An Inference-Rule-Based Decision Procedure for Verification of Heap-Manipulating Programs with Mutable Data and Cyclic Data Structures*

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

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

Abstract. Research on the automatic verification of heap-manipulating programs (HMPs) — programs that manipulate unbounded linked data structures via pointers — has blossomed recently, with many different approaches all showing leaps in performance and expressiveness. A year ago, we proposed a small logic for specifying predicates about HMPs and demonstrated that an inference-rule-based decision procedure could be performance-competitive, and in many cases superior to other methods known at the time. That work, however, was a proof-of-concept, with a logic fragment too small to verify most real programs. In this work, we generalize our previous results to be practically useful: we allow the data in heap nodes to be mutable, we allow more than a single pointer field, and we add new primitives needed to verify cyclic structures. Each of these extensions necessitates new or changed inference rules, with the concomitant changes to the proofs and decision procedure. Yet, our new decision procedure, with the more general logic, actually runs as fast as our previous results. With these generalizations, we can automatically verify many more HMP examples, including three small container functions from the Linux kernel.

1 Introduction

Heap-manipulating programs (HMPs) are programs that access and modify linked data structures consisting of an unbounded number of uniform *heap nodes*. They are a somewhat idealized model of programs with dynamic memory allocation, and given that most real software applications use dynamic memory allocation, they are an important frontier for software verification.

Research on verification of HMPs has blossomed recently, with over a dozen papers published in the past year alone, and many different approaches showing incredible progress. For example, automatically verifying the sortedness of applying bubble sort to a singly-linked list required well over 4 minutes of runtime for a state-of-the-art approach a year and a half ago [25], whereas by a year ago, we could verify sortedness (and no memory leaks or cycles) in less than 2 minutes [2]. Verifying no leaks or cycles

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

(but not sortedness) took us only 11.4 seconds, but this verification could be done in a mere 0.08 seconds a half year later! [29] While one may quibble about details when comparing performance results in this research area (e.g., machine speeds vary slightly, many papers do not report exact run times or the precise property being verified, amount of human effort is hard to quantify, etc.), the overall trend of rapid advancement is clear. Numerous approaches are now efficient enough to be potentially practically relevant.

Given the large amount of related work, we provide here only a very crude sketch of the research milieu surrounding our work. We can roughly group most work on HMP verification into three broad categories: shape analysis based on abstract interpretation [13], deductive verification using classical Floyd-Hoare-style pre- and postconditions [36] augmented with a specialized logic for heap structures, or model checking [37] using predicate abstraction [15] to deal with the infinite state space.

Perhaps most widely known is the shape analysis work, epitomized by the TVLA system [31]. As the name implies, a major strength of these approaches is in the analysis of the shape of heap structures, and they are able to handle shapes, like trees, that most other approaches cannot. Data, on the other hand, is commonly abstracted away, e.g., the impressively fast 0.08 second verification cited above ignores data in heap nodes. Earlier shape analysis work also required user assistance to specify "instrumentation predicates" and how they are affected by updates. More recent work has improved precision (e.g., [33]) and automation (e.g., [29]).

The deductive approach to verifying HMPs is the most venerable, dating back to Nelson's pioneering work [10]. Nelson was working with first-order logic, imposing a penalty in both performance and manual effort. Much more recently, PALE [18] is based on the weak, monadic, second-order logic of graph types, which is a decidable logic for which the MONA decision procedure [30] exists. Unfortunately, the complexity is non-elementary, so the decision procedure must be used with care. Separation logic [27] is apparently the key to much greater efficiency, with recent results reporting fast verification times (e.g., [6]) and interprocedural scalability [4]. A decidable fragment of separation logic is also known [5]. Deductive approaches typically require manual effort, particularly to specify loop invariants, but recent work is addressing that problem as well (e.g., [26, 28]).

Model checking, on the other hand, has always emphasized full automation, including automatic computation of invariants via fixpoints, and great precision. Model checking has revolutionized hardware verification, and with the use of predicate abstraction, has started to impact software verification as well (e.g., [15, 16, 32, 17]). Predicate abstraction conservatively abstracts a program into a Boolean program whose state space is the truth valuations of a finite set of predicates over the concrete program state. Once the predicates are specified, the method runs fully automatically. (In this paper, we do not consider heuristics for discovering predicates.) To verify HMPs, we therefore need a logic for specifying predicates about the heap state. Furthermore, to compute abstract pre- or post-images, the decision procedure for the logic must be extremely fast, since most predicate abstraction approaches make numerous queries to the decision procedure. Dams and Namjoshi were the first to explore this approach, but not having a decision procedure for their logic, they had to rely on manual guidance to assure termination [14]. Balaban et al. proposed a simple logic and small-model-theorem-based decision procedure, and demonstrated the feasibility and promise of this approach [7]. Alternatively, Lahiri and Qadeer proposed first-order axioms for their heap properties and used a first-order prover [23]. In both works, the decision procedure was a major bottleneck, and performance was substantially worse than the more established approaches. We were inspired by these pioneering works and created a simple logic and novel decision procedure that demonstrated that an approach based on model checking and predicate abstraction could be performance competitive, and often superior, to other methods available at the time [1, 2].[3] (Other recent promising logics for the predicate-abstraction-based approach include [21] and [34], but no decision procedures are available yet.)

In addition to the fast run times and low memory usage, another feature of our approach was the architecture of the decision procedure. Rather than being based on a small model theorem, it fires inference rules until saturation, backtracking as needed. Such a decision procedure promises several potential benefits: it simplifies integration into a combined satisfiability-modulo-theories solver; it suggests the ability to generate proofs automatically, which could be checked for higher assurance; and proof-generation suggests the possibility of computing interpolants, which have demonstrated enormous potential for improving model-checking efficiency [35]. Accordingly, there is value in pursuing an inference-rule-based decision procedure for HMP verification, as long as the performance is adequate, which it is.

Unfortunately, our previous work was only a proof-of-concept. The logic we proposed is too simplistic: data in heap nodes was not allowed to change, we could not specify important properties about cyclic lists, and heap nodes had only a single pointer field. These restrictions eliminated the vast majority of real programs from consideration.

Contributions: This paper expands and generalizes our previous, preliminary results to be practically useful:

– The new logic and decision procedure allow data stored in heap nodes to be mutable. With this extension, our method can in principle model any operations on data to full bit-accuracy. (In practice, of course, data fields will be downsized as much as possible, as is typical in model checking.) Changing the logic to allow data updates necessitated discovering and adding four new inference rules to the decision procedure.
– We now allow a finite number of pointer fields per heap node. This is needed by all but the most simplistic data structures. This change required all inference rules to be parameterized over the pointer fields, and the proofs must consider interacting constraints arising from the different points-to relations.
– To support cyclic data structures (e.g., cyclic singly- and doubly-linked lists), we added a generalized, ternary transitive closure between operator $\mathsf{btwn}_f(x,y,z)$, similar to Nelson's [10]. While the idea of such an operator is not new, how to support such an operator in an inference-rule-based decision procedure is completely new.

[3] The published paper has some minor errors, which are corrected in the technical report [2]. The technical report gives run times for the corrected algorithm, which are also much faster than in the paper, due to an improved implementation.

This was the most difficult change to our decision procedure, requiring the addition of 14 new inference rules, most of which are quite complicated.
– Despite the vastly increased complexity of the inference rule set, the essential structure of the decision procedure remained unchanged — the basic approach is still empirically very efficient. In fact, with continuing improvements to the implementation, performance actually improved slightly.
– The additional inference rules did greatly complicate the theoretical underpinnings of our approach. We report some theoretical results for our new logic and decision procedure: our decision procedure is sound and always terminates, and the decision procedure is complete for the fragment of the logic without updates. (In practice, completeness was not an issue, as we could verify all examples that we could specify.) The statements of the theorems are completely analogous to our previous work (e.g., "The decision procedure is sound."), but the proofs had to be completely reworked to account for the greater complexity of the expanded logic.

Overall, the contributions in this paper enable us to very efficiently verify a much larger variety of HMPs, including three small container functions from the Linux kernel.

2 Review of Our Previous Logic and Decision Procedure

To make this paper self-contained, we briefly review our original, simple logic and the proof-of-concept decision procedure. Details are in the published paper and technical report [1, 2].

One of the most fundamental concepts for verifying HMPs is unbounded *reachability* (a.k.a. *transitive closure*) between nodes, i.e., can one follow pointers from node x to node y. Several papers have previously identified the importance of transitive closure for HMPs, e.g., [9–12, 7, 23, 38]. Unfortunately, adding support for transitive closure to even simple logics often yields undecidability [12], hence our decision to start with a minimal logic and add features as needed to verify real examples.

In particular, the logic we originally proposed in [1] is as minimal as imaginable while usable to verify some non-trivial HMPs using predicate abstraction. Fig. 1 shows

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

Fig. 1. Our original, simple transitive closure logic [1]. v is any of a finite set of node variables that point to heap nodes. b is any of a finite set of Boolean variables that model data not contained in heap nodes. Each heap node has a finite set of data fields D, each able to hold a Boolean value, and $d \in D$. These model data contained in a heap node, with whatever precision is desired. There is a single pointer field f in each heap node, which points to another heap node. The term $f(x)$ denotes the heap node reached by following the f pointer from node x. Similarly, the atom $d(x)$ denotes the content of data field d in node x. Transitive closure is specified with $f^*(x,y)$, which denotes whether node x reaches node y by following 0 or more f pointers. The decision procedure decides satisfiability of conjunctions of literals.

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

Fig. 2. Inference rule example. This is a typical inference rule from the decision procedure. Above the line are antecedents; below the line are consequents. This rule says that if we get to node y by following one f pointer from node x, and if we can get from x to z by following 0 or more f pointers, then we conclude that $x = z$ or that we can get from y to z by following 0 or more f pointers.

the logic. While there can be an arbitrary amount of data, allowing modeling with bit-accurate precision, there is only a single pointer field, with a single transitive closure operator, which greatly restricts the heap properties that could be specified.

To specify the effect of program assignments that modify pointers in the heap, i.e., modify f, we need to be able to specify a transition relation between the old and new values of f. Accordingly, for each assignment of the form $f(\tau_1) := \tau_2$, we allow the user to specify a pointer function symbol f' that represents the value of f after the assignment. The semantic relationship between f and f' is

$$f' = \mathsf{update}(f, \tau_1, \tau_2) \tag{1}$$

Our decision procedure implicitly constrains f and f' appropriately, which is previous work. However, our original logic did not have the analogous constructs to allow heap data to be modified.

Conjunction and disjunction are conspicuous by their absence. The decision procedure decides satisfiability of a conjunction of literals. The satisfiability of a conjunction of predicates is the fundamental operation in computing the abstract pre- or post-image operators in predicate abstraction, potentially being called an exponential number of times per image, so we designed the decision procedure for that problem. We would handle a general formula with disjunctions by going to DNF and checking satisfiability of each disjunct separately.

The decision procedure is based on applying inference rules (IRs). Viewed from a high level, the decision procedure repeatedly searches for an applicable IR, applies it (i.e. adds one of its consequents to the set of literals), 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, it returns that the set of literals is *satisfiable*. Fig. 2 shows one sample inference rule. The decision procedure for our original logic has 17 inference rules, some of which are parameterized.

3 New Extensions to Logic and Decision Procedure

Our previous work was proof-of-concept: HMP verification based on model-checking and predicate abstraction could be performance competitive with other approaches, thanks to our efficient, inference-rule-based decision procedure. But our simplistic logic was too inexpressive for all but a few examples.

This paper addresses that problem. In the following subsections, we describe three extensions to our original logic and decision procedure. These extensions are absolutely indispensable for verifying a wide range of real programs. For each extension, we give a short example illustrating typical program constructs that motivated the extension, and then present how we changed the logic and decision procedure. The BNF for the extended logic is provided in Fig. 3.

3.1 Mutable Data Fields

Fig. 4 presents a simple example of a procedure that mutates data fields. The procedure sets the values of the data field of all nodes in the non-empty acyclic singly-linked input list *head* to `true`. Necessary assumptions are formalized by the **assume** statement on line 2 of the program. The body of the procedure is simple; it traverses the list, and on line 5 assigns `true` to the data field d at each iteration. The specification is expressed by the **assert** statement on line 8, and indicates that whenever line 8 is reached, *head* must point to an acyclic singly-linked list with data field d of all nodes set to `true`.

Assignments that modify a data field $d \in D$ have the general form $d(\tau) := b$, where τ is a term, and b is a data variable. Line 5 of the HMP of Fig. 4 is an example of such assignment. In order to be able to handle data mutations, for each data assignment we allow the user to introduce a data function symbol d' that represents d after the assignment. The semantic relationship between d and d' is

$$d' = \mathsf{update}(d, \tau, b) \tag{2}$$

Our decision procedure implicitly enforces the constraint (2) when it encounters the symbols d and d'. We accomplished this through the additional set of inference rules that capture the effects of a data field update. Fig. 5 presents these rules, and for example PRESERVEVALUE ensures the data values of nodes that are not equal to τ are preserved.

3.2 Cyclicity

We illustrate the extension for supporting cyclic lists with an example called INIT-CYCLIC in Fig. 6. The procedure takes a node *head* that points to a cyclic list and sets the data fields of all nodes in the list to `true`. Necessary assumptions are again formalized by the **assume** statement on line 2 of the program. In the predicates required for the verification of this example, 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. The fact that *head* is reachable from $f(head)$ enforces the cyclicality assumption. The body of INIT-CYCLIC is straightforward. First,

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

Fig. 3. The syntax of our new logic. Aside from the addition of the important new btwn atom, the pointer function symbol f now ranges over a *set* of names F.

the data field of *head* is set to true on line 4. 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 9, and indicates that whenever line 9 is reached, data fields of all nodes in the list have to be set to true.

Cyclic lists are commonly used data structures, and therefore supporting cyclicity is very important. In our experience and others' [10, 24], expressing "betweenness" is often necessary to construct invariants to verify cyclic list HMPs. For example, in order to prove the assertion on line 9 of INIT-CYCLE, the predicate abstraction engine must be able to construct an appropriate loop invariant (i.e. at line 5). This invariant must be strong enough to imply that all nodes x lying between *head* and *curr* on the cyclic list have $d(x) = \mathsf{true}$. It is not hard to show that our base logic of Sect. 2 is not capable of expressing this.

To solve this deficiency, we have added a generalized, ternary transitive closure between predicate $\mathsf{btwn}_f(x,y,z)$ to our logic, similar to Nelson's [10]. Formally, the interpretation of a between atom is defined as follows: 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 $\tau_2 = f^{n_0}(\tau_1)$, $\tau_3 = f^{m_0}(\tau_1)$, $n_0 \leq m_0$, and for all n, m such that $\tau_2 = f^n(\tau_1)$, $\tau_3 = f^m(\tau_1)$, we have $n_0 \leq n$ and $m_0 \leq m$.

While the idea of such a construct is not new, how to support it in an inference-rule-based decision procedure is completely new. This was also the most difficult extension of our decision procedure, requiring the addition of 14 new inference rules presented in Fig. 7, most of which are quite involved. For instance, 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. Furthermore, the introduction of the between atom broke our soundness and completeness results from the previous paper, and we had to completely redo all of our proofs. We give the intuition behind our new theoretical results in Sect. 4, while the complete proofs are presented in the technical report [3].

3.3 Multiple Pointer Fields

Fig. 8 shows a list container procedure LINUX-LIST-DEL from the Linux kernel. It illustrates the need for both multiple pointer fields and cyclic lists. The procedure takes a node *entry* and removes it from a cyclic doubly-linked list. Each node in the list has

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

Fig. 4. INIT-LIST initializes the data fields of an acyclic singly-linked list. In the **assume** and **assert** statements, variable t represents an arbitrary node (see Sect. 5).

two pointer fields: a *prev* and a *next* pointer. The body of the procedure is simple; it connects the *prev* and *next* pointers of *entry*'s neighbors, thus removing *entry* from the list. The assumptions and specifications for this example are quite involved and are given in our technical report [3].

Cyclic doubly-linked lists are widely used data structures. For instance, they are commonly used in kernels, such as the Linux kernel from where this example was taken. Handling multiple pointer fields is theoretically hard; it is a well-known result that unrestricted use of reachability in the presence of only two pointer fields is undecidable [12]. We therefore had to take special care in defining our extension. It turns out that if each individual reachability operator only refers to a single pointer field and there are no quantifiers, the decidability results still hold. This restriction prevents us from, e.g., expressing transitive closure in a tree, since that would require formulas like $(\mathit{left} \vee \mathit{right})^*(\mathit{root}, \mathit{leaf})$. However, we can still handle doubly-linked lists and similar structures.

On the logic side, this extension is reflected in symbol f being an element of a set of pointer function symbols F, rather than a single pointer function symbol (see Sect. 2). Our extended decision procedure supports for multiple pointer fields by instantiating the inference rules for each pointer field. In a sense, the decision procedure processes each field as a separate theory, and interaction between these theories is limited to communication of deduced term equalities and disequalities.

4 Correctness of the Decision Procedure

In this section, we will give the soundness and completeness theorems that show the correctness of our decision procedure. The detailed proofs of all theorems and more formal presentation of the decision procedure can be found in the technical report [3].

We'll start with noting that the problem our decision procedure solves is NP-hard, hence a polytime algorithm is unlikely to exist.

Theorem 1. *Given a set of literals Φ, the problem of deciding if Φ is satisfiable is NP-hard.*

Theorem 1 still holds when Φ contains no pointer function updates, no btwn predicates, no data fields, and only mentions a single pointer function f; hence it even applies to our simplistic original logic [1].

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

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

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

The following theorem tells us that if iterative application of the IRs in the decision procedure yields a contradiction, then we can conclude that the original set of literals is unsatisfiable.

Theorem 2. *The inference rules of Fig. 5, Fig. 7, Fig. 9, and Fig. 10 (see Appendix A) are sound.*

The proof proceeds by arguing in turn that each of the IRs given in the figures is sound.

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

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

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

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

Thanks to Theorem 3, our decision procedure can without loss of generality assume that Φ is normal. Let us call a set of literals Φ *consistent* if it does not contain a contradiction, and call Φ *closed* if none of the IRs of Fig. 7 and Fig. 9 are applicable. Our completeness theorem may then be stated as follows.

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

The proof of Theorem 4 is quite technical, and involves reasoning about the dependencies between digraphs of partial functions and the digraphs of their transitive closures.

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 4, it may correctly return *satisfiable*. We still don't have a proof that the procedure is complete when its input includes a data or pointer field update. Fortunately, not having such a theorem *does not* compromise the soundness of verification by predicate abstraction. In practice, in our experiments of Sect. 5, we never found any property violations caused by the extended decision procedure erroneously concluding that a set of literals was satisfiable.

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

Fig. 6. INIT-CYCLIC sets data fields of all nodes in a cyclic list to true. Additional predicates required for the verification: $curr = head$, $curr = f(head)$, $\mathsf{btwn}_f(curr,t,head)$, $t = head$, $\mathsf{btwn}_f(head,t,curr)$, $f^*(t,curr)$.

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

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

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

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

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

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

$$\frac{\mathsf{btwn}_f(v,u,x) \quad \mathsf{btwn}_f(v,u,y) \quad \mathsf{btwn}_f(u,x,y)}{\mathsf{btwn}_f(v,x,y)}\textsc{Btw12} \qquad \frac{\mathsf{btwn}_f(x,y,z) \qquad \neg x=z}{\mathsf{btwn}_{f'}(x,y,z) \quad \begin{array}{l} \mathsf{btwn}_f(x,\tau_1,z) \\ \neg \tau_1=z \end{array}}\textsc{UpdBtwn}$$

Fig. 7. Between inference rules. Here x, y, z, etc. range over variables V and $f \in F$ ranges over pointer fields. UpdBtwn enforces the implicit constraint $f' = \mathtt{update}(f, \tau_1, \tau_2)$, where τ_1 and τ_2 are variables (see Sect. 2).

Theorem 5. *The decision procedure always terminates.*

The theorem follows from the fact that none of the IRs create new terms, and there is only a finite number of possible literals that one could add given a fixed set of terms.

Our soundness, completeness, and termination results given in this section also ensure that the logic without pointer and data field updates is decidable. Furthermore, we believe that our logic with updates is subsumed by the slightly more general decidable logic presented in [7], and therefore also decidable.

5 Experimental Results

We ran our experiments using the new decision procedure[4] in the same verification setup as before [2]: a straightforward implementation of model checking with predicate abstraction. Once the predicates are specified, everything is fully automatic, including computation of most-precise abstract images and loop invariants.

Table 1 gives a baseline performance comparison on the same examples from our previously published work [2]. Table 2 gives results for the more than twice as many examples that we could not verify previously. We ran all experiments on a 2.6 Ghz Pentium 4 machine.

[4] The decision procedure is publicly available at `http://www.cs.ubc.ca/~zrakamar`

program	property	CFG edges	preds	DP calls	old time (s)	new time (s)
List-Reverse	NL	6	8	184	0.1	0.2
List-Add	$NL \wedge AC \wedge IN$	7	8	66	0.1	0.1
ND-Insert	$NL \wedge AC \wedge IN$	5	13	259	0.5	0.5
ND-Remove	$NL \wedge AC \wedge RE$	5	12	386	0.9	0.9
Zip	$NL \wedge AC$	20	22	9153	17.8	17.3
Sorted-Zip	$NL \wedge AC \wedge SO \wedge IN$	28	22	14251	23.4	22.8
Sorted-Insert	$NL \wedge AC \wedge SO \wedge IN$	10	20	5990	14.2	13.8
Bubble-Sort	$NL \wedge AC$	21	18	3444	11.4	11.1
Bubble-Sort	$NL \wedge AC \wedge SO$	21	24	31446	119.5	114.9

Table 1. Performance comparison against our previous work [2]. Although our extensions required adding several complex inference rules to the decision procedure, the running times stayed roughly the same: there was no practical performance penalty. “property” specifies the verified property; “CFG edges” is 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; “old time” is the total execution time from [2] (faster than [1]); “new time” is the total execution time using our new decision procedure.

The examples from Table 1 perform operations on acyclic singly linked lists — reverse, add elements, remove elements, sort, merge, etc. Therefore, we have been able to verify them without using the extensions described in this paper. The comparison supports our claim that although we greatly improved the expressiveness of the logic and therefore extended the decision procedure with a number of intricate inference rules, the practical running times haven’t changed.

Table 2 presents results of the experiments using examples that involve data field updates, cyclic lists, and doubly-linked lists. We could not handle them using the old logic and decision procedure. However, we have been successful in verifying them using the described new features added to our logic and decision procedure. These example programs are the following:

Remove-Elements – removes from a cyclic list elements whose data field is false.
Remove-Segment – removes the first contiguous segment of elements whose data field is true from a cyclic singly-linked list. This example is taken from a paper by Manevich et al. [24].

```
procedure Linux-List-Del(entry)
    $p := prev(entry)$;
    $n := next(entry)$;
    $prev(n) := p$;
    $next(p) := n$;
    $next(entry) :=$ nil;
    $prev(entry) :=$ nil;
end procedure
```

Fig. 8. Linux-List-Del is a standard function that removes a node from a cyclic doubly-linked list taken from a Linux kernel.

program	property	CFG edges	preds	DP calls	time(s)
REMOVE-ELEMENTS	NL $\wedge$ CY $\wedge$ RE	15	17	3062	8.8
REMOVE-SEGMENT	CY	17	15	902	2.2
SEARCH-AND-SET	NL $\wedge$ CY $\wedge$ DT	9	16	4892	5.3
SET-UNION	NL $\wedge$ CY $\wedge$ DT $\wedge$ IN	9	21	374	1.4
CREATE-INSERT	NL $\wedge$ AC $\wedge$ IN	9	24	3020	14.8
CREATE-INSERT-DATA	NL $\wedge$ AC $\wedge$ IN	11	27	8710	39.7
CREATE-FREE	NL $\wedge$ AC $\wedge$ IN $\wedge$ RE	19	31	52079	457.4
INIT-LIST	NL $\wedge$ AC $\wedge$ DT	4	9	81	0.1
INIT-LIST-VAR	NL $\wedge$ AC $\wedge$ DT	5	11	244	0.2
INIT-CYCLIC	NL $\wedge$ CY $\wedge$ DT	5	11	200	0.2
SORTED-INSERT-DNODES	NL $\wedge$ AC $\wedge$ SO $\wedge$ IN	10	25	7918	77.9
REMOVE-DOUBLY	NL $\wedge$ DL $\wedge$ RE	10	34	3238	24.3
REMOVE-CYCLIC-DOUBLY	NL $\wedge$ CD $\wedge$ RE	4	27	1695	15.6
LINUX-LIST-ADD	NL $\wedge$ CD $\wedge$ IN	6	25	1240	6.4
LINUX-LIST-ADD-TAIL	NL $\wedge$ CD $\wedge$ IN	6	27	1598	7.3
LINUX-LIST-DEL	NL $\wedge$ CD $\wedge$ RE	6	29	2057	24.7

Table 2. Results for HMPs that could not be handled in our previous work. "property" specifies the verified property; "CFG edges" denotes the number of edges in the control-flow graph of the program; "preds" is the number of predicates required for verification; "DP calls" is the number of decision procedure queries; "time" is the total execution time.

SEARCH-AND-SET – searches for an element with specified integer value in a cyclic singly-linked list, and initializes integer data fields of previous elements. Although this example uses merely 2-bit integers, it shows that our logic and decision procedure support any finite enumerated data type.

SET-UNION – joins two cyclic lists. This example is taken from a paper by Nelson [10].

CREATE-INSERT, CREATE-INSERT-DATA, CREATE-FREE – create new nodes (*malloc*), initialize their data fields, and insert them nondeterministically into a linked list. Also, remove nodes from a linked list and *free* them.[5]

INIT-LIST, INIT-LIST-VAR, INIT-CYCLIC – initialize data fields of acyclic and cyclic singly-linked lists, and set values of data variables.

SORTED-INSERT-DNODES – inserts an element into a sorted linked list so that sortedness is preserved. Every node in the linked list has an additional pointer to a node that contains a data field which is used for sorting.

REMOVE-DOUBLY – removes an element from an acyclic doubly-linked list.

REMOVE-CYCLIC-DOUBLY – removes an element from a cyclic doubly-linked list. This example is taken from a paper by Lahiri and Qadeer [23].

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

Our technical report [3] provides pseudocode and lists the required predicates for all examples.

The safety properties we checked (when applicable) of the HMPs are roughly:

[5] *malloc* and *free* are modelled as removing and adding nodes to an infinite cyclic list [20].

- *no leaks* (NL) – all nodes reachable from the head of the list at the beginning of the program are also reachable at the end of the program.
- *insertion* (IN) – a distinguished node that is to be inserted into a list is actually reachable from the head of the list, i.e. the insertion “worked”.
- *acyclic* (AC) – the final list is acyclic, i.e. nil is reachable from the head of the list.
- *cyclic* (CY) – list is a cyclic singly-linked list, i.e. the head of the list is reachable from its successor.
- *doubly-linked* (DL) – the final list is a doubly-linked list.
- *cyclic doubly-linked* (CD) – the final list is a cyclic doubly-linked list.
- *sorted* (SO) – list is a sorted linked list, i.e. each node’s data field is less than or equal to its successor’s.
- *data* (DT) – data fields of selected (possibly all) nodes in a list are set to a value.
- *remove elements* (RE) – for examples that remove node(s), this states that the node(s) was (were) actually removed. For the program REMOVE-ELEMENTS, RE also asserts that the data field of all removed elements is false.

Often, the properties one is interested in verifying for HMPs involve universal quantification over the heap nodes. For example, 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 introduce a Skolem constant t to represent a universally quantified variable [8, 7]. 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 program of Fig. 4, we express NL by conjoining $f^*(head, t)$ to the **assume** statement on line 2, and conjoining $f^*(head, t)$ to the assertion on line 8. Since (after the **assume**) t can be any node reachable from *head*, if the assertion is never violated, we have proven NL.

6 Future Work and Conclusions

We have introduced a logic for verifying HMPs that is expressive enough, and an inference-rule-based decision procedure for the logic that is efficient enough, to verify a wide range of small, but realistic programs. There are many directions for future research, some of which are outlined here.

We have found that even minimal support for universally quantified variables (as in the logic of Balaban et al. [7]) would allow expression of many common heap structure attributes. For example, the current logic cannot assert that two terms x and y point to disjoint linked lists; a single universally quantified variable would allow for this property (see Nelson [9, page 22]). We also found that capturing disjointedness is necessary for verifying that LIST-REVERSE always produces an acyclic list; hence we were unable to verify this property. We believe that our decision procedure can be enhanced to handle this case, either by introducing limited support for quantifiers, or by adding a new “disjoint predicate” with appropriate inference rules.

A broader expressiveness deficiency is the expression of more involved heap structure properties, such as for trees. Though our logic cannot capture “x points to a tree”, we believe that it is possible that an extension could be used to verify simple properties of programs that manipulate trees, for example that there are no memory leaks. It may also be possible to use techniques like structure simulation [22] or field constraint analysis [19], which use decidable logics to verify data structures originally beyond the scope of such logics (e.g., skip lists). We have run our decision procedure on some queries for MONA generated by the field constraint analysis tool Bohne [19], where we appear to be faster than MONA, but the queries have run so quickly on both tools that the comparison is meaningless.

We also plan on investigating how existing techniques for predicate discovery and more advanced predicate abstraction algorithms mesh with our decision procedure.

We have initial results showing the possibility of incorporating our decision procedure into a combined satisfiability-modulo-theories decision procedure and have started exploring such integration. We believe that by doing so, it would be possible to improve the precision of heap abstraction used by the existing software verification tools that employ theorem provers. We also plan to look into extending our decision procedure to generate proofs and interpolants.[6]

References

1. J. Bingham and Z. Rakamarić. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.
2. J. Bingham and Z. Rakamarić. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs, 2005. UBC Dept. Comp. Sci. Tech Report TR-2005-19, http://www.cs.ubc.ca/cgi-bin/tr/2005/TR-2005-19.
3. Z. Rakamarić, J. Bingham, and A. J. Hu. A Better Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs, 2006. UBC Dept. Comp. Sci. Tech Report TR-2006-02, http://www.cs.ubc.ca/cgi-bin/tr/2006/TR-2006-02.
4. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *Static Analysis Symposium (SAS)*, 2006.
5. J. Berdine, C. Calcagno, and P. W. O'Hearn. A Decidable Fragment of Separation Logic. In *Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2004.
6. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, 2006.
7. I. Balaban, A. Pnueli, and L. Zuck. Shape Analysis by Predicate Abstraction. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.
8. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Symp. on Principles of Programming Languages (POPL)*, 2002.
9. G. Nelson. Techniques for Program Verification. PhD thesis, Stanford University, 1979.
10. G. Nelson. Verifying Reachability Invariants of Linked Structures. In *Symp. on Principles of Programming Languages (POPL)*, 1983.
11. M. Benedikt, T. Reps, and M. Sagiv. A Decidable Logic for Describing Linked Data Structures. In *European Symposium on Programming (ESOP)*, 1999.
12. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The Boundary Between Decidability and Undecidability for Transitive Closure Logics. In *Workshop on Computer Science Logic (CSL)*, 2004.

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

13. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symp. on Principles of Programming Languages (POPL)*, 1977.
14. D. Dams and K. S. Namjoshi. Shape Analysis Through Predicate Abstraction and Model Checking. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2003.
15. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Conf. on Computer Aided Verification (CAV)*, 1997.
16. S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *Conf. on Computer Aided Verification (CAV)*, 1999.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Symp. on Principles of Programming Languages (POPL)*, 2002.
18. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2001.
19. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field Constraint Analysis. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2006.
20. T. Reps, M. Sagiv, and A. Loginov. Finite Differencing of Logical Formulas for Static Analysis. In *European Symposium on Programming (ESOP)*, 2003.
21. 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.
22. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via Structure Simulation. In *Conf. on Computer Aided Verification (CAV)*, 2004.
23. S. K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Symp. on Principles of Programming Languages (POPL)*, 2006.
24. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.
25. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction Refinement via Inductive Learning. In *Conf. on Computer Aided Verification (CAV)*, 2005.
26. D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.
27. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Symp. on Principles of Programming Languages (POPL)*, 2001.
28. S. Magill, A. Nanevski, E. M. Clarke, and P. Lee. Inferring Invariants in Separation Logic for Imperative List-processing Programs. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.
29. T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for Shape Analysis with Fast and Precise Transformers. In *Conf. on Computer Aided Verification (CAV)*, 2006.
30. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA Implementation Secrets. In *Conf. on Implementation and Application of Automata (CIAA)*, 2000.
31. T. Lev-Ami and M. Sagiv. TVLA: A System for Implementing Static Analyses. In *Static Analysis Symposium (SAS)*, 2000.
32. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2001.
33. G. Yorsh, T. Reps, and M. Sagiv. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.
34. S. Ranise and C. G. Zarba. A Theory of Singly-Linked Lists and its Extensible Decision Procedure. In *IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2006.

35. K. L. McMillan. Applications of Craig Interpolants in Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.
36. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.
37. 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.
38. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating Reachability using First-Order Logic with Applications to Verification of Linked Data Structures. In *Conf. on Automated Deduction (CADE)*, 2005.

A Inference Rules from Previous Work [1, 2]

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

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

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

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

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

Fig. 9. Basic inference rules. Here x, y, z, etc. range over variables V and $d \in D$ ranges over data fields. Note that $\textsc{Cycle}_k$ actually defines a separate rule for each $k \geq 1$.

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

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

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

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

Fig. 10. Pointer update inference rules. The rules are used to extend our logic to support a pointer function symbol 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 $f(\tau_1)$.