24:     assert f*(z,nil) ∧ f*(z,t)
25: end procedure
```

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

```
1: procedure SORTED-ZIP(x,y)
2:     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
3:     while ¬x = nil ∧ ¬y = nil do
4:         if d(x) < d(y) then
5:             if ¬temp = nil then
6:                 f(temp) := x;
7:             else
8:                 merge := x;
9:             end if
10:            temp := x;
11:            x := f(x);
12:        else
13:            if ¬temp = nil then
14:                f(temp) := y;
15:            else
16:                merge := y;
17:            end if
18:            temp := y;
19:            y := f(y);
20:        end if
21:    end while
22:    if ¬x = nil then
23:        if merge = nil then
24:            merge := x;
25:        else
26:            f(temp) := x;
27:        end if
28:    end if
29:    if ¬y = nil then
30:        if merge = nil then
31:            merge := y;
32:        else
33:            f(temp) := y;
34:        end if
35:    end if
36:    assert  f*(merge,t) ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil)
37: end procedure
```

Fig. 12. SORTED-ZIP. Necessary 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 $\neg(d(x) \wedge \neg d(y))$.

```
1:  procedure SORTED-INSERT(head, item)
2:      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)
3:      while ¬succ = nil ∧ d(item) > d(succ) do
4:          curr := succ;
5:          succ := f(curr);
6:      end while
7:      if d(head) > d(item) then
8:          f(item) := head;
9:          head := item;
10:     else
11:         f(item) := succ;
12:         f(curr) := item;
13:     end if
14:     assert f*(head, nil) ∧ f*(head, t) ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil)
15: end procedure
```

Fig. 13. SORTED-INSERT. Necessary 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 $\neg(d(x) \wedge \neg d(y))$.

```
 1: procedure BUBBLE-SORT(x)
 2:     assume f*(x, nil) ∧ f*(x,t) ∧ ¬t = nil ∧ y = x ∧ yn = f(y) ∧ prev = nil ∧ last = nil
 3:     while ¬last = f(x) do
 4:         while ¬yn = last do
 5:             if d(y) > d(yn) then
 6:                 f(y) := f(yn);
 7:                 f(yn) := y;
 8:                 if prev = nil then
 9:                     x := yn;
10:                 else
11:                     f(prev) := yn;
12:                 end if
13:                 prev := yn;
14:                 yn := f(y);
15:             else
16:                 prev := y;
17:                 y := yn;
18:                 yn := f(yn);
19:             end if
20:         end while
21:         prev := nil;
22:         last := y;
23:         y := x;
24:         yn := f(x);
25:     end while
26:     assert f*(x, nil) ∧ f*(x,t) ∧ (d(t) ≤ d(f(t)) ∨ f(t) = nil)
27: end procedure
```

Fig. 14. BUBBLE-SORT. Necessary 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 $\neg(d(x) \wedge \neg d(y))$.