A Logic and Decision Procedure for Verification of Heap-Manipulating Programs

by

Zvonimir Rakamarić

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

August 2006

Abstract

Heap-manipulating programs (HMPs), which manipulate unbounded linked data structures via pointers, are a major frontier for formal verification of software. Formal verification is the process of proving (or disproving) the correctness of a system with respect to some kind of formal specification or property. The primary contributions of this thesis are the definition of a simple transitive closure logic tailored for formal verification of HMPs, and an efficient decision procedure for this logic. To assess the effectiveness of the proposed approach, we develop an HMP verification framework, which uses our fast implementation of the decision procedure to verify a number of HMP examples. Experimental examples (including three small container functions from the Linux kernel) demonstrate that the logic is practically useful and expressive enough to prove many interesting heap properties. In addition, the decision procedure provides a substantial time and space advantage over previous approaches.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

We are witnessing how software systems are becoming a part of every segment of human life. Nowadays, software is often used to control many "mission-critical" tasks, where an error in software could cause very serious consequences. Such software systems have to meet a high reliability bar in order to prevent disasters from happening.

Back in 1972, Dijkstra realized that testing is not the solution for achieving error-free software — "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72]. In theory, testing could be used to show that a software system doesn't have any errors by exhaustively traversing all possible execution paths. In practice, however, it is usually impossible to test software for every possible execution because of the vast state space. As opposed to testing, formal verification is the process of proving (or disproving) the correctness of a system with respect to some kind of formal specification or property. Formal verification is a now well-accepted method for hardware verification, and it could be used to achieve highly reliable and error-free software as well. Therefore, there has been a lot of research recently in employing successful formal verification techniques from the hardware world on software.

Much of the success of applying formal verification to hardware comes from using

model checking [CES86]. Model checking is a method for formally verifying finite-state systems by exhaustively traversing their state space. Because the state space grows exponentially with the size of a system, model checkers face a blow up of the state space, commonly known as the state explosion problem. Different techniques are used to overcome state explosion, abstraction being one of the most successful ones. Abstraction is the process of reducing the complexity of a system by removing information which is not relevant for a particular task. It is often used in the verification of large, complex systems.

Software model checking has recently emerged as a vibrant area of formal verification research. Because of the state explosion problem, much of the success of applying model checking to software has come from using *predicate abstraction* [GS97, DDP99, BMMR01, HJMS02].

Predicate abstraction is an abstraction technique that employs a finite set of predicates in some logic. The predicates are assertions about states of the concrete system; the concrete system is usually infinite-state. In the abstraction, each predicate is represented with a boolean variable. Therefore, predicate abstraction is used to transform a typically infinite-state system, such as software, to a much smaller, finite-state, manageable over-approximation. The finite-state over-approximation is defined as a transition system over boolean variables which represent predicates, and can in turn be model checked using standard model checking techniques. The smaller, reduced system tries to preserve some of the properties of the original system that are necessary for proving system correctness. As already mentioned, predicate abstraction usually 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 very important, since most predicate abstraction approaches make numerous queries to the decision procedure.

An important class of programs are *heap-manipulating programs* (HMPs): programs that access and modify linked data structures consisting of an unbounded number of *heap nodes*. HMPs access the heap nodes through a finite number of pointers (which we

call *node variables*) and by following pointer fields between nodes. Since the number of nodes in the heap is unbounded, HMPs are infinite-state systems, so one cannot directly apply finite-state model checking to this problem without using abstraction.

To apply predicate abstraction to HMPs and assert many interesting correctness properties, one must be able to express the fact that a node is reachable from some other node by following a number of links — pointers in the data structure. For instance, to express a property that a node belongs to a particular singly-linked list, one must be able to say that the node is reachable from the head of that list. This concept is called unbounded *reachability* (a.k.a. *transitive closure*) between nodes. Several researchers have previously identified the importance of transitive closure for HMPs [Nel79, Nel83, BRS99, IRR+04a, BPZ05, LQ06, LAIR+05]. Unfortunately, adding support for transitive closure to even simple logics often yields undecidability [IRR+04a], and therefore, one must be careful when defining such a logic.[1]

Verification of HMPs has recently regained the focus of the software verification community. Many of the published approaches are based on predicate abstraction [BPZ05, LQ06, DN03, MYRS05], and thus require a transitive closure logic and a decision procedure. Furthermore, other HMP verification approaches and tools [WKL+06, YRS04] could also take advantage of a transitive closure logic. Therefore, defining such a logic and a fast decision procedure with a good implementation would be an enormous benefit. However, there exist only a handful of implemented decision procedures for logics that could be used in the verification of HMPs. This thesis defines such a logic and accompanying decision procedure, and assesses their usability and performance. In order to accomplish that, the thesis sets up and uses an HMP verification framework. The framework employs predicate abstraction, the logic, and the decision procedure to verify HMPs. The verification problem it solves can be stated as follows: given an HMP, determine whether it is the case that all executions that satisfy all initial assumptions also satisfy all assertions in the program.

[1] In the heap analysis community, the term "transitive closure logic" refers to many different logics that include transitive closure, and not strictly to the first-order transitive closure logic (FO+TC).

1.2 Contributions

The first contribution is a decidable simple transitive closure logic. This logic is a fragment of the decidable logics containing transitive closure, but we show (through many nontrivial experiments) that it is still expressive enough to verify properties of interest for HMPs using predicate abstraction. Important properties of data structures like singly-linked lists, doubly-linked lists, and cyclic lists can be easily defined in this logic.

The second and most important contribution is an efficient decision procedure for this logic[2]. Most other decision procedures for similar logics employ a small model theorem and enumerate a huge number of heap structures up to some bound. Instead of using a small-model theorem and enumerating a super-factorial number of possible models (e.g., [BRS99, BPZ05, LQ06]), our decision procedure is based on inference rules. We show that this procedure, though worst case exponential-time (a proof that satisfiability is NP-hard even for a small fragment of our logic can be found in [RBH06]), solves very quickly the vast majority of queries sent to it during predicate abstraction. The result is an approach that can have large time and memory savings over decision procedures that enumerate all models.

Overall, we have been able to verify with a short runtime and an insignificant memory consumption a large variety of interesting HMPs, such as three small container functions from the Linux kernel.

1.3 Organization of the Thesis

The material presented in this thesis is based on previously published work [BR06, RBH06] done with collaborators. The initial idea of the simple transitive closure logic and its decision procedure belongs to Jesse Bingham. My main contributions are extensions that made the logic practically usable and the fast decision procedure.

The thesis is organized as follows: Chapter 2 provides background information. It

[2]The implementation of the decision procedure, called *straclos*, which stands for Simple TRAnsitive CLOSure logic, can be downloaded from **http://www.cs.ubc.ca/~zrakamar**.

introduces heap-manipulating programs in Section 2.1, and predicate abstraction and the verification framework in Sections 2.2 and 2.3, respectively. Section 2.4 summarizes other work on verification of HMPs. Chapters 3 and 4 respectively define the transitive closure logic and its decision procedure. Chapter 5 presents experimental results, while Chapter 6 concludes and suggests some possible extensions to the logic and decision procedure. The appendices A and B provide additional, more theoretical aspects of the approach — proofs of the supporting theorems and additional details regarding the decision procedure. Proofs of the theorems are largely due to my collaborator Jesse Bingham. The thesis includes the proofs for the sake of completeness of the presentation. Appendix C gives pseudocode for the example programs and the sets of predicates needed for their verification.

Chapter 2

Background

2.1 Heap-Manipulating Programs

Heap-manipulating programs are an important and widespread class of programs. Any program that accesses and modifies linked heap[1] data structures consisting of an unbounded number of *heap nodes* falls into this class. A heap node is a dynamically allocated chunk of memory. It usually contains pointer fields and data fields of different types. Pointer fields are links to other heap nodes. HMPs access the heap nodes through a finite number of pointer variables and by following pointer fields between nodes.

In our framework, the heap consists of an unbounded number of uniform heap nodes. Uniformity of the nodes means that they all contain the same pointer and data fields. Data fields are booleans. Our HMPs can have a finite number of global node variables (pointers) and a finite number of global boolean variables. Variables or data fields of any other finite type can be modeled (or encoded as) booleans. Note that the form of HMPs that we support doesn't preclude generality, as any HMP can be translated to our form. The framework currently doesn't support procedure/function calls. Figure 2.1 formally defines the supported HMP syntax.

[1] In the thesis, the term heap always refers to a memory area used for dynamic memory allocation, and not to the tree-based data structure where the value of each node is less than or equal to the value of its parent, as used to implement, for example, sorting algorithms and priority queues.

$$
\begin{array}{rcl}
x, y, z & \in & \mathit{NodeVariables} \\
b & \in & \mathit{BooleanVariables} \\
f, g, h & \in & \mathit{PointerFields} \\
d & \in & \mathit{DataFields} \\
\mathit{HMP} & ::= & \{\mathit{Statement}\ ;\ \}^* \\
\mathit{Statement} & ::= & \textbf{while}\ (\mathit{BoolExp})\ \textbf{do}\ \mathit{HMP}\ \textbf{end while} \\
 & | & \textbf{if}\ (\mathit{BoolExp})\ \textbf{then}\ \mathit{HMP}\ \textbf{else}\ \mathit{HMP}\ \textbf{end if} \\
 & | & \mathit{Assignment} \\
 & | & \textbf{assert}\ \psi \\
 & | & \textbf{assume}\ \psi \\
 & | & \textbf{break} \\
 & | & \textbf{nop} \\
\mathit{Assignment} & ::= & \mathit{Term} := \mathit{Term} \\
 & | & \mathit{Term} := \mathsf{nil} \\
 & | & b := \mathit{BoolExp} \\
 & | & d(\mathit{Term}) := \mathit{BoolExp} \\
\mathit{Term} & ::= & x \\
 & | & f(\mathit{Term}) \\
\mathit{BoolExp} & ::= & b \\
 & | & \mathsf{true} \\
 & | & \mathsf{false} \\
 & | & \mathit{ND} \\
 & | & \mathit{Term} = \mathit{Term} \\
 & | & d(\mathit{Term}) \\
 & | & \neg \mathit{BoolExp} \\
 & | & (\mathit{BoolExp} \wedge \mathit{BoolExp}) \\
 & | & (\mathit{BoolExp} \vee \mathit{BoolExp})
\end{array}
$$

Figure 2.1: The Syntax of a Heap-Manipulating Program. ψ is a simple transitive closure logic formula (Chapter 3). *ND* is a boolean value that is nondeterministically true or false.

```
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
```

Figure 2.2: ND-Insert HMP Example. 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.

Figure 2.2 gives an HMP example called ND-INSERT. 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. We denote by $f(x)$ the node pointed to by a pointer field named f of a node x. The pointer *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 Chapter 3. [2] The fact that nil is reachable from *head* enforces the acyclicity assumption.[3]

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. The node *item* is inserted when either $ND = \text{true}$, or the end of the list is reached (detected by the disjunct $f(p) = \text{nil}$ on line 4). The specification is expressed by the **assert** statement on line 12, and indicates that

[2] In C-like syntax, $f(x)$ would be written as `x->f`. In a modal logic, f would be a next operator, and f^* would be eventuality. The notation we are using is standard in the heap analysis community.

[3] In our logical framework, nil is modeled simply as a node having $f(\text{nil}) = \text{nil}$.

```
1: procedure INIT-CYCLIC(head)
2:     assume  f*(head,t) ∧ f*(f(head),head) ∧ curr=f(head)
                ∧ btwn_f(curr,t,head) ∧ ¬head=nil
3:     d(head) := true;
4:     while ¬curr=head do
5:         d(curr) := true;
6:         curr := f(curr);
7:     end while
8:     assert f*(head,t) ∧ f*(f(head),head) ∧ d(t) ∧ ¬head=nil
9: end procedure
```

Figure 2.3: Init-Cyclic HMP Example. A program that sets data fields of all nodes in a cyclic list to true.

whenever line 12 is reached, *head* must point to an acyclic list that contains *item*.

Figure 2.3 presents another example called INIT-CYCLIC, and it captures some of the more interesting features the framework supports. 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 true. Similarly to $f(x)$, we denote by $d(x)$ the value of a data field named d of a node x. Necessary assumptions are again formalized by the **assume** statement on line 2 of the program. In the **assume** statement, 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 Chapter 3. The fact that *head* is reachable from $f(head)$ enforces the cyclicity assumption. The body of the INIT-CYCLIC procedure first sets the data field of *head* 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.

Figure 2.4 shows a list container procedure from the Linux kernel. It illustrates the need for multiple pointer fields. The procedure takes a node *entry* and removes it from a cyclic doubly-linked list. Each node in the list has a *prev* and a *next* pointer. The body of the procedure is simple; it connects *prev* and *next* pointers of the *entry*'s neighbourhood

```
1: procedure LINUX-LIST-DEL(entry)
2:     p := prev(entry);
3:     n := next(entry);
4:     prev(n) := p;
5:     next(p) := n;
6:     next(entry) := nil;
7:     prev(entry) := nil;
8: end procedure
```

Figure 2.4: Linux-List-Del HMP Example. A Linux kernel function that removes a node from a cyclic doubly-linked list.

nodes and therefore removes *entry* from the list. The assumptions and specifications for this example are very complicated and are given in the Appendix C.

Now that we have introduced heap-manipulating programs, we'll present the basics of the algorithm that our framework uses for their verification.

2.2 Predicate Abstraction

Our approach to verifying heap-manipulating programs is based on *predicate abstraction* [GS97], which is an instance of *abstract interpretation* [CC77]. In the framework of abstract interpretation, a *concrete system* is verified by constructing a finite-state over-approximation called the *abstract system.* In this thesis, the concrete system we are verifying is an HMP. 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 are the *abstract boolean variables* $b_1, \ldots, b_k$. An *abstract state a* will be a vector of truth assignments to the abstract boolean variables $b_1, \ldots, b_k$. The set of *abstract states* $\mathscr{A}$ will then be the set of assignments to the abstract boolean variables. Note that a set of states S can also be represented by its characteristic function, i.e. a logic formula ψ such that

$$s \models \psi \text{ iff } s \in S$$

In the rest of the thesis, we will use the set and formula definitions interchangeably, and it will be clear from the context to which one we are referring.

The concrete and abstract systems are connected with two functions:

(i) The *abstraction function* $\alpha : \mathscr{C} \rightarrow \mathscr{A}$, which maps a concrete state c to the abstract state a, is defined as

$$\alpha(c) = \bigwedge_{i=1}^{k} \begin{cases} b_i & \text{iff } c \models \phi_i \\ \neg b_i & \text{otherwise} \end{cases}$$

A set of concrete states C is then abstracted by

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

(ii) The *concretization function* $\gamma : \mathscr{A} \rightarrow \mathscr{C}$, which maps an abstract state a to the set of concrete states it represents, is defined as

$$\gamma(a) = \bigwedge_{i} \begin{cases} \phi_i & \text{iff } a \models b_i \\ \neg \phi_i & \text{iff } a \models \neg b_i \end{cases}$$

Because a is an abstract state, it defines every abstract boolean variable b_i, and therefore either $a \models b_i$ or $a \models \neg b_i$ always holds. A set of abstract states A is then concretized by

$$\gamma(A) = \bigvee_{a \in A} \gamma(a)$$

The abstraction function α and the concretization function γ form a Galois connection, and therefore for any concrete set of states C, the following formula is satisfied:

$$C \subseteq \gamma(\alpha(C))$$

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 infiniteness of the concrete state space arises from the unboundedness of the heap in HMPs.

Given a set of concrete initial states I_C, 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. Predicate abstraction is used to solve this problem by computing an over-approximation $R^\alpha \subseteq \mathscr{A}$ of the set of reachable abstract states such that $\alpha(R) \subseteq R^\alpha$. Verification succeeds if one can prove that the state assertion ψ holds for all members of $\gamma(R^\alpha)$. A key difference in the various approaches to predicate abstraction is how R^α is computed [GS97, DDP99, DD01, FQ02, BPZ05, DN03]. 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. The algorithm that our framework uses to compute R^α is described in the next section.

Since predicate abstraction is an incomplete approach, if it fails to verify the property, this can happen either 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 completing the verification can be a difficult task. Many works have addressed this issue of *predicate discovery* [DD02, BPR02, HJMS02, DN03], which falls under the more general framework of *abstraction refinement* [CGJ+00]. As in recent papers on this topic [BPZ05, LQ06], 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.

2.3 Over-Approximating the Reachable Abstract States

An over-approximation R^α of the set of reachable abstract states is usually computed as a fixpoint, using 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. The abstract post image operator $\mathsf{post}(A)$ is thus the set of abstract states representing concrete states that are concrete successors of those states represented by A. Given $\mathsf{post}(A)$ (or an over-approximation) and the

initial set of concrete states I_C, we compute R^α using the following least fixpoint iteration (or equivalent)

$$\begin{aligned} R_0 &= \alpha(I_C) \\ R_1 &= R_0 \cup \mathsf{post}(R_0) \\ R_2 &= R_1 \cup \mathsf{post}(R_1) \\ &\vdots \end{aligned}$$

which can be expressed using the least fixpoint operator μ as a formula

$$R^\alpha = \mu X.\, \alpha(I_C) \cup X \cup \mathsf{post}(X)$$

There exist a number of algorithms for computing post, with tradeoffs between how precisely post is computed and the number of queries to the decision procedure. The naive algorithm is straightforward. Since post distributes over disjunction,[4] computing $\mathsf{post}(A)$ is reducible to computing $\mathsf{post}(\rho)$ for each disjunct ρ in some disjunctive normal form decomposition of A. A disjunct ρ is a conjunction of possibly negated abstract boolean variables. By using a BDD [Bry86] to represent A, we can easily obtain such a decomposition. The naive algorithm cycles through all 2^k abstract states a, and checks if $a \in \mathsf{post}(\rho)$; the abstract post image operator $\mathsf{post}(\rho)$ 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:

$$\gamma(\rho) \wedge \mathsf{wp}(\gamma(a)) \tag{2.1}$$

where γ is the concretization function defined in the previous section, and wp is the *weakest precondition* operator [Gri81, Dij76]. The weakest precondition operator wp is a syntactic transformation on logic formulas that depends on the program statement under consideration [Gri81, Dij76]. 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.[5] Our approach applies wp at the granularity of individual program statements when performing predicate abstraction.

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

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

Because our predicate abstraction framework is mainly developed for testing the decision procedure, we choose to implement a simple, precise predicate abstraction algorithm. Specifically, the framework uses the described naive algorithm with several straightforward improvements by Das et al. [DDP99] that preserve the precision of the naive algorithm. First, if (2.1) 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 A representing the set of all abstract states a that won't yield such a contradiction. Second, rather than enumerating all $a \in A$, we do recursive case-splitting on the abstract variables, which allows for pruning of large portions of A. For example, let $\sigma = b_1$ be the disjunct containing only the positive occurrence of the abstract boolean variable b_1. This disjunct represents all abstract states having b_1 true. Then if $\gamma(\rho) \wedge \mathsf{wp}(\gamma(\sigma))$ 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.

2.4 Related Work

There is an extensive literature on verification of HMPs describing many different approaches. This section concentrates on the work similar to ours, and on techniques that could benefit from the logic and the decision procedure presented in this thesis. Similarly to Lahiri and Qadeer [LQ06], the described related work will be roughly divided, according to the technique(s) it is based on, in the following categories: shape analysis, predicate abstraction, logics, first-order axiomatization of reachability. These categories often overlap. For example, this thesis spans the predicate abstraction and logic based categories.

Shape Analysis. The most well-known shape analysis tool is the *Three Valued Logic Analyzer* or TVLA [LAS00]. TVLA 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 in the case that the generated transformers are not strong enough. We cannot handle all heap structures that TVLA can. On the other hand, 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, and there has been some work done to make TLVA more precise [YRS04]. However, the described improvement hasn't been tested because it requires a transitive closure logic decision procedure similar to ours, which the authors didn't have. Now, TVLA could take advantage of the decision procedure described in this thesis and make its analysis more precise.

All HMP verification approaches described in the literature require some amount of manual user's effort during the verification (e.g. providing the loop invariants, finding the right set of predicates, etc.). Recently, Loginov et al. [LRS05] presented an interesting combination of abstraction refinement using machine learning techniques, and used TVLA to fully automatically (i.e. no manual effort required) verify some HMP examples. Their technique might be a good starting point for extending this thesis with the abstraction refinement algorithm, which would even more automatize our approach.

Predicate abstraction. TVLA is also used as the underlying engine for the approach by Manevich et al. [MYRS05]. In the approach, the authors 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 cyclic singly-linked lists. The approach described in this thesis is not that limited, and can also easily handle doubly-linked lists. The defined abstraction enables them to verify a number of HMPs, though the properties they verify tend to be simpler than ours (see Chapter 5).

Balaban et al. [BPZ05] present a predicate abstraction based approach for verification of HMPs that is similar to ours. 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. We, on the other hand, use our saturation-based decision procedure which substantially improves memory consumption and computation time. Though we do not consider liveness in the thesis, it is likely that the technique of Kesten and Pnueli [KP00] for establishing termination (employed by Balaban et al.) is also compatible with our work.

Another approach based on predicate abstraction and model checking is proposed by Dams and Namjoshi [DN03]. They abstract a program by iteratively calculating weakest preconditions of shape predicates, and are able to handle second-order shape properties such 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.

Logics. The pioneer of logic-based tools is the *Pointer Assertion Logic Engine* (PALE) [MS01]. PALE specifies heap structures using graph types [KS93], which are tree-shaped data structures augmented with extra pointers that may point anywhere in the tree. The authors show that many common heap structures can be defined that way, some of which we cannot express, such as trees. PALE employs a well-known decision procedure for a monadic second-order logic on graph types called MONA [KMS00]. MONA has non-elementary complexity, and therefore, it pays a performance penalty compared to our approach (Chapter 5 gives the initial comparison). Furthermore, loop invariants must be provided by the user.

Inherent locality of the heap is employed in recent work that is based on separation logic [MNCL06, DOY06]. Both approaches utilize symbolic execution of separation logic formulas to infer invariants of heap-manipulating programs. Because there are infinitely many *symbolic heaps*, they have to use different techniques such as abstraction, widening operators, or rewrite rules to reach a fixpoint.

In addition, two new decidable logics for expressing properties of linked data structures have been proposed recently. Ranise and Zarba [RZ05] describe a decidable logic for reasoning about acyclic singly-linked lists with an NP-complete decision problem. They state a small model theorem for the logic, but the design and implementation of a practical

decision procedure is left as an area of future work. Yorsh et al. [YRS+06] define the *Logic of Reachable Patterns* (LRP), a fragment of the first-order logic over graph structures with transitive closure. Reachability in LRP is defined using regular expressions which denote paths in the heap structure reaching a certain pattern. Patterns are quantifier-free first-order formulas over graph structures used to limit the neighborhood of a reachable node. The authors prove that with suitable restrictions on patterns, the satisfiability problem for LRP is decidable. Because LRP operates on general graphs, even with those restrictions it can express complex heap data structures, such as binary trees. Restricted LRP formulas can be decided using MONA [KMS00], or alternatively by directly constructing a tree automaton. There is still no implementation available, and therefore, the authors haven't provided any experiments. The worst case complexity of the satisfiability problem is at least doubly-exponential, but the authors hope to achieve a reasonable performance because formulas that come up in practice are well-structured.

To handle more involved data structures, Wies et al. [WKL+06] introduce *field constraint analysis*, a novel technique for verifying heap structure invariants. The analysis uses decidable logics (in particular monadic second-order logic of trees and its decision procedure MONA) to handle complex data structures originally beyond the scope of these logics, such as skip lists. It generalizes a previous similar approach called *structure simulation* [IRR+04b]. While doing the verification, the analysis makes a number of queries to MONA. We have evaluated our approach on these queries[6], and the initial results are encouraging, that is, we are solving the queries faster than MONA is (see Chapter 5). Therefore, the field constraint analysis could benefit from using our faster decision procedure instead of MONA.

First-order axiomatization of reachability. First-order axiomatization of reachability was first proposed by Nelson [Nel83]. Lahiri and Qadeer [LQ06] 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. Because the

[6]Thanks to Thomas Wies and Viktor Kuncak for sending us the queries their tool generates.

given set of axioms is incomplete, they provide an induction principle that is used to derive additional axioms when necessary. They use UCLID [BLS02] as the underlying inference engine. Lev-Ami et al. [LAIR+05] 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 sacrifice in performance.

McPeak and Necula [MN05] specify heap data structures using *local equality axioms*, first-order axioms that 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 and reason about infinite scalar types without abstracting them with booleans. 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.

Chapter 3

Proposed Logic

In the previous chapter, we introduced HMPs and predicate abstraction. Predicate abstraction employs a set of predicates in a logic that has to be expressive enough to allow useful abstractions. This chapter starts by defining the syntax and semantics of our proposed simple transitive closure logic for predicate abstraction of HMPs. Next, it introduces logic extensions necessary for handling pointer and data field updates. Chapter 4 then presents our decision procedure for the described logic.

3.1 Basic Logic

Our logic assumes finite sets of *node* variables V, *data* variables B, *data function* symbols D, and *pointer function* symbols F. The *term*, *atom*, and *literal* syntactic entities are given in Figure 3.1. 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 *data variable* literals.

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

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

Figure 3.1: Syntax of the Proposed Logic. In the syntax, the symbol $v \in V$, $d \in D$, $b \in B$, and $f \in F$.

that:

- Each node variable symbol $\sigma \in V$ is interpreted as a node $\Theta(\sigma) \in N$. The variables of V model program variables that point to nodes in the data structure.
- Each data variable symbol $\sigma \in B$ is interpreted as a boolean value $\Theta(\sigma) \in \{\mathsf{true}, \mathsf{false}\}$. The variables of B model program variables of boolean type.
- Each data function symbol $\sigma \in D$ is interpreted as a function that maps nodes to booleans $\Theta(\sigma) \in [N \rightarrow \{\mathsf{true}, \mathsf{false}\}]$. Data function symbols D model data fields of nodes.
- Each pointer function symbol $\sigma \in F$ is interpreted as a mapping from nodes to nodes $\Theta(\sigma) \in [N \rightarrow N]$. Pointer function symbols F model pointers from nodes to nodes.

The *size* of H is defined to be $|N|$. Heap structures naturally model linked data structures of nodes, each node having some finite number of pointers to other nodes and some finite number of boolean-valued data fields. Clearly, program variables or node data fields of any finite enumerated type can be encoded using the booleans supported by our logic.

The interpretation function Θ extends to interpret any term, atom, or literal in a straightforward, inductive way. The interpretation of a term $\tau \in V$ is defined above, otherwise, τ has the form $f(\tau')$ for some term τ', and the interpretation is

$$\Theta(\tau) = \Theta(f)(\Theta(\tau'))$$

Atoms are interpreted by Θ as boolean values:

- A data variable atom $b \in B$ is interpreted as defined above.
- A data atom $d(\tau)$ is interpreted as $\Theta(d)(\Theta(\tau))$.
- An equality atom $\tau_1 = \tau_2$ is interpreted as true iff $\Theta(\tau_1) = \Theta(\tau_2)$.
- A reachability atom $f^*(\tau_1, \tau_2)$ is interpreted as true iff there exists some $n \geq 0$ such that $\Theta(f)^n(\Theta(\tau_1)) = \Theta(\tau_2)$.[1]
- 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 $\Theta(\tau_2) = \Theta(f)^{n_0}(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^{m_0}(\Theta(\tau_1))$, $n_0 \leq m_0$, and for all n, m such that $\Theta(\tau_2) = \Theta(f)^n(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^m(\Theta(\tau_1))$, we have $n_0 \leq n$ and $m_0 \leq m$. Note that the corner case $\mathsf{btwn}_f(x,y,x)$ holds only if $x = y$ because the "distance" between x and x is zero.

Finally, a literal that is not an atom must be of the form $\neg\phi$ where ϕ is an atom, and we simply define $\Theta(\neg\phi) = \neg\Theta(\phi)$. Figures 3.2 and 3.3 give examples of some heap structures and literals. Note that when dealing with acyclic lists, as in Figure 3.2, the fact that node *prev* is between nodes *head* and *curr* can be expressed using a conjunction of reachability literals $f^*(head, prev) \wedge f^*(prev, curr)$. However, if nodes are on a cycle, as in Figure 3.3, each node is reachable from every other node, and therefore it is not possible to express betweenness using reachability literals. For instance, $f^*(x,z) \wedge f^*(z,y)$ holds in this example, although node z is not between nodes x and y. We have introduced between literals to solve that problem: the fact that node y is between nodes x and z in Figure 3.3 can be expressed with $\mathsf{btwn}_f(x,y,z)$.

Conforming to the usual notation, given a heap structure $H = (N, \Theta)$ and a literal ϕ, we write $H \models \phi$ iff $\Theta(\phi) = \text{true}$. For a set of literals Φ, we write $H \models \Phi$ iff $H \models \phi$ for all $\phi \in \Phi$. Given Φ, if there exists H such that $H \models \Phi$, we say that Φ is *satisfiable*. In

[1] 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 $n \geq 1$.

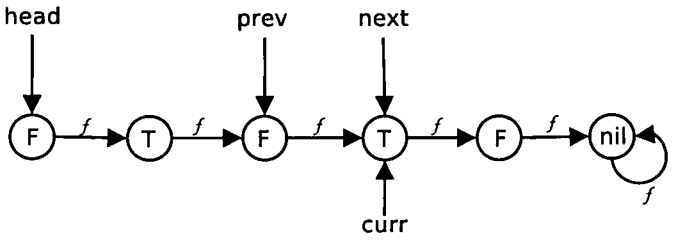

Figure 3.2: Acyclic Heap Structure Example. Arrows marked with f represent a pointer function f, while the Ts and Fs inside nodes are values of a data function d.
As examples of literals interpreted as true consider: $next = curr$ (both variables point to the same node), $f^*(head, \mathsf{nil})$ (the node nil is reachable from the node pointed to by *head* following f links), $f^*(head, prev)$ (the node pointed to by *prev* is reachable from the node pointed to by *head* following f links), $d(curr)$ (the data field of the node pointed to by *curr* is true), $f(f(curr)) = \mathsf{nil}$ (the node to which we get to by following two f links from *curr* is nil), $\mathsf{btwn}_f(head, prev, next)$ (node *prev* is between nodes *head* and *next*).
As examples of literals interpreted as false consider: $d(prev)$ (the data field of the node pointed to by *prev* is false), $next = \mathsf{nil}$ (the node pointed to by *next* is not nil), $f^*(next, prev)$ (the node pointed to by *prev* is not reachable from the node pointed to by *next* following f links).

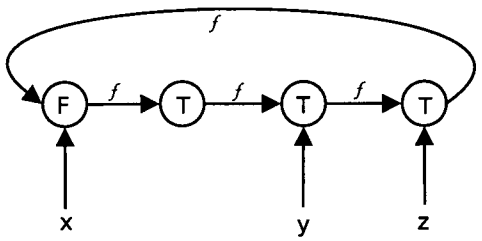

Figure 3.3: Cyclic Heap Structure Example. As examples of literals interpreted as true consider: $\mathsf{btwn}_f(x, y, z)$ (node y is between nodes x and z), $\mathsf{btwn}_f(x, y, y)$, $\mathsf{btwn}_f(x, x, y)$ (node y is reachable from node x following f links, and between includes the endpoints), $f^*(f(x), x)$ (node x is reachable from the node coming after x following f links, i.e. cyclicity).
As examples of literals interpreted as false consider: $f^*(x, \mathsf{nil})$ (nil is not reachable from x following f links), $\mathsf{btwn}_f(x, y, x)$ (the distance, i.e. the number of f links, between x and y is greater than the distance between x and x because each node has a zero-distance from itself).

Chapter 4, we will describe our decision procedure for satisfiability, which has a worst-case exponential running time. The problem it solves is NP-hard (see Appendix A.5), hence a polytime algorithm is unlikely to exist.

3.2 Handling Pointer and Data Function Updates

When doing predicate abstraction and computing reachable abstract states, we are always applying the weakest precondition operator over one program statement at a time (see Sections 2.2 and 2.3). The weakest precondition of a formula ϕ with respect to an assignment statement $x := e$, where x is a variable and e is an expression, is usually defined as the formula constructed by replacing all occurrences of x in ϕ with e. This holds under the assumption that x cannot be aliased. However, handling program assignments that modify the heap structure or data fields is not that straightforward and requires special care. Intuitively, assignments that modify heap structure influence not only literals related to the assigned terms, but also other, apparently unrelated, reachability and between literals. For instance, the assignment $f(x) := y$ might alter the reachability between any pair of nodes that link through the updated node x. Similarly, because of aliasing, updates of data fields impose additional node constraints, e.g. the values of data fields of nodes that alias the updated node have to be changed accordingly. Therefore, handling these two requires special additions to our logic that we will describe here.

To handle program assignments that modify the pointers in the heap, i.e. modify some $f \in F$, we use a special pointer function symbol f' for each modified f. The symbols f and f' model the pointer function before and after the assignment, respectively. 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 Figure 2.2 on page 8 are examples of such assignments. The necessary semantic relationship between f and f' can

be expressed using the well-known update operator:[2]

$$\Theta(f') = \mathsf{update}(\Theta(f), \Theta(\tau_1), \Theta(\tau_2)) \tag{3.1}$$

Rather than support update as an interpreted second order function symbol in the logic, our decision procedure, described in the next chapter, implicitly enforces the constraint (3.1) (see Section 4.3).

Assignments that modify a data field $d \in D$ are handled similarly by using the primed symbol d' for each modified 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 Figure 2.3 on page 9 are examples of such assignments. Analogously to (3.1), the semantic relationship between d and d' is

$$\Theta(d') = \mathsf{update}(\Theta(d), \Theta(\tau), \Theta(b)) \tag{3.2}$$

Our decision procedure also knows to implicitly enforce the constraint (3.2).

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

Chapter 4

Decision Procedure

Predicate abstraction, the previously described technique we are using for HMP verification, makes a number of queries to a decision procedure in order to verify an HMP. Each query is the conjunction of a set of literals in the employed logic. Therefore, the decision problem we aim to solve here is this: given a finite set of literals Φ from the logic proposed in the previous chapter, does there exist a heap structure H such that $H \models \Phi$, i.e. H models all the literals in the set? If there is such an H, then we say that Φ is *satisfiable*, otherwise Φ is *unsatisfiable*.

One approach to this problem would be through a small model theorem, akin to other transitive closure logics [BPZ05, BRS99, RZ05, LQ06]. 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 saturation-based approach, on the other hand, has small memory requirements, and is based on the exhaustive application of a set of inference rules.

4.1 Inference Rules

The inference rules (IRs) attempt to prove unsatisfiability by deriving a *contradiction*, meaning the inference of both an atom ϕ and its negation $\neg\phi$. Figure 4.1 gives an example of a more involved IR r. An *antecedent* of IR r is a literal appearing above the line,

$$\frac{f^*(x,y) \quad f^*(y,z) \quad f^*(z,x)}{\begin{array}{ll} \mathsf{btwn}_f(x,y,z) & \mathsf{btwn}_f(x,z,y) \\ \mathsf{btwn}_f(y,z,x) & \mathsf{btwn}_f(z,y,x) \\ \mathsf{btwn}_f(z,x,y) & \mathsf{btwn}_f(y,x,z) \end{array} \quad x{=}y \quad x{=}z \quad y{=}z}$$

Figure 4.1: Inference Rule Example. Here x, y, and z range over variables V and $f \in F$ ranges over pointer fields.

while a *consequent* is a set of vertically stacked literals appearing below the line. We say that an IR r is *applicable* (to a set of literals 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), *all* of r's antecedents appear in Φ, and *none* of r's consequents appear in Φ, where a consequent Π is defined as appearing in Φ if for each literal $\phi \in \Pi$ it is also the case that $\phi \in \Phi$. We define the formal meaning of an IR with the antecedents A and consequents C as usual:

$$\bigwedge_{\psi \in A} \psi \vdash \bigvee_{\Pi \in C} \bigwedge_{\phi \in \Pi} \phi$$

The basic IRs are presented in Figures 4.2 and 4.3. We now give a brief intuition behind the IRs given in Figure 4.2:

IDENT – states that each node variable is equal to itself.

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 $x = z$.

CYCLE$_k$ – formalizes that 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. Note that CYCLE$_k$ actually defines a separate rule for each $k \geq 1$.

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

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

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

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

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

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

Figure 4.2: Basic Inference Rules. Here x, y, z, etc. range over variables V, $f \in F$ ranges over pointer fields, and $d \in D$ ranges over data fields. Note that $\textsc{Cycle}_k$ actually defines a separate rule for each $k \geq 1$.

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

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

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

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

$$\frac{f^*(x,y)}{\begin{array}{l}\mathsf{btwn}_f(x,x,y)\\ \mathsf{btwn}_f(x,y,y)\end{array}}\textsc{Btw7}$$

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

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

$$\frac{\mathsf{btwn}_f(w,x,y) \quad \mathsf{btwn}_f(w,y,z)}{\mathsf{btwn}_f(w,x,z)}\textsc{Btw11}$$

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

Figure 4.3: Between Inference Rules. Here x, y, z, etc. range over variables V and $f \in F$ ranges over pointer fields.

NOTEQNODES – ensures that if the values of a data field of two nodes are not equal, the nodes are not equal as well.

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.

SCC – 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). It is similar to FUNC, though the two IRs are irredundant with respect to each other.

SHARE – captures the fact that in a cycle of f, no two distinct nodes x and y can have $f(x) = f(y)$.

Similarly, we give an intuition behind the between IRs given in Figure 4.3:

BTWREFLEX – enforces that any node variable is between itself.

BTW1 – if a pointer field of node z points to x (i.e. $f(z) = x$) and y is on the cycle that includes x and z, then y lies between x and z.

BTW2 – states that if node y lies between nodes x and z, then obviously y is reachable from x, and also z is reachable from y.

BTW3 – asserts that if x, w, y, and z are on the same chain and $f(x) = w$, then y is also between w and z, unless $x = y$.

BTW4 – if y is between x and z, and also z between x and y, then it has to be that y and z are equal.

BTW5 – formalizes 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.

BTW6 – handles the case when x, y, and z are all on the same cycle.

BTW7 – ensures that betweenness includes the endpoints.

BTW8 – if the distance between x and z is one, and y is between x and z, then y is either equal to x or to z.

BTW9 – asserts that if x, y, z, and w are on the same chain, y is between x and w, and $f(z)=w$, then y is also between x and z, unless $y=w$.

BTW10 – covers "lollipop" shaped structures where x is a node in the stick leading to the circle of which y and z are a part. It follows that w is on the circle as well, unless $y=z$.

BTW11 – if nodes w, x, y, and z are chained in that order, which is captured by the antecedents, then node x is between nodes w and z.

BTW12 – similarly to BTW11, if nodes v, u, x, and y are chained in that order, then node x is between nodes v and y.

Given the preceding intuition, it is easy to prove the following:

Theorem 1. *The inference rules of Figures 4.2 and 4.3 are sound.*

Proofs of the inference rules related theorems are in Appendix A.

Theorem 1 tells us that the iterative application of the IRs preserves the satisfiability status of the initial set of literals, that is, if it yields a contradiction along every branch caused by the IRs with multiple consequents, then we can conclude that the original set of literals is unsatisfiable.

To prove completeness, we first reduce the problem to sets of literals in a certain normal form, then prove completeness for only *normal sets*, defined bellow.

Definition 1 (normal). *Let* $\mathsf{Vars}(\Phi)$ *denote the subset of the node variables V appearing in* Φ. *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 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.*

The algorithm exhaustively replaces occurrences of terms of the form $f(v)$ with freshly introduced variables v_{fresh}, and consequently adds equality literals $f(v) = v_{fresh}$ to the set.

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 Figures 4.2 and 4.3 are applicable. Thus, we have our completeness theorem, which is the foundation of our decision procedure:

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

Corollary 1. *For any normal set of literals, the inference rules of Figures 4.2 and 4.3 are complete.*

Intuitively, if the iterative application of the IRs on the initial, normal set of literals saturates (i.e. generates a closed set of literals), then the initial set of literals is satisfiable, which brings us to our decision procedure algorithm.

4.2 Basic Decision Procedure

Viewed from a high level, the decision procedure first applies the transformation of Theorem 2 and transforms any initial set of literals to a normal one. Then, the procedure invokes the core algorithm that is presented in Figure 4.4, which can without loss of generality assume that Φ is normal.

The core decision procedure algorithm repeatedly searches for an applicable IR, applies it (i.e. adds the literals of one of its consequents 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*. We note that

```
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 Π of r do
7:             for each literal φ of Π do
8:                 if φ is an equality literal of the form v_i = v_j then
9:                     Φ' := Φ[v_i/v_j]
10:                else
11:                    Φ' := Φ∪{φ}
12:                end if
13:            end for
14:            if DECIDE(Φ') = SAT then
15:                return SAT
16:            end if
17:        end for
18:        return UNSAT
19:    else
20:        return SAT
21:    end if
22: end function
```

Figure 4.4: Pseudocode of the Core Decision Procedure Algorithm. The procedure DECIDE requires Φ to be a normal set of literals. The notation $\Phi[v_i/v_j]$ on line 9 represents the set obtained by replacing all occurrences of v_j in literals of Φ with v_i.

our decision procedure is guaranteed to terminate because none of the IRs introduce new terms.

Theorem 4. *The decision procedure always terminates.*

For the proof of this theorem and lemmas that demonstrate the correctness of the algorithm, see Appendix B.

4.3 Decision Procedure Extension for Handling Updates

In Section 3.2, we introduced additions for handling pointer and data function updates to our logic. Here, we enforce the introduced update constraints (3.1) and (3.2) by adding a

number of additional IRs to our decision procedure.

Initially, recall that in each pointer function update constraint

$$f' = \mathsf{update}(f, \tau_1, \tau_2), \tag{3.1}$$

the symbol f' is the pointer function after the update. Therefore, each of the IRs of Figures 4.2 and 4.3 that mention a pointer function apply to f' also.

To enforce the constraint (3.1), which involves pointer function f before the update and f' after the update, we need additional IRs that mention both pointer functions. Therefore, the decision procedure includes the eight IRs of Figure 4.5. For each pointer function update, the IR UPDATE introduces a fresh variable w that is forced to be equal to $f(\tau_1)$. The reason behind introducing w is to preserve all literals in the normal form (page 31) when applying IRs. Other IRs refer to these freshly introduced variables. This allows us to state that

$$f = \mathsf{update}(f', \tau_1, w),$$

and therefore we have the obvious symmetry between the IRs UPDFUNC1 and UPDFUNC2, between UPDTRANS1 and UPDTRANS2, and between UPDTRANS3 and UPDTRANS4. Note that some of the IRs in Figure 4.5 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 also always terminates. Next, we give a brief intuition behind the IRs in Figure 4.5:

UPDATE – enforces the update constraint (3.1) on f', and also introduces a fresh variable w that is forced to be equal to $f(\tau_1)$.

UPDBTWN – asserts that if y is between x and z before the update, then after the update, y is also between x and z, unless the node τ_1, whose pointer function is updated, is also between x and z. If τ_1 is between x and z, and not equal to z, its update can alter the fact that y is between x and z.

UPDFUNC1 – if $f(x) = y$ before the update, then after the update also $f'(x) = y$, unless x is the node τ_1.

UPDFUNC2 – analogously to UPDFUNC1, if $f'(x) = y$ after the update, then before the update also $f(x) = y$, unless x is the node τ_1.

UPDTRANS1 – if y is reachable from x before the update, then after the update y is also reachable from x, unless the node τ_1 is on the path from x to y.

UPDTRANS2 – analogously to UPDTRANS1, if y is reachable from x after the update, then before the update y is also reachable from x, unless the node τ_1 is on the path from x to y.

UPDTRANS3 – if τ_1 and y are reachable from x before and after the update, respectively, then we case-split on whether τ_1 comes before or after y following the path from x. If τ_1 comes before y, the update can influence the reachability between x and y.

UPDTRANS4 – analogously to UPDTRANS3, if τ_1 and y are reachable from x after and before the update, respectively, then we case-split on whether τ_1 comes before or after y following the path from x.

Similarly to pointer function updates, recall that in each data function update constraint

$$d' = \mathsf{update}(d, \tau, b), \tag{3.2}$$

the symbol d' is the data function after the update. Therefore, the IR NOTEQNODES of Figure 4.2 applies to d' also. To enforce the data update constraint (3.2), which mentions both d and d', we add the four IRs of Figure 4.6 to our decision procedure. Here, we give the intuition behind the added rules:

EQDATA – enforces the data update constraint (3.2), that is, the data function value of the node τ, whose data function is updated, has to be equal to the boolean variable b that is assigned to it.

PRESERVEVALUE – data values of nodes that are not equal to the node τ have to be preserved.

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

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

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

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

Figure 4.5: Pointer Update Inference Rules. These are used to extend our logic to support a second pointer function symbol f' for each updated $f \in F$, 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 $f(\tau_1)$. Note that each pointer update introduces its own, unique variable w, and therefore for each update we actually introduce a separate set of IRs.

$$\frac{d'(\tau) \quad \neg d'(\tau)}{b \quad \neg b}\ \text{EQDATA} \qquad \frac{\neg\tau = x}{d(x) \quad \neg d(x) \quad d'(x) \quad \neg d'(x)}\ \text{PRESERVEVALUE}$$

$$\frac{d(x) \quad \neg d'(x)}{\tau = x}\ \text{EQNODES1} \qquad \frac{\neg d(x) \quad d'(x)}{\tau = x}\ \text{EQNODES2}$$

Figure 4.6: Data Update Inference Rules. These are used to extend our logic to support a second data function symbol d' for each updated $d \in D$, with the implicit constraint $d' = \mathsf{update}(d, \tau, b)$, where $\tau \in V$ and b is a boolean variable.

EQNODES1, EQNODES2 – nodes whose data function value changes have to be equal to the node τ.

Using the presented intuition behind pointer and data function update rules, we prove the following:

Theorem 5. *The inference rules of Figures 4.5 and 4.6 are sound.*

The proof of this theorem is provided in Appendix A. Extending our decision procedure with these additional inference rules allows us to soundly conclude unsatisfiability of a set of literals involving both pointer and data function updates, with the implicit constraints (3.1) and (3.2).

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. Each time the predicate abstraction engine wants to determine whether some abstract state is reachable, the decision procedure is queried for the satisfiability of formula (2.1). If the decision procedure falsely infers that the formula is satisfiable, when it is actually unsatisfiable, the predicate abstraction algorithm will add the unreachable abstract state to the set of abstract states we are assuming are reachable. Therefore, the computed set of reachable abstract states won't be the most precise one, but an over-approximation. Having an over-approximated set of reachable abstract states doesn't preclude soundness of the verification, i.e. if the program is verified, it satisfies all specified properties. The

over-approximation may increase the number of falsely reported property violations (see Section 2.2). However, in our practical experiments of Chapter 5, we never found any property violations caused by the extended decision procedure erroneously concluding that a set of literals was satisfiable.

Chapter 5

Experiments

This chapter presents the results of testing our framework and the proposed simple transitive closure logic decision procedure on a number of HMP examples. We implemented the decision procedure (called *straclos*[1]) used in our experiments in C++, and the implementation is publicly available[2].

The examples we used in our experiments perform different operations on acyclic and cyclic, singly- and doubly-linked lists. Appendix C provides pseudocode and lists the required predicates for all examples. Here, we give a short summary for each example:

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

LIST-ADD first traverses a linked list. Then, it adds a node to the end of the list.

ND-INSERT nondeterministically inserts a node into the linked list. (Pseudocode for this example was previously given in Figure 2.2 on page 8 in the background section on HMPs.)

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

[1] *straclos* stands for Simple TRAnsitive CLOSure logic.

[2] *straclos* can be downloaded from **http://www.cs.ubc.ca/~zrakamar**.

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. [JJKS97].

SORTED-ZIP merges the elements of two sorted lists into one, also sorted. Here, the data elements are simply booleans, so "sorted" means that all nodes with data fields whose value is false come before nodes with data fields whose value is true. This is sufficient to express sortedness of a list of any finite enumerated type (for example `int`).

SORTED-INSERT inserts a node into a sorted linked list so that sortedness is preserved. This is a modification of the example from a paper by Lahiri and Qadeer [LQ06].[3]

BUBBLE-SORT sorts elements of a linked list using the bubble sort algorithm. It is taken from a paper by Balaban et al. [BPZ05]. The data fields on which we sort are again booleans.

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. [MYRS05].

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 combines two cyclic singly-linked lists. Each list represents a set that is uniquely defined with the data field value of its nodes. Therefore, the combination of two lists represents the set union, and the data fields of both lists have to be set to the same value. This example is taken from a paper by Nelson [Nel83].

[3]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.

CREATE-INSERT creates a new node (*malloc*[4]) and inserts it nondeterministically into a linked list.

CREATE-INSERT-DATA creates a new node, initializes its data field, and inserts it nondeterministically into a linked list.

CREATE-FREE creates a new node and inserts it nondeterministically into a linked list. Also, nondeterministically removes a node from the linked list and *frees*[5] it.

INIT-LIST initializes the data fields of an acyclic singly-linked list.

INIT-LIST-VAR similarly to INIT-LIST, initializes the data fields of an acyclic singly-linked lists, but also sets the value of a global data variable before terminating.

INIT-CYCLIC initializes data fields of a cyclic singly-linked list.

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.

REMOVE-CYCLIC-DOUBLY removes an element from a cyclic doubly-linked list. This example is taken from a paper by Lahiri and Qadeer [LQ06].

LINUX-LIST-ADD, LINUX-LIST-ADD-TAIL, LINUX-LIST-DEL are three examples from the Linux kernel list container[6], which add and remove nodes from a cyclic doubly-linked list.

The data fields and data variables in all of our examples are booleans. The safety properties the tool checked (when applicable) at the end of the HMPs are roughly:

[4] *malloc* is modeled as removing a node from the unreachable infinite cyclic list [RSL03].

[5] *free* is modeled in the same fashion as *malloc*, as adding a node to the unreachable infinite cyclic list.

[6] Linux kernel version 2.6.13; list container source file is `include/linux/list.h`; verified functions are `list_add`, `list_add_tail`, and `list_del`.

- *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, that is, the insertion "worked".
- *acyclic* (AC) – the final list is acyclic, that is, nil is reachable from the head of the list.
- *cyclic* (CY) – list is a cyclic singly-linked list, that is, 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, that is, 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, 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 [FQ02, BPZ05] 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 Figure 2.2 on page 8, we can express NL by conjoining $\neg t = \mathsf{nil} \wedge f^*(head, t)$ to the **assume** statement:

assume $\neg f^*(head, item) \wedge f^*(head, \mathsf{nil}) \wedge \neg head = \mathsf{nil} \wedge f(item) = \mathsf{nil} \wedge p = head$

on line 2, and conjoining $f^*(head, t)$ to the assertion:

assert $f^*(head, item) \wedge f^*(head, \mathsf{nil})$

on line 12. Since (after the **assume**) t can be any non-nil node reachable from *head*, if the assertion is never violated, we have proven NL.

Table 5.1 summarizes the results of the experiments, which were run on a 2.6 Ghz Pentium 4 machine. As the table shows, we were successful in verifying interesting properties of many examples quickly and in small amounts of memory.

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 vary greatly.[7] Furthermore, most tools are not publicly available, and most papers do not publish quantitative performance results. Here, we make a few comparisons to tools similar to ours, for which we know some performance results:

The BUBBLE-SORT example is from Balaban et al. [BPZ05]. 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 [Bal05].

The recent experimental results of Manevich et al. [MYRS05] report comparable execution times to us, in spite of the fact they were executed on a slower machine.[8] For most of their examples, however, their times are for verifying only the simple property of no-null-dereferences (and cyclicity for two examples). Our times are for verifying more complicated properties, for instance NL. In addition, for most of the examples, we verify more than one property in a single run.

For the examples in common with Lahiri and Qadeer [LQ06], we are vastly faster at verifying the same properties, with speed-ups of roughly 1 to 3 orders of magnitude on all but one example. It should be noted, however, that we used a slightly faster machine, and

[7] For instance, some of the tools put the burden of providing loop invariants on the users, while we compute those automatically, which is a costly operation.

[8] Unfortunately, their tool hasn't been publicly released, and therefore we couldn't make a more thorough comparison.

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

Table 5.1: Results of Verifying HMPs. "Program" is the verified HMP; "Property" specifies the verified property; "CFG" denotes the number of edges in the control-flow graph of the program; "Preds" is the number of predicates required for verification; "DP calls" is the number of decision procedure queries; "Time(sec)" is the total execution time in seconds. The experiments were executed on a 2.6 Ghz Pentium 4 machine. The memory usage for each of the experiments was less than 20 MB.

also that our data fields are booleans whereas theirs are abstract integers. 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. The limited support for universally quantified variables we are planning to add would solve this problem.

In addition to these experiments, we ran *straclos* on a couple of queries for MONA generated by field constraint analysis tool Bohne [WKL+06]. Initial results show that *straclos* is faster than MONA, but the queries we have are too simple to make a more serious comparison.

Chapter 6

Conclusions and Future Work

This thesis has presented a novel logic and accompanying decision procedure that are used in the verification of heap-manipulating programs using predicate abstraction techniques. A number of experiments demonstrate the usability and effectiveness of our work for verification of HMPs that occur in practice. Of special note is our verification of three small, but real, Linux kernel list container functions, which use cyclic doubly-linked lists. These results clearly show the potential that this work has in the world of HMP verification.

There are some obvious directions left for further improving the proposed logic. First is the addition of quantifiers. We have found that even minimal support for universally quantified variables (e.g. allowing universal quantification over variables only in the beginning of a formula) would allow expression of many common heap structure attributes. For example, even if we introduce Skolem constants, as described in Chapter 5, the current logic cannot assert that two terms x and y point to disjoint linked lists. Intuitively, using skolemization we can express that some unconstrained node from the list x is not reachable from y, which doesn't necessarily mean that all nodes from the list x are not reachable from y. A single universally quantified variable would allow for this property (see Nelson [Nel79, page 22]). We also found that capturing disjointedness is necessary for verifying that LIST-REVERSE example always produces an acyclic list; hence we were unable to verify this property. Our decision procedure can be enhanced to soundly support

universal quantification using heuristic quantifier instantiation techniques, i.e., eliminating quantifiers by instantiating them with terms appearing in a formula.

The other logic limitation, that we see no immediate solution to, is our inability to express more involved heap structure properties, in particular trees. Our logic cannot capture "x points to a tree" because expressing that requires using transitive closure over multiple pointer functions (i.e. expressing the fact that some node is reachable from the tree root following a sequence of *left* or *right* pointers). It is likely that adding such a transitive closure operator to our logic would cause undecidability [IRR+04a]. However, we believe that it is possible that an extension could be used to verify simple properties of programs that manipulate trees. For example, by supporting relations instead of pointer functions, we could verify that there are no memory leaks in HMPs that manipulate trees.

Besides improving the logic, we also plan on investigating how existing techniques for predicate discovery and more advanced predicate abstraction algorithms mesh with our decision procedure. For instance, predicate abstraction algorithm could be enhanced to support quantifiers by employing indexed predicate abstraction [LB04]. We would also like to look into possible ways of extending our decision procedure to generate interpolants [Cra57].[1] Interpolants have been successfully used for refining abstractions in software model checking [HJMM04], and can be efficiently generated from proofs of unsatisfiability. Furthermore, by incorporating our decision procedure into a Nelson-Oppen style theorem prover [NO79, MZ03], it would be possible to improve the precision of a heap abstraction used by the existing software verification tools that employ theorem provers [BMMR01, HJMS02, FLL+02]. We already have the initial results (the sketch of the proof that our logic is *stably infinite*, which is an important Nelson-Oppen requirement) showing that incorporating the decision procedure can be done.

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

Bibliography

[Bal05] I. Balaban, 2005. Personal correspondence.

[BLS02] 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 *14th International Conference on Computer Aided Verification (CAV)*, pages 78 – 92, 2002.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[BPR02] T. Ball, A. Podelski, and S.K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.

[BPZ05] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.

[BR06] J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages

207–221, 2006. Extended version: UBC Department of Computer Science Technical Report TR-2005-19.

[BRS99] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming (ESOP)*, 1999.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV)*, pages 154–169, 2000.

[Cra57] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.

[DD01] S. Das and D. L. Dill. Successive approximation of abstract transition relations,. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2001.

[DD02] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2002.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer Aided Verification (CAV)*, 1999.

[Dij72] E. W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, 1972.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DN03] D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 310–323, 2003.

[DOY06] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic, 2006.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 191–202, 2002.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[Gri81] D. Gries. *The Science of Programming*. Springer, New York, 1981.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV)*, 1997.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–244, 2004.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[IRR+04a] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *18th International Workshop on Computer Science Logic (CSL)*, pages 160–174, 2004.

[IRR+04b] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Conf. on Computer Aided Verification (CAV)*, 2004.

[JJKS97] J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 226–236, 1997.

[KMS00] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *5th International Conference on Implementation and Application of Automata (CIAA)*, 2000.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KS93] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 196–205, 1993.

[LAIR+05] 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 *Conference on Automated Deduction (CADE)*, 2005.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *7th International Static Analysis Symposium (SAS)*, pages 280–301, 2000.

[LB04] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 267–281, 2004.

[LQ06] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 115–126, 2006.

[LRS05] A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *17th International Conference on Computer Aided Verification (CAV)*, pages 519–533, 2005.

[MN05] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *17th International Conference on Computer Aided Verification (CAV)*, pages 476–490, 2005.

[MNCL06] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *3rd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.

[MS01] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *6th International Confer-*

ence on Verification, Model Checking and Abstract Interpretation (VMCAI), pages 181–198, 2005.

[MZ03] Z. Manna and C. G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003.

[Nel79] G. Nelson. *Techniques for program verification.* PhD thesis, Stanford University, 1979.

[Nel83] G. Nelson. Verifying reachability invariants of linked structures. In *10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–47, 1983.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[RBH06] Z. Rakamarić, J. Bingham, and A. J. Hu. A better logic and decision procedure for predicate abstraction of heap-manipulating programs. Technical Report TR-2006-02, UBC Department of Computer Science, January 2006.

[RSL03] 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.

[RZ05] S. Ranise and C. G. Zarba. A decidable logic for pointer programs manipulating linked lists. Unpublished manuscript, 2005.

[WKL+06] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.

[YRS04] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.

[YRS+06] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Foundations of Software Science and Computation Structures (FOSSACS)*, 2006.

Appendix A

Proofs

In this appendix, we prove all theorems except the decision procedure related Theorem 4, which is proven in Appendix B. The proofs presented here are largely due to Jesse Bingham, who was a coauthor of the published papers on the topic. I have included the proofs in my thesis for the sake of completeness of the presentation.

A.1 Proof of Theorem 1

Our proof of Theorem 1 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 $\Theta(x)$ (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 1. *The inference rules of Figures 4.2 and 4.3 are sound.*

Proof. We argue in turn that each inference rule is sound. Most between inference rule cases involve an implicit appeal to Lemma 1.

- IDENT, REFLEX, TRANS1, TRANS2. These rules are clearly sound.
- 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.
- CYCLE$_k$, $k \geq 1$. Suppose all the antecedents hold; then $y = f^n(x_1)$ for some $n \geq 0$ and thus $y = x_{1+(n \bmod k)}$.
- NOTEQNODES. The antecedents imply that $d(x) \neq d(y)$, which implies $x \neq y$.
- 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 $z = f^{m-n}(y)$.
- 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 $f*(z,x)$.

- 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 $f(x) = f(y)$.
- 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 therefore $\mathsf{btwn}_f(w,y,z)$.
- 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.2 Proof of Theorem 2

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. 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 Definition 1, is satisfiable if and only if Φ is, and is constructed in polynomial time. □

A.3 Proof of Theorem 3

In order to prove Theorem 3, 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 pointer 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 that $\mathsf{btwn}_f(v,w,w) \in \Phi$. 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 1 (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 2 (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. 1, $\eta_f(v) = v$. 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. 1, there exists $w \in V$ such that $\text{btwn}_f(v,u,w) \in \Phi$, 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. 1 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. 1, the fact that $\eta_f(v) = u$, and Lemma 5 we have that $u \leq_f^v w$ and hence $\text{btwn}_f(v,u,w) \in \Phi$. 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

$\text{btwn}_f(u,x,y) \in \Phi$. Since u is the $\leq_f^v$-minimal element of $R_f(v) \setminus \{v\}$, we also have that $\text{btwn}_f(v,u,x) \in \Phi$ and $\text{btwn}_f(v,u,y) \in \Phi$. It follows that $\text{btwn}_f(v,x,y) \in \Phi$, since BTW12 is disabled, and thus $x \leq_f^v y$. □

Definition 3 ($seq_f(v)$). *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 $\left|R_f(v)\right|$ 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. 3. Otherwise, the Lemma follows trivially from Def. 3. □

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. 3. 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 $|R_f(v)|$ 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. 3. 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 $|R_f(u)|$ 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 $|R_f(v)|$ 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 $|R_f(u)|$ 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 4. *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*

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

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

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

Theorem 3. *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 Definition 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. 4.
- b : satisfied by H^Φ from Def. 4.
- $f^*(v,u)$: 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. 4.
- b : not satisfied by H^Φ from Def. 4.

- $f^*(v,u)$: 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.4 Proof of Theorem 5

Theorem 5. *The inference rules of Figures 4.5 and 4.6 are sound.*

Proof. We use the same abuse of notation used in the proof of Theorem 1.

- UPDATE. This IR is sound since w is a fresh variable.
- UPDBTWN. We employ the notation $\delta(\cdot,\cdot)$ used in the proof of Theorem 1, 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 $\mathsf{btwn}_f(x,\tau_1,z)$.
- UPDFUNC1. The inference rule clearly respect the facts that $f' = \mathsf{update}(f,\tau_1,\tau_2)$ and $f = \mathsf{update}(f',\tau_1,w)$.
- UPDFUNC2. Analogous to UPDFUNC1.
- 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. Analogous to UPDTRANS1.
- 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. Analogous to UPDTRANS3.
- 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.

□

A.5 Complexity of the Satisfiability Problem

The proof of complexity of the satisfiability problem for our logic uses a reduction from the following decision problem, which is known to be NP-complete [GJ90].

Definition 5 (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 6. *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 pointer 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 $distinct(T)$ be the set of $\binom{|T|}{2}$ literals that enforces that the terms of T are pair-wise unequal, for example

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

Similarly, given two sets of terms T_1 and T_2, let $distinct(T_1, T_2)$ 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 $n = |A|$:

$$\{\neg f^{n-2}(h) = f^{n-1}(h), f^{n-1}(h) = f^n(h)\} \tag{A.1}$$

$$distinct(A) \tag{A.2}$$

$$\{f^*(h, e) \mid e \in A\} \tag{A.3}$$

$$\bigcup_{(a,b,c)\in C} \{f^*(y_{(a,b,c)}, b), f^*(b, z_{(a,b,c)}), f^*(h, y_{(a,b,c)}), f^*(h, z_{(a,b,c)})\} \tag{A.4}$$

$$\bigcup_{(a,b,c)\in C} distinct(\{y_{(a,b,c)}, z_{(a,b,c)}\}, A \setminus \{a, c\}) \tag{A.5}$$

(A.1) says that h is the head of a non-cyclic list of exactly n nodes $h, f(h), \ldots, f^{n-1}(h)$. (A.2) and (A.3) 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 (A.3) implies that for each $e \in A$ we have $e = f^i(h)$ for some $0 \leq i < n$ while (A.2) enforces that there is no $e' \in A \setminus \{e\}$ such that $e' = f^i(h)$. (A.4) 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 (A.5) 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. 5 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. 5 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. □

Appendix B

Formalization of the Decision Procedure

The core decision procedure algorithm takes a normal set of literals Φ; this restriction does not lose us any generality thanks to Lemma 18. The pseudocode of the core of the decision procedure is given in Figure 4.4 on page 32. 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 Figure 3.1 on page 20; they do not apply to the extended logic of Section 3.2. The proofs of Lemmas 18, 19, and Theorem 4 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 14 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 9, and clearly performing the substitution preserves normality. In all other cases, Φ' is assigned by line 11, 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 18, addition of all consequents of an applicable IR yielded UNSAT from the recursive calls. The proof thus depends on the Theorem 1, which states that the IRs of Figure 4.2 and Figure 4.3 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 20. Let $\hat{\Phi}$ be the set of literals that caused line 20 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 3, $\hat{\Phi}$ is satisfiable, which implies the satisfiability of Φ also. □

B.1 Proof of Theorem 4

Theorem 4. *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. □

Appendix 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
```

Figure C.1: **LIST-REVERSE** is a classical HMP example that performs in-place reversal of a linked list. 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
```

Figure C.2: **LIST-ADD** first traverses a linked list. Then, it adds a node to the end of the list. 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
```

Figure C.3: **ND-INSERT** nondeterministically inserts a node into the linked list. 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
```

Figure C.4: ND-REMOVE nondeterministically chooses a node and removes it from the list. 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
```

Figure C.5: 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. 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;   x := f(x);
11:        else
12:            if ¬temp=nil then
13:                f(temp) := y;
14:            else
15:                merge := y;
16:            end if
17:            temp := y;   y := f(y);
18:        end if
19:    end while
20:    if ¬x=nil then
21:        if merge=nil then
22:            merge := x;
23:        else
24:            f(temp) := x;
25:        end if
26:    end if
27:    if ¬y=nil then
28:        if merge=nil then
29:            merge := y;
30:        else
31:            f(temp) := y;
32:        end if
33:    end if
34:    assert f*(merge,t) ∧ (d(t) ≤ d(f(t)) ∨ f(t)=nil)
35: end procedure
```

Figure C.6: **SORTED-ZIP** merges the elements of two sorted lists into one, also sorted. 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
```

Figure C.7: SORTED-INSERT inserts a node into a sorted linked list so that sortedness is preserved. 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
```

Figure C.8: BUBBLE-SORT sorts elements of a linked list using the bubble sort algorithm. Predicates used to verify the example: $f^*(x, \text{nil})$, $f^*(x,t)$, $t = \text{nil}$, $y = x$, $yn = f(y)$, $prev = \text{nil}$, $last = \text{nil}$, $d(t)$, $d(f(t))$, $f(t) = \text{nil}$, $f(x) = last$, $yn = last$, $d(y)$, $d(yn)$, $f^*(yn,t)$, $f^*(last,t)$, $f^*(x,prev)$, $f(yn) = y$, $f(prev) = y$, $t = y$, $f(yn) = f(y)$, $f^*(x,yn)$, $d(last)$, $prev = y$. Comparison between data values is defined as a formula over boolean data predicates. For instance, $d(x) \leq d(y)$ is defined as $\neg(d(x) \wedge \neg d(y))$.

```
 1: procedure REMOVE-ELEMENTS(x)
 2:     assume ¬x=nil ∧ f*(f(x),x) ∧ f*(x,t) ∧ btwn_f(curr,t,x) ∧ prev=x ∧ curr=f(x)
 3:     while ¬curr=x do
 4:         if d(curr)=false then
 5:             curr := f(curr);
 6:             f(prev) := curr;
 7:         else
 8:             prev := curr;
 9:             curr := f(curr);
10:         end if
11:     end while
12:     if d(x)=false then
13:         if ¬prev=x then
14:             x := f(x);
15:             f(prev) := x;
16:         else
17:             x := nil;
18:         end if
19:     end if
20:     assert f*(f(x),x) ∧ (    (f*(x,t) ∧ d(t)=true)
                             ∨  (¬f*(x,t) ∧ d(t)=false))
21: end procedure
```

Figure C.9: REMOVE-ELEMENTS removes from a cyclic list elements whose data field is false. Predicates used to verify the example: $f^*(f(x),x)$, $curr=x$, $prev=x$, $d(curr)$, $d(x)$, $f^*(x,t)$, $d(t)$, $x=\mathsf{nil}$, $curr=f(x)$, $\mathsf{btwn}_f(curr,t,x)$, $f(prev)=curr$, $f(f(prev))=curr$, $f^*(f(x),prev)$, $prev=curr$, $f^*(t,prev)$, $f(prev)=t$, $f(curr)=x$.

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

Figure C.10: **REMOVE-SEGMENT** removes the first contiguous segment of elements whose data field is true from a cyclic singly-linked list. Predicates used to verify the example: $f^*(f(x),x)$, $x=\mathsf{nil}$, $temp=\mathsf{nil}$, $y=x$, $z=\mathsf{nil}$, $x=temp$, $z=x$, $z=y$, $d(y)$, $d(z)$, $f^*(z,x)$, $f^*(z,y)$, $\mathsf{btwn}_f(y,z,x)$, $\mathsf{btwn}_f(y,temp,x)$, $f^*(y,x)$.

```
 1: procedure SEARCH-AND-SET(x, data_1, data_2)
 2:     assume ¬x=nil ∧ f*(f(x),x) ∧ f(x)=curr ∧ f*(x,t) ∧ btwn_f(curr,t,x) ∧ true
 3:     d_1(x) := true;
 4:     d_2(x) := true;
 5:     while ¬curr=x do
 6:         if d_1(curr)=data_1 ∧ d_2(curr)=data_2 then
 7:             break;
 8:         else
 9:             d_1(curr) := true;
10:             d_2(curr) := true;
11:             curr := f(curr);
12:         end if
13:     end while
14:     assert f*(f(x),x) ∧ f*(x,t) ∧ ¬x=nil ∧ true
15:     assert (x=curr ∧ d_1(t) ∧ d_2(t))
            ∨ ( ¬x=curr ∧ d_1(curr)=data_1 ∧ d_2(curr)=data_2
              ∧ ( (btwn_f(x,t,curr) ∧ ¬t=curr ∧ d_1(t) ∧ d_2(t))
                ∨ (btwn_f(x,t,curr) ∧ t=curr)
                ∨ ¬btwn_f(x,t,curr)))
16: end procedure
```

Figure C.11: **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. Predicates used to verify the example: $f^*(f(x),x)$, $f(x)=curr$, $d_1(curr)$, $d_2(curr)$, $data_1$, $data_2$, $x=curr$, $f^*(x,t)$, $x=\mathsf{nil}$, $\mathsf{btwn}_f(curr,t,x)$, $\mathsf{btwn}_f(x,t,curr)$, $d_1(t)$, $d_2(t)$, $true$, $curr=t$, $x=t$.

```
1: procedure SET-UNION(a,b)
2:     assume f(a)=curr ∧ f*(f(a),a) ∧ f*(f(b),b)
3:     assume f*(a,t) ∧ ¬f*(b,t) ∧ ¬d(t)
4:     assume ¬f*(a,s) ∧ f*(b,s) ∧ d(s)
5:     assume ¬d(a) ∧ d(b)
6:     assume btwn_f(f(a),t,a) ∧ btwn_f(f(b),s,b)
7:     assume ¬t=nil ∧ ¬s=nil
8:     tmpd := d(b);
9:     d(a) := tmpd;
10:    while ¬curr=a do
11:        d(curr) := tmpd;
12:        curr := f(curr);
13:    end while
14:    tmp := f(a);
15:    f(a) := f(b);
16:    f(b) := tmp;
17:    assert f*(f(b),b) ∧ f*(b,t) ∧ d(t) ∧ f*(b,s) ∧ d(s) ∧ d(b) ∧ ¬t=nil ∧ ¬s=nil
18: end procedure
```

Figure C.12: **SET-UNION** combines two cyclic singly-linked lists. 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 ∧ ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧
           ¬f(malloc) = pmalloc ∧ item = nil ∧ f(pmalloc) = malloc ∧ ¬malloc = nil ∧
           f*(malloc,pmalloc)
3:     assume (f*(head,t) ∧ ¬f*(malloc,t)) ∨ (¬f*(head,t) ∧ f*(malloc,t))
4:     assume ¬f(malloc)=pmalloc;
5:     item := malloc;
6:     malloc := f(malloc);
7:     f(pmalloc) := malloc;
8:     while true do
9:         if ND ∨ f(p)=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 ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧ ¬item=nil ∧
           f*(head,item) ∧ f(pmalloc)=malloc ∧ ¬malloc=nil ∧ f*(malloc,pmalloc)
18:    assert (f*(head,t) ∧ ¬f*(malloc,t)) ∨ (¬f*(head,t) ∧ f*(malloc,t))
19: end procedure
```

Figure C.13: **CREATE-INSERT** creates a new node (*malloc*) and inserts it nondeterministically into a linked list. Predicates used to verify the example: $f^*(head,t)$, $nil=f(p)$, $p=head$, $t=nil$, $f^*(head,nil)$, $head=nil$, $f^*(f(malloc),malloc)$, $f(malloc)=pmalloc$, $f^*(malloc,t)$, $item=nil$, $f^*(head,item)$, $f(pmalloc)=malloc$, $malloc=nil$, $f^*(malloc,pmalloc)$, $f^*(head,p)$, $f^*(item,nil)$, $f^*(item,p)$, $f^*(item,t)$, $f^*(f(p),t)$ $item=t$, $f(pmalloc)=item$, $f(f(pmalloc))=malloc$, $f^*(malloc,item)$, $nil=f(item)$.
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 ∧ ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧
           ¬f(malloc) = pmalloc ∧ item = nil ∧ f(pmalloc) = malloc ∧ ¬malloc = nil ∧
           f*(malloc,pmalloc)
3:     assume (f*(head,t) ∧ ¬f*(malloc,t) ∧ ¬(d(head) ⊕ d(t)))
           ∨ (¬f*(head,t) ∧ f*(malloc,t))
4:     assume ¬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 ∨ f(p)=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 ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧ ¬item=nil ∧
           f*(head,item) ∧ f(pmalloc)=malloc ∧ ¬malloc=nil ∧ f*(malloc,pmalloc)
20:    assert (f*(head,t) ∧ ¬f*(malloc,t) ∧ ¬(d(head) ⊕ d(t)))
           ∨ (¬f*(head,t) ∧ f*(malloc,t))
21: end procedure
```

Figure C.14: **CREATE-INSERT-DATA** creates a new node, initializes its data field, and inserts it nondeterministically into a linked list. Predicates used to verify the example: $t = \text{nil}$, $f^*(head,t)$, $\text{nil} = f(p)$, $p = head$, $f^*(head,\text{nil})$, $head = \text{nil}$, $f^*(f(malloc),malloc)$, $f(malloc) = pmalloc$, $f^*(malloc,t)$, $item = \text{nil}$, $f^*(head,item)$, $f(pmalloc) = malloc$, $malloc = \text{nil}$, $f^*(malloc,pmalloc)$, $d(head)$, $d(t)$, $f^*(item,p)$, $f^*(item,t)$, $f^*(f(p),t)$ $item = t$, $f(pmalloc) = item$, $f(f(pmalloc)) = malloc$, $f^*(malloc,item)$, $\text{nil} = f(item)$, $f^*(head,p)$, $f^*(item,\text{nil})$, $tmpd$.
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 ∧ ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧
           ¬f(malloc) = pmalloc ∧ item = nil ∧ f(pmalloc) = malloc ∧ ¬malloc = nil ∧
           f*(malloc,pmalloc)
3:     assume (f*(head,t) ∧ ¬f*(malloc,t)) ∨ (¬f*(head,t) ∧ f*(malloc,t))
4:     assume ¬f(malloc)=pmalloc;
5:     item := malloc;
6:     malloc := f(malloc);
7:     f(pmalloc) := malloc;
8:     while true do
9:         if ND ∨ f(p)=nil then
10:            f(item) := f(p);
11:            f(p) := item;   break;
12:        else
13:            p := f(p);
14:        end if
15:    end while
16:    p := head;   r := f(head);
17:    while true do
18:        if ND ∨ f(r)=nil then
19:            f(p) := f(r);
20:            f(pmalloc) := r;
21:            f(r) := malloc;
22:            malloc := r;   break;
23:        else
24:            p := r; r := f(r);
25:        end if
26:    end while
27:    assert ¬t=nil ∧ f*(head,nil) ∧ ¬head=nil ∧ f*(f(malloc),malloc) ∧ ¬item=nil ∧
           f(pmalloc)=malloc ∧ ¬malloc=nil ∧ f*(malloc,pmalloc) ∧ ¬f*(head,r)
28:    assert (f*(head,t) ∧ ¬f*(malloc,t) ∧ ¬r=t ∧ (f*(head,item) ⊕ r=item))
           ∨ (¬f*(head,t) ∧ f*(malloc,t) ∧ (f*(head,item) ⊕ r=item))
29: end procedure
```

Figure C.15: **CREATE-FREE** creates a new node and inserts it into a linked list. Also, removes a node from the list and *frees* it. 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 20-22 model a *free* statement by returning a node to the infinite cyclic list of unallocated nodes.

```
1: procedure INIT-LIST(x)
2:     assume f*(x,t) ∧ f*(x,nil) ∧ curr=x ∧ ¬t=nil ∧ true
3:     while ¬curr=nil do
4:         d(curr) := true;
5:         curr := f(curr);
6:     end while
7:     assert f*(x,t) ∧ f*(x,nil) ∧ d(t) ∧ true ∧ ¬t=nil
8: end procedure
```

Figure C.16: **INIT-LIST** initializes the data fields of an acyclic singly-linked list. Predicates used to verify the example: $curr = \mathsf{nil}$, $curr = x$, $f^*(x,t)$, $f^*(x,\mathsf{nil})$, $d(t)$, $true$, $t = \mathsf{nil}$, $f^*(curr,t)$, $f^*(t,curr)$.

```
1: procedure INIT-LIST-VAR(x,tmp)
2:     assume f*(x,t) ∧ f*(x,nil) ∧ curr=x ∧ ¬t=nil ∧ true
3:     while ¬curr=nil do
4:         d(curr) := true;
5:         curr := f(curr);
6:     end while
7:     tmp := d(x);
8:     assert f*(x,t) ∧ f*(x,nil) ∧ d(t) ∧ true ∧ tmp ∧ ¬t=nil
9: end procedure
```

Figure C.17: **INIT-LIST-VAR** initializes the data fields of an acyclic singly-linked lists, and also sets the value of a global data variable before terminating. 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.

```
1: procedure INIT-CYCLIC(x)
2:     assume f*(x,t) ∧ f*(f(x),x) ∧ curr=f(x) ∧ btwn_f(curr,t,x) ∧ ¬x=nil ∧ true
3:     d(x) := true;
4:     while ¬curr=x do
5:         d(curr) := true;
6:         curr := f(curr);
7:     end while
8:     assert f*(x,t) ∧ f*(f(x),x) ∧ d(t) ∧ true ∧ ¬x=nil
9: end procedure
```

Figure C.18: **INIT-CYCLIC** initializes data fields of a cyclic singly-linked list. Predicates used to verify the example: $curr = x$, $curr = f(x)$, $f^*(x,t)$, $f^*(f(x),x)$, $d(t)$, $true$, $\mathsf{btwn}_f(curr,t,x)$, $x = \mathsf{nil}$, $t = x$, $\mathsf{btwn}_f(x,t,curr)$, $f^*(t,curr)$.

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

Figure C.19: **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. 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 $\neg(d(x) \wedge \neg d(y))$.

```
1:  procedure REMOVE-DOUBLY(head, tail, node)
2:      assume next*(head, tail) ∧ prev*(tail, head) ∧ nil = next(tail) ∧ nil = prev(head)
3:      assume next*(head, t) ∧ prev*(tail, t)
4:      assume next*(head, node) ∧ prev*(tail, node)
5:      assume next*(head, prev(node)) ∧ prev*(tail, next(node))
6:      assume ¬node = nil ∧ ¬t = nil
7:      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))))
8:      if prev(node) = nil then
9:          head := next(node);
10:     else
11:         temp := prev(node);   next(temp) := next(node);
12:     end if
13:     if next(node) = nil then
14:         tail := prev(node);
15:     else
16:         temp := next(node);   prev(temp) := prev(node);
17:     end if
18:     assert next*(head, tail) ∧ prev*(tail, head) ∧ nil = next(tail) ∧ nil = prev(head)
19:     assert ¬next*(head, node) ∧ ¬prev*(tail, node)
20:     assert ¬node = nil ∧ ¬t = nil
21:     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))))
22: end procedure
```

Figure C.20: REMOVE-DOUBLY removes an element from an acyclic doubly-linked list. Predicates used to verify the example:
$nil = prev(node)$, $nil = next(node)$, $next^*(head, tail)$, $prev^*(tail, head)$,
$nil = next(tail)$, $nil = prev(head)$, $next^*(head, t)$, $prev^*(tail, t)$,
$head = t$, $tail = t$, $t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, node)$, $prev^*(tail, node)$, $head = node$, $tail = node$,
$node = prev(next(node))$, $node = next(prev(node))$,
$next^*(head, prev(node))$, $prev^*(tail, next(node))$,
$nil = prev(next(node))$, $nil = next(prev(node))$,
$node = nil$, $node = t$, $t = nil$, $head = next(node)$, $head = temp$, $temp = prev(node)$,
$next^*(head, temp)$, $temp = t$, $temp = next(prev(node))$, $next(node) = next(prev(node))$,
$prev(temp) = prev(node)$, $prev(node) = t$.

```
 1: procedure REMOVE-CYCLIC-DOUBLY(head, entry)
 2:     assume next*(next(head), head) ∧ prev*(prev(head), head)
 3:     assume prev*(head, next(head)) ∧ next*(head, prev(head))
 4:     assume next*(head, t) ∧ prev*(head, t)
 5:     assume next*(head, prev(t)) ∧ prev*(head, next(t))
 6:     assume next*(head, entry) ∧ prev*(head, entry)
 7:     assume next*(head, prev(entry)) ∧ prev*(head, next(entry))
 8:     assume t = prev(next(t)) ∧ t = next(prev(t))
 9:     assume head = prev(next(head)) ∧ head = next(prev(head))
10:     assume entry = prev(next(entry)) ∧ entry = next(prev(entry))
11:     assume ¬entry = head
12:     p := prev(entry);
13:     n := next(entry);
14:     prev(n) := p;
15:     next(p) := n;
16:     assert next*(next(head), head) ∧ prev*(prev(head), head)
17:     assert ¬next*(head, entry) ∧ ¬prev*(head, entry)
18:     assert prev*(head, next(head)) ∧ next*(head, prev(head))
19:     assert head = prev(next(head)) ∧ head = next(prev(head))
20:     assert ¬entry = prev(next(entry)) ∧ ¬entry = next(prev(entry))
21:     assert ¬entry = head
22:     assert (   next*(head, t) ∧ prev*(head, t) ∧ t = prev(next(t)) ∧ ¬entry = t
                 ∧ t = next(prev(t)) ∧ next*(head, prev(t)) ∧ prev*(head, next(t)))
            ∨ (   ¬next*(head, t) ∧ ¬prev*(head, t)
                 ∧ ¬t = prev(next(t)) ∧ ¬t = next(prev(t)) ∧ entry = t)
23: end procedure
```

Figure C.21: **REMOVE-CYCLIC-DOUBLY** removes an element from a cyclic doubly-linked list. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, entry)$, $prev^*(head, entry)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$head = prev(next(head))$, $head = next(prev(head))$,
$entry = prev(next(entry))$, $entry = next(prev(entry))$,
$next^*(head, prev(entry))$, $prev^*(head, next(entry))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = entry$, $entry = head$, $next(entry) = n$, $prev(entry) = p$, $n = head$, $prev(n) = p$, $n = t$,
$next(head) = entry$, $p = t$.

```
 1: procedure LINUX-LIST-ADD(head, new)
 2:     assume next*(next(head), head) ∧ prev*(prev(head), head)
 3:     assume ¬next*(head, new) ∧ ¬prev*(head, new)
 4:     assume prev*(head, next(head)) ∧ next*(head, prev(head))
 5:     assume next(new) = nil ∧ prev(new) = nil
 6:     assume head = prev(next(head)) ∧ head = next(prev(head))
 7:     assume (   next*(head, t) ∧ prev*(head, t) ∧ t = prev(next(t)) ∧ t = next(prev(t))
                 ∧ ¬t = new ∧ next*(head, prev(t)) ∧ prev*(head, next(t)))
            ∨ (   ¬next*(head, t) ∧ ¬prev*(head, t) ∧ ¬t = prev(next(t)) ∧ t = new
                 ∧ ¬t = next(prev(t)) ∧ ¬next*(head, prev(t)) ∧ ¬prev*(head, next(t)))
 8:     p := head;
 9:     n := next(head);
10:     prev(n) := new;
11:     next(new) := n;
12:     prev(new) := p;
13:     next(p) := new;
14:     assert next*(next(head), head) ∧ prev*(prev(head), head)
15:     assert prev*(head, next(head)) ∧ next*(head, prev(head))
16:     assert next*(head, t) ∧ prev*(head, t)
17:     assert next*(head, prev(t)) ∧ prev*(head, next(t))
18:     assert next*(head, new) ∧ prev*(head, new)
19:     assert t = prev(next(t)) ∧ t = next(prev(t))
20:     assert ¬next(new) = nil ∧ ¬prev(new) = nil
21:     assert head = prev(next(head)) ∧ head = next(prev(head))
22: end procedure
```

Figure C.22: **LINUX-LIST-ADD** adds a node to a cyclic doubly-linked list. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, new)$, $prev^*(head, new)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$next(new) = nil$, $prev(new) = nil$,
$head = prev(next(head))$, $head = next(prev(head))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = new$, $p = head$, $next(new) = n$, $prev(n) = new$, $n = t$, $head = t$, $prev(new) = head$, $n = head$,
$n = next(head)$.

```
 1: procedure LINUX-LIST-ADD-TAIL(head, new)
 2:     assume next*(next(head), head) ∧ prev*(prev(head), head)
 3:     assume prev*(head, next(head)) ∧ next*(head, prev(head))
 4:     assume ¬next*(head, new) ∧ ¬prev*(head, new)
 5:     assume next(new) = nil ∧ prev(new) = nil
 6:     assume head = prev(next(head)) ∧ head = next(prev(head))
 7:     assume (   next*(head, t) ∧ prev*(head, t) ∧ t = prev(next(t)) ∧ t = next(prev(t))
                 ∧ ¬t = new ∧ next*(head, prev(t)) ∧ prev*(head, next(t)))
            ∨ (   ¬next*(head, t) ∧ ¬prev*(head, t) ∧ ¬t = prev(next(t)) ∧ t = new
                 ∧ ¬t = next(prev(t)) ∧ ¬next*(head, prev(t)) ∧ ¬prev*(head, next(t)))
 8:     p := prev(head);
 9:     n := head;
10:     prev(n) := new;
11:     next(new) := n;
12:     prev(new) := p;
13:     next(p) := new;
14:     assert next*(next(head), head) ∧ prev*(prev(head), head)
15:     assert prev*(head, next(head)) ∧ next*(head, prev(head))
16:     assert next*(head, t) ∧ prev*(head, t)
17:     assert next*(head, prev(t)) ∧ prev*(head, next(t))
18:     assert next*(head, new) ∧ prev*(head, new)
19:     assert t = prev(next(t)) ∧ t = next(prev(t))
20:     assert ¬next(new) = nil ∧ ¬prev(new) = nil
21:     assert head = prev(next(head)) ∧ head = next(prev(head))
22: end procedure
```

Figure C.23: **LINUX-LIST-ADD-TAIL** adds a node to the tail of a cyclic doubly-linked list. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$,
$next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$,
$next^*(head, new)$, $prev^*(head, new)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$,
$next(new) = nil$, $prev(new) = nil$,
$head = prev(next(head))$, $head = next(prev(head))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = new$, $p = t$, $prev(new) = p$, $next^*(head, p)$, $next(p) = head$, $new = prev(head)$,
$prev^*(p, next(head))$, $n = head$, $prev(new) = head$, $prev^*(p, next(t))$, $n = t$.

```
 1: procedure LINUX-LIST-DEL(head, entry)
 2:     assume next*(next(head), head) ∧ prev*(prev(head), head)
 3:     assume prev*(head, next(head)) ∧ next*(head, prev(head))
 4:     assume next*(head, t) ∧ prev*(head, t)
 5:     assume next*(head, prev(t)) ∧ prev*(head, next(t))
 6:     assume next*(head, entry) ∧ prev*(head, entry)
 7:     assume next*(head, prev(entry)) ∧ prev*(head, next(entry))
 8:     assume t = prev(next(t)) ∧ t = next(prev(t))
 9:     assume head = prev(next(head)) ∧ head = next(prev(head))
10:     assume entry = prev(next(entry)) ∧ entry = next(prev(entry))
11:     assume ¬next(entry) = nil ∧ ¬prev(entry) = nil
12:     assume ¬entry = head
13:     p := prev(entry);
14:     n := next(entry);
15:     prev(n) := p;
16:     next(p) := n;
17:     next(entry) := nil;
18:     prev(entry) := nil;
19:     assert next*(next(head), head) ∧ prev*(prev(head), head)
20:     assert prev*(head, next(head)) ∧ next*(head, prev(head))
21:     assert ¬next*(head, entry) ∧ ¬prev*(head, entry)
22:     assert ¬next*(head, prev(entry)) ∧ ¬prev*(head, next(entry))
23:     assert next(entry) = nil ∧ prev(entry) = nil
24:     assert head = prev(next(head)) ∧ head = next(prev(head))
25:     assert ¬entry = prev(next(entry)) ∧ ¬entry = next(prev(entry))
26:     assert ¬entry = head
27:     assert (   next*(head, t) ∧ prev*(head, t) ∧ t = prev(next(t)) ∧ t = next(prev(t))
                 ∧ ¬entry = t ∧ next*(head, prev(t)) ∧ prev*(head, next(t)))
              ∨ (   ¬next*(head, t) ∧ ¬t = prev(next(t)) ∧ ¬t = next(prev(t)) ∧ entry = t
                 ∧ ¬prev*(head, t) ∧ ¬next*(head, prev(t)) ∧ ¬prev*(head, next(t)))
28: end procedure
```

Figure C.24: **LINUX-LIST-DEL** removes a node from a cyclic doubly-linked list. Predicates used to verify the example:
$next^*(next(head), head)$, $prev^*(prev(head), head)$, $next^*(head, t)$, $prev^*(head, t)$,
$t = prev(next(t))$, $t = next(prev(t))$, $next^*(head, entry)$, $prev^*(head, entry)$,
$prev^*(head, next(head))$, $next^*(head, prev(head))$, $next(entry) = \text{nil}$, $prev(entry) = \text{nil}$,
$head = prev(next(head))$, $head = next(prev(head))$,
$entry = prev(next(entry))$, $entry = next(prev(entry))$,
$next^*(head, prev(entry))$, $prev^*(head, next(entry))$,
$next^*(head, prev(t))$, $prev^*(head, next(t))$,
$t = entry$, $entry = head$, $next(entry) = n$, $prev(entry) = p$, $n = head$, $prev(n) = p$, $n = t$,
$next(head) = entry$, $p = t$.