RGSep Action Inference

Viktor Vafeiadis

Microsoft Research Cambridge, UK

Abstract. We present an automatic verification procedure based on RGSep that is suitable for reasoning about fine-grained concurrent heap-manipulating programs. The procedure computes a set of RGSep actions overapproximating the interference that each thread causes to its concurrent environment. These inferred actions allow us to verify safety, liveness, and functional correctness properties of a collection of practical concurrent algorithms from the literature.

1 Introduction

Low level C programmers constantly rely on two very error-prone programming features: manual memory management (`malloc`/`free`) and concurrency. While there are several verification techniques for reasoning about either feature in isolation, few techniques can handle programs using both features.

One such technique is RGSep [20], a recent extension of rely-guarantee reasoning [11] that incorporates separation logic [15]. RGSep specifications describe the updates to the shared state using two binary relations: the rely and the guarantee. A thread's rely relation under-approximates the interference it can tolerate from its environment (that is, the updates that other threads are allowed to do), whereas the guarantee over-approximates the updates the thread can do, i.e. the interference that it causes to its concurrent environment. RGSep represents these binary relations as the reflexive and transitive closure of a set of *actions*, which are precondition-postcondition pairs describing the possible small updates.

On its own RGSep is just a program logic: users must prove their programs correct with pencil and paper using RGSep's proof rules. As constructing such proofs manually is quite tedious and often error-prone, there has been some work on constructing such proofs semi-automatically [5, 19] by letting the programmer supply the rely and guarantee relations and doing abstract interpretation to figure out the more tedious aspects of the proof.

Here, we extend the aforementioned work to be fully automatic. We present an algorithm (Infer-Actions, §3) that calculates the rely and guarantee relations as a set of actions, each of which is extended with a special context assertion describing the part of the state that is not affected by the action. These contexts arise naturally during action inference and allow us to define a useful join on actions (§4).

In the process of inferring these actions, our algorithm also proves memory safety, discovers shape invariants, and discharges any user-supplied assertions. The output of action inference has been used to prove advanced safety properties, such as linearizability, and conditional termination [9].

2 Preliminaries

We consider programs in a first-order subset of C. Programs consist of an initialization phase followed by a top-level parallel composition of a possibly unbounded number of threads. The programs for the initialization phase and for each thread are converted to the following simpler language of commands:

$$\begin{aligned} C ::= {} & \texttt{skip} \mid x := E \mid x := [E] \mid [E] := E' \mid x := \texttt{malloc}() \\ & \mid \texttt{assume}(E) \mid C_1; C_2 \mid C_1 \oplus C_2 \mid C^* \mid \texttt{atomic}\ C \end{aligned}$$

where x ranges over program variables and E over arithmetic expressions. Program commands, C, include the empty command, variable assignments, memory loads and stores, memory allocation, assume statements, sequential composition, non-deterministic choice, loops, and atomic commands.

An important aspect of our intermediate language is that the atomicity of memory accesses is explicit. By default, we assume that memory accesses are non-atomic. When, however, a memory access is guaranteed to be atomic by the memory model (for example, a single-word memory access to a volatile variable or field), we make this explicit by enclosing it in an atomic block. Similarly, we also use atomic blocks to encode complex atomic instructions such as compare-and-swap. As data races on non-atomic memory accesses can lead to incoherent results, our proof system ensures that there are no races on non-atomic memory accesses, but permits races between two atomic commands.

2.1 Underlying Separation Logic Domain

Our verification is parametric with respect to an underlying separation logic abstract domain. Elements of a separation logic domain are assertions belonging to a fragment of separation logic and are ordered by logical implication. Further, we assume that this fragment of separation logic includes $\mapsto$-assertions, disjunction, $*$-conjunction and that assertions can have free logical variables. We shall use uppercase italic letters (P, Q, R) to range over such separation logic assertions. Their meaning with respect to an interpretation ($\mathcal{I}$) mapping logical variables to values is a set of heaps (partial finite maps from addresses to values):

$$\begin{aligned} [\![\mathsf{emp}]\!]_{\mathcal{I}} &\stackrel{\text{def}}{=} \{h \mid \mathbf{dom}\ h = \emptyset\} \\ [\![E \mapsto E']\!]_{\mathcal{I}} &\stackrel{\text{def}}{=} \{h \mid \mathbf{dom}\ h = \{[\![E]\!]_{\mathcal{I}}\} \wedge h([\![E]\!]_{\mathcal{I}}) = [\![E']\!]_{\mathcal{I}}\} \\ [\![P * Q]\!]_{\mathcal{I}} &\stackrel{\text{def}}{=} \{h_1 \uplus h_2 \mid h_1 \in [\![P]\!]_{\mathcal{I}} \wedge h_2 \in [\![Q]\!]_{\mathcal{I}}\} \end{aligned}$$

where $h_1 \uplus h_2$ denotes the union of the functions h_1 and h_2 if their domains are disjoint, and is undefined if their domains overlap. Finally, the abstract domain must support the following three operations:

Abstraction: $\textsc{Abstract}(P)$ over-approximates P ($[\![P]\!]_{\mathcal{I}} \subseteq [\![\textsc{Abstract}(P)]\!]_{\mathcal{I}}$) ensuring that fixpoint calculations of the form $P \leftarrow P \vee \alpha(\mathit{transform}(P))$ terminate. This is usually achieved by $\textsc{Abstract}$ having a finite range.

(Must-)Subtraction: $\textsc{Subtract}(P, Q, A)$ is an enhanced entailment checking procedure and is also known as 'frame inference' [2]. It takes two assertions (P, Q) and a set of logical variables (A) that are implicitly existentially quantified in Q. Subtraction tries to find an assertion F such that $P \implies \exists A.\ Q * F$. If such an assertion exists, it returns it; otherwise, it throws an exception (usually resulting in a verification failure). Note that the frame F may provide witnesses for the existentially quantified variables A.

May-Subtraction: $\textsc{May-Subtract}(P, Q, R)$ takes three assertions P, Q, and R and returns an assertion S denoting the left-over state if Q and R are removed from P and the R-part is added back. Formally, for all $\mathcal{I}$, h_1, and h_2, if $(h_1 \uplus h_2) \in [\![P]\!]_\mathcal{I}$ and $h_1 \in [\![Q]\!]_\mathcal{I}$ and $h_2 \in [\![R * \mathsf{true}]\!]_\mathcal{I}$, then $h_2 \in [\![S]\!]_\mathcal{I}$. May-subtraction is an overapproximation of the separation logic formula $(Q \mathbin{-\!\circledast} P) \wedge (R * \mathsf{true})$, where $\mathbin{-\!\circledast}$ is the 'septraction' operator [20, 5].

The difference between $\textsc{Subtract}$ and $\textsc{May-Subtract}$ is rather important. $\textsc{Subtract}(P, Q, \emptyset)$ proves that Q can be removed from P and returns the remaining part of the state. In contrast, $\textsc{May-Subtract}(P, Q, \mathsf{emp})$ considers all the ways that Q might be removed from P and returns the remaining parts of the state. Consider the following example:

Example 1. Let $P \equiv x \mapsto 1 * y \mapsto 2$, $Q \equiv a \mapsto b$. Calling $\textsc{Subtract}(P, Q, \emptyset)$ would throw an exception because P does not imply that a is allocated. In contrast, $\textsc{May-Subtract}(P, Q, \mathsf{emp})$ would return $(a = x \wedge b = 1 \wedge y \mapsto 2) \vee (a = y \wedge b = 2 \wedge x \mapsto 1)$. Similarly, $\textsc{May-Subtract}(P, \mathsf{emp}, Q)$ would return $(a = x \wedge b = 1 \wedge x \mapsto 1 * y \mapsto 2) \vee (a = y \wedge b = 2 \wedge x \mapsto 1 * y \mapsto 2)$.

During action inference, we shall use $\textsc{Subtract}$ to calculate the effect of the atomic commands of the current thread, and $\textsc{May-Subtract}$ to calculate the effect of interference (i.e., of the commands of the other threads).

Our implementation uses the abstract domains from Distefano et al. [6] and Vafeiadis [19] as underlying separation logic domains. For $\textsc{Subtract}$, we used the entailment algorithm of Berdine et al. [2], and for $\textsc{May-Subtract}$ an improvement over the septraction elimination algorithm of Calcagno et al. [5], which is reported in Appendix A.

2.2 RGSep

RGSep [20] is the program logic on top of which our verification is based. RGSep logically partitions the state of the program into a number of (disjoint) components, which are called regions. Each thread owns one region for its local data, and there is also one region containing data that is shared among threads. RGSep assertions describe only the shared region and the current thread's region and are given by the following grammar:

$$p, q ::= P_\mathrm{L} * \boxed{P_\mathrm{S}} \mid p \vee q \mid \exists x.\ p$$

The first assertion form says that the thread's local state satisfies P_L and that the shared state is disjoint and satisfies P_S. Formally,

$$[\![P_\mathrm{L} * \boxed{P_\mathrm{S}}]\!]_\mathcal{I} \stackrel{\mathrm{def}}{=} \{(h_\mathrm{L}, h_\mathrm{S}) \mid h_\mathrm{L} \in [\![P_\mathrm{L}]\!]_\mathcal{I} \wedge h_\mathrm{S} \in [\![P_\mathrm{S}]\!]_\mathcal{I} \wedge \mathrm{defined}(h_\mathrm{L} \uplus h_\mathrm{S})\}$$

Note that the separation logic formulas P_L and P_S can have common variables. Such common variables keep track of the correlation between each thread's local state and the shared state. In contrast, there is no way of expressing correlations between the local states of two threads.

The concurrent behaviour of a thread is abstracted by a set of precondition-postcondition pairs, $P \rightsquigarrow Q$, known as actions. Actions summarise what modifications the atomic statements of a thread can perform on the shared state. Their semantics is formally defined as follows:

$$\mathcal{A}[\![P \rightsquigarrow Q]\!] \stackrel{\text{def}}{=} \{(s \uplus s_0, s' \uplus s_0) \mid \exists \mathcal{I}.\ \ s \in [\![P]\!]_\mathcal{I} \wedge s' \in [\![Q]\!]_\mathcal{I}\}$$

The action's precondition and postcondition describe only the part of the state that changes; the remaining part (s_0) is assumed not to change, and is not further constrained. The assertions P and Q can have some free logical variables (in the domain of $\mathcal{I}$): these are implicitly existentially quantified and their scope extends over both P and Q.

In this paper, we extend the notion of actions with a context assertion, R, restricting when the action can execute. Contexts are very useful during action inference and, in particular, for defining a good join operation (see §4). Formally, their meaning is:

$$\begin{aligned}\mathcal{A}[\![R \mid P \rightsquigarrow Q]\!] &\stackrel{\text{def}}{=} \mathcal{A}[\![P \rightsquigarrow Q]\!] \cap \mathcal{A}[\![P * R \rightsquigarrow Q * R]\!] \\ &= \{(s \uplus s_0, s' \uplus s_0) \mid \exists \mathcal{I}.\ \ s \in [\![P]\!]_\mathcal{I} \wedge s' \in [\![Q]\!]_\mathcal{I} \wedge s_0 \in [\![R * \mathsf{true}]\!]_\mathcal{I}\}\end{aligned}$$

The meaning of a set of actions is the reflexive and transitive closure of the union of the meanings of the individual actions:

$$[\![\{a_1, \ldots, a_n\}]\!] \stackrel{\text{def}}{=} (\mathcal{A}[\![a_1]\!] \cup \ldots \cup \mathcal{A}[\![a_n]\!])^*$$

Reflexive and transitive closure models any arbitrary interleaving of any number of repetitions of the actions a_1 to a_n.

RGSep judgments are of the form $\mathit{Rely}, \mathit{Guar} \vdash_{\mathrm{RGSep}} \{p\}\ C\ \{q\}$, where *Rely* and *Guar* are sets of actions and p and q are RGSep assertions. Informally, this specification says that if the initial state satisfies p and all environment transitions are included in *Rely*, then (a) C does not fault, (b) all of C's transitions are included in *Guar*, and (c) if C terminates, then the final state satisfies q. RGSep provides a collection of proof rules for deriving such judgments, which we omit for brevity. These can be found in [20, 18].

Stabilization. An important requirement of the RGSep proof rules is that certain assertions appearing in the proof of a thread are stable under the rely condition. Stability is formally defined as follows:

Definition 1 (Stability). *An assertion P about the shared state is* stable *under the binary relation R, if and only if interference with R cannot falsify P: i.e. for all $\mathcal{I}$, s, s', if $s \in [\![P]\!]_\mathcal{I}$ and $(s, s') \in R$, then $s' \in [\![P]\!]_\mathcal{I}$.*

Algorithm 1 $\textsc{Stabilize}(S, \mathit{Rely})$

```
1: repeat
2:     $S_{\text{old}} \leftarrow S$
3:     for all $(R \mid P \rightsquigarrow Q) \in \mathit{Rely}$ do
4:         $S \leftarrow S \vee \textsc{Abstract}(\textsc{May-Subtract}(S, P, R) * Q)$
5: until $S = S_{\text{old}}$
6: return $S$
```

Given a rely condition *Rely* and a possibly unstable assertion S, Alg. 1 computes a weaker assertion S' that is stable under *Rely*. It does so by taking into account interference with each action in *Rely* until a fixpoint is reached. To ensure that the fixpoint calculation converges, we apply abstraction at each loop iteration.

Theorem 1 (Stabilization Soundness). *If* $\textsc{Stabilize}(S, \mathit{Rely}) = S'$*, then for all* $\mathcal{I}$*,* $[\![S]\!]_{\mathcal{I}} \subseteq [\![S']\!]_{\mathcal{I}}$ *and* S' *is stable under Rely.*

The proof of this theorem follows directly from the definitions of stability, RGSep actions, and the specification of May-Subtract.

Note that the execution time of $\textsc{Stabilize}(S, \mathit{Rely})$ is linear in the number of actions in *Rely*. Since stabilization is the most time-consuming component of action inference, it is important that action inference infers small sets of actions. We shall return to this point in Sect. 4.

3 Action Inference Algorithm

A library consists of an initialization method, *init*, and a number of access methods, *Ms*, which can be executed concurrently after the initialization method has finished. The most general concurrent client of a library is defined as follows:

Definition 2 (Most General Client). *The* most general client *of a library executes its initialization method followed by an unbounded number of threads, each executing any number of the access methods in any order:*

$$\textsc{mgc}(\mathit{init}, \{C_1, \ldots, C_n\}) \stackrel{\text{def}}{=} \mathit{init}; \|(C_1 \oplus \ldots \oplus C_n)^*$$

The most general client over-approximates all legal clients of the library in that concrete clients will use the module in a more constrained way than the most general client.

Some libraries require that their methods are called in a more constrained fashion than the most general client above. For example, a lock library typically assumes that threads do not attempt to acquire any locks that they already hold nor to release any locks that they do not hold. These requirements can be formalized with a simple state machine per thread describing which methods the thread is allowed to call at each time. To verify the lock library, one can encode the state machine in the body of the `acquire` and `release` methods using an auxiliary thread-local variable.

Algorithm 2 INFER-ACTIONS($init, Ms$)

```
1: $G \leftarrow \emptyset$
2: $(-, Inv) \leftarrow$ SYMB-EXEC($\mathsf{emp}, \emptyset, init$)
3: repeat
4:     $G_{\mathrm{old}} \leftarrow G$
5:     $Inv \leftarrow$ STABILIZE($Inv, G$)
6:     for all $C \in Ms$ do
7:         $(G_{\mathrm{new}}, -) \leftarrow$ SYMB-EXEC($\boxed{Inv}, G, C$)
8:         $G \leftarrow G \cup G_{\mathrm{new}}$
9: until $G = G_{\mathrm{old}}$
10: return $(G, Inv)$
```

Algorithm 3 Memory reads: SYMB-EXEC($\exists \boldsymbol{z}.\ P_{\mathrm{L}} * \boxed{P_{\mathrm{S}}}, Rely, \mathtt{x} := [E]$)

```
1: if SUBTRACT($P_{\mathrm{L}}, E \mapsto \alpha, \{\alpha\}$) $= R_{\mathrm{L}}$ then
2:     return $(\emptyset,\ \exists \boldsymbol{z}\, \alpha\, \beta.\ \mathtt{x} = \alpha \wedge E \mapsto \alpha * R_{\mathrm{L}}[\beta/\mathtt{x}] * \boxed{P_{\mathrm{S}}[\beta/\mathtt{x}]})$
3: else if inside an atomic block and SUBTRACT($P_{\mathrm{S}}, E \mapsto \alpha, \{\alpha\}$) $= R_{\mathrm{S}}$ then
4:     return $(\emptyset,\ \exists \boldsymbol{z}\, \alpha\, \beta.\ \mathtt{x} = \alpha \wedge P_{\mathrm{L}}[\beta/\mathtt{x}] * \boxed{E \mapsto \alpha * R_{\mathrm{S}}[\beta/\mathtt{x}]})$
5: else
6:     return ERROR
```

INFER-ACTIONS (see Alg. 2) takes a library and computes the total interference caused by its access methods (G) and its data structure invariant (Inv) by considering the library's most general client and doing a fixpoint computation. The algorithm assumes that clients of the library cannot directly access the library's internal state; thus, there is no external rely condition.

To calculate the interference produced by a command, INFER-ACTIONS calls our new symbolic execution procedure, SYMB-EXEC. This takes a precondition p, a rely R, and a command C and tries to prove memory safety returning a guarantee G and a postcondition q such that $R, G \vdash_{\mathrm{RGSep}} \{p\}\ C\ \{q\}$. If SYMB-EXEC($p, R, C$) fails to prove memory safety, then it returns ERROR.

We consider memory safety to be the most basic property that all programs should have, and thus fail verification if this property cannot be established. In addition to memory safety, however, action inference can prove much more interesting properties, such as data structure invariants, and discharge user-supplied assertions.

Symbolic execution is defined by induction on the command, C.

Memory Reads. When symbolic execution encounters a memory read (see Alg. 3), it tries to apply the memory read axiom of separation logic. If it is a non-atomic read, the memory location must be in the local state: this is to prevent race conditions. Otherwise, if the read is inside an atomic block (e.g. because the read is atomic, or it used to implement a complex atomic instruction such as CAS), the memory location can also be in the shared state. As memory reads do not change the heap, the guarantee condition is empty. If SYMB-EXEC cannot prove that the memory cell exists, it returns ERROR. This is consistent with

Algorithm 4 Memory writes: $\textsc{Symb-Exec}(\exists \boldsymbol{z}.\ P_{\mathrm{L}} * \boxed{P_{\mathrm{S}}}, \mathit{Rely}, [E] := E')$

```
1: if $\textsc{Subtract}(P_{\mathrm{L}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{L}}$ then
2:     return $(\emptyset,\ \exists \boldsymbol{z}.\ E \mapsto E' * R_{\mathrm{L}} * \boxed{P_{\mathrm{S}}})$
3: else if inside an atomic block and $\textsc{Subtract}(P_{\mathrm{S}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{S}}$ then
4:     $(P_{\mathrm{L2S}}, P'_{\mathrm{L}}) \leftarrow \textsc{Reachable-Split}(P_{\mathrm{L}}, E \mapsto E')$
5:     $\mathit{act} \leftarrow \textsc{A-Abs}(R_{\mathrm{S}} \mid E \mapsto \alpha \rightsquigarrow E \mapsto E' * P_{\mathrm{L2S}})$
6:     return $(\{\mathit{act}\},\ \exists \boldsymbol{z}.\ P'_{\mathrm{L}} * \boxed{E \mapsto E' * P_{\mathrm{L2S}} * R_{\mathrm{S}}})$
7: else
8:     return $\textsc{Error}$
```

the standard memory model, where programs fail when they access unallocated memory.

If the precondition is disjunctive, Symb-Exec does the obvious case split:

```
$\textsc{Symb-Exec}(\bigvee_i \exists \boldsymbol{z}_i.\ P_i * \boxed{Q_i}, \mathit{Rely}, \mathtt{x} := [E]) \overset{\mathrm{def}}{=}$
    for each $i$ do
        $(G_i, q_i) \leftarrow \textsc{Symb-Exec}(\exists \boldsymbol{z}_i.\ P_i * \boxed{Q_i}, \mathit{Rely}, \mathtt{x} := [E])$
    return $(\bigcup_i G_i,\ \bigvee_i q_i)$
```

Memory Writes. If the precondition is disjunctive, symbolic execution does the same case split as for memory reads above. For non-disjunctive preconditions, see Alg. 4. If the write is local, it does not affect the shared state; so $G = \emptyset$. If, however, the write is on the shared state (and hence the write is required to be within an atomic block), then its effect is an action, *act*, which might include some transfer of ownership that re-adjusts the boundary between the local and the shared states. The algorithm relies on a simple reachability heuristic to decide how to re-adjust this boundary. After the memory write, any part of the local state that is reachable from E' is accessible from the shared memory location E, and thus can be accessed by other threads. Therefore, symbolic execution splits the local assertion P_{L} into two parts: P_{L2S} that becomes shared, and P'_{L} that remains local.

As a final step, symbolic execution calls action abstraction, A-Abs, which over-approximates the inferred action. Its input an action $R \mid P \rightsquigarrow Q$ and returns a larger action $R' \mid P' \rightsquigarrow Q'$; i.e. $\mathcal{A}[\![R \mid P \rightsquigarrow Q]\!] \subseteq \mathcal{A}[\![R' \mid P' \rightsquigarrow Q']\!]$. Over-approximation is necessary in order to ensure convergence of the algorithm.

Our implementation of A-Abs consists of two steps. First, it existentially quantifies over local program variables and forgets any pure facts involving them. Second, it applies the underlying abstraction of the separation logic domain to R, P, and Q. We have also experimented with a more aggressive abstraction that removes any list segments appearing in the context. This was partly motivated by our experience: actions containing list segments are rarely needed in manual proofs. Nevertheless, they are necessary for some examples.

Other Program Constructs. Dealing with the other constructs is easy and follows directly from the corresponding RGSep proof rules (see Alg. 5).

Assignments: For simplicity, we assume that all shared variables are allocated in the heap. This is easy to achieve by a preprocessing step which allocates global variables at statically known memory addresses, converting any assignments to global variables into memory writes. Thus, the remaining assignments affect only local variables, and their guarantee condition is empty.

Allocation Commands: Allocated cells are part of the local state.

Sequencing: The guarantee condition of a sequential composition, $C_1; C_2$, is the union of the guarantee conditions of the two commands, C_1 and C_2.

Choice: Similarly, the guarantee condition of $C_1 \oplus C_2$ is the union of the guarantee conditions of the two branches, C_1 and C_2.

Loops: Calculating the loop invariant involves a standard fixpoint computation which applies widening after each iteration. The guarantee condition of the loop is the guarantee condition of the last iteration in the fixpoint computation. To ensure that the fixpoint converges, at each iteration p is abstracted by performing the abstraction of the underlying separation logic domain (Abstract) to all its P_L and P_S components.

Atomic Commands: Symbolic execution runs the body of the atomic command assuming that there is no interference ($\mathit{Rely} = \emptyset$) and then does a stabilization step to take into account interference from other threads. The guarantee condition of the atomic block is just the guarantee condition of its body.

Symbolic execution and action inference are sound in the following sense:

Theorem 2 (Symbolic Execution Soundness). *If* $\textsc{Symb-Exec}(p, \mathit{Rely}, C)$ *returns* (G, q), *then* $\mathit{Rely}, G \vdash_{\text{RGSep}} \{p\}\ C\ \{q\}$.

Theorem 3 (Action Inference Soundness). *If* $\textsc{Infer-Actions}(\mathit{init}, \mathit{Ms})$ *returns* (G, Inv), *then* $\emptyset, G \vdash_{\text{RGSep}} \{\mathsf{emp}\}\ \textsc{mgc}(\mathit{init}, \mathit{Ms})\ \{\boxed{\mathit{Inv}}\}$.

To prove these theorems, we first have to prove the following simpler lemma:

Lemma 1. *If* $\textsc{Symb-Exec}(p, \emptyset, C)$ *returns* (G, q), *then* $\emptyset, G \vdash_{\text{RGSep}} \{p\}\ C\ \{q\}$.

This follows from the RGSep proof rule for atomic blocks and the proof rules in Section 4.2 of Vafeiadis's thesis [18]. The theorems then follow easily from the RGSep proof rules [20] and from Lemma 1.

Incompleteness. There are three sources of incompleteness to consider.

First, without auxiliary variables rely-guarantee reasoning is intentionally incomplete. This incompleteness is exactly what makes rely-guarantee reasoning tractable. In practice, auxiliary variables are rarely needed for the sort of programs we have looked at. (None of the memory safety benchmarks of Sect. 5 needed auxiliary variables, except that in the algorithms using locks we modelled locks as storing the identifier of the thread holding the lock. In the linearizability benchmarks, auxiliary variables are used as part of the specification.) Symbolic execution does not attempt to infer such auxiliary variables.

Algorithm 5 SYMB-EXEC(p, Rely, C) where $p \equiv \bigvee_i \exists \boldsymbol{z}_i.\ P_i * \boxed{Q_i}$

```
 1: if $C$ is skip then
 2:     return $(\emptyset, p)$
 3: else if $C$ is assume$(E)$ then
 4:     return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, E{\neq}0 \land P_i * \boxed{Q_i})$
 5: else if $C$ is $\mathtt{x} := E$ then
 6:     return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, \exists \beta.\ \mathtt{x}{=}E[\beta/\mathtt{x}] \land P_i[\beta/\mathtt{x}] * \boxed{Q_i[\beta/\mathtt{x}]})$
 7: else if $C$ is $\mathtt{x} :=$ malloc() then
 8:     return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, \exists \alpha\, \beta.\ \mathtt{x}{\mapsto}\alpha * P_i[\beta/\mathtt{x}] * \boxed{Q_i[\beta/\mathtt{x}]})$
 9: else if $C$ is $(C_1; C_2)$ then
10:     $(G_1, q_1) \leftarrow$ SYMB-EXEC$(p, \mathit{Rely}, C_1)$
11:     $(G_2, q_2) \leftarrow$ SYMB-EXEC$(q_1, \mathit{Rely}, C_2)$
12:     return $(G_1 \cup G_2, q_2)$
13: else if $C$ is $(C_1 \oplus C_2)$ then
14:     $(G_1, q_1) \leftarrow$ SYMB-EXEC$(p, \mathit{Rely}, C_1)$
15:     $(G_2, q_2) \leftarrow$ SYMB-EXEC$(p, \mathit{Rely}, C_2)$
16:     return $(G_1 \cup G_2, q_1 \lor q_2)$
17: else if $C$ is $(C_0)^*$ then
18:     repeat
19:         $p_{\text{old}} \leftarrow p$
20:         $(G_{\text{new}}, p) \leftarrow$ ABS-POST(SYMB-EXEC$(p, \mathit{Rely}, \mathtt{skip} \oplus C_0)$)
21:     until $p = p_{\text{old}}$
22:     return $(G \lor G_{\text{new}}, p)$
23: else if $C$ is atomic $C_0$ then
24:     $(G, \bigvee_i \exists \boldsymbol{x}_i.\ P_i * \boxed{Q_i}) \leftarrow$ SYMB-EXEC$(p, \emptyset, C_0)$
25:     return $(G, \bigvee_i \exists \boldsymbol{x}_i.\ P_i * \boxed{\text{STABILIZE}(Q_i, \mathit{Rely})})$
```

Second, symbolic execution of atomic blocks is incomplete if the body of an atomic block contains an execution path with more than one memory write. For example, consider the atomic block `atomic` ([a] := 10; [b] := 10). Assuming the two memory locations a and b were initialized to α and β respectively, then the atomic block does the action:

$$A \stackrel{\text{def}}{=} (\mathsf{emp} \mid \mathsf{a}{\mapsto}\alpha * \mathsf{b}{\mapsto}\beta \rightsquigarrow \mathsf{a}{\mapsto}10 * \mathsf{b}{\mapsto}10)$$

However, calling SYMB-EXEC would return two actions:

$$G \stackrel{\text{def}}{=} \{(\mathsf{b}{\mapsto}\beta \mid \mathsf{a}{\mapsto}\alpha \rightsquigarrow \mathsf{a}{\mapsto}10),\ (\mathsf{a}{\mapsto}10 \mid \mathsf{b}{\mapsto}\beta \rightsquigarrow \mathsf{b}{\mapsto}10)\}$$

It is easy to show that $[\![\{A\}]\!] \subsetneq [\![G]\!]$. Action A can be simulated by doing the two actions of G in sequence. In the other direction, G allows us to change one field at a time, whereas A demands that both fields are modified in one step.

Normally this form of incompleteness is harmless because atomic commands arise from a single memory read, write or CAS and hence contain at most one memory write. More advanced atomic commands, such as those due to a DCAS or ones containing assignments to auxiliary variables, can contain more than one memory writes. To deal with such atomic commands precisely we introduce a parallel memory write command, which writes to multiple heap locations in one step. This is analogous to the parallel assignment statement present in some programming languages. Symbolic execution of parallel memory writes executes each write separately, but collects all the updates together and returns one action describing all the updates.

Third, the sub-procedures used by the analysis are often incomplete. This includes SUBTRACT, MAY-SUBTRACT, the abstraction of separation logic assertions and of RGSep actions, and the reachability heuristic for deciding ownership transfer. From all these, incompleteness arising from the abstraction of separation logic assertions is the most frequent.

Small Example. To illustrate our symbolic execution and action inference algorithms, consider a trivial shared stack which supports only a push operation:

$$\begin{aligned}
\mathit{init} &\stackrel{\text{def}}{=} \mathtt{S} := \mathtt{malloc}();\ [\mathtt{S}] := \mathtt{NULL} \\
\mathit{push} &\stackrel{\text{def}}{=} \mathtt{y} := \mathtt{malloc}();\ \mathtt{b} := \mathtt{false}; \\
&\quad \left(\begin{array}{l} \mathtt{assume}(\neg\mathtt{b});\ \mathtt{atomic}\langle \mathtt{x} := [\mathtt{S}]\rangle;\ \mathtt{atomic}\langle [\mathtt{y}] := \mathtt{x}\rangle; \\ \mathtt{atomic}\left\langle \mathtt{t} := [\mathtt{S}];\left(\begin{array}{l} (\mathtt{assume}(\mathtt{t} = \mathtt{x}); [\mathtt{S}] := \mathtt{y}; \mathtt{b} := \mathtt{true}) \\ \oplus\, \mathtt{assume}(\mathtt{t} \neq \mathtt{x}) \end{array}\right)\right\rangle \end{array}\right)^{*}; \\
&\quad \mathtt{assume}(\mathtt{b})
\end{aligned}$$

The stack is implemented as a linked list starting from address S. The initialization method, *init*, creates an empty stack. The method *push* creates a new node (y) and tries to add it at the beginning of the stack using a compare&swap (CAS) instruction inside a loop. The big atomic block inside the loop results from desugaring the CAS instruction. Similarly, the variable b arises from a break statement.

Let us execute action inference on this example: SYMB-EXEC(emp, $\emptyset$, *init*) returns the postcondition $\mathtt{S} \mapsto \mathtt{NULL}$. As every assertion is stable under the empty rely, stabilization does nothing and returns the same assertion. Then, action inference calls SYMB-EXEC($\boxed{\mathtt{S} \mapsto \mathtt{NULL}}$, $\emptyset$, *push*). Symbolically executing the first two commands of *push* results in the state $\exists\alpha.\ \neg\mathtt{b} * \mathtt{y}\mapsto\alpha * \boxed{\mathtt{S} \mapsto \mathtt{NULL}}$.

Now consider the loop of *push*. The first memory read is from the shared state and gives us the postcondition: $\exists\alpha.\ \mathtt{x} = \mathtt{NULL} * \neg\mathtt{b} * \mathtt{y}\mapsto\alpha * \boxed{\mathtt{S} \mapsto \mathtt{NULL}}$. Next, there is a local write, which gives the postcondition: $\mathtt{x}{=}\mathtt{NULL} * \neg\mathtt{b} * \mathtt{y}\mapsto\mathtt{x} * \boxed{\mathtt{S} \mapsto \mathtt{NULL}}$. Then, there is the big atomic block representing a CAS. After the memory read, $\mathtt{t} := [\mathtt{S}]$, we get: $\mathtt{t}{=}\mathtt{x} * \mathtt{x}{=}\mathtt{NULL} * \neg\mathtt{b} * \mathtt{y}\mapsto\mathtt{x} * \boxed{\mathtt{S} \mapsto \mathtt{NULL}}$. Therefore, from the two conditional branches, only the first one is possible. In this branch, the memory write is shared; so symbolic execution has to compute an action. According to reachability heuristic, the memory cell $\mathtt{y}\mapsto\mathtt{x}$ becomes shared, as it is reachable from y. The postcondition is $\mathtt{t}{=}\mathtt{x} * \mathtt{x}{=}\mathtt{NULL} * \neg\mathtt{b} * \boxed{\mathtt{S} \mapsto \mathtt{y} * \mathtt{y}\mapsto\mathtt{NULL}}$ and the inferred action is

$$A_1 \stackrel{\text{def}}{=} \mathsf{emp} \mid \mathtt{S} \mapsto \mathtt{NULL} \rightsquigarrow \mathtt{S} \mapsto y * y \mapsto \mathtt{NULL}$$

Then, as b becomes true, symbolic execution exits the loop, and returns.

Therefore, in the first iteration of its fixpoint loop, action inference has computed $G = \{A_1\}$ and $Inv = \mathtt{S} \mapsto \mathtt{NULL}$. In the second iteration, Inv is no longer stable. Stabilization returns $Inv = (\mathtt{S} \mapsto \mathtt{NULL} \vee \exists y.\ \mathtt{S} \mapsto y * y \mapsto \mathtt{NULL})$, and symbolic execution also returns the action $A_2 \overset{\text{def}}{=} x \mapsto \mathtt{NULL} \mid \mathtt{S} \mapsto x \rightsquigarrow \mathtt{S} \mapsto y * y \mapsto x$. In the third iteration, Inv becomes $\mathit{listseg}(\mathtt{S}, \mathtt{NULL})$[1] and symbolic execution also returns: $A_3 \overset{\text{def}}{=} \mathit{listseg}(x, 0) \mid \mathtt{S} \mapsto x \rightsquigarrow \mathtt{S} \mapsto y * y \mapsto x$. In the fourth iteration, Inv is already stable, and symbolic execution returns no new actions. Therefore, action inference terminates after four iterations and having found three actions.

4 Non-Standard Join

As presented above, the Infer-Actions and Symb-Exec algorithms use set union to combine sets of actions. Using set union, however, produces too many actions, many of which are unnecessary. For instance, in the stack example above, the actions A_1 and A_2 are both included in the action A_3, and hence are unnecessary once A_3 is discovered.

As remarked in Sect. 2, having a large set of actions makes stabilization calculations slower, which in turn slows down action inference. More importantly, however, the output of action inference becomes difficult to read and slows down any verification procedures that use action inference as their first step (e.g. [9]).

Therefore, we shall replace set union with a more aggressive join operation. The idea is to define a 'lossless' join that removes actions that are already included in other actions. Note that there is a natural inclusion order on actions: action a is semantically included in action b if and only if $\mathcal{A}[\![a]\!] \subseteq \mathcal{A}[\![b]\!]$. In general, testing whether $\mathcal{A}[\![a]\!] \subseteq \mathcal{A}[\![b]\!]$ is undecidable. We can, however, define the following decidable approximation to action inclusion:

Definition 3. $(R_1 \mid P_1 \rightsquigarrow Q_1) \sqsubseteq (R_2 \mid P_2 \rightsquigarrow Q_2)$ *if and only if there exists a substitution* σ *of the logical variables such that* $P_1 = \sigma(P_2)$, $Q_1 = \sigma(Q_2)$, *and* $R_1 \vdash \sigma(R_2) * \mathsf{true}$.

It is easy to check that if $a \sqsubseteq b$, then $\mathcal{A}[\![a]\!] \subseteq \mathcal{A}[\![b]\!]$. To calculate $(R_1 \mid P_1 \rightsquigarrow Q_1) \sqsubseteq (R_2 \mid P_2 \rightsquigarrow Q_2)$, we run first order unification to find a substitution σ such that $P_1 = \sigma(P_2)$ and $Q_1 = \sigma(Q_2)$, and then call Subtract$(R_1, \sigma(R_2), \emptyset)$ to decide whether $R_1 \vdash \sigma(R_2) * \mathsf{true}$.

Example 2. Consider the actions $A \overset{\text{def}}{=} (\mathtt{y} \mapsto 3 \mid \mathtt{x} \mapsto 0 \rightsquigarrow \mathtt{x} \mapsto 1)$, $A' \overset{\text{def}}{=} (\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 3 \rightsquigarrow \mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 3)$, and $B \overset{\text{def}}{=} (\mathtt{x} \mapsto a \rightsquigarrow \mathtt{x} \mapsto 1)$. Clearly, $\mathcal{A}[\![A]\!] = \mathcal{A}[\![A']\!] \subseteq \mathcal{A}[\![B]\!]$, because A and A' allow us to write 1 to x only when it previously contained 0 and y contained 3, whereas B allows us to write 1 to x regardless of the original value of x and the value of y. It is also easy to check that $A \sqsubseteq B$; just take σ to be the substitution mapping a to 0. In contrast, $A' \not\sqsubseteq B$, $A \not\sqsubseteq A'$, and $A' \not\sqsubseteq A$. In principle, we could have defined a finer approximation to inclusion so that

[1] The list segment comes from applying Distefano's abstraction to the formula $\exists yz.\ \mathtt{S} \mapsto y * y \mapsto z * z \mapsto \mathtt{NULL}$, which arises during the initial stabilization.

Data structure	**No join**			**Lossless join**			**Man**
	#I	**#A**	**Time**	**#I**	**#A**	**Time**	**#A**
Treiber stack [17]	4	5	0.09s	4	2	0.08s	2
M&S two-lock queue [13]	5	26	0.33s	5	12	0.25s	6
M&S non-blocking queue [13]	5	10	1.69s	5	6	1.45s	3
DGLM non-blocking queue [7]	5	12	2.23s	5	8	1.97s	3
Lock-coupling list [10]	4	21	0.98s	4	10	0.81s	4
Optimistic list [10]	5	30	109.06s	5	10	52.29s	4
Lazy list [10]	5	48	59.98s	5	13	26.21s	5
CAS-based set [21]	3	9	24.74s	2	5	8.80s	3
DCAS-based set [21]	2	6	0.31s	2	4	0.27s	2

Fig. 1. Verification times for the memory safety benchmarks.

Data structure	**No join**			**Lossless join**			**Man**
	#I	**#A**	**Time**	**#I**	**#A**	**Time**	**#A**
Treiber stack [17]	4	5	0.14s	4	2	0.09s	2
M&S two-lock queue [13]	6	39	0.70s	6	13	0.48s	6
M&S non-blocking queue [13]	6	14	4.37s	6	7	3.76s	3
DGLM non-blocking queue [7]	6	16	4.88s	6	9	4.22s	3

Fig. 2. Verification times for the linearizability benchmarks.

the latter three inclusions were also true, but such a finer approximation would have been significantly slower to compute. Instead, we simply avoid generating problematic actions such as A'.

From this computable check for action inclusion, we define the following 'lossless' join operator:

$$\begin{aligned} A \sqcup \{b\} &\stackrel{\text{def}}{=} \textbf{if } \exists a \in A.\ b \sqsubseteq a \textbf{ then } A \textbf{ else } \{b\} \cup \{a \in A \mid a \not\sqsubseteq b\} \\ A \sqcup \{b_1, \dots, b_n\} &\stackrel{\text{def}}{=} (\cdots (A \sqcup \{b_1\}) \sqcup \dots) \sqcup \{b_n\} \end{aligned}$$

The join $A \sqcup B$ inserts the actions of B into A one at a time. For every such action, b, if it is already included in A, it is discarded; otherwise, b is added into A and every action of A that is included in b is removed.

Finally, we prove that join does not forget any information.

Lemma 2. *For all sets of actions A and B, $[\![A \sqcup B]\!] = [\![A \cup B]\!]$.*

The proof of this lemma follows from the observation that if $a \sqsubseteq b$, then $[\![\{a, b\}]\!] = [\![\{b\}]\!]$. Lemma 2 means that we can replace union by the lossless join in the INFER-ACTIONS and SYMB-EXEC algorithms without any loss in precision.

5 Evaluation

We have run action inference on a number of fine-grained concurrent algorithms from the literature. For the first set of benchmarks (Fig. 1), we have proved memory safety and inferred the expected data structure shape invariants (e.g., in all cases we can show the data structures are acyclic). For the second set of benchmarks (Fig. 2), we have taken the stack and queue algorithms from Fig. 1 and have proved linearizability using the method described in [19, 1], which is to instrument the algorithms by manually inserting auxiliary code describing the linearization points of each algorithm.

We have run our tool in two modes: with no join enabled, and with lossless join enabled. For each run, we have recorded the number of iterations that Infer-Action takes in order to reach a fixpoint (**#I**), the number of actions inferred (**#A**), and the total verification time (**Time**). The final column reports the minimum number of actions needed for a manual proof of the algorithm. The tests were conducted on a 3.4GHz Pentium 4 processor running Windows Vista.

Enabling join significantly reduces the number of actions inferred, and hence also the verification times especially for the more difficult benchmarks. The number of actions inferred using the lossless join is still quite larger than what would have been written by hand. This is mainly due to a number of unnecessary case splits present in the set of inferred actions. Normally, enabling lossless join does not affect the number of iterations taken by Infer-Action. This is expected, because the action set calculated using lossless join is semantically equivalent to the set calculated using normal set union. Somewhat counter-intuitively, however, the CAS-based set example finishes in fewer iterations when lossless join is used. This is probably due to the incompleteness of entailment checking between separation logic formulas.

Trying to further reduce the number of inferred actions, we have experimented with a more aggressive action abstraction that drops all list segments from the actions' contexts. While this abstraction works well for most of the examples, the resulting actions are too weak to prove functional correctness of the linked list benchmarks. (They are sufficient for proving memory safety.)

Other Uses of Action Inference. Action inference has already been used as a subcomponent in two related verification procedures. The first use was in verifying liveness properties of non-blocking algorithms by Gotsman et al. [9]. There, one first runs action inference to prove memory safety and to compute a set of RGSep actions. Then, one does a layered proof search attempting to show that certain actions are not executed infinitely often and that certain operations terminate. This proof search is quadratic in the number of inferred actions; so inferring few actions is necessary for achieving good performance.

The second use is in a new verification procedure for linearizability that does not require linearization point annotations. This procedure constructs a list of candidate linearization point assignments, and then searches through the list checking whether any of those assignments is valid. In this case, action inference is executed both as an initial phase in order to find candidate linearization point assignments and at each step of the proof search in order to determine whether a given linearization point assignment is valid. This procedure can verify the benchmarks in Fig. 2 within 10 seconds each.

6 Related Work

Action inference extends the original work on RGSep shape analysis [5]. It is similar in spirit to the thread-local shape analysis by Gotsman et al. [8], but technically quite different. Both works attempt to verify one thread at a time and do a global fixpoint calculation to compute the interaction between threads. The major difference is that Gotsman calculates resource invariants, whereas we calculate a set of actions. The shift from invariants to sets of actions makes our method more expressive and thus able to reason about fine-grained concurrency, but also required us to introduce concepts such as stabilization. In contrast, Gotsman et al. can handle only coarse-grained concurrency.

Manevich et al. [12], Berdine et al. [3], and Segalov et al. [16] have developed a series of related shape analyses that suitably restrict the correlations between the states of different threads that are tracked. This gives them a very strong thread-modular flavour. Their main difference is that action inference does an abstract interpretation over both invariants and actions, whereas the other three analyses do abstract interpretation only over invariants. The second important difference is the underlying abstract domain: we use separation logic, whereas they use three value logic. Using action inference, we can verify roughly the same programs and properties as the other three analyses, but our verification times have so far been significantly faster.

7 Conclusion

We have presented an algorithm for computing the interference caused by a program enabling us to verify safety properties of concurrent heap-manipulating programs. Our action inference algorithm forms the basis of more advanced verification methods for proving certain liveness properties [9] and linearizability.

In the future, we would like to apply action inference to larger and more complex concurrent libraries. The main technical obstacle in achieving this is to make the sequential shape analyses expressive enough to describe the invariants of such libraries. We would also like to consider program verification in the context of relaxed memory models, and to replace the reachability heuristic for determining ownership transfer with a more robust technique possibly based on footprint analysis [14] or bi-abduction [4].

Acknowledgements I would like to thank Alexey Gotsman, Matthew Parkinson, Mohammad Raza, Mooly Sagiv, and Hongseok Yang for useful discussions and comments, and especially the anonymous reviewers for their constructive and detailed feedback.

References

1. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E. Comparison under abstraction for verifying linearisability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)

2. Berdine, J., Calcagno, C., O'Hearn, P. W. Symbolic execution with separation logic. In: APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
3. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S. Thread quantification for concurrent shape analysis. In Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H. Compositional shape analysis by means of bi-abduction. In: POPL 2009, pp. 289–300. ACM (2009)
5. Calcagno, C., Parkinson, M., Vafeiadis, V. Modular safety checking for fine-grained concurrency. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
6. Distefano, D., O'Hearn, P.W., Yang, H. A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
7. Doherty, S., Groves, L., Luchangco, V., Moir, M. Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
8. Gotsman, A., Berdine, J., Cook, B., Sagiv, M. Thread-modular shape analysis. In: PLDI 2007. ACM (2007)
9. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V. Proving that non-blocking algorithms don't block. In: POPL 2009, pp. 16–28. ACM (2009)
10. Herlihy, M., Shavit, N. The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
11. Jones, C.B. Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
12. Manevich, R., Lev-Ami, T., Ramalingam, G., Sagiv, M., Berdine, J. Heap decomposition for concurrent shape analysis. In Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 363–377. Springer, Heidelberg (2008)
13. Michael, M., Scott, M. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC 1996. ACM (1996)
14. Raza, M., Calcagno, C., Gardner, P. Automatic parallelization with separation logic. In Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009)
15. Reynolds, J.C. Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002)
16. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M. Efficiently tracking thread correlations. In: APLAS 2009. Springer, Heidelberg (2009)
17. Treiber, R.K. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.
18. Vafeiadis, V. Fine-grained concurrency verification. PhD dissertation, University of Cambridge Computer Laboratory. Tech. report UCAM-CL-TR-726 (2007)
19. Vafeiadis, V. Shape-value abstraction for verifying linearizability. In Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)
20. Vafeiadis, V., Parkinson, M. A marriage of rely/guarantee and separation logic. In Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
21. Vechev, M., Yahav, E. Deriving linearizable fine-grained concurrent objects. In: PLDI 2008, pp. 125–135. ACM (2008)

A May-Subtraction Implementation

This section describes an efficient implementation of may-subtraction for the following simple list segment domain:

$$
\begin{array}{rll}
P, Q, R ::= \mathsf{false} \mid (\exists \boldsymbol{z}.\, \Pi \wedge \Sigma) \mid P \vee Q & & \text{Full assertions} \\
\Pi ::= \mathsf{true} \mid E = E' \mid E \neq E' \mid \Pi \wedge \Pi & & \text{Pure part} \\
\Sigma ::= \mathsf{emp} \mid \mathsf{true} \mid E \mapsto_A E' \mid ls_A(E, E') \mid \Sigma * \Sigma & & \text{Spatial part}
\end{array}
$$

To implement may-subtraction efficiently, each primitive assertion is annotated with a permission set, $\emptyset \neq A \subseteq \{1, 2, 3\}$, represented as a bit-vector. These permission annotations are used only internally within the may-substraction calculation; its interface does not expose the permission annotations.

Similar to Berdine et al. [2], we represent formulas in a canonical form up to the usual properties of $*$, $\wedge$, and $\vee$ (commutativity, associativity, distribution of $*$ and $\wedge$ over disjunction, identity and nullary elements, $\mathsf{true} * \mathsf{true} = \mathsf{true}$), substitution of equated terms, and the following three new normalization rules:

$$
\begin{array}{rcl}
x{\mapsto_A}y * x{\mapsto_B}z & \Longleftrightarrow & y = z \wedge x{\mapsto_{A \odot B}}y \\
x{\mapsto_A}y * ls_B(x, z) & \Longleftrightarrow & x = z \wedge x{\mapsto_A}y \vee x{\mapsto_{A \odot B}}y * ls_B(y, z) \\
ls_A(x, y) * ls_B(x, z) & \Longleftrightarrow & ls_{A \odot B}(x, y) * ls_B(y, z) \vee ls_{A \odot B}(x, z) * ls_A(z, y)
\end{array}
$$

where we take $x{\mapsto_{A \odot B}}y$ to mean $x{\mapsto_{A \cup B}}y$ if $A \cap B = \emptyset$ and false otherwise. Similarly, $ls_{A \odot B}(x, y)$ stands for $ls_{A \cup B}(x, y)$ if $A \cap B = \emptyset$, and $x = y \wedge \mathsf{emp}$ otherwise. These rules check whether there are any overlapping spatial conjuncts, and perform case splits to eliminate such conjuncts. (Repeated application of these rules terminates, because within each disjunct each rule either reduces the number of spatial conjuncts, or keeps the same number of spatial conjuncts, but increments one of their permission annotations.) Our rules are better than the normalization rules of Berdine et al. [2], as they resolve all 'spooky' disjuncts and avoid a quadratic expansion of the formula in the common case.

Our implementation of may-subtraction uses the permission annotations to exploit the above normalization rules. It is defined in terms of a helper function:

$$
\textsc{May-Subtract}(P, Q, R) \overset{\text{def}}{=} \textsc{MaySubHelper}(P_{\{1\}} * R_{\{2\}} * Q_{\{2,3\}})
$$

where P_A marks the spatial conjuncts of the (non-annotated) assertion P with A. Permission $\{1\}$ indicates that the conjunct belongs the P; permission $\{2\}$ says that it belongs to either Q or the context R (and has to be matched with something in P), whereas permission $\{3\}$ indicates the conjunct has to be matched with something in P and then removed for the result.

The helper function, MaySubHelper, is defined in Alg. 6. First, it applies the normalization rules. If all conjuncts are matched (i.e., none remain with a label not containing 1), it returns all the conjuncts that must not be removed (i.e., those whose label does not contain 3). If, there is an unmatched $\mapsto$, then MaySubHelper does a case split as to which primitive conjunct the $\mapsto$ belongs, and continues. Otherwise, if an unmatched list segment remains,

Algorithm 6 MaySubHelper(P)

```
  $Res \leftarrow \mathsf{false}$
  for each disjunct $\Pi \wedge \Sigma$ in Normalize($P$) do
    if $\exists x, y, A$ such that $(x \mapsto_A y) \in \Sigma$ and $1 \notin A$ then
      $Res \leftarrow Res \vee$ MaySubHelper($\bigvee_{S \in \Sigma} \Pi \wedge$ Expose$(x, y, S) * \circledast(\Sigma \setminus S)$)
    else if $\exists x, y, A$ such that $(ls_A(x, y)) \in \Sigma$ and $1 \notin A$ and $3 \notin A$ then
      $Res \leftarrow Res \vee$ MaySubHelper($\Pi \wedge \Sigma$)
    else if $\exists x, y, A$ such that $(ls_A(x, y)) \in \Sigma$ and $1 \notin A$ then
      $Res \leftarrow Res \vee (\Pi \wedge \mathsf{true})$
    else
      $Res \leftarrow Res \vee (\Pi \wedge \circledast\{S \mid S_A \in \Sigma \wedge 3 \notin A\})$
  return $Res$
```

where

Expose$(x, y, \mathsf{true}) \overset{\text{def}}{=} x \mapsto_{\{1\}} y * \mathsf{true}$ Expose$(x, y, z \mapsto_B w) \overset{\text{def}}{=} x = z \wedge x \mapsto_B w$

Expose$(x, y, ls_B(z, w)) \overset{\text{def}}{=} ls_B(z, x) * x \mapsto_B y * ls_B(y, w)$

MaySubHelper conservatively assumes that it could match any part of the formula.

Our May-Subtract algorithm is a significant improvement over the septraction elimination algorithm by Calcagno et al. [5], as it can handle contextual matches (i.e., the R component) and it delays the application of the expensive rule that exposes $\mapsto$-assertions. The soundness of our algorithm follows from the semantics of separation logic assertions and permissions.