A Decision Procedure for Well-Founded Reachability

Shuvendu K. Lahiri Shaz Qadeer

April 20, 2007

Technical Report
MSR-TR-2007-43

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

A Decision Procedure for Well-Founded Reachability

Shuvendu K. Lahiri and Shaz Qadeer

Microsoft Research

Abstract. In earlier work, we introduced the logic of well-founded reachability for reasoning about linked data structures. In this paper, we present a rewriting-based decision procedure for the ground (quantifier-free) logic. We also extend the logic with restricted set constraints to allow specifications involving unbounded collections of objects. We have implemented this decision procedure within a satisfiability modulo theories (SMT) framework. Our implementation substantially improves the automation and the time taken for verifying our benchmarks compared to our earlier approach based on an incomplete axiomatization of well-founded reachability.

1 Introduction

First-order theorem provers [7, 3] have been a fundamental component of many scalable program verification tools [9, 2]. The theorem provers are primarily used for checking the validity of verification conditions characterizing the correctness of a program, or for computing abstractions during predicate abstraction [10]. The appeal of first-order reasoning comes from the ability to *combine* various useful theories required for program analysis e.g., arithmetic, arrays, and uninterpreted functions, in a systematic manner [17]. More recently, advances in Satisfiability-Modulo-Theories (SMT) solvers [23][1] have created the opportunity for scaling automated verification to deep properties of complex software.

Despite recent advances, automated verification of programs manipulating linked lists has remained outside the scope of first-order reasoning. Analysis of such programs typically requires a *reachability predicate* to capture the unbounded number of dynamically allocated cells present in a linked list. For a given cell u, the reachability predicate characterizes the set of cells $\{u, u.\mathtt{f}, u.\mathtt{f}.\mathtt{f}, \ldots\}$ reachable from u using the transitive closure of `f`. There are several reasons for the difficulty of reasoning about reachability using first-order logic. First, transitive closure cannot be expressed in first-order logic [5]; consequently, any first-order axiomatization of this predicate will be imprecise. Second, it is difficult to obtain a precise update to the reachability relation in response to an update to f (for a statement `x.f := y`). Finally, the reachability predicate by itself is not

[1] SMT solvers combine advances in Boolean satisfiability (SAT) solvers with powerful first-order theory reasoning, most recently with quantifiers

expressive enough to capture interesting properties lists viewed as collections, especially in the presence of cycles.

In earlier work, we proposed an alternate formulation of the reachability predicate known as the *well-founded reachability* [12]. We generalized the special cell *null* that terminates an acyclic list, to a set BS of *blocking* cells. The set BS is an auxiliary variable used by the programmer to indicate the head cells of cyclic lists. We enforce, by automatically instrumenting the program with simple assertions, that the program heap is well-founded with respect to every linking field `f` used to construct lists in the program, i.e., each cycle formed by `f` contains at least one cell from BS. We define a reachability relation R_f^{BS}, such that $R_f^{BS}(u, v)$ holds if and only if v is reachable, in zero or more steps using `f`, from u without visiting any cell in BS. We also define a function B_f^{BS}, such that $B_f^{BS}(u)$ is the first cell in BS encountered by following one or more steps using `f` from u. The well-founded reachability relation enjoys several useful properties. First, R_f^{BS} and B_f^{BS} allow us to express many interesting properties of both acyclic and cyclic lists uniformly. Second, the update to R_f^{BS} and B_f^{BS} in response to an update to `f` is a simple quantifier-free expression. This property guarantees that the verification condition of a program with quantifier-free assertions remains quantifier-free, thereby allowing efficient verification.

The R_f^{BS} predicate, similar to its traditional counterpart, cannot be expressed using first-order axioms. We showed that the satisfiability problem for the ground fragment over R_f^{BS} and B_f^{BS} is NP-complete [12], and provided a small-model theorem. However, using the small-model theorem to encode a ground formula into a Boolean formula and using a SAT solver did not yield an efficient decision procedure. Instead, we provided a small set of six quantified first-order axioms that sufficed for most of the representative linked list examples in the literature.

In this work, we provide an efficient decision procedure for the ground logic over the symbols R_f^{BS} and B_f^{BS}. The decision procedure is based on a set of rewrite rules that are shown to be sound, complete, and terminating (Section 2). Unlike previous ground decision procedures [20, 19], we show that the decision procedure is complete with respect to updates to `f` (Section 3). We also extend our logic to add constraints involving sets of cells, which allows us to specify rich specification without sacrificing decidability (Section 4). The logic can be combined with other first-order theories (such as arithmetic) using Nelson-Oppen combination. We provide a prototype implementation of the decision procedure within an SMT solver using first-order axioms that mimic the rewrite rules (Section 5). Finally, we report our experience using the decision procedure on a set of C verification benchmarks [6] that were earlier verified using the incomplete axiomatization(Section 6). Our results are encouraging: the new technique substantially improves the automation and the time taken for verifying our benchmarks compared to our earlier approach based on an incomplete axiomatization. We conclude with a discussion of the related work in Section 7. The proofs of the various theorems in the paper are given in the appendix.

2 Logic of well-founded reachability

In Figure 1, we present the syntax of the quantifier-free logic with signature $(BS, f, R, B, null)$. Since BS and f are clear from the context, we use R and B to represent R_f^{BS} and B_f^{BS} respectively.

$$
\begin{array}{lll}
x, y \in \mathit{Variable} \cup \{null\} & & \\
\quad l \in \mathit{Literal} & ::= & x = y \mid R(x,y) \mid BS(x) \mid f(x) = y \mid B(x) = y \mid \\
& & x \neq y \mid \neg R(x,y) \mid \neg BS(x) \\
\quad \varphi \in \mathit{Formula} & ::= & l \mid \varphi \vee \varphi \mid \varphi \wedge \varphi
\end{array}
$$

Fig. 1. Quantifier-free logic with signature $(BS, f, R, B, null)$

A *model* is a finite domain *Cell* together with interpretations $I_x : \mathit{Cell}$ for each variable in *Variable*, $I_f : \mathit{Cell} \to \mathit{Cell}$ for f, $I_{BS} : \mathit{Cell} \to \mathit{Boolean}$ for BS, $I_R : \mathit{Cell} \times \mathit{Cell} \to \mathit{Boolean}$ for R, $I_B : \mathit{Cell} \to \mathit{Cell}$ for B, and $I_{null} : \mathit{Cell}$ for *null* that satisfy the following properties:

1. $I_f(null) = I_{null}$.
2. If there exists $n > 0$ and elements $u_0, u_1, \ldots, u_n \in \mathit{Cell}$ such that $I_f(u_i) = u_{i+1}$ for all $0 \leq i < n$ and $u_0 = u_n$, then there exists i such that $0 \leq i < n$ and $I_{BS}(u_i)$.
3. I_R is the least fixpoint of the equation

$$X \;\equiv\; \lambda u, v \in \mathit{Cell}.\; u = v \vee (\neg I_{BS}(I_f(u)) \wedge X(I_f(u), v)).$$

4. I_B satisfies the equation

$$I_B \;\equiv\; \lambda u \in \mathit{Cell}.\; \mathit{let}\; v = I_f(u)\; \mathit{in}\; \mathit{ite}(I_{BS}(v),\; v,\; I_B(v)).$$

   Here the construct $\mathit{ite}(a,\ b,\ c)$ evaluates to b if a is true and evaluates to c otherwise.

In the rest of the paper, we assume (without any loss of generality) that any formula φ contains the literal $f(null) = null$ needed to model the first requirement on a model. For any such formula φ, we write $M \models \varphi$ to denote that the model M satisfies φ.

2.1 Decision procedure

We now present an algorithm that decides for any conjunction of literals φ, whether there exists a model M such that $M \models \varphi$. The algorithm maintains a *context*, which is a conjunction of literals currently asserted to be true. Our decision procedure is given as a collection of rewrite rules (Figure 2) that operate over the context. In each step of the algorithm, an applicable rewrite rule is applied which may cause a case-split together with the addition to the context

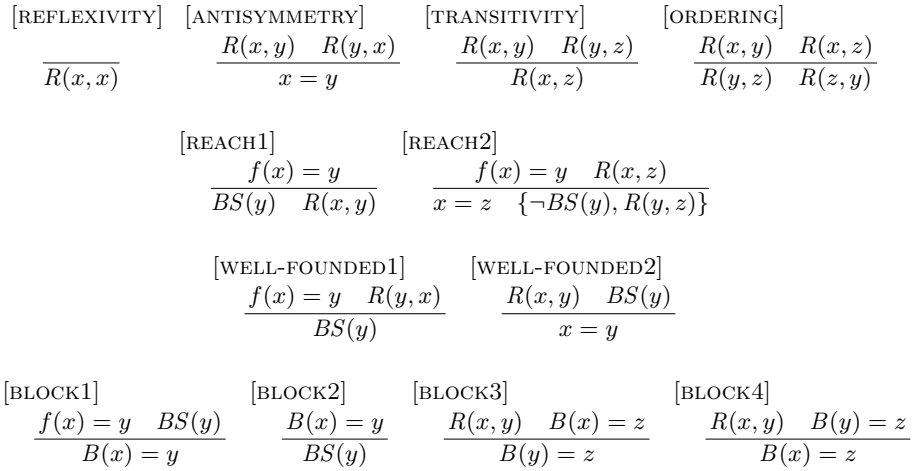

Fig. 2. Rewrite rules

of one or more literals. Figure 2 does not include the standard rules for equality propagation and congruence closure which are also needed by our algorithm.

Each rule in Figure 2 is written as a conjunction of antecedents above the line and a disjunction of consequents below the line. For the case of the rule [REACH2], the literals inside the {} are interpreted as conjoined. If there is a rule such that the current context contains all the literals above the line, then the properties of $(BS, f, R, B, null)$ guarantee that the disjunction of the literals below the line is entailed by the context. In this case, we say that the rule *matches* the context. For example, the rule [TRANSITIVITY] matches if both the literals $R(x, y)$ and $R(y, z)$ are present in the context and the literal $R(x, z)$ is added to the context when this rule is applied. A context is called *convex* if for every matching rule, the context contains all the literals in one of the disjuncts below the line. A context φ is *consistent* if it is not the case that $l \in \varphi$ and $\neg l \in \varphi$ for some literal l. A context is *inconsistent* if it is not consistent.

Our algorithm essentially explores a decision tree while maintaining a context. It initializes the context to the input formula φ. At each step, if the current context is inconsistent, the algorithm backtracks to the last untried decision if there remains one and otherwise returns unsatisfiable. Otherwise, if the current context is convex, the algorithm reports that φ is satisfiable. Otherwise, there is a matching rule such that none of the literals below the line are present in the context. If there is only one literal below the line, it is added to the context. Otherwise, a case split is performed with one literal added to the context for each case.

The rules [REFLEXIVITY], [ANTISYMMETRY], [TRANSITIVITY], and [ORDERING] capture basic properties of the reachability relation, independent of its connection with f and B. While the standard notion of reachability has the reflexivity,

transitivity and ordering properties, only well-founded reachability has the antisymmetry property. The rules [REACH1] and [REACH2] capture the connection between BS, f, and R. The rule [WELL-FOUNDED1] ensures that every cycle contains an element in BS. The rule [WELL-FOUNDED2] ensures that a blocking cell is reachable only from itself. The rules [BLOCK1] and [BLOCK2] establish the connection between BS, f, and B. The rules [BLOCK3] and [BLOCK4] state that the function B is congruent with respect to the R relation.

2.2 Termination

We argue that the size of the context remains bounded during the execution of the algorithm. The context consists of a collection of literals, each of which is a relation over terms. The only rules that may require creation of new terms are [BLOCK1], [BLOCK3], and [BLOCK4] and they only create new terms of the form $B(x)$ for some variable x.[2] Thus, the number of terms can at most double during the execution of the algorithm. Therefore, the number of possible distinct literals remains bounded as well. Bounded number of literals implies that the number of case splits performed by the algorithm is bounded. Hence, the algorithm terminates.

Due to the case-splitting, the complexity of our algorithm is clearly exponential in the size of the input formula. In earlier work [12], we showed that the problem of checking the satisfiability of a conjunction of literals in the logic in Figure 1 is NP-complete. Therefore, an algorithm with better than exponential-time complexity in the worst case is unlikely. As we show in Section 6, we have found that our algorithm performs well on our benchmarks.

2.3 Correctness

It is straightforward to argue that our algorithm is sound. We need to reason locally and prove that each deduction rule in Figure 2 is sound. To prove completeness, we must show that if a conjunction of literals φ is consistent, closed under congruence and convex under the rewrite rules in Figure 2, then there is a model M such that $M \models \varphi$. We now show how to construct such a model M from φ.

We create a partition $A = \{\alpha_1, \ldots, \alpha_n\}$ of the set of variables occurring in φ satisfying the following condition: for all $i \in [1, \ldots, n]$ and for all variables x and y, $x \in \alpha_i$ and $y \in \alpha_i$ iff $x = y \in \varphi$. For each $\alpha \in A$, let $[\![\alpha]\!]$ denote a fixed representative member of α. For each class $\alpha \in A$ such that $(B([\![\alpha]\!]) = x) \notin \varphi$ for all x, we add the literal $B([\![\alpha]\!]) = \mathit{null}$ to φ. Let the resulting conjunction of literals be called φ'. It is simple to argue that φ' is consistent, closed under congruence, and convex under the rewrite rules in Figure 2.

We now define a model M for φ' which will be a model for φ as well. Since the extra literals in φ' over φ do not include any equalities between variables,

[2] Also note, that the set of literals in the context is always in *purified* form [18], where each term is equated with some variable.

the partition A over the variables remains unchanged. Moreover, the set φ' has the property that for each equivalence class $\alpha \in A$, there is a unique equivalence class β such that $(B(\lVert\alpha\rVert) = \lVert\beta\rVert) \in \varphi'$.

To define M, we first need to define relations $\mathcal{R}$ and $\mathcal{F}$ as follows. Let $\mathcal{R} = \{(\alpha, \beta) \in A \times A \mid R(\lVert\alpha\rVert, \lVert\beta\rVert) \in \varphi\}$. Since φ is convex with respect to the rules [REFLEXIVITY], [ANTISYMMETRY], and [TRANSITIVITY], we get that $\mathcal{R}$ is a partial order. Let $\mathcal{F} \subseteq \mathcal{R}$ be a minimal relation over A whose reflexive-transitive closure is equal to $\mathcal{R}$. We prove by contradiction that $\mathcal{F}$ is functional, that is, for all α, β, and β', if $\mathcal{F}(\alpha, \beta)$ and $\mathcal{F}(\alpha, \beta')$, then $\beta = \beta'$. Suppose $\mathcal{F}(\alpha, \beta)$, $\mathcal{F}(\alpha, \beta')$, and $\beta \neq \beta'$. If either $\alpha = \beta$ or $\alpha = \beta'$, then the minimality of $\mathcal{F}$ is violated. Therefore, we have $\alpha \neq \beta$ and $\alpha \neq \beta'$. Then $R(\lVert\alpha\rVert, \lVert\beta\rVert) \in \varphi$ and $R(\lVert\alpha\rVert, \lVert\beta'\rVert) \in \varphi$. Since φ is convex, we get from rule [ORDERING] that either $R(\lVert\beta\rVert, \lVert\beta'\rVert) \in \varphi$ or $R(\lVert\beta'\rVert, \lVert\beta\rVert) \in \varphi$. We get $\mathcal{R}(\beta, \beta')$ in the first case, and $\mathcal{R}(\beta', \beta)$ in the second case. Both cases contradict the minimality of $\mathcal{F}$.
Defining I_x: For any variable x, let I_x be the equivalence class containing x.
Defining I_{BS}: For each equivalence class α, let $I_{BS}(\alpha) = \mathit{true}$ if $BS(\lVert\alpha\rVert) \in \varphi'$, and $I_{BS}(\alpha) = \mathit{false}$ otherwise.
Defining I_f: For each equivalence class α, we define $I_f(\alpha)$ as follows. If $f(\lVert\alpha\rVert) = \lVert\beta\rVert \in \varphi'$, then $I_f(\alpha) = \beta$. Otherwise, if $\mathcal{F}(\alpha, \beta)$ holds, then $I_f(\alpha) = \beta$. Otherwise, $I_f(\alpha) = \beta$ for the unique equivalence class β such that $B(\lVert\alpha\rVert) = \lVert\beta\rVert \in \varphi'$.
Defining I_R: Define $I_R = \mathcal{R}$.
Defining I_B: For each equivalence class α, we define $I_B(\alpha) = \beta$ for the unique class β such that $B(\lVert\alpha\rVert) = \lVert\beta\rVert \in \varphi'$.
Defining I_{null}: We define I_{null} to be the equivalence class containing *null*.

Theorem 1. *Let φ be a conjunction of literals that is consistent, closed under congruence, and convex under the rewrite rules in Figure 2. Let M be the tuple $(I_{BS}, I_f, I_R, I_B, I_{null})$ defined above. Then $M \models \varphi$.*

It also follows easily from the model constructed above that the logic is *stably infinite*, i.e., any quantifier-free satisfiable formula in this logic also has an infinite model. The stably infinite property allows us to combine this theory with other theories in the Nelson-Oppen cooperating theory framework.

Theorem 2. *The logic presented in Figure 1 is stably infinite.*

3 From programs to formulas

In Section 2, we described a logic with the signature $(BS, f, R, B, \mathit{null})$ and a procedure for deciding the satisfiability of formulas in that logic. We are interested in using this logic for verification of heap-manipulating programs. In such programs, the heap consists of objects, each of which might contain several fields. The fields of interest for this paper are those that are used to link together members of a list. Such a field (called a *linking* field) can be modeled using the symbol f.

When the field f of an object is updated by the program, we must update the reachability relation R accordingly. If all linked lists in the program are

terminated by *null*, then the new value of R can be defined *precisely* in first-order logic in terms of the old value of R. However, in the presence of cyclic linked lists, providing such a simple update to R is not possible. Thus, programs (e.g., operating systems) that use circular linked lists are difficult to analyze precisely and efficiently. We solve this problem by introducing the methodology of well-founded reachability. In this methodology, the programmer uses a reachability relation that is parameterized not only by the linking field f but also by a set of blocking cells BS. The symbol BS models an auxiliary variable in the program, updates to which must be explicitly provided by the programmer. The methodology ensures that the program invariant every cell in the heap reaches an element in the blocking set by following a finite number of applications of f. This variable is initialized to the singleton *null*. The program is required to add the the head cell of a cyclic linked list to this variable just before the cycle is created and remove an appropriate cell from the variable when two linked lists are spliced together. The main advantage of this approach is that we regain the ability to provide a simple first-order update to R when f is updated at some heap cell.

Since programs might use several linking fields to build lists, we augment the logic from the previous section to allow multiple linking fields from the set *LinkField* and multiple blocking sets from the set *BlockSet*. Instead of a single theory with the signature $(BS, f, R, B, null)$, we have a collection of theories $(BS, f, R_f^{BS}, B_f^{BS}, null)$. Note that these theories share only variables and the equality symbol since each linking field f is associated with its own blocking set BS. We also know that each of these theories is stably infinite (Theorem 2). Since they only share equality and variables, they can be combined in the framework of cooperating decision procedures [18].

3.1 Verification condition generation

$$
\begin{array}{lll}
S \in \mathit{Stmt} & ::= & \mathit{Assert}(e) \mid \mathit{Assume}(e) \mid x := f(y) \mid S_1; S_2 \mid S_1 \Box S_2 \mid \\
& & f(x) := y \mid \mathit{AddBlock}(BS, x) \mid \mathit{RemoveBlock}(BS, x) \mid \\
& & \{I\}\ \mathit{While}\ e\ S
\end{array}
$$

Fig. 3. Program

We now describe how a program S is translated into a verification condition φ such that if φ is unsatisfiable then S is correct. We explain the procedure of verification condition generation on programs constructed according to Figure 3. We now define $wp(S, \varphi)$, the weakest precondition of a formula φ with respect to a program S.

While the statement $\mathit{Assert}(e)$ is used to model intermediate assertions and postconditions, the statement $\mathit{Assume}(e)$ is used to model preconditions and conditional statements. We have $wp(\mathit{Assert}(e), \varphi) = e \wedge \varphi$ and $wp(\mathit{Assume}(e), \varphi) =$

$e \Rightarrow \varphi$. The statement $x := f(y)$ reads the value of field f at cell y into a variable x. We have $wp(x := f(y), \varphi) = \varphi[f(y)/x]$, where $\varphi[f(y)/x]$ is the formula in which x is syntactically replaced everywhere with $f(y)$. The statement $S_1; S_2$ evaluates S_1 followed by S_2 and we have $wp(S_1; S_2, \varphi) = wp(S_1, wp(S_2, \varphi))$. The statement $S_1 \Box S_2$ executes either S_1 or S_2 nondeterministically. This statement, together with the assume statement, is used to model conditional execution. We have $wp(S_1 \Box S_2, \varphi) = wp(S_1, \varphi) \wedge wp(S_2, \varphi)$.

The statements $f(x) := y$, $AddBlock(BS, x)$, and $RemoveBlock(BS, x)$ are the most interesting because they affect the values of R_f^{BS} and B_f^{BS}. The statement $f(x) := y$ updates the value of field f at cell x to y. To incorporate its effect on the R_f^{BS} and B_f^{BS}, it is desugared into the following statement sequence, denoted by S_1, and we get $wp(f(x) := y, \varphi) = wp(S_1, \varphi)$.

$$
\begin{array}{lcl}
Assert(R_f^{BS}(y,x) \Rightarrow BS(y)) \ ; & & \\
B_f^{BS} & := & \lambda\, u.\ ite(R_f^{BS}(u,x),\ ite(BS(y),\ y,\ B_f^{BS}(y)),\ B_f^{BS}(u)) \ ; \\
R_f^{BS} & := & \lambda\, u, v. \\
 & & ite(\ R_f^{BS}(u,x), \\
 & & \quad (R_f^{BS}(u,v) \wedge \neg R_f^{BS}(x,v)) \ \vee\ v = x \ \vee\ (\neg BS(y) \wedge R_f^{BS}(y,v)), \\
 & & \quad R_f^{BS}(u,v)\ )\ ; \\
f & := & \lambda\, u.\ ite(u = x,\ y,\ f(u))
\end{array}
$$

The first statement is an assertion ensuring that the heap remains well-founded. The next three statements capture the update to B_f^{BS}, R_f^{BS}, and f as lambda expressions. These expressions use the $ite(a,\ b,\ c)$ term, which can be easily eliminated in a post-processing step by introducing boolean connectives. The weakest precondition $wp(f := \lambda\, u.\ ite(u = x,\ y,\ f(u)), \varphi)$ is obtained by replacing every occurrence of a term $f(t)$ in φ by the term $ite(t = x,\ y,\ f(t))$. The weakest precondition of the other lambda expressions is calculated similarly. The desugaring for $AddBlock(BS, x)$ and $RemoveBlock(BS, x)$ is similar and given in Appendix B.

A loop $\{I\}$ *While* e S is first desugared into the following code:

$$Havoc; Assume(I); (Assume(e); S; Assert(I); Assume(false)) \Box Assume(\neg e)$$

The *Havoc* statement is a special statement whose effect is to update every program variable, including all the fields and the blocking sets, to nondeterministic values. The weakest precondition $wp(Havoc, \varphi)$ is obtained by replacing every occurrence of a variable x by a fresh variable x', every occurrence of f with a fresh f', every occurrence of BS with a fresh BS', every occurrence of R_f^{BS} with $R_{f'}^{BS'}$, and every occurrence of B_f^{BS} with $B_{f'}^{BS'}$.

Remark 1. The ability to provide quantifier-free updates to R_f^{BS}, BS and B_f^{BS} for the different statements in a program ensures that the formula generated by performing the $wp(S, \varphi)$, for a formula φ over a program S remains simple. In particular, if φ is a formula in the extended logic, then the resultant formula $wp(S, \varphi)$ also falls into the logic, and thereby can be decided efficiently.

4 Set constraints

For many programs, the reachability predicate by itself is insufficient for expressing the appropriate specification. For example, consider the following program.

```
class ListCell { int data; ListCell next; }
ListCell head;

iter = head;
while (iter != null) {
    iter.data = 42;
    iter = iter.next;
}
```

In this program, the variable `head` points to the beginning of a `null`-terminated list. We would like that when the loop terminates, the `data` field of every element in the list is 42. Proving such a postcondition also requires proving a loop invariant stating that every element reachable from `head` is either reachable from `iter` or its `data` field equals 42. Both of these assertions state a fact about an entire collection of heap cells; consequently, there is a universal quantifier hidden in them. While general universal quantification in first-order logic makes the satisfiability problem undecidable, we have observed that the most common and useful data structure specifications use a restricted form of quantification which can be viewed as expressing set constraints. Therefore, we extend the logic from last section in Figure 4 to allow such set constraints to be expressed.

$$
\begin{array}{rll}
i \in \mathit{Integer} & & \\
d \in \mathit{DataField} & & \\
c \in \mathit{DataVariable} & & \\
BS \in \mathit{BlockSet} & & \\
f \in \mathit{LinkField} & & \\
v \in \mathit{Variable} & & \\
x, y \in \mathit{Variable} \cup \{\mathit{null}\} & & \\
X, Y, Z \in \mathit{SetVariable} & & \\
t, u \in \mathit{DataTerm} & ::= i \mid c \mid d(x) \mid t + t \mid t - t & \\
\psi \in \mathit{SetFormula} & ::= x = y \mid R_f^{BS}(x, y) \mid BS(x) \mid t \leq u \mid & \\
 & \quad x \neq y \mid \neg R_f^{BS}(x, y) \mid \neg BS(x) \mid t < u & \\
m \in \mathit{SetLiteral} & ::= X(x) \mid \neg X(x) \mid X = \lambda v.\psi \mid X \subseteq Y \cup Z & \\
l \in \mathit{Literal} & ::= x = y \mid R_f^{BS}(x, y) \mid BS(x) \mid f(x) = y \mid B_f^{BS}(x) = y \mid & \\
 & \quad x \neq y \mid \neg R_f^{BS}(x, y) \mid \neg BS(x) & \\
\varphi \in \mathit{Formula} & ::= l \mid m \mid \varphi \vee \varphi \mid \varphi \wedge \varphi &
\end{array}
$$

Fig. 4. Set logic

The logic in Figure 4 makes two significant extensions to the logic from Figure 1. First, we also allow formulas to refer to data fields from the set *DataField*.

The interpretation of each function $d \in \mathit{DataField}$ is a function from *Cell* to *Integer*. Second, we allow formulas to refer to variables whose interpretation is a subset of *Cell*. These set variables appear in subset and equality constraints. This logic provides the set constructor $\lambda v.\psi$, where $\psi \in \mathit{SetFormula}$, to represent the set $\{v \in \mathit{Cell} \mid \psi(v)\}$. The bound variable v is a member of *Variable* and can never be equal to *null*. Therefore, *null* is a free variable in any formula. Note that the empty set, the universal set and other set constraints such as $X = Y \cup Z$, $X \cap Y = Z$, and $X = Y \setminus Z$ can all be expressed using the above syntax.

Using this logic, we write the precondition of the program above as $B^{BS}_{next}(\mathit{head}) = \mathit{null}$, the loop invariant as

$$\lambda x.(R^{BS}_{next}(\mathit{head}, x) \wedge \mathit{head} \neq \mathit{null}) \subseteq \begin{pmatrix} \lambda x.(R^{BS}_{next}(\mathit{iter}, x) \wedge \mathit{iter} \neq \mathit{null}) \\ \cup\ \lambda x.\mathit{data}(x) = 42 \end{pmatrix}$$

and the postcondition as

$$\lambda x.(R^{BS}_{next}(\mathit{head}, x) \wedge \mathit{head} \neq \mathit{null}) \subseteq \lambda x.\mathit{data}(x) = 42.$$

Let $FV(\varphi) \subseteq \mathit{Variable} \cup \{\mathit{null}\}$ be the set of variables occurring in φ that are not bound by a lambda term. To check whether a formula φ is satisfiable, we obtain a formula φ' by performing the following operations on φ.

1. Replace each occurrence of a set literal $X = \lambda v.\psi$ with the conjunction
$$\bigwedge_{x \in FV(\varphi)} X(x) \Leftrightarrow \psi[x/v].$$
2. Replace each occurrence of a set literal $X \subseteq Y \cup Z$ with the conjunction
$$\bigwedge_{x \in FV(\varphi)} X(x) \Rightarrow Y(x) \vee Z(x).$$

Since the formula ψ used in the set constructor $\lambda x.\psi$ does not refer to any linking fields, instantiating it on a variable does not result in any new terms of type *Cell*. As a result, the formulas φ and φ' are equisatisfiable. Furthermore, the formula φ' is in the combined logic of (i) the logic in Figure 1 for which we already have a decision procedure, and (ii) the decidable logic with signature (d, $+$, $-$, $\leq$, $<$) over the *DataField*. Since these logics have disjoint signatures, are stably infinite, we obtain a decision procedure for the combined logic.

Theorem 3. *The satisfiability problem for a conjunction of literals in the set logic of Figure 4 is NP-complete.*

It is easy to generalize our logic to support n-ary relations with subset constraints among them; a set is just the special case of a unary relation. The logic remains decidable since we can eliminate constraint relating n-ary relations by instantiating that constraint with all possible n-tuples of free variables.

Although our logic is expressive enough for most specifications, there are some specifications which cannot be expressed such as the following invariant

[REFLEXIVITY]	$\forall x : \{Element(x)\}\ R(x,x)$
[ANTISYMMETRY]	$\forall x, y : \{R(x,y), R(y,x)\}\ R(x,y) \wedge R(y,x) \Rightarrow x = y$
[TRANSITIVITY]	$\forall x, y, z : \{R(x,y), R(y,z)\}\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$
[ORDERING]	$\forall x, y, z : \{R(x,y), R(x,z)\}\ R(x,y) \wedge R(x,z) \Rightarrow R(y,z) \vee R(z,y)$
[REACH1]	$\forall x : \{f(x)\}\ BS(f(x)) \vee R(x, f(x))$
[REACH2]	$\forall x, z : \{f(x), R(x,z)\}\ R(x,z) \Rightarrow x = z \vee (\neg BS(f(x)) \wedge R(f(x), z))$
[WELL-FOUNDED1]	$\forall x : \{R(f(x), x)\}\ R(f(x), x) \Rightarrow BS(f(x))$
[WELL-FOUNDED2]	$\forall x, y : \{R(x,y), BS(y)\}\ R(x,y) \wedge BS(y) \Rightarrow x = y$
[BLOCK1]	$\forall x : \{BS(f(x))\}\ BS(f(x)) \Rightarrow B(x) = f(x)$
[BLOCK2]	$\forall x : \{B(x)\}\ BS(B(x))$
[BLOCK3]	$\forall x, y : \{R(x,y), B(x)\}\ R(x,y) \Rightarrow B(x) = B(y)$
[BLOCK4]	$\forall x, y : \{R(x,y), B(y)\}\ R(x,y) \Rightarrow B(x) = B(y)$

Fig. 5. Reachability axioms

of a doubly-linked list whose forward and backward linking fields are *next* and *prev* respectively.

$$\lambda x.R^{BS}_{next}(head, x) \subseteq \lambda x.(x = next(prev(x)) \wedge x = prev(next(x)))$$

Of course, our verification system described in Section 5, accepts such specifications as well. In practice, we have observed that the theorem provers used by our verifier is able to prove many such specifications; however termination is not guaranteed in general.

5 Implementation

The goal of our work is to enable the verification of program specifications that use the reachability predicate. To perform such a verification task for realistic programs, a decision procedure must be able to reason about not only the reachability predicate but also other theories such as arithmetic and propositional logic. Hence, to be useful for program verification, our decision procedure must be implemented in cooperation with decision procedures for other theories. Implementing a theory inside a satisfiability modulo-theory (SMT) framework is a considerable implementation which must ultimately be undertaken for good performance. To create an initial prototype, we instead chose to implement our decision procedure by encoding our rewrite rules using universally-quantified first-order axioms with appropriate matching *triggers*. Most SMT solvers support such axioms; our implementation is based on Simplify [7] and Z3.

The axioms encoding the rewrite rules in Figure 2 are given in Figure 5. To avoid the use of excessive parentheses, we use the convention that $\Rightarrow$ and $\Leftrightarrow$ have lower precedence than $\wedge$ and $\vee$. For each axiom, a set of triggers is specified using curly braces. Each trigger is a collection of terms enclosed within $\{\cdot\}$, which together must refer to all of the universally-quantified variables. The axiom is instantiated for those terms which if substituted for the quantified variables

$\forall x, S, T : \{In(x, Union(S,T))\}\{Union(S,T), In(x,S)\}\{Union(S,T), In(x,T)\}$
$\quad In(x, Union(S,T)) \Leftrightarrow In(x,S) \vee In(x,T)$
$\forall x, S, T : \{In(x, Intersection(S,T))\}\{Intersection(S,T), In(x,S), In(x,T)\}$
$\quad In(x, Intersection(S,T)) \Leftrightarrow In(x,S) \wedge In(x,T)$
$\forall x, S, T : \{In(x, Difference(S,T))\}\{Difference(S,T), In(x,S)\}$
$In(x, Difference(S,T)) \Leftrightarrow In(x,S) \wedge \neg In(x,T)$
$\forall S, T : \{Disjoint(S,T)\}\ Disjoint(S,T) \Leftrightarrow (\forall x : \{Element(x)\}\ \neg(In(x,S) \wedge In(x,T)))$
$\forall S, T : \{Subset(S,T)\}\ Subset(S,T) \Leftrightarrow (\forall x : \{Element(x)\}\ In(x,S) \Rightarrow In(x,T))$

Fig. 6. Set axioms

in the trigger terms result in terms that are all present in ground formulas. Typically, each rewrite rule results in an axiom in which the conjunction of the literals above the line implies the disjunction of the literals below the line and the terms in the literals above the line appear in the trigger. However, literals such as $f(x) = y$ above the line are eliminated by adding a term $f(x)$ to the trigger and substituting $f(x)$ for y below the line. The trigger for the axiom [RELEXIVITY] is explained below.

In addition to axioms for reachability, we also need axioms for set constraints. To write these axioms, we introduce two uninterpreted types *Cell* and *Set* and the following uninterpreted functions, essentially acting as set constructors.

$$Union : Set \times Set \rightarrow Set$$
$$Intersection : Set \times Set \rightarrow Set$$
$$Difference : Set \times Set \rightarrow Set$$

In addition, we introduce the following uninterpreted predicates that express relationships among sets and set elements.

$$In : Cell \times Set \rightarrow Boolean$$
$$Disjoint : Set \times Set \rightarrow Boolean$$
$$Subset : Set \times Set \rightarrow Boolean$$

The axioms constraining these functions and predicates are given in Figure 6.

We also allow sets to be defined using the lambda notation $X = \lambda x.\psi$. These set variables can then be used in the specification. The set and the definition are related by adding constraints $\forall x : \{Element(x)\}\ In(x, X) \Leftrightarrow \psi$ to the resulting formula.

Finally, we also introduce a predicate $Element : Cell \rightarrow Boolean$, which provides a mechanism for informing the theorem prover of terms that are potentially set elements. A few of the axioms constraining *Element* that we have found sufficient for our examples are:

$$\forall x : \{Reach(x)\}\ Element(x)$$
$$\forall x : \{BS(x)\}\ Element(x)$$
$$\forall x : \{B(x)\}\ Element(x) \wedge Element(B(x))$$
$$\forall x, S : \{In(x,S)\}\ Element(x)$$

Example	Old Time (s)	New Time (s)
`iterate`	1.7	1.4
`iterate_acyclic`	1.7	1.5
`array_iterate`	1.3	1.3
`slist_add`	1.4	1.3
`reverse_acyclic`	1.7	1.4
`slist_sorted_insert`	14.2	3.1
`dlist_add`	32.6	7.1
`dlist_remove`	37.9	2.4
`allocator`(1)	901.8	563.2
`allocator`(2)	*	2421
`init`	*	15.86
`removeList`	*	110.95

Fig. 7. Results of assertion checking. SIMPLIFY was used as the theorem prover. The experiments were conducted on a 3.6GHz, 2GB machine running Windows XP. A timeout (indicated by *) of 5000 seconds was set for each experiment.

Note that the term $Element(x)$ is part of the trigger for the axioms for [REFLEXIVITY] in Figure 5, defining sets using lambda expressions, and the disjointness and subset axioms in Figure 6.

As mentioned earlier, there are many advantages to implementing a rewriting-based decision procedure using first-order axioms. However, we would like to note that such an approach does have some drawbacks which must be addressed eventually. First, matching in typical SMT solvers is expensive. Second, matching is performed purely on the basis of certain terms being present in the ground formulas, which is not the desired behavior for many axioms in Figure 5. For example, we want the [TRANSITIVITY] axiom to be instantiated only if both $R(x, y)$ and $R(y, z)$ are asserted to be *true* in the current context. However, the matcher will instantiate these axioms even otherwise and introduce an unnecessary case-split.

6 Evaluation

We have implemented the decision procedure described in this paper in the tool HAVOC [6]. HAVOC is a tool for checking properties of heap-manipulating C programs. The tool generates a formula representing the verification condition that is checked for validity using a first-order theorem prover. The logic and the decision procedure presented in this paper had to be suitably extended to account for low-level C operations such as pointer arithmetic and internal pointers. The set of rewrite rules for the decision procedure are very similar to the rules in Figure 2; the main differences being that fields are modeled as offsets within an object and the reachability relation is defined with respect to these offsets [6].

Figure 7 presents a set of C benchmarks that manipulate singly and doubly linked lists. These benchmarks use pointer arithmetic, internal pointers into objects and cast operations in addition to linked data structures. The examples `iterate` and `iterate_acyclic` respectively initialize the data elements of a

cyclic and acyclic lists respectively; `slist_add` adds a node to a singly linked list; `reverse_acyclic` is a routine for in-place reversal of an acyclic list. The example `slist_sorted_insert` inserts a node into a sorted (by the data field) linked list. Similarly, `dlist_add` and `dlist_remove` are the insertion and deletion routines for cyclic doubly-linked lists. Finally, `allocator` is a low-level custom storage allocator. The version `allocator`(1) refers to the case where triggers were specified manually for a few invariants and the verification condition was split into two parts; these changes were required to prove the example using our previous method. The version `allocator`(2) corresponds to the case without these changes. Details of these examples and properties checked are described in earlier work [6]. The examples `init` and `removeList` are routines from an example with multiple doubly linked lists and complex data structure invariants.

The second column in Figure 7 gives the verification time for our earlier approach based on an incomplete axiomatization [6]. The third column gives the verification time with the approach presented in this paper. The results indicate that approach presented in this paper is consistently more robust and scalable compared to the earlier approach based on the incomplete axiomatization. [3]

On more complex problems, we obtain significant improvement in the runtime. We also obtain more automation in `allocator`(2) as the user does not have to manually provide triggers and decompose the proof. The new approach has also enabled us to verify new examples (`init` and `removeList`) that were beyond the capability of the old axiomatization. We believe that the main reason for the limited scalability of our old approach on complex problems is the large number of instantiations generated in the theorem prover. The significant runtime to prove the `allocator` example can be attributed to the interaction of arithmetic and quantifier instantiation inside the theorem prover. Nevertheless, the results are encouraging and we believe a customized implementation of the decision procedure inside the theorem prover will provide additional efficiency. We have also verified some of these examples with Z3, a new SMT solver being developed at Microsoft Research, with substantially better runtimes compared to Simplify.

7 Related work

Balaban et al. [1] present a logic that allows reachability over singly-linked lists to be expressed. Their decision procedure is based on a small-model property of the logic. The logic is restricted to quantifier-free facts about lists, e.g., disjointness of two lists cannot be expressed. Moreover, our experience has shown that a decision procedure based on small-model encoding is inefficient compared to one based on rewriting and inference [12].

Nelson [19] presents an logic with a ternary reachability predicate $R_f(x, y, z)$, which informally means that "x reaches y without visiting z". This predicate

[3] It was not possible to evaluate other decision procedures (e.g. [20]) because of (a) our use of well-founded reachability in the annotations, and (b) the presence of pointer arithmetic etc. requires a more general version of the reachability predicate [6].

allows properties of cyclic linked lists to be expressed. However, disjointness properties cannot be expressed without the use of explicit quantifiers. The paper presented a set of first-order axioms or rewrite rules to reason about this logic, but lacked a proof of completeness. There were several subsequent attempts, all of them incomplete, to provide first-order axiomatization of reachability [13, 15, 12]. Rakamarić et al. [20] provide a rewriting-based decision procedure for a logic similar to Nelson's. Their decision procedure is incomplete in relating R_f and $R_{f'}$, where f' is the result of updating f at some location. Our work improves upon this earlier work both by being precise with respect to heap updates over well-founded heaps and expressive with respect to properties over collections of objects. In addition, the rewrite rules in our decision procedure are fewer and simpler resulting in a simpler proof of completeness.

Ranise and Zarba [21] present a logic that is more expressive than our logic; their reachability relation contains a witness path for the reachability of one cell from another. They provide a decision procedure for the logic based on translation to a set of simpler logics without reachability. But they provide no implementation to evaluate the feasibility of their approach. Kuncak and Rinard [11] provide a logic with sets for reasoning about data structures. Unlike our logic, their logic does not allow sets to be constructed from the reachability predicate.

Unlike the papers discussed so far which have essentially used first-order logic for reasoning about linked data structures, other approaches have used higher-order logic for the same purpose. The pointer assertion logic engine (PALE) [16] uses monadic second-order logic to express properties involving reachability. Although the logic can express more complex shape properties than that allowed by our logic, the decision procedure for it has high complexity. Further, it is unclear how to combine it in a program verification framework with reasoning about other theories such as arithmetic. The work of Yorsh et al. [24] on the logic of reachable patterns is in a similar direction. They also provide a logic for expressing complex shape properties and provide a decision procedure by translation to monadic second-order logic.

Separation logic [22] has been proposed to reason about heap-manipulating programs. Berdine et al. [4] describe a rewrite-based decision procedure for a fragment of separation logic with linked lists. Similar to the work on monadic second-order logic, it is not clear how to combine this decision procedure with theory reasoning.

Automatic computation of intermediate invariants for programs with linked data structures (*shape analysis*) has also received considerable attention in recent years. This work is orthogonal and complementary to our work and we only discuss it briefly. Most of this work is based on specialized abstract domains for the heap [14, 8] or use predicate abstraction [10] with decision procedures for logics with reachability [1, 20, 12]. Better decision procedures are crucial for the latter approaches, but they can also be used to improve the imprecision of the underlying abstract domain in the former approaches [13].

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *Verification, Model checking, and Abstract Interpretation (VMCAI '05)*, LNCS 3385, pages 164–180, 2005.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
3. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 04: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 2004.
5. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.
6. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, LNCS 4424, pages 19–33, 2007.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. 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 '06)*, LNCS 3920, pages 287–302, 2006.
9. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254. Springer-Verlag, June 1997.
11. V. Kuncak and M. C. Rinard. Decision procedures for set-valued fields. *Electr. Notes Theor. Comput. Sci.*, 131:51–62, 2005.
12. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Principles of Programming Languages (POPL '06)*, pages 115–126, 2006.
13. T. Lev-Ami, N. Immerman, T. W. Reps, S. 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 '05)*, LNCS 3632, pages 99–115, 2005.
14. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS '00)*, LNCS 1824, pages 280–301, 2000.
15. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Computer-Aided Verification (CAV '05)*, LNCS 3576, pages 476–490, 2005.
16. Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI 01: Programming Language Design and Implementation*, pages 221–231, 2001.
17. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.

18. G. Nelson and D. C. Oppen. Fast decision procedures based on the congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
19. Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 38–47. ACM Press, 1983.
20. Z. Rakamarić, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, LNCS 4349, pages 106–121, 2007.
21. S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*, pages 206–215. IEEE Computer Society, 2006.
22. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 02: Logic in Computer Science*, pages 55–74, 2002.
23. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
24. Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *FoSSaCS '06: Foundations of Software Science and Computation Structures*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer-Verlag, 2006.

A Proofs

Theorem 1. *Let φ be a conjunction of literals that is consistent, closed under congruence, and convex under the rewrite rules in Figure 2. Let M be the tuple $(I_{BS}, I_f, I_R, I_B, I_{null})$ defined in Section 2.3. Then $M \models \varphi$.*

Proof. We now argue that this model satisfies the four properties required of any model of the logic in Figure 1.

1. Since the constraint on I_{null} is explicitly present in φ, the model satisfies the constraint for *null* trivially.
2. We show that if there exists $n > 0$ and equivalence classes $\alpha_0, \alpha_1, \ldots, \alpha_n \in A$ such that $I_f(\alpha_i) = \alpha_{i+1}$ for all $0 \leq i < n$ and $\alpha_0 = \alpha_n$, then there exists i such that $0 \leq i < n$ and $I_{BS}(\alpha_i) = \mathit{true}$. If this statement is false then $BS(\llbracket \alpha_i \rrbracket) \notin \varphi'$ for all $0 \leq i < n$. The convexity of φ' w.r.t. [REACH1] gives us that $R(\alpha_i, \alpha_{i+1}) \in \varphi'$ for all $0 \leq i < n$. If $n = 1$, the convexity of φ' w.r.t. [REFLEXIVITY] and [WELL-FOUNDED1] gives a contradiction. Otherwise, the convexity of φ' w.r.t. [TRANSITIVITY] gives us $R(\alpha_1, \alpha_n)$ and then convexity w.r.t. [WELL-FOUNDED1] gives a contradiction.
3. We show that I_R is a fixpoint of the equation

$$X \;\equiv\; \lambda u, v \in A.\; u = v \vee (\neg I_{BS}(I_f(u)) \wedge X(I_f(u), v)).$$

   ($\Rightarrow$) Suppose $I_R(\alpha, \beta)$ for equivalence classes $\alpha, \beta \in A$. Then $R(\llbracket \alpha \rrbracket, \llbracket \beta \rrbracket) \in \varphi'$ and $f(\llbracket \alpha \rrbracket) = \llbracket \gamma \rrbracket \in \varphi'$ for some equivalence class γ. Since φ' is convex w.r.t. rule [REACH2], we have either $\llbracket \alpha \rrbracket = \llbracket \beta \rrbracket \in \varphi'$ or $\neg BS(\llbracket \gamma \rrbracket) \in \varphi'$ and

$R(\llbracket\gamma\rrbracket, \llbracket\beta\rrbracket) \in \varphi'$. In the first case, we get $\alpha = \beta$. In the second case, we get $I_{BS}(\gamma) = \mathit{false}$ and $I_R(\gamma, \beta) = \mathit{true}$.
($\Leftarrow$) Suppose $\alpha = \beta$. Since φ' is convex w.r.t. rule [REFLEXIVITY], we have $R(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket) \in \varphi'$ and therefore $I_R(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket) = \mathit{true}$. Suppose $\neg I_{BS}(I_f(\alpha))$ and $I_R(I_f(\alpha), \beta))$. Suppose $I_f(\alpha) = \gamma$. Then $R(\llbracket\gamma\rrbracket, \beta) \in \varphi'$. Since φ' is convex w.r.t. rule [REACH1], either $BS(\llbracket\gamma\rrbracket) \in \varphi'$ or $R(\llbracket\alpha\rrbracket, \llbracket\gamma\rrbracket) \in \varphi'$. The first case is not possible because then $I_{BS}(\gamma) = \mathit{true}$ which is a contradiction. Therefore $R(\llbracket\alpha\rrbracket, \llbracket\gamma\rrbracket) \in \varphi'$ and since φ' is closed w.r.t. rule [TRANSITIVITY] we get $R(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket) \in \varphi'$ and therefore $I_R(\alpha, \beta) = \mathit{true}$.
(*Least fixpoint*) We show that I_R is the least fixpoint of the equation above by well-founded induction on I_R. For any relation X that is a fixpoint of the above equation, we show that $\mathcal{R} \subseteq X$.
- For the base case, we consider an α such that $\mathcal{R}(\alpha, \alpha)$ and for all α', if $\mathcal{R}(\alpha, \alpha')$ then $\alpha = \alpha'$. Clearly $(\alpha, \alpha) \in X$.
- For the inductive case, we consider an α such that $\mathcal{R}(\alpha, \alpha')$ for some α' such that $\alpha \neq \alpha'$. Consider a $\beta \in A$, such that $\mathcal{R}(\alpha, \beta)$. Since $\mathcal{R}$ is a fixpoint of the above equation, we know that either (i) $\alpha = \beta$, or (ii) $\neg I_{BS}(I_f(\alpha))$ and $\mathcal{R}(I_f(\alpha), \beta)$. In the first case $X(\alpha, \beta)$ holds and hence we get the desired result. In the second case, let $\alpha' = I_f(\alpha)$. Since φ' is convex w.r.t. [REACH1], and $I_{BS}(\alpha') = \mathit{false}$, we have $R(\llbracket\alpha\rrbracket, \llbracket\alpha'\rrbracket) \in \varphi'$, which in turn implies $\mathcal{R}(\alpha, \alpha')$. By the inductive hypothesis, for any $\gamma \in A$, if $\mathcal{R}(\alpha', \gamma)$ then $X(\alpha', \gamma)$, and therefore $X(\alpha', \beta)$. From the fixpoint equation, this implies $X(\alpha, \beta)$.

4. We show that I_B satisfies the equation

$$I_B \;\equiv\; \lambda u \in A'.\ \mathit{let}\ v = I_f(u)\ \mathit{in}\ \mathit{ite}(I_{BS}(v),\ v,\ I_B(v)).$$

Let $\alpha \in A'$ be an equivalence class and let $\beta = I_f(\alpha)$.
($I_{BS}(\beta) = \mathit{true}$) In this case, we have $BS(\llbracket\beta\rrbracket) \in \varphi'$. Since φ' is convex w.r.t. [BLOCK1], we get $B(\llbracket\alpha\rrbracket) = \llbracket\beta\rrbracket \in \varphi'$ and therefore $I_B(\llbracket\alpha\rrbracket) = \llbracket\beta\rrbracket$.
($I_{BS}(\beta) = \mathit{false}$) Since φ' is convex w.r.t. [REACH1], either $BS(\llbracket\beta\rrbracket) \in \varphi'$ or $R(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket) \in \varphi'$. The first case is not possible because then $I_{BS}(\beta) = \mathit{true}$ which is a contradiction. Therefore $R(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket) \in \varphi'$. Since φ' is convex w.r.t. [BLOCK3], we have $B(\llbracket\beta\rrbracket) = B(\llbracket\alpha\rrbracket)$.

□

Theorem 2. *The logic presented in Figure 1 is stably infinite.*

Proof. The proof follows from the proof of Theorem 1. Consider a satisfiable formula φ in this logic. By theorem 1, there is a model M whose domain A is a partitioning of the variables in φ, and interpretation I to the various symbols as described above. We define another model M' with an infinite domain A' (such that $A \subset A'$) with the following interpretation J:

- $J_{null} = I_{null}$.
- For any variable $x \in \varphi$, $J_x = I_{null}$.
- For any $\alpha \in A'$, $J_{BS}(\alpha) = I_{BS}(\alpha)$ if $\alpha \in A$, and $J_{BS}(\alpha) = \mathit{true}$ otherwise.

– For any $\alpha \in A'$, $J_f(\alpha) = I_f(\alpha)$ if $\alpha \in A$, and $J_f(\alpha) = \alpha$ otherwise.
– For any $\alpha \in A'$, $J_B(\alpha) = I_B(\alpha)$ if $\alpha \in A$, and $J_B(\alpha) = \alpha$ otherwise.
– For any $\alpha, \beta \in A'$, $J_R(\alpha, \alpha) = \mathit{true}$ for all $\alpha \in A'$, $J_R(\alpha, \beta) = I_R(\alpha, \beta)$ if both $\alpha, \beta \in A$, and $J_R(\alpha, \beta) = \mathit{false}$ otherwise.

□

Lemma 1. *Instantiating the set constraints in φ on all possible free variables of φ results in a formula φ' that is equisatisfiable with φ.*

Proof. Recall from the proof of Theorem 1, that the domain of the model of φ' is the partition of the set of variables in φ' induced by the set of variables equalities in φ'. In other words, each element of the domain of the model is mapped to at least one variable from φ'. By instantiating the set constraints on all the free variables in φ (or the variables in φ'), we ensure that the set constraints are instantiated on every element of the domain. Hence every model of φ' can be extended to a model for φ. Hence, φ and φ' are equisatisfiable. □

Theorem 3. *The satisfiability problem for a conjunction of literals in the set logic of Figure 4 is NP-complete.*

Proof. The logic is clearly NP-hard because it is a superset of well-founded reachability. Lemma 1 shows that a formula φ in this logic is equisatisfiable to a formula φ' without any set constraints. The formula φ' can subsequently be translated into an equisatisfiable formula ψ in purified form. Clearly, the size of ψ is polynomial in the size of φ.

From Theorem 2, we get that the theory of well-founded reachability is stably infinite. Therefore the formula ψ can be decided by combining the theory of well-founded reachability, the theory of arithmetic, and the theory of uninterpreted functions in the Nelson-Oppen framework. Let ψ be the disjoint union of ψ_1, a formula in the logic of well-founded reachability and ψ_2, a formula in the theory of arithmetic, and ψ_3 a formula in the theory of uninterpreted functions. The completeness of the the Nelson-Oppen framework guarantees that to provide a model for ψ, it is enough to provide a model for ψ_1, ψ_2, and ψ_3 together with an a truth assignment to every possible equality between variables. Clearly, the size of such a witness is polynomial in the size of ψ. □

B Weakest preconditions

The statement $\mathit{AddBlock}(BS, x)$ adds the cell x to the blocking set BS. To incorporate its effect on the R_f^{BS} and B_f^{BS}, it is desugared into the following statement sequence, denoted by S_2, and we get $wp(\mathit{AddBlock}(BS, x), \varphi) = wp(S_2, \varphi)$.

$$
\begin{array}{rcl}
\mathit{Assert}(\neg BS(x)) \ ; & & \\
B_f^{BS} & := & \lambda\, u.\ \mathit{ite}(R_f^{BS}(u, x) \wedge u \neq x,\ x,\ B_f^{BS}(u)) \ ; \\
R_f^{BS} & := & \lambda\, u, v.\ R_f^{BS}(u, v) \wedge (\neg R_f^{BS}(u, x) \vee u = x \vee \neg R_f^{BS}(x, v)) \ ; \\
BS & := & \lambda\, u.\ \mathit{ite}(u = x,\ \mathit{true},\ BS(u))
\end{array}
$$

The statement $\mathit{RemoveBlock}(BS, x)$ removes the cell x from the blocking set BS. To incorporate its effect on the R_f^{BS} and B_f^{BS}, it is desugared into the following statement sequence, denoted by S_3, and we get $wp(\mathit{RemoveBlock}(BS, x), \varphi) = wp(S_3, \varphi)$.

$$
\begin{array}{l}
\mathit{Assert}(BS(x)) \ ; \\
\mathit{Assert}(B_f^{BS}(x) \neq x) \ ; \\
R_f^{BS} \ := \ \lambda\, u, v.\ R_f^{BS}(u, v) \vee (B_f^{BS}(u) = x \wedge R_f^{BS}(x, v)) \ ; \\
B_f^{BS} \ := \ \lambda\, u.\ \mathit{ite}(B_f^{BS}(u) = x,\ B_f^{BS}(x),\ B_f^{BS}(u)) \ ; \\
BS \ := \ \lambda\, u.\ \mathit{ite}(u = x,\ \mathit{false},\ BS(u))
\end{array}
$$