Back to the Future

Revisiting Precise Program Verification using SMT Solvers

Shuvendu K. Lahiri Shaz Qadeer
Microsoft Research
{shuvendu, qadeer}@microsoft.com

Abstract

This paper takes a fresh look at the problem of *precise* verification of heap-manipulating programs using first-order Satisfiability-Modulo-Theories (SMT) solvers. We augment the specification logic of such solvers by introducing the *Logic of Interpreted Sets and Bounded Quantification* for specifying properties of heap-manipulating programs. Our logic is expressive, closed under weakest preconditions, and efficiently implementable on top of existing SMT solvers. We have created a prototype implementation of our logic over the solvers SIMPLIFY and Z3 and used our prototype to verify many programs. Our preliminary experience is encouraging; the completeness and the efficiency of the decision procedure is clearly evident in practice and has greatly improved the user experience of the verifier.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Algorithms, Reliability, Verification

Keywords Software verification, SMT solvers, decision procedures, heap-manipulating programs, reachability, linked lists

1. Introduction

First-order theorem provers like SIMPLIFY (Detlefs et al. 2005) are a fundamental component of many scalable program verification tools. These provers are used in many ways—to solve the verification condition of each procedure in a modular analysis (Flanagan et al. 2002; Barnett et al. 2005) and to compute and refine abstractions in a whole-program analysis (Ball et al. 2001; Henzinger et al. 2002). First-order reasoning has the important ability to combine various useful theories required for program verification, e.g., arithmetic, arrays, and uninterpreted functions, in a systematic manner (Nelson and Oppen 1979). Recently, Satisfiability-Modulo-Theories (SMT) solvers (Satisfiability Modulo Theories Library (SMT-LIB)) such as YICES (Dutertre and de Moura 2006) and Z3 (de Moura and Bjorner 2007), have combined advances in Boolean satisfiability solvers with powerful first-order theory reasoning using decision procedures. We believe that these powerful solvers have created an opportunity for scaling automated verification to deep properties of complex software.

POPL '08 January 7–12, 2008, San Francisco, California, USA.

Despite these recent advances, automated verification of heap-manipulating programs remains difficult with first-order reasoning. The main reason behind this difficulty is that the specification logic supported by SMT solvers is not expressive enough. In particular, it is usually cumbersome and often impossible to specify properties of unbounded lists and trees and non-aliasing invariants of deeply-nested heap structures. Previous attempts (Flanagan et al. 2002) at reasoning about these programs using first-order provers relied heavily on the use of quantifiers both for expressing assertions about (unbounded) data structures and for axiomatizing theories for linked lists and trees. The result has been unsatisfactory for two reasons. First, most recursive predicates useful for expressing invariants about unbounded data-structures cannot be axiomatized in first-order logic (Börger et al. 1997). Consequently, these axiomatizations tend to be incomplete leading to an unacceptable frequency of failed proofs. Second, quantifier-reasoning in first-order SMT solvers remains incomplete, heuristic-driven, and brittle. To properly use these solvers, considerable user ingenuity is required for writing carefully crafted quantified assertions. Such expertise is usually beyond the capability of normal programmers.

In this paper, we revisit the problem of *precise* verification of heap-manipulating programs using first-order SMT solvers. Our work is motivated by our desire to analyze systems software such as device drivers and operating systems code, which make heavy use of linked lists and deeply-nested linked data structures. We are interested in building an assertion checker for correctness properties of such programs such as memory-safety and data-structure invariants.

Towards this end, we present the *Logic of Interpreted Sets and Bounded Quantification* for specifying properties of heap-manipulating programs and a verifier for proving these properties. Our logic uses first-order logic as a substrate. In addition to providing useful but conventional theories such as arithmetic and equality with uninterpreted functions, the logic also provides several novel features that alleviate, to a significant extent, the aforementioned difficulties faced by first-order solvers in verifying data-structure properties. The contributions of this paper can be categorized along the following dimensions:

Logic. We introduce a new logic that facilitates precise, automated, and efficient reasoning about many heap-intensive programs. The logic provides an interpreted recursive predicate to reason about lists and two interpreted set constructors useful for writing specifications involving bounded quantification over the constructed sets.

1. The logic is *expressive*. In addition to describing rich data structure invariants (such as disjointness of two lists), and properties of entire collections (such as sortedness of a list), the logic is expressive enough for describing concisely *object invariants* over a given type and non-aliasing constraints.

2. The logic is *closed under weakest precondition*. Given a loop-free and call-free program annotated with preconditions, post-conditions and assertions in our logic, we present a procedure to generate a formula also in our logic that is unsatisfiable *if and only if* the program does not go wrong by failing any assertions.
3. The logic is *simple*. In spite of its expressiveness, the decision problem for the logic is NP-complete.

The increased expressiveness of our logic due to quantifiers and closure under weakest precondition make it much more attractive for program verification, compared to other similar logics (Nelson 1983; Rakamarić et al. 2007).

Decision procedure. We describe an efficient decision procedure for the logic using a set of sound, complete and terminating inference rules. The resulting decision procedure can leverage theory reasoning (for arithmetic and uninterpreted functions) and conflict-clause driven backtracking search of modern SMT solvers. The presence of bounded quantification over interpreted sets allows us to instantiate the quantifiers in a *lazy* manner, an attribute that is essential for good performance. Lazy instantiation greatly improves the performance of the decision procedure as (often useless) quantifier instantiation is one of the bottlenecks for first-order SMT provers supporting general quantifiers. We have implemented an initial prototype of the decision procedure over existing SMT solvers SIMPLIFY and Z3, using universally quantified first-order axioms with matching *triggers*.

Evaluation. We have used our decision procedure to verify many small to medium-sized C programs. Our preliminary experience is encouraging; the completeness and efficiency of the decision procedure is clearly evident in practice and has improved the robustness of the verification efforts manifold.

Although we have applied our verifier to annotated programs, where the user supplies the annotations, the ability to perform precise and automated verification is the cornerstone of many other verification techniques. Predicate abstraction techniques (Graf and Saïdi 1997) make calls to a theorem prover to construct an abstraction. Refinement of abstractions (Kurshan 1995; Clarke et al. 2000) relies on computing the weakest precondition and solving the generated verification condition. Symbolic execution of programs (Godefroid et al. 2005) requires solving path constraints precisely. The contributions of this paper are applicable, not only to modular program verification, but to these other domains as well.

Proofs for the lemmas and theorems in this paper have been omitted for lack of space. They can be found in a technical report (Lahiri and Qadeer 2007a).

2. Motivating example

We consider the linked data structures present in an real-world application called `muh` (Muh). `muh` is an Internet Relay Chat (IRC) bouncer, a program that acts as a middleman between an IRC-client and an IRC-server. The application is written in C.

The main data structures, described in Figure 1, consists of two acyclic doubly-linked lists, pointed to by `log_list.head` and `channel_list.head`, containing a list of `logentry` and `channel` nodes respectively. Figure 2 and Figure 3 describe the lists and their contents. Each node in the `log_list`, pointed to by the `data` field in the `dlink_node`, contains two character arrays `channel_name` and `filename` and an integer `logtype`. Similarly, each node in the `channel_list` contains two character arrays `name`, and `topic`, and a pointer to a `channel_log` node. The `channel_log` structure further contains an integer `ftype` and a pointer to a `FILE` called `logfile`.

Note that the list node `dlink_node` uses its `void * data` field polymorphically (Figure 1). When the node participates in the `log_list`, the `data` field is cast to a (`logentry`$*$) pointer, and

```
typedef struct _dlink_node
{
  struct _dlink_node *next;
  struct _dlink_node *prev;
  void *data;
} dlink_node;

typedef struct _dlink_list
{
  dlink_node *head;
  dlink_node *tail;
} dlink_list;

typedef struct _logentry
{
  char *channel_name;
  char *filename;
  int logtype;
} logentry;

extern dlink_list log_list;

typedef struct _channel_log
{
  int ftype;
  FILE *logfile;
} channel_log;

typedef struct _channel
{
  char *name;
  char *topic;
  channel_log *log;
} channel;

extern dlink_list channel_list;
```

Figure 1. Main data structures of muh

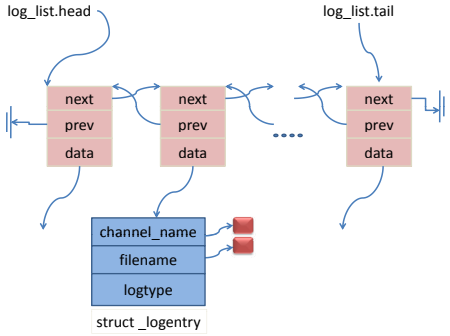

Figure 2. The log list

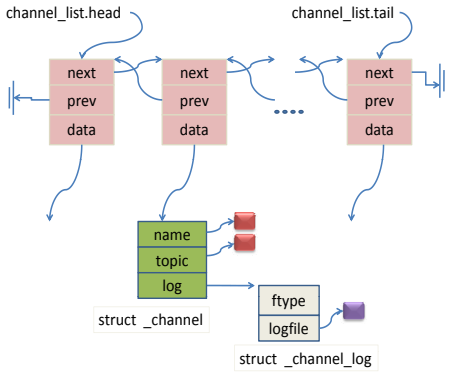

Figure 3. The channel list

```
void clear_logs(int clear)
{
    dlink_node *ptr;
    dlink_node *next_ptr;
    struct _logentry *logptr;

    .......

    /* then clear the loglist */
    for(ptr = log_list.head; ptr; ptr = next_ptr)
    {
        next_ptr = ptr->next;
        logptr = ptr->data;

        xfree(logptr->channel_name);
        xfree(logptr->filename);
        xfree(logptr);

        dlink_delete(ptr, &log_list);
        dlink_free(ptr);
    }
}

void rem_channel(struct _channel *chptr)
{
    dlink_node *ptr;

    /* close the logfile if we have one */
    if(chptr->log != NULL)
    {
        .....
        xfree(chptr->log);
    }

    if((ptr = dlink_find(chptr, &channel_list)) == NULL)
        return;

    dlink_delete(ptr, &channel_list);
    dlink_free(ptr);

    xfree(chptr->name);
    xfree(chptr->topic);
    xfree(chptr);
}
```

Figure 4. Freeing entries from `log_list` and `channel_list`.

when it participates in the `channel_list`, the `data` field is cast to a (`channel`$*$) pointer.

The example is representative of real-world applications written in C, which consist of a combination of multiple linked data structures. These data structures can either be recursively defined (e.g. `dlink_node`) or deeply-nested (e.g. `channel`).

During the lifetime of the application, various operations mutate these data structures through a set of functions. These functions correspond to adding or deleting a log to a list, adding or deleting a channel to a list, opening or closing a `FILE`, or freeing a set of entries of a list. In this section, we focus on the routines `clear_logs` and `rem_channel`, which free objects present in the data structures. Figure 4 describe parts of the procedures that free elements (using a procedure `xfree`) from the `log_list` and the `channel_list`. An important memory-safety property to enforce is the following:

> *Absence of double-free*: An object is not freed twice in the applications lifetime.

Let us consider the procedure `clear_logs` in Figure 4. The procedure iterates over the linked list pointed to by `log_list.head`, freeing the objects in each node. It first frees the `channel_name` and `filename` objects, then frees the `logentry` object pointed to by the `data` pointer and finally deletes the `dlink_node` object from the list and frees it. The procedure `rem_channel` similarly removes an entry from the `channel_list` and frees the objects.

Let us examine a few scenarios to understand why it is non-trivial to establish the absence of double-free for this routine:

1. Consider a given iteration of the loop, where `channel_name` and `filename` are freed. If both these pointers are aliased, then `xfree(logptr->filename)` would free an object that has already been freed by `xfree(logptr->channel_name)`. Hence, we would like to enforce that the `channel_name` and `filename` pointers in a `logentry` node do not alias.
2. Now consider two different iterations of the loop operating on two linked list nodes u and v. Let us imagine that u`->data` and v`->data` are aliased. In this case, each of the three `xfree` calls on the later iteration would free an already freed object.
3. Another scenario causing a double-free could arise if the `channel_name` or `filename` fields in some node in `log_list` aliases with `name` or `topic` fields in some node in `channel_list`. In this scenario, a call to `rem_channel` followed by a call to `clear_logs` would cause a double-free.
4. Finally, if the `data` field in a `dlink_node` object aliases with the `channel_name` fields in a `logentry` object, a double-free may be erroneously reported. This situation is avoided by enforcing that pointers to objects of two different types can never be equal.

In Section 3, we present a simple and natural specification logic that can express the necessary invariants to prove the absence of double-free error in the program. In Section 4, we revisit this example to illustrate the use of our logic. In spite of the expressiveness, the decision problem of the logic still remains NP-complete and we propose an efficient decision procedure for the logic using SMT solvers.

$$
\begin{array}{rcll}
c & \in & \mathit{Integer} & \\
x & \in & \mathit{Variable} & \\
f & \in & \mathit{Function} & \\
\varphi & \in & \mathit{Formula} & ::= \alpha \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \\
\alpha & \in & \forall\mathit{Formula} & ::= \gamma \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid \forall x \in S.\alpha \\
\gamma & \in & \mathit{GFormula} & ::= t_1 = t_2 \mid t_1 < t_2 \mid \\
 & & & \quad t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \mid \neg\gamma \\
t & \in & \mathit{Term} & ::= c \mid x \mid t_1 - t_2 \mid t_1 + t_2 \mid \\
 & & & \quad f(t) \mid \mathit{ite}(t = t',\ t_1,\ t_2) \\
S & \in & \mathit{Set} & ::= g^{-1}(t) \mid \mathit{Btwn}(f, t_1, t_2)
\end{array}
$$

Figure 5. *Logic of Interpreted Sets and Bounded Quantification* (*LISBQ*)

3. Logic

Our logic, presented in Figure 5, is interpreted over a finite partially-ordered set $\mathcal{D}$ of sorts. The set $\mathcal{D}$ contains the sort $\mathit{Integer}$ of integers. Each variable x has some sort $D \in \mathcal{D}$. Each uninterpreted function f has a sort $D \to E$ for sorts $D, E \in \mathcal{D}$. A model M for a formula in the logic provides an interpretation M_D for each sort D, where M_{Integer} is just the set of integers. The model also provides an interpretation $M_x \in M_D$ for each variable x of sort D and an interpretation $M_f : M_D \to M_E$ for each function f of sort $D \to E$. The interpretation is extended to arbitrary terms in the logic in the natural way.

The logic provides a ternary reachability predicate $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ for each function f of sort $D \to D$ where $D \in \mathcal{D}$. The model $M \models t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ iff there are *distinct* $u_0, u_1, \ldots, u_n \in M_D$

$$T \in Stmt ::= Assert(\varphi) \mid Assume(\varphi) \mid x := new \mid free(x) \mid x := t \mid f(x) := y \mid T_1; T_2 \mid T_1 \square T_2$$

Figure 6. Program

such that $M_f(u_i) = u_{i+1}$ for all $i \in [0, n)$, $u_0 = M_{t_1}$, $u_n = M_{t_3}$, and $u_i = M_{t_2}$ for some $i \in [0, n]$. We often refer to the binary reachability predicate $t_1 \xrightarrow{f} t_2$ which is defined to be $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_2$.

The logic allows bounded universal quantification over two different kinds of (potentially unbounded) sets. The set $f^{-1}(t)$ is constructed using a function f of sort $D \to E$ where $D, E \in \mathcal{D}$ and $D \neq E$. The model $M \models x \in f^{-1}(t)$ iff $M_{f(x)} = M_t$. The set $Btwn(f, t_1, t_2)$ is constructed using a function f of sort $D \to D$ where $D \in \mathcal{D}$. The model $M \models x \in Btwn(f, t_1, t_2)$ iff $M \models t_1 \xrightarrow{f} x \xrightarrow{f} t_2$. Note that our sorting requirements preclude the use of the same function symbol f in a term $f^{-1}(t)$ and a term $Btwn(f, t_1, t_2)$. This restriction is important for the completeness of our decision procedure; it ensures that models for the sets $g^{-1}(t)$ and $Btwn(f, t_1, t_2)$ can be computed independently.

The set of formulas *Formula* in our logic is closed under boolean combination. We have both universal and existential quantification but do not have alternation of quantifiers.

We require that logical formulas satisfy the *sort-restricted* property. This property is crucial for the termination of the decision procedure described in Section 3.3.

> In every formula $\forall x \in S.\alpha$, the sort of any term in α containing x as a strict sub-term is less than the sort of x.

We assume the existence of a type checking or type inference algorithm that can check for any formula φ in our logic that it is type-correct and satisfies the property described above.

3.1 From programs to formulas

In this section, we show how we translate a program T without loops and procedure calls into a formula φ in our logic. The translation has the property that φ is unsatisfiable iff S does not go wrong by failing an assertion. We assume that a preprocessing phase has eliminated all loops either soundly by using a programmer-supplied loop invariant or unsoundly by unrolling each loop a bounded number of times. Similarly, all procedure calls have been eliminated either soundly by using programmer-supplied pre- and post-conditions or unsoundly by inlining upto a bounded depth.

The syntax of the programs we consider is given in Figure 6. The statement $Assert(\varphi)$ is used to introduce intermediate assertions and postconditions. The statement $Assume(\varphi)$ is used to introduce preconditions and conditional statements. The statement $x := new$ creates a new object. Allocation of objects whose addresses are in sort D is modeled using a map $Alloc_D : D \to Integer$. The statement $x := new$ gets desugared into the code sequence

$$Assume(Alloc_D(k) = 0); Alloc_D(k) := 1; x := k$$

where k is a fresh variable (not used anywhere else) introduced per allocation site. The statement $free(x)$ frees an object. If x is of sort D, this statement is desugared into the code sequence

$$Assert(Alloc_D(x) = 1); Alloc_D(x) := 2.$$

The statement $x := t$ evaluates t and writes it into a variable x. The statement $f(x) := y$ writes the value in variable y into the field f at cell x. The statement $T_1; T_2$ evaluates T_1 followed by T_2. The statement $T_1 \square T_2$ executes either T_1 or T_2 nondeterministically.

$$\begin{aligned} wp(Assert(\psi), \varphi) &= \psi \wedge \varphi \\ wp(Assume(\psi), \varphi) &= \psi \Rightarrow \varphi \\ wp(x := t, \varphi) &= \varphi[x/t] \\ wp(f(x) := y, \varphi) &= \Gamma(\varphi, f, x, y) \\ wp(T_1; T_2, \varphi) &= wp(T_1, wp(T_2, \varphi)) \\ wp(T_1 \square T_2, \varphi) &= wp(T_1, \varphi) \wedge wp(T_2, \varphi) \end{aligned}$$

Figure 7. Weakest precondition

$$\begin{aligned} \Gamma(\varphi_1 \wedge \varphi_2, f, p, q) &\equiv \Gamma(\varphi_1, f, p, q) \wedge \Gamma(\varphi_2, f, p, q) \\ \Gamma(\varphi_1 \vee \varphi_2, f, p, q) &\equiv \Gamma(\varphi_1, f, p, q) \vee \Gamma(\varphi_2, f, p, q) \\ \Gamma(\neg\varphi, f, p, q) &\equiv \neg\Gamma(\varphi, f, p, q) \end{aligned}$$

Figure 8. Definition of $\Gamma(\varphi, f, p, q)$

This statement, together with the assume statement, is used to model conditional execution.

The weakest precondition computation for the various statements in our language (other than $x := new$ and $free(x)$ which are desugared) is given in Figure 7. As we discuss below, the set of formulas in our logic is closed under the computation of weakest precondition with respect to any statement T. Let $true$ denote the formula $x = x$ for some designated variable x. We first compute $wp(T, true)$ as described in the figure. The translation has the property that $\neg wp(T, true)$ is unsatisfiable iff T does not go wrong. In Sections 3.2 and 3.3, we present a procedure to decide whether $\neg wp(T, true)$ is satisfiable.

We use the notation $\varphi[x/t]$ to indicate the result of syntactically replacing x everywhere with t in the formula φ. The computation described in Figure 7 is straightforward and follows the classical description (Dijkstra 1976) except in the case of the statement $f(x) := y$. It is nontrivial to provide the weakest precondition of a formula φ with respect to $f(x) := y$ because of the use of the predicate $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ in φ. This predicate may be used either as an atom $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ or in defining the bound set $Btwn(f, t_1, t_2)$ of a universally quantified fact. The main difficulty is that a local change in the value of the function f at x might cause global change in the value of this predicate. However, for closure under weakest precondition, we must capture this change using the vocabulary of the logic.

The function Γ defined in Figures 8-11 is used for computing the weakest precondition of φ with respect to the statement $f(p) := q$. Figure 8 computes $\Gamma(\varphi, f, p, q)$ for a formula φ by straightforward recursion on the structure of the formula. Figure 9 computes $\Gamma(\alpha, f, p, q)$ for a formula α. The first two rules are straightforward recursion but to understand the next four rules, we must first understand $\Gamma(\gamma, f, p, q)$ defined in Figure 10 and $\Gamma(t, f, p, q)$ defined in Figure 11.

The rules for computing $\Gamma(t, f, p, q)$ in Figure 11 are mostly straightforward. Let f^p_q denote the function that is identical to f except at p where its value is q. The only nontrivial rule $\Gamma(f(t), f, p, q)$ states that the value of $f^p_q(t)$ is q if $p = t$ and $f(\Gamma(t, f, p, q))$ otherwise.

In Figure 10, $\Gamma(t_1 = t_2, f, p, q)$ and $\Gamma(t_1 < t_2, f, p, q)$ are obtained in a straightforward fashion by recursively computing $\Gamma(t_1, f, p, q)$ and $\Gamma(t_2, f, p, q)$. The computation of $\Gamma(t_1 \xrightarrow{g} t_2 \xrightarrow{g} t_3, f, p, q)$ is easy since g is different from f. We need to work harder, however, to compute $\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q)$. Suppose f is a function of sort $D \to D$. Let $u \xrightarrow[w]{f} v$ be the predicate defined

$$\Gamma(\alpha_1 \wedge \alpha_2, f, p, q) \equiv \Gamma(\alpha_1, f, p, q) \wedge \Gamma(\alpha_2, f, p, q)$$

$$\Gamma(\alpha_1 \vee \alpha_2, f, p, q) \equiv \Gamma(\alpha_1, f, p, q) \vee \Gamma(\alpha_2, f, p, q)$$

$$\begin{array}{l}\Gamma(\forall x \in g^{-1}(t).\alpha, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t' = \Gamma(t, f, p, q), \alpha' = \Gamma(\alpha, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \forall x \in g^{-1}(t').\alpha'\end{array}$$

$$\begin{array}{l}\Gamma(\forall x \in f^{-1}(t).\alpha, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t' = \Gamma(t, f, p, q), \alpha' = \Gamma(\alpha, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \wedge \quad q = t' \Rightarrow \alpha'[x/p] \wedge \forall x \in f^{-1}(t').\alpha' \\ \quad\quad \wedge \quad q \neq t' \Rightarrow \forall x \in f^{-1}(t').x = p \vee \alpha'\end{array}$$

$$\begin{array}{l}\Gamma(\forall x \in \mathit{Btwn}(g, t_1, t_2).\alpha, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t_1' = \Gamma(t_1, f, p, q), t_2' = \Gamma(t_2, f, p, q), \alpha' = \Gamma(\alpha, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \forall x \in \mathit{Btwn}(g, t_1', t_2').\alpha'\end{array}$$

$$\begin{array}{l}\Gamma(\forall x \in \mathit{Btwn}(f, t_1, t_2).\alpha, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t_1' = \Gamma(t_1, f, p, q), t_2' = \Gamma(t_2, f, p, q), \alpha' = \Gamma(\alpha, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \wedge \quad t_1' \xrightarrow[p]{f} t_2' \Rightarrow \forall x \in \mathit{Btwn}(f, t_1', t_2').\alpha' \\ \quad\quad \wedge \quad p \neq t_2' \wedge t_1' \xrightarrow[t_2']{f} p \wedge q \xrightarrow[p]{f} t_2' \Rightarrow \forall x \in \mathit{Btwn}(f, t_1', p).\alpha' \\ \quad\quad \wedge \quad p \neq t_2' \wedge t_1' \xrightarrow[t_2']{f} p \wedge q \xrightarrow[p]{f} t_2' \Rightarrow \forall x \in \mathit{Btwn}(f, q, t_2').\alpha'\end{array}$$

Figure 9. Definition of $\Gamma(\alpha, f, p, q)$

as follows:

$$u \xrightarrow[w]{f} v \quad \equiv \quad u \xrightarrow{f} v \xrightarrow{f} w \vee (u \xrightarrow{f} v \wedge \neg u \xrightarrow{f} w)$$

Intuitively, $u \xrightarrow[w]{f} v$ holds iff u can reach v by following zero or more f links without going through w. Formally, the model $M \models t_1 \xrightarrow[t_3]{f} t_2$ iff there are *distinct* $u_0, u_1, \ldots, u_n \in M_D$ such that $M_f(u_i) = u_{i+1}$ for all $i \in [0, n)$, $u_0 = M_{t_1}$, $u_n = M_{t_2}$, and $u_i \neq M_{t_3}$ for all $i \in [0, n)$. Then, we have the following identity:

$$\begin{array}{rll}u \xrightarrow{f_q^p} v \xrightarrow{f_q^p} w \equiv & \vee & u \xrightarrow{f} v \xrightarrow{f} w \wedge u \xrightarrow[p]{f} w \\ & \vee & p \neq w \wedge u \xrightarrow[w]{f} p \wedge u \xrightarrow{f} v \xrightarrow{f} p \wedge q \xrightarrow[p]{f} w \\ & \vee & p \neq w \wedge u \xrightarrow[w]{f} p \wedge q \xrightarrow{f} v \xrightarrow{f} w \wedge q \xrightarrow[p]{f} w\end{array}$$

The first disjunct captures the case when p cannot destroy the path witnessing $u \xrightarrow{f} v \xrightarrow{f} w$ and therefore $u \xrightarrow{f_q^p} v \xrightarrow{f_q^p} w$ holds as well. The second disjunct captures the case when there is a path from u to v and the update to f creates a path from v to w. The third disjunct captures the case when there is a path from v to w and the update to f creates a path from u to v.

The identity given above provides a precise update for the predicate $u \xrightarrow{f} v \xrightarrow{f} w$ with respect to the statement $f(p) := q$. Consequently, it is crucial for the closure of our logic under weakest preconditions. Finally, this identity provides a simple way to compute $\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q)$ as shown in Figure 10.

$$\Gamma(t_1 = t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) = \Gamma(t_2, f, p, q)$$

$$\Gamma(t_1 < t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) < \Gamma(t_2, f, p, q)$$

$$\begin{array}{l}\Gamma(t_1 \xrightarrow{g} t_2 \xrightarrow{g} t_3, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t_1' = \Gamma(t_1, f, p, q), t_2' = \Gamma(t_2, f, p, q), t_3' = \Gamma(t_3, f, p, q) \\ \quad \mathit{in} \\ \quad\quad t_1' \xrightarrow{g} t_2' \xrightarrow{g} t_3'\end{array}$$

$$\begin{array}{l}\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t_1' = \Gamma(t_1, f, p, q), t_2' = \Gamma(t_2, f, p, q), t_3' = \Gamma(t_3, f, p, q) \\ \quad \mathit{in} \\ \quad\quad t_1' \xrightarrow{f_q^p} t_2' \xrightarrow{f_q^p} t_3'\end{array}$$

$$\Gamma(\neg\gamma, f, p, q) \equiv \neg\Gamma(\gamma, f, p, q)$$

Figure 10. Definition of $\Gamma(\gamma, f, p, q)$

$$\Gamma(c, f, p, q) \equiv c$$

$$\Gamma(x, f, p, q) \equiv x$$

$$\Gamma(t_1 + t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) + \Gamma(t_2, f, p, q)$$

$$\Gamma(t_1 - t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) - \Gamma(t_2, f, p, q)$$

$$\Gamma(g(t), f, p, q) \equiv g(\Gamma(t, f, p, q))$$

$$\begin{array}{l}\Gamma(f(t), f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t' = \Gamma(t, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \mathit{ite}(p = t',\ q,\ f(t'))\end{array}$$

$$\begin{array}{l}\Gamma(\mathit{ite}(t_1 = t_2,\ t_3,\ t_4), f, p, q) \equiv \\ \quad \mathit{let} \\ \quad\quad t_1' = \Gamma(t_1, f, p, q), t_2' = \Gamma(t_2, f, p, q), \\ \quad\quad t_3' = \Gamma(t_3, f, p, q), t_4' = \Gamma(t_4, f, p, q) \\ \quad \mathit{in} \\ \quad\quad \mathit{ite}(t_1' = t_2',\ t_3',\ t_4')\end{array}$$

Figure 11. Definition of $\Gamma(t, f, p, q)$

We now go back to Figure 9 to understand the last four rules for computing $\Gamma(\alpha, f, p, q)$. The third and fifth rules are straightforward because f and g are different functions. The intuition for $\Gamma(\forall x \in f^{-1}(t).\alpha, f, p, q)$ rests on the identity

$$(f_q^p)^{-1}(t') = \mathit{ite}(q = t',\ f^{-1}(t') \cup \{p\},\ f^{-1}(t') \setminus \{p\}),$$

where $t' = \Gamma(t, f, p, q)$ and we have extended the $\mathit{ite}(\cdot,\ \cdot,\ \cdot)$ constructor to sets for brevity. The intuition for the rule $\Gamma(\forall x \in \mathit{Btwn}(f, t_1, t_2).\alpha, f, p, q)$ rests on the identity relating $u \xrightarrow{f_q^p} v \xrightarrow{f_q^p} w$ and $u \xrightarrow{f} v \xrightarrow{f} w$ defined above; there is a one-one correspondence between the disjuncts there and the conjuncts in the definition of $\Gamma(\forall x \in \mathit{Btwn}(f, t_1, t_2).\alpha, f, p, q)$.

$$[\text{AND}]\ \frac{\varphi_1 \wedge \varphi_2}{\varphi_1, \varphi_2} \qquad [\text{OR}]\ \frac{\varphi_1 \vee \varphi_2}{\varphi_1 \quad \varphi_2}$$

$$[\text{ITE}]\ \frac{\varphi[ite(t_1 = t_2,\ t_3,\ t_4)]}{t_1 = t_2, \varphi[t_3] \quad t_1 \neq t_2, \varphi[t_4]} \qquad [\text{EQ}]\ \frac{\varphi_1[t_1] \quad \varphi_2[t_2]}{t_1 = t_2 \quad t_1 \neq t_2}$$

Figure 12. Basic inference

The contribution of this section can be summarized in the following theorem about the definition of the $wp(T, \varphi)$ captured by Figures 7-11.

THEOREM 1. *For any program T, the formula $\neg wp(T, true)$ is in the logic LISBQ. Moreover, the program T goes wrong iff $\neg wp(T, true)$ is satisfiable.*

3.2 Decision procedure for ground logic

In this section, we provide a decision procedure for checking satisfiability of a formula φ in our logic. First, we convert φ into negation normal form and skolemize the resulting existential quantifiers that result from moving a negation inside a universal quantifier. The resulting formula remains in our logic. We first present a decision procedure for the case of ground formulas (Figures 12 and 13) and then augment the procedure to deal with quantifiers (Figure 14).

Our algorithm maintains a *context*, which is a conjunction of formulas currently asserted to be true. The algorithm provides a collection of rewrite rules 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 of one or more formulas.

Each inference rule is written as a conjunction of antecedents above the line and a disjunction of consequents below the line. In some cases such as [AND], a consequent below the line might have several comma-separated formulas which are interpreted as conjoined. If there is a rule such that the current context contains all the formulas above the line, then the it is guaranteed that the disjunction of the formulas below the line is entailed by the context. In this case, we say that the rule *matches* the context. A context is called *saturated* if for every matching rule, the context contains all the formulas in one of the disjuncts below the line.

Let U denote the quantifier-free theory of equality with uninterpreted functions and relations. The signature of U contains all symbols of our logic *LISBQ* (Figure 5) except $+$, $-$, $<$, and the constants in $Integer$. Note that while *LISBQ* interprets the relation $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$, the logic U treats it as uninterpreted. Let V denote the logic with the same signature as U in which the relation $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ is interpreted. Let A denote the quantifier-free theory of linear arithmetic. The signature of A contains all symbols of *LISBQ* except the function symbols, such as f, and the relation symbols, such a $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$. The only symbols shared among U and A are the variables in $Variable$. Let UA denote the combination of U and A. The signature of UA is exactly the same as the logic in Figure 5 except that in UA the relation $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ is treated as uninterpreted.

A *literal* is a quantifier-free formula that is free of boolean connectives $\wedge$ and $\vee$, and $ite(\cdot,\ \cdot,\ \cdot)$ terms. We assume the existence of an oracle for the theory UA that can decide whether the set of literals in the current context is satisfiable. The context is *consistent* if the oracle decides that set of literals in the context is satisfiable. Otherwise, the context is *inconsistent*.

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 saturated, the algorithm reports that φ is satisfiable. Otherwise, there is a matching rule such that none of the formulas below the line are present in the context. If there is only one formula below the line, it is added to the context. Otherwise, a case split is performed with one formula added to the context for each case.

$$[\text{REFLEXIVE}]\ \frac{}{t \xrightarrow{f} t} \qquad [\text{STEP}]\ \frac{f(t)}{t \xrightarrow{f} f(t)} \qquad [\text{REACH}]\ \frac{f(t_1) \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2 \quad t_1 \xrightarrow{f} f(t_1) \xrightarrow{f} t_2}$$

$$[\text{CYCLE}]\ \frac{f(t_1) = t_1 \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2} \qquad [\text{SANDWICH}]\ \frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_1}{t_1 = t_2}$$

$$[\text{ORDER1}]\ \frac{t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \quad t_1 \xrightarrow{f} t_3 \xrightarrow{f} t_2} \qquad [\text{ORDER2}]\ \frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2,\ t_2 \xrightarrow{f} t_3}$$

$$[\text{TRANSITIVE1}]\ \frac{t_1 \xrightarrow{f} t_2 \quad t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_3} \qquad [\text{TRANSITIVE2}]\ \frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t \xrightarrow{f} t_2}{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t,\ t_0 \xrightarrow{f} t \xrightarrow{f} t_2}$$

$$[\text{TRANSITIVE3}]\ \frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_0 \xrightarrow{f} t \xrightarrow{f} t_1}{t_0 \xrightarrow{f} t \xrightarrow{f} t_2,\ t \xrightarrow{f} t_1 \xrightarrow{f} t_2}$$

Figure 13. Reachability

Figure 12 gives the basic inference rules. The rules [AND] and [OR] are straightforward and follow from the logical meaning of $\wedge$ and $\vee$. The rule [ITE] is applicable whenever the context contains a ground formula φ containing a term $ite(t_1 = t_2,\ t_3,\ t_4)$, in which case we perform a case split on $t_1 = t_2$. If $t_1 = t_2$ we replace $ite(t_1 = t_2,\ t_3,\ t_4)$ with t_3 in φ, otherwise we replace $ite(t_1 = t_2,\ t_3,\ t_4)$ with t_4. The rule [EQ] performs a case split on the equality between any two terms t_1 and t_2 that exist in the context.

The rules in Figure 13 are dedicated to proving facts about the ternary reachability predicate $u \xrightarrow{f} v \xrightarrow{f} w$. The rules make extensive use of the binary reachability predicate $u \xrightarrow{f} v$ which, as mentioned earlier, is equivalent to $u \xrightarrow{f} v \xrightarrow{f} v$. Rule [REFLEXIVE] says that $\cdot \xrightarrow{f} \cdot$ is a reflexive relation. In rule [STEP], as in a few other rules discussed later, we take a notational liberty by putting a term $f(t)$ above the line. Rule [STEP] is applicable whenever a term $f(t)$ occurs in any ground formula in the context and concludes the obvious fact that $f(t)$ is reachable from t. Rule [REACH] draws conclusions based on the presence of $f(t_1)$ in the context and the reachability from t_1 of another term t_2.

Rules [CYCLE] and [SANDWICH] entail an equality without introducing a case split. Both rules draw conclusions from the presence of cycles in the graph of the reachability relation. Rules [ORDER1] and [ORDER2] connect the ternary and binary reachability predicates. Rule [ORDER1] says that if t_1 reaches both t_2 and t_3, then either t_1 reaches t_2 followed by t_3 or t_1 reaches t_3 followed by t_2. Rule [ORDER2] draws the more obvious conclusion in the other direction. Rules [TRANSITIVE1], [TRANSITIVE2], and [TRANSITIVE3] state various facts related to the transitivity of $\cdot \xrightarrow{f} \cdot$ and $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$.

Given a quantifier-free formula φ as input, the procedure defined by the rules in Figures 12 and 13 terminates because the number of new terms created is bounded by the number of $ite(\cdot, \cdot, \cdot)$ terms in φ. Since the procedure simply combines backtracking with the creation of new facts among this bounded set of terms, we conclude that the procedure will terminate. Soundness of the algorithm is proved by reasoning locally about each inference rule to verify that the conjunction of antecedents indeed implies the disjunction of the consequents.

The argument for the completeness of the algorithm is as follows. Suppose during the execution of the algorithm, we arrive at a consistent and saturated context C. We create a model that satisfies each formula in C using the following steps: Remove each formula in C that is not a literal. Due to the rules [AND], [OR], and [ITE], it suffices to find a model for the resulting set of formulas. Purify the remaining literals by introducing fresh variables and new equalities and saturate the context with all derived facts in the theory U using congruence closure. For each sort D, introduce a fresh variable $\perp_D$ and add literals $\perp_D \neq x$, for every other variable x of sort D. The role of these variables will become clear later when we define the model. Note that the addition of these disequalities to the context does not create any fresh implications. Split the context into the set C_V containing literals only from theory V and the set C_A containing literals only from theory A. Due to the rule [EQ], both C_V and C_A are *convex* and entail the same set of equalities. In addition, both theories V and A are stably infinite. Therefore, in order to get a model for $C_V \wedge C_A$, we only need to get models separately C_V and C_A (Nelson and Oppen 1979). Here we only show how to generate a model for C_V, since a model for C_A can be generated from the decision procedure for arithmetic.

We now show how to construct a model M for a consistent and saturated C_V. In order to define a model M for C_V, we need for each sort D different from the sort $Integer$, a domain M_D, an assignment $M_x \in D$ for every variable x of sort D, and an assignment $M_f : M_D \to M_D$ for every function f of sort $D \to D$. We start in the usual way and define M_D to be the partition $\{u_1, \ldots, u_n\}$ of the set of variables of sort D satisfying the following condition: for all $i \in [1, \ldots, n]$ and for all variables x and y of sort D, $x \in u_i$ and $y \in u_i$ iff $x = y \in C_V$. For each $u \in M_D$, let $[\![u]\!]$ denote a fixed representative member of u. Now we define M_x to be the unique $u \in M_D$ such that $x \in u$. Note that the equivalence class containing the variable $\perp_D$ is the singleton $\{\perp_D\}$ because C_V contains disequalities differentiating $\perp_D$ from every variable of sort D.

We would like to define $f(u)$ for an arbitrary element $u \in M_D$. If $(f([\![u]\!]) = [\![v]\!]) \in C_V$ for some $v \in M_D$, then we define $M_f(u) = v$. However, if $(f([\![u]\!]) = [\![v]\!]) \notin C_V$ for any $v \in M_D$, then we must pick some element of M_D to be $f(u)$. The main difficulty is that the interpretation of the function f is tied to the interpretation of the relation $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$. We must be careful not to define $f(u)$ to be inconsistent with the constraints in C_V. To help us in this task, we define for each $u \in M_D$, the relation $R_u = \{(v, w) \in M_D \times M_D \mid [\![u]\!] \xrightarrow{f} [\![v]\!] \xrightarrow{f} [\![w]\!] \in C_V\}$.

LEMMA 1. *For all $u \in M_D$, the following facts hold:*

1. $(u, u) \in R_u$.
2. *R_u is reflexive over $dom(R_u)$.*
3. *R_u is transitive.*
4. *R_u is totally-ordered over $dom(R_u)$.*

With the aid of R_u, we now provide a complete interpretation for M_f. Let u be an arbitrary element of M_D. If $(f([\![u]\!]) = [\![v]\!]) \in C_V$ for some $v \in M_D$, then define $M_f(u) = v$. Otherwise, if $R_u = \{(u, u)\}$ then define $M_f(u) = u$. Otherwise, define $M_f(u)$ to be the least element, with respect to R_u, of $dom(R_u) \setminus \{u\}$.

$$\text{[INV]}\ \frac{f(t') = t \quad \forall x \in f^{-1}(t).\alpha}{\alpha[x/t']} \qquad \text{[BTWN]}\ \frac{t_1 \xrightarrow{f} t \xrightarrow{f} t_2 \quad \forall x \in Btwn(f, t_1, t_2).\alpha}{\alpha[x/t]}$$

Figure 14. Quantifier instantiation

Lemma 1 guarantees that defining f in the last two cases does not create any contradictions with the context C_V. The interpretation for $\cdot \xrightarrow{f} \cdot \xrightarrow{f} \cdot$ is now defined as

$$u \xrightarrow{f} v \xrightarrow{f} w \Leftrightarrow [\![u]\!] \xrightarrow{f} [\![v]\!] \xrightarrow{f} [\![w]\!] \in C_V.$$

In addition to defining the interpretation for each function f of sort $D \to D$, we also need to define an assignment $M_g : M_D \to M_E$ for every function g whose sort is $D \to E$ where D and E are different. Let u be an arbitrary element of M_D. If $(g([\![u]\!]) = [\![v]\!]) \in C_V$ for some $v \in M_E$, then we define $M_g(u) = v$. Otherwise, we define $M_g(u) = \{\perp_E\}$.

LEMMA 2. *The model M defined above satisfies $M \models C_V$.*

Based on Lemma 2, we obtain the following theorem.

THEOREM 2. *Let $\varphi \in$ Formula be quantifier-free. Then the procedure described by the rules in Figures 12 and 13 terminates and decides the satisfiability of φ.*

In Section 3.3, we extend our decision procedure to the full logic with quantification. The following lemma captures an important property of the rules in Figures 12 and 13 that is used to prove the completeness of our decision procedure for the full logic.

LEMMA 3. *Let X be any collection of facts of the form $t_1 = t_2$. Let Y be any collection of facts of the form $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$. Let C be any consistent and saturated context. If C entails $\bigvee(X \cup Y)$, then one of the following must hold:*

1. *$t_1 = t_2 \in C$ for some $t_1 = t_2 \in X$.*
2. *$t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \in C$ for some $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \in Y$.*

3.3 Decision procedure for quantified logic

We now extend our decision procedure to handle quantified facts by adding the rules in Figure 14. The first rule [INV] handles quantification over the set constructor $f^{-1}(t)$; if the current context contains the fact $f(t') = t$ then this rule instantiates the body of the quantifier at t'. The rule [BTWN] works in similarly for the set constructor $Btwn(f, t_1, t_2)$.

Our decision procedure terminates even after adding the quantifier instantiation rules in Figure 14 because the input formula is required to be *sort-restricted*. There is a partial-order on the set of sorts and whenever a quantifier $\forall x \in S.\varphi$ is instantiated, any new terms generated are of a sort less than the sort of x. By well-founded induction over the partial-order on the set of sorts, we can show that for each sort D there is a decision depth in the backtracking search beyond which the number of terms of sort D remains unchanged. In fact, for an input formula φ, the number of terms of sort D generated by our algorithm is bounded by $|\varphi| \cdot K^{|\mathcal{D}|}$, where K is the number of function symbols. Since the number of sorts is finite, the procedure will terminate. Since the size of the model constructed in Section 3.2 is linear in the number of terms and the number of terms is linear in the size of the formula, we have the following theorem about the complexity of our logic.

THEOREM 3. *The satisfiability problem for the logic LISBQ is NP-complete.*

The local soundness of the rules [INV] and [BTWN] is obvious. For completeness, we appeal to Lemma 3. Consider a context C that is consistent and saturated with respect to all the rules in Figures 12, 13, and 14. Let D_1 be the conjunction $\bigwedge t_1 \neq t_2$ for all t_1 and t_2 such that $t_1 = t_2 \notin C$. Let D_2 be the conjunction $\bigwedge \neg t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ for all t_1, t_2, and t_3 such that $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \notin C$. Lemma 3 guarantees that $C \wedge D_1 \wedge D_2$ is satisfiable, which in turn implies that all the quantified facts in C have been instantiated enough. Therefore, a model for the set of literals in C is a model for all the facts in C. We have already shown, earlier in this section, how to construct a model for the set of literals in C. Thus, we have the following theorem.

THEOREM 4. *Let $\varphi \in$ Formula. Then the procedure described by the rules in Figures 12, 13, and 14 terminates and decides the satisfiability of φ.*

It is important to note that set-bounded quantification is not essential for either termination or completeness of our decision procedure. Termination is ensured by the *sort-restricted* property. Completeness could be ensured simply by instantiating quantified facts on all ground terms of the appropriate sort. However, such a procedure would lead to a huge number of instantiations and would consequently be very expensive in practice. Set-bounded quantification allows us to instantiate quantifiers lazily and yields an efficient implementation.

3.4 Expressiveness

In this section, we show various examples to illustrate the expressiveness of our logic.

Cyclic lists. We specify that `hd` points to the head of a cyclic list as follows:

$$\texttt{hd} \neq \texttt{null} \wedge \texttt{f}(\texttt{hd}) \xrightarrow{\texttt{f}} \texttt{hd}$$

Suppose each element of this list contains a field `data`. The invariant for a loop that iterates, using a variable `i`, over this list setting the `data` field to 0 is specified as follows:

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}, \texttt{f}(\texttt{hd}), \texttt{i}) \setminus \{\texttt{i}\}.\ \texttt{data}(\texttt{u}) = 0$$

Sortedness. Suppose `hd` points to a null-terminated list. The invariant that the values stored in the `data` field of the list members are sorted is specified as follows:

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}, \texttt{f}(\texttt{hd}), \texttt{null}) \setminus \{\texttt{null}\}.\\ \forall \texttt{v} \in \texttt{Btwn}(\texttt{f}, \texttt{u}, \texttt{null}) \setminus \{\texttt{null}\}.\ \texttt{data}(\texttt{u}) \leq \texttt{data}(\texttt{v})$$

List of lists. Suppose `hd` is pointing to a null-terminated list linked by the field `f` and each member of the list has a field `l` that points to a distinct null-terminated list linked by the field `g`. The disjointness of these lists is specified as follows:

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}, \texttt{hd}, \texttt{null}) \setminus \{\texttt{null}\}.\\ \forall \texttt{v} \in \texttt{Btwn}(\texttt{f}, \texttt{hd}, \texttt{null}) \setminus \{\texttt{null}\}.\\ \texttt{u} = \texttt{v} \vee \forall \texttt{w} \in \texttt{Btwn}(\texttt{g}, \texttt{l}(\texttt{u}), \texttt{null}) \setminus \{\texttt{null}\}. \neg \texttt{l}(\texttt{v}) \xrightarrow{\texttt{g}} \texttt{w}$$

The ability to specify such invariants is useful for verifying systems software that uses composite data structures (Berdine et al. 2007).

List union. Suppose `a`, `b`, and `c` are null-terminated lists constructed using linking fields $\texttt{f}_1$, $\texttt{f}_2$, and $\texttt{f}_3$ respectively. We specify that `a` is the union of `b` and `c` as follows:

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}_1, \texttt{a}, \texttt{null}).\texttt{b} \xrightarrow{\texttt{f}_2} \texttt{u} \vee \texttt{c} \xrightarrow{\texttt{f}_3} \texttt{u}$$
$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}_2, \texttt{b}, \texttt{null}).\texttt{a} \xrightarrow{\texttt{f}_1} \texttt{u}$$
$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{f}_3, \texttt{c}, \texttt{null}).\texttt{a} \xrightarrow{\texttt{f}_1} \texttt{u}$$

This invariant is useful in proving the correctness of the in-place list reversal program.

4. Revisiting the motivating example

In this section, we revisit the example presented in Section 2 and describe the various invariants required to prove the absence of double-free property. We show that these invariants can be naturally expressed using the simple logic we presented in Section 3. To easily follow the specifications, the reader should reexamine the program in Section 2.

For the purpose of illustration, we consider programs written in a subset of C that is simple yet rich enough to express the program in Section 2. The language precludes performing arithmetic on pointers, taking the address of (using the & operator) a stack variable or a field inside a structure, and the use of arrays, unions and nested structures. Before we describe our specifications, we briefly describe the memory model and how we define the operational semantics of the program.

- The memory is partitioned into a set of maps $\texttt{f}_1, \texttt{f}_2, \ldots, \texttt{f}_\texttt{n}$, one corresponding to each field declared in the program, and indexed by the objects or references. Without loss of generality, we assume that the field names are globally distinct.
- The value of the C expression `x->f` is obtained by looking up the map `f` at the index obtained by evaluating `x`. An update `x->f = y` updates the map `f` at the location obtained by evaluating `x`, with the value obtained by evaluating `y`.

4.1 Sorts

To generate a well-sorted formula from a program, we assign sorts to the different fields and variables in the program. The sorts are generated by analyzing the type structure of the program. Initially, a sort is assigned to each distinct type in the program. If two types can be the target of a `void`$*$ field (e.g. `data` in our example), we merge the sorts for the two types. For our example, the set of sorts $\mathcal{D}$ consists of at least the following sorts:

$$\mathcal{D} \equiv \{\mathit{Integer}, \texttt{P\_dlink\_node}, \texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle, \\ \texttt{P\_channel\_log}, \texttt{P\_FILE}, \texttt{P\_char}, \ldots\}$$

`P_dlink_node` is the sort for a pointer to a `dlink_node`. The sort $\texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle$ is for a pointer to either `logentry` or `channel`, unified due to the presence of the polymorphic `data` field in `dlink_node`. Given the sort set $\mathcal{D}$, we can assign sorts to the different variables and fields in the program, by substituting the sort corresponding to each type in the program.

For each sort $D \in \mathcal{D}$, we maintain a map $\texttt{Type}_D : D \rightarrow \mathit{Integer}$, that maps an object of sort D to an integer constant denoting the dynamic type of object. We introduce a constant for each type, by prefixing a @ to the type name (e.g. `@logentry` for an object of type `logentry`, `@channel_log` for an object of type `channel_log`). The dynamic type of each object is assigned during the allocation of the object. All the casts in the program are checked to see that they match the dynamic type of the objects.

Finally, the partial order $\prec$ on $\mathcal{D}$ is generated by analyzing the signature of the fields in the program, apart from the linking fields like `next` and `prev`. For each field $\texttt{f} : D \rightarrow E$ in the program, we add the constraint that $E \prec D$. The partial order for our program is the following:

$$\begin{array}{lcl} \texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle & \prec & \texttt{P\_dlink\_node} \\ \texttt{P\_char} & \prec & \texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle \\ \mathit{Integer} & \prec & \texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle \\ \texttt{P\_channel\_log} & \prec & \texttt{P\_}\langle\texttt{logentry}, \texttt{channel}\rangle \\ \mathit{Integer} & \prec & \texttt{P\_channel\_log} \\ \texttt{P\_FILE} & \prec & \texttt{P\_channel\_log} \end{array}$$

4.2 Specifications using $\texttt{g}^{-1}$ set constructor

The $\texttt{g}^{-1}$ set constructor is useful for expressing both non-aliasing of heap objects and *type invariants*.

To prove the absence of double-free property in our example, we need to ensure non-aliasing of various fields of the same sort (e.g. the char$*$ fields channel_name, filename, etc.). For a field f, and a set of fields F, we first define a macro NotAliased(f, F, u) as follows:

$$\texttt{NotAliased}(\texttt{f},\texttt{F},\texttt{u}) \equiv \\ \texttt{f}^{-1}(\texttt{f}(\texttt{u})) = \{\texttt{u}\} \wedge \quad \bigwedge_{g \in \texttt{F}} \texttt{g}^{-1}(\texttt{f}(\texttt{u})) = \{\}$$

This macro specifies that the object f(u) pointed to by a given field f, cannot be also pointed to by any of the fields in F. The set F usually contains a set of fields that have the same sort as f. Note that we have used set equality as a syntactic sugar for the more elaborate formula using bounded quantification.

We can use this macro to specify that any object pointed to by the char$*$ field channel_name is distinct from the objects pointed to by the other char$*$ fields as follows:

$$\forall \texttt{u} \in \texttt{Type}_{\texttt{P\_}\langle \texttt{logentry},\texttt{channel}\rangle}{}^{-1}(@\texttt{logentry}). \\ \texttt{NotAliased}(\texttt{channel\_name}, \{\texttt{filename}, \texttt{name}, \texttt{topic}\}, \texttt{u})$$

It means for any object u of sort P_⟨logentry, channel⟩ with a dynamic type logentry, the object channel_name(u) of sort P_char, can't be pointed to by any of the other char$*$ fields.

Observe that the use of $\texttt{Type}_{\texttt{D}}{}^{-1}(@T)$ allows us to describe *type invariants* for any given dynamic type T within the sort D, such as logentry in the previous example.

4.3 Specifications using Btwn(f, x, y) set constructor

Let us now illustrate the use of the set constructor Btwn(f, x, y) to describe properties of linked lists.

- *Disjointness of lists*: To specify that the two linked lists have disjoint elements, we can exploit the fact that the nodes in the two linked lists have different dynamic types:

  $$\forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{log\_list.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{Type}_{\texttt{P\_}\langle \texttt{logentry},\texttt{channel}\rangle}(\texttt{data}(\texttt{u})) = @\texttt{logentry} \\ \forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{channel\_list.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{Type}_{\texttt{P\_}\langle \texttt{logentry},\texttt{channel}\rangle}(\texttt{data}(\texttt{u})) = @\texttt{channel}$$

  These invariants describe that for any node u in the linked list between log_list.head (respectively, channel_list.head) and null, but excluding null, the type of the object pointed to by data(u) is @logentry (respectively, @channel). By the property of functions, this ensures that the set of nodes in the two lists are disjoint. The interesting nature of this specification is that we can specify the disjointness of the two lists by stating an invariant locally for each list.

- *Non-aliasing for lists*: We also need to ensure that each node in each linked list points to a distinct object. We use both the set constructors in the following specification:

  $$\forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{log\_list.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{NotAliased}(\texttt{data}, \{\}, \texttt{u}) \\ \forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{channel\_list.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{NotAliased}(\texttt{data}, \{\}, \texttt{u})$$

  The first invariant describes that for any node u in the linked list between log_list.head and null, but excluding null, the object pointed to by data(u) has exactly one object (namely u) pointing into it using the data field. This ensures that the data field for each object in the list points to a distinct object. The second invariant states this property for the second list.

- *Data structure invariant*: In Figure 4, the routines clear_logs and rem_channel delete pointers from the doubly-linked lists using the dlink_delete routine. The correctness of the routine relies on the input list being a doubly-linked list. We use the following macro to describe the invariants for a generic acyclic doubly-linked list DlistInv(dlist, next, prev):

  $$\begin{aligned} &\texttt{DlistInv}(\texttt{dlist}, \texttt{next}, \texttt{prev}) \equiv \\ &\quad \wedge \quad \texttt{prev}(\texttt{dlist.head}) = \texttt{null} \\ &\quad \wedge \quad \texttt{next}(\texttt{dlist.tail}) = \texttt{null} \\ &\quad \wedge \quad \texttt{Btwn}(\texttt{next}, \texttt{dlist.head}, \texttt{null}) = \\ &\qquad\qquad \texttt{Btwn}(\texttt{prev}, \texttt{dlist.tail}, \texttt{null}) \\ &\quad \wedge \quad \texttt{null} \in \mathit{Btwn}(\texttt{next}, \texttt{dlist.head}, \texttt{null}) \\ &\quad \wedge \quad \forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{dlist.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ &\qquad\qquad \texttt{u} = \texttt{dlist.head} \vee \texttt{next}(\texttt{prev}(\texttt{u})) = \texttt{u} \\ &\quad \wedge \quad \forall \texttt{u} \in \texttt{Btwn}(\texttt{prev}, \texttt{dlist.tail}, \texttt{null}) \setminus \{\texttt{null}\}. \\ &\qquad\qquad \texttt{u} = \texttt{dlist.tail} \vee \texttt{prev}(\texttt{next}(\texttt{u})) = \texttt{u} \end{aligned}$$

  The first two invariants are self-explanatory. The third invariant states that the set of objects reachable following the next field from the dlist.head is the same as the set reachable following the prev field from the dlist.tail. The fourth invariant states that the lists obtained by following the next and prev fields are both acyclic. The last two invariants constrain the fields next and prev to be fields of a doubly-linked list.

  Although the invariant looks complex, these data structure invariants have to be written only once for each type of doubly-linked list. and then instantiated for the different lists (e.g. log_list and channel_list) in the program. This predicate can be reused across all other programs that manipulate acyclic doubly-linked lists as well.

In addition to these invariants, we also need invariants stating that all objects reachable from the two lists are allocated. Moreover, for the loops iterating over the lists, we need to specify that the iterator (e.g. ptr in clear_logs) points to an object in the list. All these invariants are expressible in our logic.

4.4 Sort-restriction

To enable the algorithm described in Section 3.3 to terminate on the above queries, we need to ensure that the formulas are *sort-restricted* as well. In this section, we show that almost all the formulas in this section meet the requirement.

Let us consider the following invariant, described in Section 4.3.

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{channel\_list.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{Type}_{\texttt{P\_}\langle \texttt{logentry},\texttt{channel}\rangle}(\texttt{data}(\texttt{u})) = @\texttt{channel}$$

For this formula, the variable u of sort P_dlink_node appears as a subterm of data(u), which has a sort P_⟨logentry, channel⟩ and a subterm of $\texttt{Type}_{\texttt{P\_}\langle \texttt{logentry},\texttt{channel}\rangle}(\texttt{data}(\texttt{u}))$, which has a sort $\mathit{Integer}$. In both cases, the sorts of the subterms are less than the sort for u, according to the partial order $\prec$ described in Section 4.1.

However, consider the following invariant also described in the previous section:

$$\forall \texttt{u} \in \texttt{Btwn}(\texttt{next}, \texttt{dlist.head}, \texttt{null}) \setminus \{\texttt{null}\}. \\ \texttt{u} = \texttt{dlist.head} \vee \texttt{next}(\texttt{prev}(\texttt{u})) = \texttt{u}$$

In this formula, u appears as a strict subterm of prev(u) and next(prev(u)), both of which have the same sort as u. In fact, any legal sort assignment would equate the sorts for the terms next(prev(u)) and u, and therefore the formula can not be *sort-restricted* for any sort assignment.

It turns out that for this example (and also for the rest of the examples we consider this paper), the only two formulas that do not meet the sort restrictions are the last two invariants of DlistInv. This is not surprising because the invariant constrains the two fields next and prev that form singly-linked lists. In Section 6, we describe our solution for ensuring that the algorithm terminates on such ill-behaved formulas as well.

$$
\begin{array}{lll}
[\text{ORDER1}] & \forall x, y, z: & \{Reach(f, x, y, y), Reach(f, x, z, z)\} \\
 & & Reach(f, x, y, y) \wedge Reach(f, x, z, z) \Rightarrow \\
 & & Reach(f, x, y, z) \vee Reach(f, x, z, y) \\
[\text{TRANSITIVE1}] & \forall x, y, z: & \{Reach(f, x, y, y), Reach(f, y, z, z)\} \\
 & & Reach(f, x, y, y) \wedge Reach(f, y, z, z) \Rightarrow \\
 & & Reach(f, x, z, z)
\end{array}
$$

Figure 15. Encoding inference rules using axioms with triggers

5. Implementation

We have created an initial prototype of the decision procedure framework over existing SMT solvers, where we encode our inference rules using universally-quantified first-order axioms with appropriate matching *triggers*. Our implementation translates annotated C programs into the BoogiePL intermediate language (DeLine and Leino 2005). Each procedure in a BoogiePL program is translated into a verification condition by the Boogie verifier (Barnett and Leino 2005). Finally, the verification conditions are proved by the SIMPLIFY (Detlefs et al. 2005) and Z3 (de Moura and Bjorner 2007) automated theorem provers.

Figure 15 gives the axioms encoding two illustrative rewrite rules from Figure 13. We use predicates $Reach(f, x, y, z)$ and $In(x, y)$ to stand for the relations $x \xrightarrow{f} y \xrightarrow{f} z$ and $\in$ respectively. To avoid the use of excessive parentheses, we use the convention that $\Rightarrow$ has 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 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.

In addition to encoding the rules of our decision procedure as axioms, we also provide triggers for the universally-quantified assertions in the program. To encode the reasoning for the rule [BTWN] (Figure 14), we infer a trigger $\{Reach(f, t_1, x, t_2)\}$ for the formula $\forall x \in Btwn(f, t_1, t_2).\alpha$. To encode the reasoning for the rule [INV], we infer a trigger $In(x, f^{-1}(t))$ for the formula $\forall x \in f^{-1}(t).\alpha$. To generate the term $In(x, f^{-1}(t))$, we add the following axiom:

$$[\text{ININV}] \quad \forall y: \quad \{f(y)\} \quad In(y, f^{-1}(f(y)))$$

We automatically generate the appropriate triggers for any universally-quantified assertions that belongs to the *sort-restricted* fragment of our logic.

There are many advantages to implementing a rewriting-based decision procedure using first-order axioms over SMT solvers:

1. First, it allows us to quickly create an initial prototype for evaluation.
2. Second, it allows us to leverage efficient ground reasoning for equality, uninterpreted functions and arithmetic.
3. Finally, we can leverage the advances in matching based quantifier instantiation using triggers (Detlefs et al. 2005; de Moura and Bjorner 2007). This is useful not only for the implementation of the rewrite rules, but also allows us to express quantified invariants outside our logic in the rare cases when required. We present the need for such invariants, and our solution to deal with them in Section 6.

However, our approach has some drawbacks over a custom implementation. First, matching in typical SMT solvers is expensive. Second, a custom implementation of our decision procedure would allow us to perform additional optimizations, e.g. ordering the various rules to detect unsatisfiability faster in common cases.

Example	SIMPLIFY Old Time (s)	SIMPLIFY New Time (s)	Z3 Time (s)
`iterate`	1.8	1.4	1.5
`iterate_acyclic`	1.7	1.5	1.43
`slist_add`	1.5	1.3	1.36
`reverse_acyclic`	2.0	1.4	1.37
`slist_sorted_insert`	16.4	3.1	4.85
`dlist_add`	38.9	7.1	1.75
`dlist_remove`	45.4	2.4	1.65
`allocator`	* (901.8)	57.1	2.0
`list_appl`	*	200.1	30.22
`muh_free`	*	*	8.2

Figure 16. Results of assertion checking. 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. For `allocator`, time inside the parenthesis denotes the runtime after manual decomposition.

6. Evaluation

We have used the decision procedure presented in this paper in the tool HAVOC (Chatterjee et al. 2007), and performed a set of preliminary experiments for verifying small to medium sized C benchmarks. HAVOC is a tool for checking properties of heap-manipulating C programs. The memory model in HAVOC accounts for additional complications of low-level C programs, including pointer arithmetic, internal pointers, nested structures, unions and arrays. The main differences over the memory model presented in this paper are: (i) each expression evaluates to a pointer type `ptr` : (`Obj`, `int`) consisting of an object and an offset; and (ii) there is a single map `Mem` : `ptr` $\rightarrow$ `ptr` for the entire memory. This low-level model is required to maintain soundness across pointer arithmetic and internal pointers in C. HAVOC also uses a alternate variant of the reachability predicate presented in this paper, called *well-founded* reachability predicate (Lahiri and Qadeer 2006; Chatterjee et al. 2007).[1] The rules presented in the paper were suitably extended to account for this memory model and reachability predicate.

Figure 16 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. `dlist_add` and `dlist_remove` are the insertion and deletion routines for cyclic doubly-linked lists. `allocator` is a low-level custom storage allocator; it maintains a list of freed regions in an object and returns a region whose size satisfies the clients request. `list_appl` is a simple application with multiple doubly-linked lists, parent pointers, and uses the primitive doubly-linked list operations. `muh_free` is a simplified version of the `muh` example presented in Section 2. The examples range from 10 to 150 lines of C code. For all these examples, we check a set of partial correctness properties including

[1] The well-founded reachability predicate also enjoys most of the properties of $x \xrightarrow{f} y \xrightarrow{f} z$, such as closure under weakest-precondition, and a rewriting-based decision procedure for the ground fragment. The results are present in a recent technical report (Lahiri and Qadeer 2007b).

(but not limited to) the implicit memory-safety requirements. For instance, we check that the output list of `slist_sorted_insert` is sorted; for `reverse_acyclic`, we verify that the input and the output lists have the same nodes; for `allocator`, we verify that the region returned by the application was already present in the free list and meets the size requirement; for `list_appl`, we verify that the disjoint lists satisfies certain data invariants; finally for `muh` we check the absence of double-free property.

In an earlier work (Chatterjee et al. 2007), we verified a subset of the examples in Figure 16 using an incomplete axiomatization of the reachability predicate, with universally quantified invariants. For most of the examples, we had to write down the triggers for the quantified invariants carefully; the theorem provers were quickly overwhelmed without such restrictions. The second column in Figure 16 denotes the runtime using our previous approach, using the SIMPLIFY theorem prover (reported from (Chatterjee et al. 2007)). The third and the fourth column denotes the runtime using the algorithm described in this paper (using SIMPLIFY and Z3 as the SMT provers respectively); for these cases, the triggers for the quantified invariants (with a couple of exceptions below) were generated automatically, using the scheme of Figure 15.

The results clearly indicate that the new algorithm outperforms the older axiomatization in terms of efficiency. We can now solve several new examples (`list_appl`, `muh_free`) that were not amenable to be solved by our previous approach. For the `allocator` example, the time reported inside the parenthesis (901.8 seconds) denotes the time taken to verify the example with our previous approach, using additional triggers and manual decomposition of the proof into two VCs — without these changes the example did not verify within the time limit. It illustrates the brittleness of our previous approach. The improved results with Z3 (over SIMPLIFY) also indicate that the recent advances in SMT solvers are crucial to scale better. However, the recent advances alone are not sufficient to solve these problems (as we learned from our failed attempts with Z3 with our old axiomatization). In most of these cases, the theorem provers quickly ran out of memory due to large number of (often useless) instantiations of the quantifiers. However, the real gain (not evident from the results) was in the predictability of the new approach. In our experience, most of the failed proofs in our verification effort with the new framework points at insufficient assertions or bugs in the program.

For these examples, the main source of formulas that do not fit the *sort-restricted* fragment of $LISBQ$ comes from specification of the doubly-linked list invariant. For the following doubly-linked list assertion mentioned in Section 2:

$$\forall \mathtt{u} \in \mathtt{Btwn}(\mathtt{next}, \mathtt{dlist.head}, \mathtt{null}) \setminus \{\mathtt{null}\}.\\ \mathtt{u} = \mathtt{dlist.head} \vee \mathtt{next}(\mathtt{prev}(\mathtt{u})) = \mathtt{u}$$

our solution has been to add a trigger $\{\mathtt{prev}(\mathtt{u})\}$ to ensure that this assertion never generates a new term $\mathtt{prev}(\mathtt{t})$ after instantiating $\mathtt{u}$ with $\mathtt{t}$. Note that even though a new term $\mathtt{next}(\mathtt{prev}(\mathtt{t}))$ could still be generated after instantiation, asserting this literal $\mathtt{next}(\mathtt{prev}(\mathtt{t})) = \mathtt{t}$ in the context would cause this term to be equated with an existing term $\mathtt{t}$. This restriction ensures that the instantiations terminates even in the presence of such formulas.

7. Related work

In this work, we have augmented first-order SMT solvers with useful theories for precise verification of heap-manipulating programs. We discuss the various works that are similar in spirit to our goal of automatically verifying such programs.

Nelson (1983) presents a ground logic with the ternary predicate $x \xrightarrow[z]{f} y$, and an axiomatization for the logic. No claim is made about the completeness of the axiomatization, but the paper provides the weakest precondition for the predicate. Rakamarić et al. (2007) provide a rewriting-based decision procedure for the ground fragment of our logic with $x \xrightarrow{f} y \xrightarrow{f} z$. However, they do not provide the weakest precondition for the predicate, and are imprecise across updates to the linking fields. In addition, the rewrite rules in our decision procedure are fewer and simpler resulting in a simpler proof of completeness. Balaban et al. (2005) 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. In all these cases, the logics are strictly less expressive than ours since they do not have any support for quantifiers — as a result they cannot express most of the properties that we discuss in this paper.

Ranise and Zarba (2006) present a decidable ground logic that combines reachability constraints with arithmetic. But they provide no implementation to evaluate the feasibility of their approach. Moreover, the logic can't express many properties of collections (such as sortedness of lists), since it does not provide support for quantifiers. Kuncak and Rinard (2005) 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.

There have been several other attempts at first-order axiomatization of reachability (Lev-Ami et al. 2005; Lahiri and Qadeer 2006), which are incomplete. McPeak and Necula (2005) use decidable fragment of first-order logic augmented with arithmetic on scalar field to specify properties of data structures. However, they do not provide any theories for recursive predicates like reachability, and rely on user provided *ghost* variables to express properties of data structures — the updates to these ghost variables have to be inserted manually by the user to generate the verification conditions. However, they demonstrate completeness of quantifier instantiation for certain syntactic class of formulas that could help extend our decision procedure for doubly-linked list assertions.

Unlike the papers discussed so far that 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) (Møller and Schwartzbach 2001) 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 logic precludes the use of integer valued functions and the decision procedure for the logic has high complexity. The work of Yorsh et al. (2006) on the logic of reachable patterns is in a similar direction. They provide a logic for expressing complex shape properties, show how to generate precise verification conditions and provide a decision procedure by translation to monadic second-order logic.

Separation logic (Reynolds 2002) has been proposed to reason about heap-manipulating programs. Berdine et al. (2004) describe a rewrite-based decision procedure for a fragment of separation logic with linked lists. Among other things, it is not clear how to harness efficient arithmetic theory reasoning in this framework.

Automatic computation of (shape) 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 (Lev-Ami and Sagiv 2000; Distefano et al. 2006) or use predicate abstraction (Graf and Saïdi 1997) with decision procedures for logics with reachability (Balaban et al. 2005; Rakamarić et al. 2007; Lahiri and Qadeer 2006). 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 (Lev-Ami et al. 2005).

8. Conclusions

In this paper, we revisit the problem of *precise* verification of heap-manipulating programs using first-order SMT solvers. To solve this problem, we present the *Logic of Interpreted Sets and Bounded Quantification* for specifying properties of heap-manipulating programs and a verifier for proving these properties. The verification is fully precise within a procedure and loop body, and is scalable across typical loop-free code fragments found in practice.

We are currently working on extending our work in two directions: First, we would like to extend our logic to support range of indices of an array as another interpreted set constructor — this would allow reasoning about rich properites of the most common data structures (arrays and lists) in a single framework. Second, we would like to perform abstraction across loop and procedure boundaries to reduce the annotation requirement by automatically inferring many annotations. The recent advances in SMT solvers and the results of this paper that leverage these advances have created a strong foundation for carrying forward this work.

Acknowledgments

We would like to thank Nikolaj Bjorner and Leonardo de Moura for help with Z3 and Amit Goel and Sava Krstić for suggesting improvements to our decision procedure.

References

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.

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.

M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.

J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification (CAV '07)*, LNCS 4590, pages 178–192, 2007.

J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS '04: Foundations of Software Technology and Theoretical Computer Science*, LNCS 3328, pages 97–109, 2004.

E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

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.

E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV '00)*, LNCS 1855, pages 154–169, 2000.

L. de Moura and N. Bjorner. Efficient Incremental E-matching for SMT Solvers. In *Conference on Automated Deduction (CADE '07)*, LNCS 4603, pages 183–198, 2007.

R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

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.

B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification (CAV '06)*, LNCS 4144, pages 81–94, 2006.

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.

P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI '05)*, pages 213–223. ACM, 2005.

S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, June 1997.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.

V. Kuncak and M. C. Rinard. Decision procedures for set-valued fields. *Electr. Notes Theor. Comput. Sci.*, 131:51–62, 2005.

R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Principles of Programming Languages (POPL '06)*, pages 115–126, 2006.

S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. Technical Report MSR-TR-2007-88, Microsoft Research, 2007a.

S. K. Lahiri and S. Qadeer. A decision procedure for well-founded reachability. Technical Report MSR-TR-2007-43, Microsoft Research, 2007b.

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.

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.

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.

Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI '01)*, pages 221–231, 2001.

Muh. Available at `http://muh.sourceforge.net/`.

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

Greg Nelson. Verifying reachability invariants of linked structures. In *Principles of Programming Languages (POPL '83)*, pages 38–47, 1983.

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.

S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Software Engineering and Formal Methods (SEFM '06)*, pages 206–215, 2006.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS '02)*, pages 55–74, 2002.

Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.

G. Yorsh, A. M. 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 '06)*, LNCS 3921, pages 94–110, 2006.