Modular Reasoning about Heap Paths via Effectively Propositional Formulas

Shachar Itzhaky
Tel Aviv University
shachar@tau.ac.il

Anindya Banerjee
IMDEA Software Institute, Madrid, Spain
anindya.banerjee@imdea.org

Neil Immerman
University of Massachusetts, Amherst, USA
immerman@cs.umass.edu

Ori Lahav
Tel Aviv University
orilahav@post.tau.ac.il

Aleksandar Nanevski
IMDEA Software Institute, Madrid, Spain
aleks.nanevski@imdea.org

Mooly Sagiv
Tel Aviv University
msagiv@acm.org

Abstract

First order logic with transitive closure, and separation logic enable elegant interactive verification of heap-manipulating programs. However, undecidabilty results and high asymptotic complexity of checking validity preclude complete automatic verification of such programs, even when loop invariants and procedure contracts are specified as formulas in these logics. This paper tackles the problem of procedure-modular verification of reachability properties of heap-manipulating programs using efficient decision procedures that are complete: that is, a SAT solver must generate a counterexample whenever a program does not satisfy its specification. By (a) requiring each procedure modifies a fixed set of heap partitions and creates a bounded amount of heap sharing, and (b) restricting program contracts and loop invariants to use only deterministic paths in the heap, we show that heap reachability updates can be described in a simple manner. The restrictions force program specifications and verification conditions to lie within a fragment of first-order logic with transitive closure that is reducible to effectively propositional logic, and hence facilitate sound, complete and efficient verification. We implemented a tool atop Z3 and report on preliminary experiments that establish the correctness of several programs that manipulate linked data structures.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Dynamic storage management

Keywords linked list; SMT; verification

1. Introduction

This paper shows how to harness existing SAT solvers for proving that a potentially recursive procedure satisfies its specification and for automatically producing counterexamples when it does not. We concentrate on proving safety properties of imperative programs manipulating linked data structures which is challenging since we need to reason about unbounded memory and destructive pointer updates. The tricky part is to identify a logic which is expressive enough to enable the modular verification of interesting procedures and properties and weak enough to enable sound and complete verification using SAT solvers.

Recently it was shown [11] how to employ effectively propositional logic (or BSR logic[1]) for verifying programs manipulating linked lists. It decides the validity of formulas of the form $\forall^*\exists^* q$ using SAT solvers where q is a quantifier free relational formula (or equivalently decides the satisfiability of $\exists^*\forall^* q$ formulas). It has been successfully used in many other contexts [18].

In this paper we show that effectively propositional logic does not suffice to naturally express the effect on the global heap when the local heap of a procedure is accessible via shared nodes from outside. For example, Fig. 1 shows a pre- and post-heap before a list pointed-to by h is reversed. The problem is how to express the change in reachability between nodes such as z_i and list nodes $1, 2, \dots, 5$: note that, e.g., nodes $3, 4, 5$ are unreachable from z_1 in the post-heap.

This paper shows that in many cases, including the above example, reachability can be checked precisely using SAT solvers. Our solution is based on the following principles:

- We follow the standard techniques (e.g., see [2, 14, 25, 29]) by requiring that the programmer defines the set of potentially modified elements.
- The programmer only specifies postconditions on local heaps and ignores the effect of paths from the global heap.
- We provide a general and exact **adaptation rule** for adapting postconditions to the global heap. This adaptation rule is expressible in a generalized version of BSR called AE^{AR}. AE^{AR} allows an extra **entry** function symbol which maps each node u in the global heap into the first node accessible from u in the local heap. In Fig. 1, z_1, z_2 and z_3 are mapped to $2, 3$ and 4, respectively. The key facts are that AE^{AR} suffices to precisely define the global reachability relationship after each procedure call and yet any AE^{AR} formula can be simulated by a BSR

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535854

[1] Due to Bernays, Schönfinkel and Ramsey.

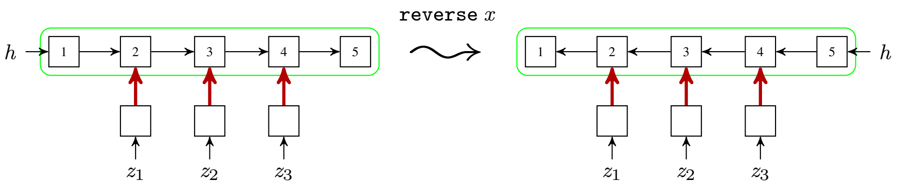

Figure 1. Reversing a list pointed to by a head h with many shared nodes accessible from outside the local heap (surrounded by a rounded rectangle).

formula. Thus the automatic methods of [11] still apply to this significantly more general setting.

- We restrict the verified procedures in order to guarantee that the generated verification condition of every procedure remains in AE^{AR}. The main restrictions are: type correctness, deterministic paths in the heap, limited number of changed list segments in the local heap (each of which may be unbounded) and limited amount of newly created heap sharing by each procedure call. These restrictions are enforced by the generated verification condition in AE^{AR}. This formula is automatically checked by the SAT solver.

1.1 Main Results

The results in this paper can be summarized as follows:

- We define a new logic, AE^{AR}, which extends BSR with a limited idempotent function and yet is equi-satisfiable with BSR.
- We provide a precise adaptation rule in AE^{AR}, which expresses the locality property of the change, and in conjunction with the postcondition on the local heap, precisely updates the reachability relation of the global heap.
- We generate a modular verification formula in AE^{AR} for each procedure, asserting that the procedure satisfies its pre- and post-conditions and the above restrictions. This verification condition is sound and complete, i.e., it is valid if and only if the procedure adheres to the restrictions and satisfies its requirements. We implemented this tool on top of Z3.
- We show that many programs can be modularly verified using our methods. They satisfy our restrictions and their BSR invariants can be naturally expressed.

1.2 A Running Example

To make the discussion more clear, we start with an example program. We use the union-find data structure[2], which maintains a forest using a parent pointer at each node (see Fig. 2) [24].

The method `find` requires that the argument x is not null. The formula $\text{tail}_f(x, r_x)$ asserts that the auxiliary variable r_x is equal to the root of x. The procedure changes the pointers of some nodes in the closed interval $[x, r_x]_f$ to point directly to r_x. Intervals are formally defined later (Definition 5). Intuitively, the closed interval $[a, b]_f$ denotes the set of nodes pointed to by a, $a.f$, $a.f.f$ and so on up until b inclusive.

The return value of `find` (denoted by retval) is r_x. The post-condition uses the symbol $\underline{f}$ that denotes the value of f before the method was invoked. Since `find` compresses paths from ancestors of x to single edges to r_x, this root may be shared via new parent pointers. Fig. 3 depicts a typical run of `find`.

@ requires $x \neq \text{null} \wedge \text{tail}_f(x, r_x)$
@ mod $[x, r_x]_f$
@ ensures $\text{retval} = r_x \wedge$ $\forall \alpha, \beta \in \text{mod} : \alpha \langle f^* \rangle \beta \leftrightarrow \alpha = \beta \vee \beta = r_x$

```
Node find(Node x) {
  Node i = x.f;
  if (i != null) {
    i = find(i);
    x.f = i;
  }
  else {
    i = x;
  }
  return i;
}
```

@ requires $x \neq \text{null} \wedge y \neq \text{null} \wedge \text{tail}_f(x, r_x) \wedge \text{tail}_f(y, r_y)$
@ mod $[x, r_x]_f \cup [y, r_y]_f$
@ ensures

$$\forall \alpha, \beta \in \text{mod} : \begin{array}{l} (x \langle \underline{f}^* \rangle \alpha \rightarrow (\alpha \langle f^* \rangle \beta \leftrightarrow \\ \qquad\qquad \beta = \alpha \vee \beta = r_x \vee \beta = r_y)) \\ \wedge (y \langle \underline{f}^* \rangle \alpha \rightarrow (\alpha \langle f^* \rangle \beta \leftrightarrow \beta = \alpha \vee \beta = r_y)) \end{array}$$

```
void union(Node x, Node y){
  Node t = find(x); Node s = find(y);
  if (t != s) t.f = s;
}
```

Figure 2. An annotated implementation of Union-Find in Java. f is the backbone field denoting the parent of a tree node.

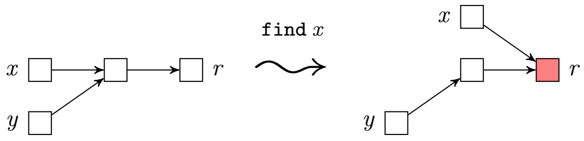

Figure 3. An example scenario of running `find` (🟥 = return value).

The method `union` requires that both its arguments are not null. It potentially modifies the ancestors of x and y, i.e., $[x, r_x]_f \cup [y, r_y]_f$. Fig. 4 depicts a typical run of `union`. Notice that we support an unbounded number of cutpoints [21] (see Section 8).

1.3 Working Assumptions

Type correct The procedure manipulates references to dynamically created objects in a type-safe way. For example, we do not support pointer arithmetic.

Deterministic Reachability The specification may use arbitrary uninterpreted relations. It may also use the reachability formula $\alpha \langle f^* \rangle \beta$ meaning that β is reachable from α via zero or more

[2] We have simplified `union` by not keeping track of the sizes of sets in order to attach the smaller set to the larger.

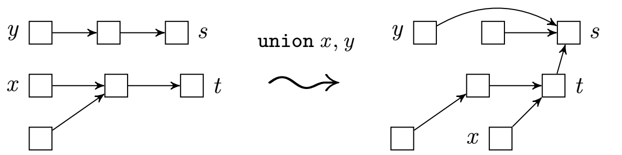

Figure 4. An example scenario of running `union`.

steps along the functional backbone field f. It may not use f in any other way. Until Section 6, we require f to be acyclic and we restrict our attention to only one backbone field.

Precondition There is a **requires clause** defining the precondition which is written in alternation-free relational first-order logic (AF^R) and may use the relation $f^\star$ [11].

Mod-set There is a **modifies clause** defining the mod-set (mod_f), which is the set of potentially changed memory locations (We include both source and target of every edge that is added or deleted). The modified set may have an unbounded number of vertices, but we require it to be the union of a bounded number of f-intervals, that is chains of vertices through f-pointers.

Postcondition There is an **ensures clause** which exactly defines the new reachability relation $f^\star$ restricted to mod_f. The ensures clause, written in AF^R, may use two vocabularies (employing both $\underline{f}$ and f to refer to the reachability relations before and after.

Bounded new sharing All the new shared nodes — nodes pointed to by more than one node — must be pointed to by local variables at the end of the procedure's execution. This requires that only a bounded number of new shared nodes can be introduced by each procedure call. Note that many heap-manipulating programs exhibit limited sharing as noted in the experimental measurements of Mitchell [16]. A similar restriction is also used in shape analysis techniques for device driver programs [27].

Loop-free We assume that all code is loop free, with loops replaced by recursive calls.

1.4 Outline of the rest of this paper

Section 2 provides a rule for adapting local changes to states containing a global heap. The idea is that the programmer only specifies changes in a small, local area of the heap. Section 3 introduces a new logic called AE^{AR}. Section 4 formalizes the requirements for specifying the meaning of commands and procedures. The technique for generating verification conditions is presented in Section 5. Extensions to the frameworks are discussed in Section 6. Our preliminary verification experience appears in Section 7. Section 8 discusses related work and Section 9 concludes. Details of the logical proof are contained in [10, Appendix A].

2. Adaptation of Local Effect to the Global Heap

Our goal is to reason modularly about a procedure that modifies a subset of the heap. We wish to automatically update the reachability relation in the entire heap based on the changes to the modified subset. We remark that in this paper we are concerned with reachability between any two nodes in the heap, as opposed to only those pointed to by program variables. When we discuss sharing we mean sharing via pointer fields in the heap as opposed to aliasing from stack variables, which does not concern us in this paper.

2.1 Non-Local Effects

Reachability is inherently non-local: a single edge mutation can affect the reachability of an unbounded number of points that are an unbounded distance from the point of change. Fig. 5 contains a typical run of `find`. Two kinds of "frames" are depicted: (i) $\mathrm{mod}_f = [x, r_x)_f$, specified by the programmer, denotes the nodes whose edges can be directly changed by `find`— this is the standard notion of a frame condition; (ii) mod_{f^*} denotes nodes for which f^*, the path relation, has changed. We do not and, in general we cannot, specify mod_{f^*} in a modular way because it usually depends on variables outside the scope of this function such as y in Fig. 5. In the example shown, there is a path from y to c before the call which does not exist after the call. Furthermore, mod_{f^*} can be an arbitrarily large set: in particular, it may not be expressible as the union of a bounded set of intervals: for example, when adding a subtree as a child of some node in another tree, mod_f spans only one edge, whereas mod_{f^*} is the entire subtree added — which may contain an unbounded number of branches.

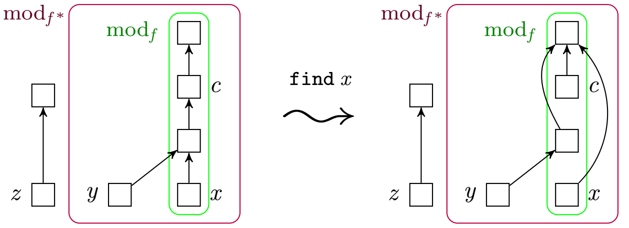

Figure 5. A case where changes made by `find` have a non-local effect: $y\langle \underline{f}^*\rangle c$, but $\neg y\langle f^*\rangle c$.

The postcondition of `find` is sound (every execution of `find` satisfies it), but incomplete: it does not provide a way to determine information concerning paths outside mod, such as from y to c in Fig. 5. Therefore, this rule is often not enough in order to verify the correctness of programs that invoke `find` in larger contexts.

Notice the difficulty of updating the global heap, especially the part $\mathrm{mod}_{f^*} \setminus \mathrm{mod}_f$. In particular, using only the local specification of `find`, one would not be able to prove that $\neg y\langle f^*\rangle c$. Indeed, the problem is updating the reachability of elements that are **outside** mod; in more complex situations, these elements may be far from the changed interval, and their number may be unbounded.

One possibility to avoid the problem of incompleteness is to specify a postcondition which is specific to the context in which the invocation occurs. However, such a solution requires reasoning per call site and is thus not modular. We wish to develop a rule that will fit in all contexts. Reasoning about all contexts is naturally done by quantification.

2.2 An FO(TC) Adaptation Rule

A standard way to modularize specifications is to specify the local effect of a procedure and then to use a general adaptation rule (or frame rule) to derive the global effect. In our case, we know that locations outside mod are not modified. Therefore, for example, after a call to `find`, a new path from node σ to node τ is either an old path from σ to τ, or it consists of an old path to a node $\alpha \in \mathrm{mod}$, a new path from α to a node $\beta \in \mathrm{mod}$ and an old path from β to τ. We express this below, first letting q denote an old edge that is not inside mod:

$$\begin{array}{lcl} \forall\alpha,\beta : \alpha\langle q\rangle\beta & \leftrightarrow & \alpha\langle \underline{f}\rangle\beta \wedge (\alpha \notin \mathrm{mod} \vee \beta \notin \mathrm{mod}) \\ \forall\sigma,\tau : \sigma\langle f^*\rangle\tau & \leftrightarrow & \sigma\langle q^*\rangle\tau \ \vee\ \exists\alpha,\beta \in \mathrm{mod} : \\ & & \sigma\langle q^*\rangle\alpha \ \wedge\ \alpha\langle f^*\rangle\beta \ \wedge\ \beta\langle q^*\rangle\tau \end{array} \quad (1)$$

eq (1) is a completely general adaptation rule: it defines f^* on the global heap assuming we know f^* on the local heap and we also have access to the old path relation q^*. The problem with this rule is that it uses a logic that is too expressive and thus hard for automated reasoning: FO(TC) is not decidable (in fact, not even

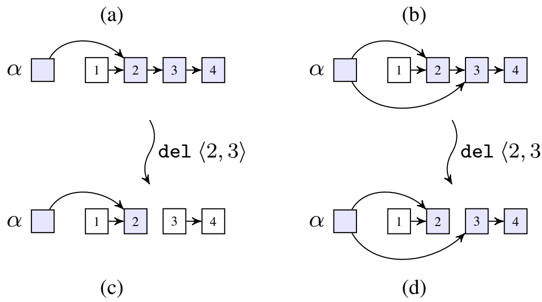

Figure 6. Memory states with non-unique pointers where global reasoning about reachability is hard. In the memory state (a), there is one edge from α into the modified-set $\{1, 2, 3, 4\}$, and in memory state (b), there are two edges from α into the same modified-set, $\{1, 2, 3, 4\}$. The two memory states have the same reachability relation and therefore are indistinguishable in terms of reachability. The memory states (c) and (d) are obtained from the memory states (a) and (b), respectively, by deleting the edge $\langle 2, 3\rangle$. The reachability in (c) is not the same as in (d), which shows it is impossible to update reachability in general w.r.t. edge deletion, without using the edge relation.

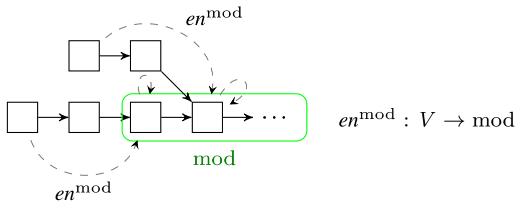

Figure 7. The function en^{mod} maps every node σ to the first node in mod reachable from σ. Notice that for any $\alpha \in \mathrm{mod}$, $en^{\mathrm{mod}}(\alpha) = \alpha$ by definition.

recursively enumerable). The first problem is that the q^* relation is not usually first order expressible and generally requires transitive closure. For example, Fig. 6 shows that in general the adaptation rule is not necessarily definable using only the reachability relation, when there are multiple outgoing edges per node. We avoid this problem by only reasoning about functional fields, f.

The second problem with eq (1) is that it contains quantifier alternation. α matches an arbitrary node in mod which may be of arbitrary size. Therefore, it is not completely obvious how to avoid existential quantifications.

2.3 An Adaptation Rule in a Restricted Logic

We now present an equivalent adaptation rule in a restricted logic, without transitive closure or extra quantifier-alternations. This is possible due to our assumptions from Section 1.3 and it greatly simplifies reasoning about modified paths in the entire heap. We require a new function symbol, en^{mod}. We call $en^{\mathrm{mod}}(\sigma)$ the **entry point** of σ in mod, i.e., the first node on the (unique) path from σ that enters mod, and null if no such node exists (see Figure 7).

Note that since transitive closure is only applied to functions, entry points such as α in eq (1) are uniquely determined by σ, the origin of the path. A key property of en^{mod} is that on mod itself, en^{mod} is the identity, and therefore for any $\sigma \in V$ it holds that $en^{\mathrm{mod}}(en^{\mathrm{mod}}(\sigma)) = en^{\mathrm{mod}}(\sigma))$ — that is, the function en^{mod} is **idempotent**. It is important to note that en^{mod} does not change as a result of local modifications in mod. Hence, we do not need to worry about en^{mod} in the pre-state as opposed to the post-state. Formally, en^{mod} is characterized by the following formula:

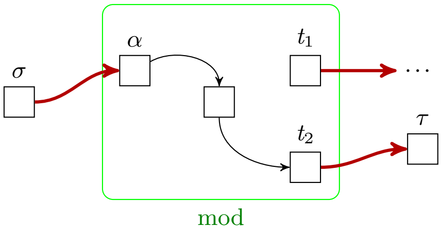

Figure 8. This diagram depicts how an arbitrary path from $\sigma \notin \mathrm{mod}$ to $\tau \notin \mathrm{mod}$ is constructed from three segments: $[\sigma, \alpha]_{\underline{f}}$, $[\alpha, t_i]_f$, and $[t_i, \tau]_{\underline{f}}$ (here $i = 2$). Arrows in the diagram denote paths; thick arrows entering and exiting the box denote paths that were not modified since they are outside of mod. Here, $\alpha = en^{\mathrm{mod}}(\sigma)$ is an entry-point and t_1, t_2 are exit-points.

$$\begin{aligned}\forall\sigma :\quad & (en^{\mathrm{mod}}(\sigma) = \mathrm{null} \land \forall\alpha \in \mathrm{mod} : \neg\sigma\langle \underline{f}^*\rangle\alpha) \lor \\ & (\sigma\langle \underline{f}^*\rangle en^{\mathrm{mod}}(\sigma) \land en^{\mathrm{mod}}(\sigma) \in \mathrm{mod} \land \\ & \forall\alpha \in \mathrm{mod} : \sigma\langle \underline{f}^*\rangle\alpha \to en^{\mathrm{mod}}(\sigma)\langle \underline{f}^*\rangle\alpha)\end{aligned} \qquad (2)$$

Using en^{mod} the new adaptation rule $\mathrm{adapt}[\mathrm{mod}]$ is obtained by considering, for every source and target, the following three cases:

Out-In: The source is out of mod; the target is in;
In-Out: The source is in mod; the target is out;
Out-Out: The source and target are both out of mod.

The full adaptation rule is obtained by taking the conjunction of the formulas for each case (eq (3), eq (4), eq (5)), that are described below, and the formula defining en^{mod} (eq (2)).

Out-In Paths Using en^{mod} we can easily handle paths that enter mod. Such paths originate at some $\sigma \notin \mathrm{mod}$ and terminate at some $\tau \in \mathrm{mod}$. Any such path therefore has to go through $en^{\mathrm{mod}}(\sigma)$ as depicted in Fig. 8. Thus, the following simple formula can be used:

$$\forall\sigma \notin \mathrm{mod}, \tau \in \mathrm{mod} : \sigma\langle f^*\rangle\tau \leftrightarrow en^{\mathrm{mod}}(\sigma)\langle f^*\rangle\tau \qquad (3)$$

Observe that for any $\beta \in \mathrm{mod}$, the atomic formula used above, $en^{\mathrm{mod}}(\sigma)\langle f^*\rangle\beta$, corresponds to the FO(TC) sub-formula $\exists\alpha \in \mathrm{mod} : \sigma\langle q^*\rangle\alpha \land \alpha\langle f^*\rangle\beta$ from eq (1).

In-Out Paths We now shift attention to paths that exit mod. Exit points, that is, last points on some path that belong to mod, are more subtle since both ends of the path are needed to determine them. The end of the path is not enough since it can be shared, and the origin of the path is not enough since it can exit the set multiple times, because a path may exit mod and enter it again later. Therefore, we cannot define a function in a similar manner to en^{mod}. The fact that transitive closure is only applied to functions is useful here: every interval $[\alpha, \beta]$ has at most one exit β. We therefore utilize the fact that mod is expressed as a bounded union of intervals — which bounds the potential exit points to a bounded set of terms. We will denote the exit points of mod by t_i.

For example, in the procedure `swap` shown in Fig. 9, $\mathrm{mod} = [x, f_x^3]$ and there is one exit point $t_1 = f_x^3$ (f_x^3 is a constant set by the precondition to have the value of $f(f(f(x)))$ using the inversion formula eq (6) to be introduced formally in Section 3.1).

Any path that originates in mod and terminates outside mod must leave through a last exit point t_i (see Fig. 10). Notice that the exit points also do not change as a result of modifying edges between nodes in mod. Let p be a path from σ to τ and let t_i be the last exit point along p. Note that the part of the path from t_i to

@ requires $E_f(x, f_x^1) \wedge E_f(f_x^1, f_x^2) \wedge E_f(f_x^3, f_x^2) \wedge$
$x \neq \text{null} \wedge f_x^1 \neq \text{null} \wedge f_x^2 \neq \text{null}$
@ mod $[x, f_x^3]$
@ ensures …

```
void swap(Node x) {
  Node t = x.f;
  x.f = t.f;
  t.f = x.f.f;
  x.f.f = t;
}
```

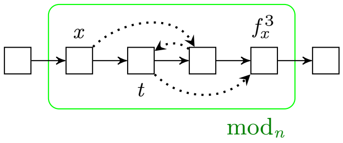

Figure 9. A simple function that swaps two adjacent elements following x in a singly-linked list. Dotted lines denote the new state after the swap. The notation e.g. $E_f(x, f_x^1)$ denotes the single edge from x to f_x^1 following the f field.

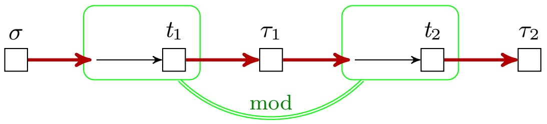

Figure 10. A subtle situation occurs when the path from σ passes through multiple exit-points. In such a case, the relevant exit-point for $\sigma\langle f^*\rangle\tau_1$ is t_1, whereas for $\sigma\langle f^*\rangle\tau_2$ and $\tau_1\langle f^*\rangle\tau_2$ it would be t_2.

```
void swap_two(Node a, Node b) {
  swap(a); swap(b);
}
```

Figure 11. An example of a procedure where the mod-set is not (essentially) convex.

τ consists only of unchanged edges — since they are all outside of mod. We can therefore safely use $\underline{f}^*$, rather than q^*, to characterize it. The part of the path from σ to t_i can be characterized by f^*, because σ and t_i are both in mod. Therefore the entire path can be expressed as $\sigma\langle f^*\rangle t_i \wedge t_i\langle \underline{f}^*\rangle\tau$. Thus, we obtain the following formula:

$$\begin{array}{l}\forall\sigma \in \text{mod}, \tau \notin \text{mod}: \ \sigma\langle f^*\rangle\tau \ \leftrightarrow \\ \quad \bigvee_{t_i}(\sigma\langle f^*\rangle t_i \wedge t_i\langle \underline{f}^*\rangle\tau \ \wedge \ \bigwedge_{t_j\neq t_i} t_j \notin [t_i, \tau]_{\underline{f}})\end{array} \tag{4}$$

Note that eq (4) corresponds the sub-formula $\exists\beta : \alpha\langle f^*\rangle\beta \wedge \beta\langle q^*\rangle\tau$ in eq (1).

Out-Out Paths For paths between σ and τ, both outside mod, there are two possible situations:

- The path goes **through** mod (as in Fig. 8). In this case, we can reuse the in-out case, by taking $en^{\text{mod}}(\sigma)$ instead of σ.
- The path is entirely outside of mod (see Fig. 12).

The corresponding formula in this case is:

$$\begin{array}{l}\forall\sigma \notin \text{mod}, \tau \notin \text{mod}: \ \sigma\langle f^*\rangle\tau \ \leftrightarrow \\ \quad \bigvee_{t_i}(en^{\text{mod}}(\sigma)\langle f^*\rangle t_i \wedge t_i\langle \underline{f}^*\rangle\tau \wedge \bigwedge_{t_j\neq t_i} t_j \notin [t_i, \tau]_{\underline{f}}) \\ \quad \vee \quad en^{\text{mod}}(\sigma) = en^{\text{mod}}(\tau) \ \wedge \ \sigma\langle \underline{f}^*\rangle\tau\end{array} \tag{5}$$

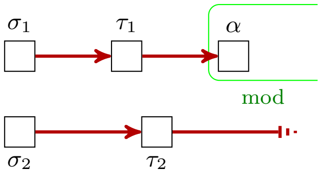

Figure 12. Paths that go entirely untouched. $en^{\text{mod}}(\sigma_1) = \alpha$, whereas $en^{\text{mod}}(\sigma_2) = \text{null}$.

Notice that the second disjunct covers the case where there is a path from τ to mod ($en^{\text{mod}}(\sigma) = en^{\text{mod}}(\tau) \neq \text{null}$) and the case where there is none ($en^{\text{mod}}(\sigma) = en^{\text{mod}}(\tau) = \text{null}$).

In conclusion, our adaptation rule, adapt[mod], is the conjunction of the three formulas in eq (3), eq (4), eq (5), and the formula defining en^{mod} (eq (2)). We need some more formalism, introduced in the next section, before we show that adapt[mod] meets our needs.

3. Adaptable Heap Reachability Logic

In this section we introduce an extension of AE^R from [11], called **adaptable heap reachability logic**, and denoted by AE^{AR}. This extension still has the attractive property of AE^R, as it is effectively reducible to the function-free $\forall^*\exists^*$-fragment of first-order logic, and thus its validity can be checked by a SAT-solver.

3.1 Preliminaries

This section reviews the AF^R (alternation free) and AE^R ($\forall\exists$) logics from [11]. They are decidable for validity since their negation corresponds to the BSR fragment [18]. These logics include the relation f^* (the reflexive transitive closure of f) but forbid the explicit use of function symbols including f. Until Section 6 we will use at most one designated backbone function (f) per formula.

Definition 1. A **vocabulary** $\mathcal{V} = \langle \mathcal{C}, \{f\}, \mathcal{R}\rangle$ is a triple of constant symbols, function symbol, relation symbols.
A **term**, t, is a variable or constant symbol.
An **atomic formula** is one of the following: (i) $t_1 = t_2$;
(ii) $r(t_1, t_2, \ldots, t_a)$ where r is a relation symbol of arity a;
(iii) $t_1\langle f^*\rangle t_2$.

A **quantifier-free formula** (QF^R) is a boolean combination of atomic formulas. A **universal formula** begins with zero or more universal quantifiers followed by a quantifier-free formula. An **alternation-free formula** (AF^R) is a boolean combination of universal formulas. AE^R consists of formulas with quantifier-prefix $\forall^*\exists^*$.

In particular, $QF^R \subset AF^R \subset AE^R$. The preconditions and the postconditions in Fig. 2 are all AF^R formulas.

Decidability and Inversion Every AE^R formula can be translated to a first-order $\forall^*\exists^*$ formula via the following steps [11]. (i) Add a new uninterpreted relation R_f which is intended to represent $\langle f^*\rangle$, the reflexive transitive closure of reachability via f, (ii) Add the consistency rule Γ_{linOrd} shown in Table 1, which requires that R_f is a total order, i.e., reflexive, transitive, acyclic, and linear, and (iii) Replace all occurrences of $t_1\langle f^*\rangle t_2$ by $R_f(t_1, t_2)$.

Proposition 1 (Simulation of AE^R)**.** *[12, Proposition 3, Appendix A.1] Consider AE^R formula φ over vocabulary $\mathcal{V} = \langle \mathcal{C}, \{f\}, \mathcal{R}\rangle$. Let $\varphi' \overset{\text{def}}{=} \varphi[R_f(t_1, t_2)/t_1\langle f^*\rangle t_2]$. Then φ' is a first-order formula over vocabulary $\mathcal{V}' = \langle \mathcal{C}, \emptyset, \mathcal{R} \cup \{R_f\}\rangle$ and $\Gamma_{linOrd} \rightarrow \varphi'$ is valid if and only if the original formula φ is valid.*

$\forall\alpha : R_f(\alpha,\alpha) \quad \wedge$	reflexivity
$\forall\alpha,\beta,\gamma : R_f(\alpha,\beta) \wedge R_f(\beta,\gamma) \rightarrow R_f(\alpha,\gamma) \quad \wedge$	transitivity
$\forall\alpha,\beta : R_f(\alpha,\beta) \wedge R_f(\beta,\alpha) \rightarrow \alpha = \beta \quad \wedge$	acyclicity
$\forall\alpha,\beta,\gamma : R_f(\alpha,\beta) \wedge R_f(\alpha,\gamma) \rightarrow (R_f(\beta,\gamma) \vee R_f(\gamma,\beta))$	linearity

Table 1. A universal formula Γ_{linOrd} requiring that all points reachable from a given point are linearly ordered.

When the graph of f is acyclic, the relation E_f characterizing the function f can be recovered from its reflexive transitive closure, f^*, at the cost of an extra universal quantifier:

$$E_f(\alpha,\beta) \overset{\text{def}}{=} \alpha\langle f^+\rangle\beta \wedge \forall\gamma : \alpha\langle f^+\rangle\gamma \rightarrow \beta\langle f^*\rangle\gamma \tag{6}$$

Here $\alpha\langle f^+\rangle\beta \overset{\text{def}}{=} \alpha\langle f^*\rangle\beta \wedge \alpha \neq \beta$.

3.2 Adaptable Heap Reachability Logic

The new logic AE^{AR} is obtained by augmenting AE^R with unary function symbols, denoted by $g, h_1, \ldots, h_n$ where:

- g must be interpreted as an idempotent function.
- The images $h_1, \ldots, h_n$ are all bounded by some pre-determined parameter N, that is: each h_i takes at most N distinct values.
- Function symbols may not be nested, i.e., all terms involving function symbols have the form $f(z)$, where z is a variable.

We later show that AE^{AR} suffices for expressing the verification conditions of the programs discussed above. In the typical use case, the function g assigns the entry point in the mod-set for every node (called en^{mod_f} above), and the functions $h_1, \ldots, h_n$ are used for expressing the entry points in inner mod-sets. The main attractive feature of this logic is given in the following theorem.

Theorem 1. *Any AE^{AR}-formula φ can be translated to an equi-valid (first-order) function-free $\forall^*\exists^*$-formula.*

The proof of Theorem 1, given in [10, Appendix A], begins by translating φ to a $\forall^*\exists^*$-formula φ' as described in Proposition 1, without modifying the function symbols $g, h_1, \ldots, h_n$. The function symbols are then replaced by new relation and constant symbols. We add new universal formulas to express the above semantic restrictions on the functions.

4. Modular Specifications of Procedure Behaviours

4.1 Notations

Definition 2. Let $\mathcal{V} = \langle \mathcal{C}, \{f\}, \mathcal{R}\rangle$ be a vocabulary including the constant symbol null. A **state**, M, is a logical structure with domain $|M|$, including null, and $\text{null}^M = \text{null} = f^M(\text{null})$, where s^M is the interpretation of the symbol s in the structure M. A state is **appropriate** for an annotated procedure *proc* if its vocabulary includes every symbol occurring in its annotations, and constants corresponding to all of the program variables occurring in *proc*.

The diagrams in this paper denote states. For example Fig. 1 shows a transition between two states.

Below we define the notion of two-vocabulary structures, which are useful to describe relations between pre- and post-states.

Definition 3 (Two-vocabulary Structure)**.** *For states $\underline{M}$ and M over the same vocabulary $\mathcal{V} = \langle \mathcal{C}, \{f\}, \mathcal{R}\rangle$ and with the same domain ($|\underline{M}| = |M|$), we denote by $\underline{M}/M$ the structure over the two-vocabulary $\mathcal{V}' = \langle \mathcal{C} \cup \underline{\mathcal{C}} \setminus \{\underline{\text{null}}\}, \{f, \underline{f}\}, \mathcal{R} \cup \underline{\mathcal{R}}\rangle$ obtained by combining $\underline{M}, M$ in the following way: f is interpreted as f^M and $\underline{f}$ is interpreted as $f^{\underline{M}}$, and similarly for all the other symbols.*

Definition 4 (Backbone Differences)**.** *For states $\underline{M}$ and M over the same vocabulary $\mathcal{V} = \langle \mathcal{C}, \{f\}, \mathcal{R}\rangle$ and with the same domain ($|\underline{M}| = |M|$), the set $\underline{M} \oplus M$ consists of the "differences between $\underline{M}$ and M w.r.t. f, excluding null", i.e. all elements u of $|M|$ such that $f^M(u) \neq f^{\underline{M}}(u)$, as well as their non-$\text{null}$ images in f^M and $f^{\underline{M}}$.*

For example, in Fig. 3, let $\underline{M}$ be the left structure and M the right structure. Then $M \oplus \underline{M} = \{x, r, \Box\}$, where $\Box$ is the unlabeled node with an edge from x in the left diagram.

4.2 Modification Set

We must specify the **mod-set**, mod, containing all the endpoints of edges that are modified. Therefore when adding or deleting an edge $\langle s, t\rangle$, both ends — the source, s, and the target, t — are included in mod. Often in programming language semantics, only the source is considered modified. However, thinking of the heap as a graph, it is useful to consider both ends of a modified edge as modified. For example, in the running example program `find` (Fig. 2), since new references to r_x may be introduced as a result of path compression, the root r_x is also considered as part of mod.

Our mod-sets are built from two kinds of intervals:

Definition 5 (Intervals)**.** The **closed interval** $[a, b]_f$ is

$$[a, b]_f \overset{\text{def}}{=} \{\gamma \mid a\langle f^*\rangle\gamma \wedge \gamma\langle f^*\rangle b\}$$

and the **half-open interval** $[a, \text{null})_f$ is:

$$[a, \text{null})_f \overset{\text{def}}{=} \{\gamma \mid a\langle f^*\rangle\gamma \wedge \gamma\langle f^+\rangle \text{null}\}$$

(notice that $\gamma\langle f^*\rangle\text{null}$ is always true in acyclic heaps).

Definition 6 (mod-set)**.** *The mod-set,* mod*, of a procedure is a union $I_1 \cup I_2 \cup \ldots \cup I_k$, where each I_i may be $[s_i, t_i]_f$ or $[s_i, \text{null})_f$, s_i, t_i are parameters of the procedure or constant symbols occurring in the pre-condition.*

In our examples, the mod-sets of `find` and `union` are written above each procedure, preceded by the symbol "@ mod' (Fig. 2). Note that it follows from Definition 6 that $\alpha \in \text{mod}$, is expressible as a quantifier-free formula.

Definition 7. *Given an appropriate state M for proc with modset* mod*,* mod^M *is the set of all elements in $|M|$ that are in one of the intervals defining* mod *(see Definition 5).*

4.3 Pre- and Post-Conditions

The programmer specifies AF^R pre- and post-conditions. Two-vocabulary formulas may be used in the post-conditions where $\underline{f}$ denotes the value of f before the call.

4.4 Specifying Atomic Commands

Table 2 provides specification of atomic commands. They describe the memory changed by atomic statements and the changes on the local heap.

Accessing a pointer field The statement `ret = y.f` reads the content of the f-field of `y`, into `ret`. It requires that `y` is not `null` and that an auxiliary variable `s` points to the f-field of `y` (which may be `null`). It does not modify the heap at all. It sets `ret` to `s`.

Command	Pre	Mod	Post
retval = `y.f`	$y \neq \text{null} \wedge E_f(y, s)$	$\emptyset$	$\text{retval} = s$
`y.f = null`	$y \neq \text{null} \wedge E_f(y, s)$	$[y, s]_f$	$\neg y\langle f^*\rangle s \wedge \neg s\langle f^*\rangle y$

`assume y.f==null;`
`y.f = x`
	$y \neq \text{null} \wedge E_f(y, \text{null}) \wedge \neg x\langle f^*\rangle y$	$[y, y]_f \cup [x, x]_f$	$y\langle f^*\rangle x \wedge \neg x\langle f^*\rangle y$

Table 2. The specifications of atomic commands. s is a local constant denoting the f-field of y. E_f is the inversion formula defined in eq (6).

It is interesting to note that the postcondition is quantfier free and much simpler than the one provided in [11]. The reason is that we do not need to specify the effect on the whole heap.

Edge Removals The statement `y.f = null` sets the content of the f-field of `y`, to `null`. It requires that `y` is not `null` and that an auxiliary variable `s` points to the f-field of `y` (which may be `null`). It modifies the node pointed-to by `y` and potentially the node pointed-to by `s`. Notice that the modset includes the elements pointed to by y and s, the two end-points of the edge. It removes paths between `y` and `s`. The postcondition asserts that there are no paths from y to s. Also, since s is potentially modified, it asserts that there are no paths from s to y.

Edge Additions The statement `y.f = x` is specified assuming without loss of generality, that the statement `y.f = null` was applied before it. Thus, it only handles edge additions. It therefore requires that `y` is not `null` and its f-field is `null`. It modifies the node pointed-to by `y` and potentially the node pointed-to by `x`. It creates a new path from `y` to `x` and asserts the absence of new paths from `x` back to `y`. Again the absence of back paths (denoted by $\neg x\langle f^*\rangle y$) is needed for completeness. The reason is that both the node pointed-to by `x` and `y` are potentially modified. Since `x` is potentially modified, without this assertion, a post-state in which `x.f == y` will be allowed by the postcondition.

4.4.1 Soundness and Completeness

We now formalize the notion of soundness and completeness of modular specifications and assert that the specifications of atomic commands are sound and complete.

Definition 8 (Soundness and Completeness of Procedure Specification)**.** *Consider a procedure proc with precondition P, modset* mod*, post-condition Q. We say that $\langle P, \text{mod}, Q\rangle$* ***is sound with respect to*** *proc if for every appropriate pre-state $\underline{M}$ such that $\underline{M} \models P$, and appropriate post-state M which is a potential outcome of the body of proc when executed on $\underline{M}$: (i) $\underline{M} \oplus M \subseteq \text{mod}^{\underline{M}}$, (ii) $\underline{M}/M \models Q$. Such a triple $\langle P, \text{mod}, Q\rangle$* ***is complete with respect to*** *proc if for every appropriate states $\underline{M}$, M such that (i) $\underline{M} \models P$, (ii) $\underline{M}/M \models Q$, and (iii) $\underline{M} \oplus M \subseteq \text{mod}^{\underline{M}}$, then there exists an execution of the body of proc on $\underline{M}$ whose outcome is M.*

The following proposition establishes the correctness of atomic statements.

Proposition 2 (Soundness and Completeness of Atomic Commands)**.** *The specifications of atomic commands given in Table 2 are sound and complete.*

The following lemma establishes the correctness of `find` and `union`, which is interesting since they update an unbounded amount of memory.

Lemma 1 (Soundness and Completeness of Union-Find)**.** *The specification of `find` and `union` in Fig. 2 is sound and complete.*

We can now state the following proposition:

Proposition 3 (Soundness and Completeness of $\text{adapt}[]$)**.** *Let* mod *be a mod-set of some procedure proc. Let M and $\underline{M}$ be two appropriate states for proc. Then, $\underline{M} \oplus M \subseteq \text{mod}^{\underline{M}}$ iff $\underline{M}/M$ augmented with some interpretation for the function symbol en^{mod} is a model of* $\text{adapt}[\text{mod}]$.

$$\begin{array}{rcl}
wp[\![\mathsf{skip}]\!](Q) & \overset{\text{def}}{=} & Q \\
wp[\![x := y]\!], Q) & \overset{\text{def}}{=} & Q[y/x] \\
wp[\![S_1 \ ; \ S_2]\!](Q) & \overset{\text{def}}{=} & wp[\![S_1]\!]\big(wp[\![S_2]\!](Q)\big) \\
wp[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!](Q) & \overset{\text{def}}{=} & [\![B]\!] \wedge wp[\![S_1]\!](Q) \ \vee \\
 & & \neg[\![B]\!] \wedge wp[\![S_2]\!](Q)
\end{array}$$

Table 3. Standard rules for computing weakest liberal preconditions for loop free code without pointer accesses. $[\![B]\!]$ is the AF^R formula for program conditions and Q is the postcondition expressed as an AF^R formula.

5. Generating Verification Condition for Procedure With Sub-calls in AE^{AR}

We follow the standard procedures, e.g. [26], which generates a verification condition for a procedure annotated with specification using weakest (liberal) preconditions. Roughly speaking, for a Hoare triple $\{P\}\mathit{prog}\{Q\}$, we generate the usual verification condition $vc[\mathit{prog}] = P \to wp[\![\mathit{prog}]\!](Q)$.

For the basic commands (assignment, composition, and conditional) we employ the standard definitions, given in Table 3.

5.1 Modular Verification Conditions

The modular verification condition would also contain a conjunct for checking that mod affected by the invoked procedure is a subset of the "outer" mod. This way the specified restriction can be checked in AE^{AR} and the SMT solver can therefore be used to enforce it automatically.

5.2 Weakest-precondition of Call Statements

As discussed in Section 2, the specification as it appears in the "ensures" clause of a procedure's contract is a local one, and in order to make reasoning complete we need to adapt it in order to handle arbitrary contexts. This is done by conjoining Q_{proc} occurring in the "ensures" clause from the specification of $proc$ with the universal adaptation rule $\text{adapt}[\text{mod}]$, where mod is replaced with the mod-set as specified in the "modifies" clause of $proc$.

Table 4 presents a formula for the weakest-precondition of a statement containing the single procedure call, where the invoked procedure has the specifications as in Fig. 13, where "$proc$" has the formal parameters $\overline{x} = x_1, \dots, x_k$, and it is used with $\overline{a} = a_1, \dots, a_k$ (used in the formula) as the actual arguments for a specific procedure call; we assume w.l.g. that each a_i is a local variable of the calling procedure.

In general it is not obvious how to enforce that the set of locations modified by inner calls is a subset of the set of locations declared by the outer procedure. Moreover, this can be tricky to check since it depends on aliasing and paths between nodes pointed

@ requires P_{proc}
@ mod $\quad \mathrm{mod}_{proc}$
@ ensures Q_{proc}
return-type $proc(\overline{x})$ `{ ... }`

Figure 13. Specification of *proc* with placeholders.

$$\begin{aligned} wp[\![r := proc(\overline{a})]\!](Q) \overset{\text{def}}{=} &\\ & P_{proc}[\overline{a}/\overline{x}] \wedge \\ & \forall \alpha : \alpha \in \mathrm{mod}_{proc}[\overline{a}/\overline{x}] \rightarrow \alpha \in \mathrm{mod}_{prog} \wedge \\ & \forall \zeta : Q_{proc}[\overline{a}/\overline{x}, \widehat{f}/f, f/\underline{f}, \zeta/\mathrm{retval}] \wedge \\ & \quad \mathrm{adapt}\left[\mathrm{mod}_{proc}[f/\underline{f}]\right][\overline{a}/\overline{x}, \widehat{f}/f, f/\underline{f}] \rightarrow \\ & \quad Q[\widehat{f}/f, \zeta/r] \end{aligned}$$

Table 4. Computing the weakest (liberal) precondition for a statement containing a procedure call. r is a local variable that is assigned the return value; $\overline{a}$ are the actual arguments passed. $\widehat{f}$ is a fresh function symbol.

to by different variables. Fortunately, the sub-formula $\forall \alpha : \alpha \in \mathrm{mod}_{proc}[\overline{a}/\overline{x}] \rightarrow \alpha \in \mathrm{mod}_{prog}$ captures this property, ensuring that the outer procedure does not exceed its own mod specification, taking advantage of the interval-union structure of the mod. Since all the modifications (even atomic ones) are done by means of procedure calls, this ensures that no edges incident to nodes outside mod are changed.

Proposition 4. *The rule for wp*$[\![]\!]$ *of call statements is sound and complete, that is, when proc is a procedure with specification as in Fig. 13, called in the context of prog whose mod-set is* mod*:*

$$\begin{gathered} \underline{M} \models wp[\![r := proc(\overline{a})]\!](Q) \\ \Updownarrow \\ \underline{M} \models P_{proc}[\overline{a}/\overline{x}] \wedge \mathrm{mod}_{proc}^{\underline{M}} \subseteq \mathrm{mod}^{\underline{M}} \wedge \\ \forall M : \left(\underline{M}/M \models Q_{proc}[\overline{a}/\overline{x}] \wedge \underline{M} \oplus M \subseteq \mathrm{mod}_{proc}^{\underline{M}}\right) \\ \Rightarrow M[r \mapsto \mathrm{retval}^M] \models Q \end{gathered} \tag{7}$$

5.3 Reducing Function Symbols

Notice that when we apply the adaptation rule for AE^{AR}, as discussed above, it introduces a new function symbol en^{mod} depending on the concrete mod-set of the called procedure. This introduces a complication: the mod-sets of separate procedure calls may differ from the one of the top procedure, hence multiple applications of Table 4 naturally require a separate function symbol en^{mod} for every such invocation. Consider for example the situation of `union` making two invocations to `find`. In Fig. 14 one can see that the mod of `union` is $[x, r_x]_f \cup [y, r_y]_f$, while the mod of the first call `t := find(x)` is $[x, r_x]_f$, which may be a proper subset of the former. The mod of the second invocation is $[y, r_y]_f$, which may overlap with $[x, r_x]_f$.

To meet the requirement of AE^{AR} concerning the function symbols, we observe that: (a) the amount of sharing any particular function call creates, as well as the entire call, is bounded, and we can pre-determine a bound for it; (b) the modification set of any sub-call must be a subset of the top call, as otherwise it violates the obligation not to change any edge outside mod. These two properties allow us to express the functions en^{mod} of the sub-calls using en^{mod} of the top procedure and extra intermediate functions with bounded image. Thus, we replace all of the function symbols en^S introduced by $\mathrm{adapt}[S]$ for different S's, with a single (global, idempotent) function symbol together with a set of bounded function symbols.

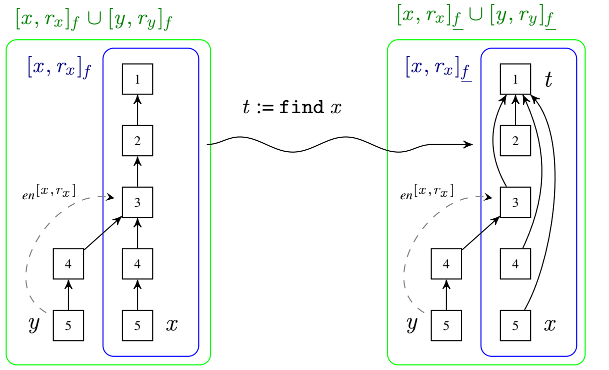
$[x, r_x]_f \cup [y, r_y]_f$
$[x, r_x]_f$
1
2
3
4
4
5
5
$en^{[x,r_x]}$
y
x
$t :=$ `find` x
$[x, r_x]_{\underline{f}} \cup [y, r_y]_{\underline{f}}$
$[x, r_x]_{\underline{f}}$
1
t
2
3
4
4
5
5
$en^{[x,r_x]}$
y
x

Figure 14. An example invocation of `find` inside `union`.

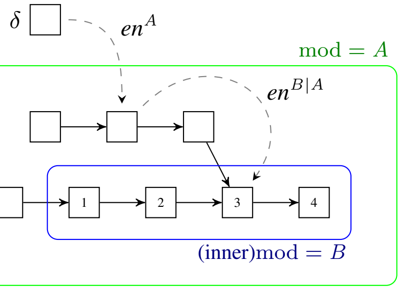

Figure 15. The inner en^{mod} is constructed from the outer one by composing with an auxiliary function $en^{B|A}$.

Consider a statement $r := proc(\overline{a})$ in a procedure *prog*. Let A denote the mod-set of *prog*, and B — the mod-set of *proc*. We show how en^B can be expressed using en^A and one more function, where the latter has a bounded range. We define $en^{B|A} : A \setminus B \rightarrow B$ a new function that is the restriction of en^B to the domain $A \setminus B$. $en^{B|A}$ is defined as follows:

$$en^B(\sigma) \overset{\text{def}}{=} \begin{cases} en^A(\sigma) & en^A(\sigma) \in B \\ en^{B|A}(en^A(\sigma)) & \text{otherwise} \end{cases} \tag{8}$$

Using equality the nesting of function symbols can be reduced (without affecting the quantifier alternation).

Consult Fig. 15; notice that $en^{B|A}(\sigma)$ is always either:

- The beginning s_i of one of the intervals $[s_i, t_i]_f$ of B (such as $\boxed{1}$ in the figure);
- A node that is **shared** by backbone pointers from two nodes in A (such as $\boxed{3}$);
- The value null.

A bound on the number of s_i's is given in the modular specification of *proc*. A bound on the number of shared nodes is given in the next subsection. This bound is effective for all the procedure calls in *prog*; hence $en^{B|A}$ can be used in the restricted logic AE^{AR}.

5.4 Bounding the Amount of Sharing

We show that under the restrictions of the specification given in Section 2 and Section 4, the number of shared nodes inside mod — that is, nodes in mod that are pointed to by more than one f-pointer of other nodes in mod — has a fixed bound throughout the procedure's execution.

Consider a typical loop-free program *prog* containing calls of the form $v_i := proc_i(\overline{a}_i)$. Assume that the mod-set of *prog* is a union of k intervals. We show how to compute a bound on the number of shared nodes inside the mod-set. Since there are k start points, at most $\binom{k}{2}$ elements can be shared when *prog* starts executing. Each sub-procedure invoked from *prog* may introduce, by our restriction, at most as many shared nodes as there are local variables in the sub-procedure. Therefore, by computing the sum over all invocation statements in *prog*, plus $\binom{k}{2}$, we get a fixed bound on the number of shared nodes inside the mod-set.

$$N_{\text{shared}} = k + \binom{k}{2} + \sum_{proc_i} |Pvar_{proc_i}|$$

$Pvar^{proc_i}$ signifies the set of local variables in the procedure $proc_i$. Notice that if the same procedure is invoked twice, it has to be included twice in the sum as well.

5.5 Verification Condition for the Entire Procedure

Since every procedure on its own is loop-free, the verification condition is straightforward:

$$vc[prog] = P_{prog} \rightarrow wp[\![prog]\!]\big(Q_{prog} \wedge \text{“}shared \subseteq Pvar\text{”}\big)$$

where “*shared* ⊆ *Pvar*” is a shorthand for the (AE^R) formula:

$$\begin{array}{c}\forall \alpha, \beta, \gamma \in \mathrm{mod} : E_f(\alpha, \gamma) \wedge E_f(\beta, \gamma) \rightarrow \\ \alpha = \beta \vee \bigvee_{v \in Pvar} \gamma = v\end{array} \qquad (9)$$

See eq (6) for the definition of E_f. It expresses the obligation mentioned in Section 1.3 that all the shared nodes in mod should be pointed to by local variables, effectively limiting newly introduced sharing to a bounded number of memory locations.

Now $vc[prog]$ is expressed in AE^{AR}, and it is valid if-and-only-if the program meets its specification. Its validity can be checked using effectively-propositional logic according to Section 3.

6. Extensions

6.1 Explicit Memory Management

This section sketches how to handle explicit allocation and reclamation of memory and exemplifies it on simple procedures shown in Fig. 16 and Fig. 17. The procedures `push` and `pushMany` extend a list at the beginning by an unbounded of fresh elements allocated using `new`. The procedure `deleteAll` takes as argument a list that has no foreign pointers into it, and explicitly frees all elements. We verify that the procedures do not introduce dangling pointers.

Table 5 updates the specification of atomic commands (provided in Table 2) to handle explicit memory management operations. For simplicity, we follow Ansii C semantics but do not handle arrays and pointer arithmetic. The allocation statement assigns a freed location denoted by s to retval and sets its value to be non-freed. All accesses to memory locations pointed-to by y in statements retval `= y.f`, `y.f = x`, and `x = y` are required to access non-freed memory. Finally, `free(y)` sets the free predicate to true for the node pointed-to by y. As a result, all the nodes reachable from y cannot be accessed.

The adaptation rule needs to be augmented in order to accommodate the change. Since nodes that are about to be allocated do not have names, the mod shall refer only to allocated nodes; free nodes can always be changed and they need not be specified in the mod. Of course, the procedure's post-condition should describe the new structure of the allocated area in terms of reachability, if modular reasoning is desired.

The change would be as follows: whenever there is a reference to some $\sigma \notin \mathrm{mod}$ (in eq (3), eq (4), and eq (5)), we would now consider only $\sigma \notin (\mathrm{mod} \cup \underline{free})$. This way the adaptation rule makes no claims regarding the free nodes. Everything else remains just the same.

@ requires $\neg free(h)$
@ mod $[h, h]_f$
@ ensures $\forall \alpha : free(\alpha) \leftrightarrow \underline{free}(\alpha) \wedge \alpha \neq \text{retval}$
$\text{retval}\langle f^* \rangle h \wedge \neg h\langle f^* \rangle \text{retval}$

```
Node push(Node h) {
  Node e = new Node();
  e.f = h;
  return e;
}
```

@ requires $\neg free(h)$
@ mod $[h, h]_f$
$\forall \alpha : free(\alpha) \rightarrow \underline{free}(\alpha)$
@ ensures $\forall \alpha : \underline{free}(\alpha) \wedge \neg free(\alpha) \rightarrow$
$h\langle f^* \rangle \alpha \wedge \alpha\langle f^+ \rangle \underline{h}$

```
Node pushMany(Node h) {
  if (?) {
    h = push(h);
    h = pushMany(h);
  }
  return h;
}
```

Figure 16. The procedure `push` allocates a new element and inserts it to the beginning of the list. The procedure `pushMany` calls `push`on the same list an arbitrary number of times.

@ requires $\forall \alpha, \beta : h\langle f^* \rangle \alpha \wedge \beta\langle f^* \rangle \alpha \rightarrow h\langle f^* \rangle \beta$ */* dominance */*
$\forall \alpha : \neg \alpha\langle f^+ \rangle h$
@ mod $[h, \text{null})_f$
@ ensures $\forall \alpha, \beta \in \mathrm{mod} : \neg \alpha\langle f^+ \rangle \beta$
$\forall \alpha : free(\alpha) \leftrightarrow \underline{free}(\alpha) \vee h\langle \underline{f}^* \rangle \alpha$

```
Node deleteAll(Node h) {
  if (h != null) {
    j = h.f;
    h.f = null;
    free(h);
    deleteAll(j);
  }
}
```

Figure 17. The procedure `deleteAll` explicitly reclaims the elements of a list dominated by its head, where no other pointers to this list exist.

6.2 Cyclic Linked-Lists

For data structures with a single pointer, the acyclicity restriction may be lifted by using an alternative formulation that keeps and maintains more auxiliary information [7, 13]. This can be easily done in AF^R, see [11].

6.3 Doubly-linked lists

To verify a program that manipulates a doubly-linked list, we need to support two fields b and f. AF^R supports this as long as the only atomic formulas used in assertions are $\alpha\langle f^* \rangle \beta$ and $\alpha\langle b^* \rangle \beta$ (and not, for example, $\alpha\langle (b|f)^* \rangle \beta$). In particular, we can specify the doubly-linked list property:

$$\forall \alpha, \beta : h\langle f^* \rangle \alpha \wedge h\langle f^* \rangle \beta \rightarrow (\alpha\langle f^* \rangle \beta \leftrightarrow \beta\langle b^* \rangle \alpha).$$

Command	Pre	Mod	Post
retval = `new()`	$free(s)$	$\emptyset$	retval $= s \land \neg free(s)$
`access y`	$\neg free(y)$	$\emptyset$	
`free(y)`	$y \neq \text{null} \land \neg free(y)$	$\emptyset$	$free(y)$

Table 5. The specifications of atomic commands for resource allocations in a C like language.

Unfortunately modularity presents another challenge: how the modset should be specified, and how to formulate the adaptation rule. Since there are two pointer fields (forward and backward), the adaptation rule (eq (1)) has to be instantiated twice. However that would require mod to be defined as a union of intervals also according to b in addition to its being defined as such using f; otherwise our logical arguments from Section 2.3 no longer hold.

When the input is a valid doubly-linked list this can always be done, since $[\alpha, \beta]_f = [\beta, \alpha]_b$. In cases such where the input is somewhat altered or corrupt (for example [23]), the logic will have to be modified to incorporate the volatile exit points of back-pointers potentially pointing to arbitrary nodes. This extension is out of the current scope.

7. Experimental Results

7.1 Implementation Details

A VC generator described in Section 5 is implemented in Python, and PLY (Python Lex-Yacc) is employed at the front-end to parse modular recursive procedure specifications as defined in Section 4. The tool checks that the pre and the post-conditions are specified in AF^R and that the modset is defined. SMT-LIB v2 [3] standard notation is used to format the VC and to invoke Z3. The validity of the VC can be checked by providing its negation to Z3. If Z3 exhibits a satisfying assignment then that serves as counterexample for the correctness of the assertions. If no satisfying assignment exists, then the generated VC is valid, and therefore the program satisfies the assertions.

The output model/counterexample (S-Expression), if one is generated, is then also parsed and f^* is evaluated on all pairs of nodes. This structure represents the state of the program either at the input or at the beginning of a loop iteration: running the program from this point will violate one or more invariants. To provide feedback to the user, f is recovered by computing eq (6)), and then the `pygraphviz` tool is used to visualize and present to the user a directed graph, whose vertices are nodes in the heap, and whose edges are the f pointer fields.

7.2 Verification Examples

We have written modular specifications for the example procedures shown in Table 7. We are encouraged by the fact that it was not difficult to express assertions in AF^R for these procedures. The annotated examples and the VC generation tool are publicly available with the submission. We only picked examples with interesting cutpoints to show the benefits of our approach in contrast to [11].

To give some account of the programs' sizes, we observe the program summary specification given as pre- and postcondition, count the number of atomic formulas in each of them, and note the depth of quantifier nesting; all our samples had only universal quantifiers in the specification. We did the same for the generated VC; naturally, the the VC is much larger than the specification even for small programs. Still, the time required by Z3 to prove that the VC is valid is short.

Thanks to the fact that FOL-based tools, and in particular SAT solvers, permit multiple relation symbols we were able to express ordering properties in sorted lists, and thus in the sorting routines implementing Quicksort and insertion-sort.

UF: find, UF: union	— Implementation of a Union-Find dynamic data structure.
SLL: filter	— Takes a linked list and deletes all elements not satisfying some predicate C.
SLL: quicksort	— Sorts a linked-list in-place using the Quicksort algorithm.
SLL: insert-sort	— Creates a new, sorted linked-list from a given list by repeatedly running `insert` on the elements of the input list.

Table 6. Description of some pointer manipulating programs verified by our tool.

Benchmark	Formula size					Solving time
	P,Q		mod	VC		
	#	∀	#	#	∀	(Z3)
SLL: filter	7	2	1	217	6	0.48s
SLL: quicksort	25	2	1	745	9	1.06s
SLL: insert-sort	21	2	1	284	11	0.37s
UF: find	13	2	1	203	6	0.40s
UF: union	20	2	2	188	6	1.39s

Table 7. Implementation Benchmarks; P,Q — program's specification given as pre- and post-condition, mod— mod-set, VC — verification condition, # — number of atomic formulas/intervals, ∀ — quantifier nesting The tests were conducted on a 1.7GHz Intel Core i5 machine with 4GB of RAM, running OS X 10.7.5. The version of Z3 used was 4.2, complied for 64-bit Intel architecture (using `gcc` 4.2, LLVM). The solving time reported is wall clock time of the execution of Z3.

Checked properties. For Table 6, apart from find and union, we also checked full functional correctness of the other examples (filter, quicksort, insertion sort). For filter, we checked that elements remain in the same order and that only the elements satisfying the filtering predicate were removed. For the sorting routines, we checked that the resulting list contains the same elements and is indeed sorted (via an order relation).

7.3 Buggy Examples

We also applied the tool to erroneous programs and programs with incorrect assertions. The results, including run-time statistics and formula sizes, are reported in Table 8. The table lists four kinds of deliberately-introduced bugs that were provided as input to the tool. Formula sizes are measured in the same way as in Table 7.

In addition, for every detected bug, our tool generates a concrete counterexample depicting a state of the heap violating some assertion. We measured the size of the model generated, by observing the size of the generated domain — which reflects the number of nodes in the heap. As expected, Z3 was able to produce concrete counterexample of reasonable size, producing output which is readable for the programmer and useful for debugging. In fact, our tool converts Z3 models into directed graph diagrams which facili-

	Formula size				Solving	C.e.
Benchmark	P,Q		VC		time	size
(+ Nature of defect)	#	∀	#	∀	(Z3)	($\lvert L\rvert$)
UF: find Incorrect handling of corner case	27	3	201	6	1.60s	2
UF: union Incorrect specification	19	2	186	6	0.70s	8
SLL: filter Uncontrolled sharing	36	4	317	6	0.49s	14
SLL: insert-sort Unmet call precondition	21	2	283	9	0.88s	8

Table 8. Information about benchmarks that demonstrate detection of several kinds of bugs in pointer programs. In addition to the previous measurements, the last column lists the size of the generated counterexample in terms of the number of vertices — linked-list or tree nodes.

tate debugging our assertions. Since the counterexamples are slight variations of the correct programs, size and running time statistics are similar.

8. Related Work

8.1 Modular Verification

The area of modular procedure specification is heavily studied. Many of these works require that the user declare potential changes similar to the modset (e.g., see [2, 14, 25, 29]). The frame rule of separation logic [9] naturally supports modular reasoning where the separating conjunction combines the local postcondition with the assertion at the call site. Unlike separation, reachability is a higher abstraction which relies on type correctness and naturally abstracts operations such as garbage collection. Nevertheless, in Section 6.1 we show that it can also deal with explicit memory reclamations.

We believe that our work in this paper pioneers the usage of an effectively propositional logic which is a weak logic to perform modular reasoning in a sound and complete way. Our adaptation rule is more complex than the frame rule as it automatically updates reachability. The idea of using the idempotent entry point function to enable local reasoning about list-manipulating programs (including an EPR reduction) has been explored independently by Piskac et al. [19], where it was used to automate the frame rule in separation logic. In this paper we identify a general EPR fragment of assertions for which this idea of the idempotent entry point function is sound and complete.

8.1.1 Cutpoints

Rinetzky et al. [21] introduce cutpoint objects which are objects that can reach the area of the heap accessible by the procedure without passing through objects directly pointed-to by parameters. Cutpoints complicate program reasoning. They are used in model checking [1] and static analysis [6, 22]. Examples such as the ones in [23] which include (unbounded) cutpoints from the stack are handled by our method without any changes. These extra cutpoints cannot change the reachabilty and thus have no effect. Interestingly, we can also handle certain programs which manipulate unbounded cutpoints. Instead, we do limit the amount of new sharing in paths which are necessary for the verification. For example, the find procedure shown in Fig. 2 includes unbounded sharing which can be created by the client program. A typical client such as a spanning tree construction algorithm will indeed create unbounded sharing. In the future, we plan to verify such clients by abstracting away the pointers inside the union-find tree.

8.2 Decision Procedures

Many decision procedures for reasoning about linked lists have been proposed [4, 15, 17, 28]. All these logics are based on monadic second-order logic on trees which has a non-elementary time (and space) asymptotic complexity. We follow [11] in using a weak logic which permits sound and complete reasoning using off the shelf SAT solvers which are efficient in practice and can be implemented in polynomial space. Indeed our preliminary experimental results reported Section 7 show that Z3 is fast enough and may be even useful for automatically generating abstract interpreters as suggested by [20].

Interestingly, the adaptation rule drastically simplifies the Weakest-Precondition rules given in [11]. Notice the specifications in Table 2 do not use quantifiers at all, whereas in [11] the formulas contain quantifiers with alternations. Indeed the appropriate quantifiers are added in a generic manner by the adaptation rule and weakest-precondition.

8.3 Incremental Reachability Update

Formulas for updating transitive closure w.r.t., graph mutations have been developed by various authors (e.g., [5, 7, 8, 13]). These works assume that a single edge is added and deleted. This submission generalizes these results to procedures which perform unbounded mutations. Indeed our adaptation rule generalizes [7, 11, 13]) which provides reachability update formula w.r.t. the removal of a single edge.

9. Conclusion

A crucial method for simplifying the reasoning about linked data structures is partitioning them into basic blocks, where each basic block has only one entry point and one exit point. This paper slightly generalizes by reasoning about blocks with a potentially unbounded number of entry points, as demonstrated by `find` and `union`. Notice that this unboundedness supports modularity: even in the case where in every particular call context there is a bounded number of paths (e.g. when there is a bounded number of roots in the heap), the bound is not known in advance, therefore the programmer has to prepare for an unbounded number of cases.

It is important to note that the adaptation rule adds expressive power to verifying programs: it is in general impossible for the programmer to define, in AF^R, a modular specification for all the procedures. Generation of a verification condition requires coordination between the separate call sites as mentioned above, in particular taking note of potential sharing. This coordination requires per-call-site instantiation, which, thanks to having the adaptation rule in the framework, is done automatically.

Finally we remark that there is a trade-off between mod and the post-condition: defining a simpler, but larger mod may cause the post-condition to become more complicated, sometimes not even AF^R-expressible. Also notice that if $\mathrm{mod} = V$ (the entire heap), modular reasoning becomes trivial since it can be done by relational composition, but this puts the burden of writing the most complete post-conditions on the programmer, which sometimes is not even possible in a limited logic.

Therefore, we believe that this paper takes a step towards modular reasoning about reachability in programs that manipulate linked lists. Lifting such reasoning to more complex data structures such as trees and graphs remains future work.

Acknowledgement. Thanks to the anonymous referees for their comments. Itzhaky, Lahav and Sagiv were funded by the European Research Council under the European Union's Seventh Framework Program (FP7/2007-2013) / ERC grant agreement no. [321174-VSSC] and by a grant from the Israel Science Foundation (652/11). Banerjee and Nanevski were partially supported by by Spanish MINECO projects TIN2009-14599-C03-02 Desafios, TIN2010-20639 Paran10, TIN2012-39391-C04-01 Strongsoft, EU NoE Project 256980 Nessos, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743. Immerman was partially supported by NSF grant CCF 1115448.

References

[1] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. Logical Methods in Computer Science, 7(4), 2011.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the spec# experience. Commun. ACM, 54(6):81–91, 2011.

[3] C. Barrett, A. Stump, , and C. Tinelli. SMTLIB: Satisfiability Modulo Theories Library, 2013. http://smtlib.cs.uiowa.edu/docs.html.

[4] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In ATVA, pages 167–182, 2012.

[5] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. Inf. & Comput., 120:101–106, 1995.

[6] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In SAS, pages 240–260, 2006.

[7] W. Hesse. Dynamic Computational Complexity. PhD thesis, UMass in Computer Science, June 2003.

[8] N. Immerman. Descriptive complexity. Graduate texts in computer science. Springer, 1999.

[9] S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In POPL, pages 14–26, 2001.

[10] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv. Modular reasoning about heap paths via effectively propositional formulas. Technical report, Tel Aviv University, 2013. `http://www.cs.tau.ac.il/~shachar/dl/tr-2013b.pdf`.

[11] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In CAV, pages 756–772, 2013.

[12] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. Technical report, Tel Aviv University, 2013. `http://www.cs.tau.ac.il/~shachar/dl/tr-2013.pdf`.

[13] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In POPL, pages 171–182, 2008.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes, 31(3):1–38, 2006.

[15] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In POPL, pages 611–622. ACM, 2011.

[16] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In ECOOP, pages 77–97, 2009.

[17] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In PLDI, pages 221–231. ACM, 2001.

[18] R. Piskac, L. M. de Moura, and N. Bjørner. Deciding effectively propositional logic using dpll and substitution sets. J. Autom. Reasoning, 44(4):401–424, 2010.

[19] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In CAV, pages 773–789, 2013.

[20] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In VMCAI, pages 252–266, 2004.

[21] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In POPL, pages 296–309, 2005.

[22] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In SAS, pages 284–302, 2005.

[23] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In POPL, pages 173–186, 2011.

[24] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. Journal of the ACM, 22(2):215–225, 1975.

[25] J. Wing. The CMU Larch Project. 1995. URL `http://www.cs.cmu.edu/afs/cs/project/larch/www/home.html`.

[26] G. Winskel. The Formal Semantics of Programming Languages: An Introduction. Foundations of Computing Series. Zone Books, U.S., 1993. ISBN 9780262731034.

[27] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In CAV, pages 385–398, 2008.

[28] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program., 73(1–2):111–142, 2007.

[29] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In PLDI, pages 349–361, 2008.