EUROPEAN COMMISSION
SEVENTH FRAMEWORK PROGRAMME
FP7-ICT-2011-C
Grant agreement no.: 308830

Deliverable D6.1

Intermediate report for WP6: Automation

Project acronym	ADVENT
Project title	Architecture-driven verification of systems software
Funding scheme	FP7 FET Young Explorers
Scientific coordinator	Dr. Alexey Gotsman, IMDEA Software Institute, Alexey.Gotsman@imdea.org, +34 911 01 22 02

1 Summary

In Work Package 6 we work towards the automation of the program verification activity. Building on the logics developed in Work Packages 2-5, we develop algorithms and tool support with the goal of improving the economic viability of sound formal verification of complex real-world systems software.

We have already obtained significant results in this area, both with respect to tool support for verification of deep correctness properties, and with respect to fully automatic program analyses.

- We have implemented the aspect-oriented approach for proving linearisability of concurrent queues developed in WP5 inside the Cave thread-modular concurrent program verifier, and used it to verify automatically the linearisability of the Herlihy and Wing queue.
- Bringing decomposition ideas (see also WP2) into program analysis, we built a compositional analysis that is as precise as a top-down analysis. We developed a compositional version of the connection pointer analysis by Ghiya and Hendren and applied it to real-world Java programs. As expected, the compositional analysis scales much better than the original top-down version. The loss of precision ranges only between 2-5%.
- We have developed an approach for extending a separation logic-based verifier to support modular specification and verification of the interactive behavior of C programs. Our approach is compositional in two senses: it supports building higher-level interaction APIs on top of lower-level ones; and it supports programs that use multiple independently-developed interaction APIs.
- We have developed a preliminary encoding of the abstract reachability-based approach for modular verification of TSO relaxed memory programs developed in WP3 into the annotation language of VeriFast, and used it to verify three important concurrency patterns.
- We have extended VeriFast with support for rely-guarantee reasoning, in the form of *shared boxes*. We have used them to verify a number of fine-grained concurrent algorithms.
- We have developed an interprocedural shape analysis for effectively cutpoint-free programs. We also present a variation of our technique which allows procedure invocations to have up to a bounded number of live references to cutpoint objects. Our work shows that most of the benchmarks used for evaluating shape analysis algorithms are comprised of effectively cutpoint-free programs.
- In preparation for extending VeriFast with annotation inference capabilities, we have developed a detailed formalization and soundness proof of an approach for local shape analysis based on separation logic from the literature.

2 Extension of Cave

Automatic Verification of Concurrent Queues [5] ***(attached to D5.1)*** As part of WP6 we have implemented the aspect-oriented approach for proving linearisability of concurrent queues developed in WP5 inside Cave, and used it to verify automatically the linearisability of the Herlihy and Wing queue.

Cave is a thread-modular concurrent program verifier. Its input is a program consisting of some initialisation code and a number of concurrent methods, which are all executed in parallel an unbounded number of times each. When successful, it produces a proof in RGSep that the program has no memory errors and none of its assertions are violated at runtime.

The main modifications we had to perform to the tool were: (1) to add code that instruments dequeue methods with a prophecy argument guessing its return value; (2) to improve Cave's

abstraction function so that it can remember properties of the form $v \notin X$, as these are needed to verify the basic aspect-oriented properties of queues; and (3) to add some glue code that constructs the verification conditions corresponding to the aspect-oriented linearisability proof technique and runs the underlying prover to verify them.

As Cave does not support arrays (it only supports linked lists), we gave the tool a linked-list version of the Herlihy and Wing queue, for which it successfully verified that there are no linearisability violations.

3 Program Analyses

Modular Lattices for Compositional Interprocedural Analysis [3] ***(attached)*** Interprocedural analyses are *compositional* when they compute over-approximations of procedures in a bottom-up fashion. These analyses are usually more scalable than top-down analyses which compute a different procedure summary for every calling context. However, compositional analyses are rare in practice, because it is difficult to develop such analyses with enough precision. In this paper, we establish a connection between a restricted class of compositional analyses and so called *modular lattices*, which require certain associativity between the lattice join and meet operations. Our connection provides sufficient conditions for building a compositional analysis that is as precise as a top-down analysis.

We developed a compositional version of the connection pointer analysis by Ghiya and Hendren which is slightly more conservative than the original top-down analysis in order to meet our modularity requirement. We implemented and applied our compositional connection analysis to real-world Java programs. As expected, the compositional analysis scales much better than the original top-down version. The top-down analysis times out in the largest two of our five programs, and the loss of precision due to the modularity requirement in the remaining programs ranges only between 2-5%.

Our work is a case study in a new way for designing and developing interprocedural analyses that can handle real code: It demonstrates that it is possible to design analyses that are as precise as top-down analyses and as scalable as bottom-up ones. In parallel, it also shows the challenges in developing such analyses and, in particular, to the importance of designing the right join operator.

Interprocedural Shape Analysis for Effectively Cutpoint-Free Programs [11] ***(attached)*** We present a framework for local interprocedural shape analysis that computes procedure summaries as transformers of procedure-local heaps (the parts of the heap that the procedure may reach). A main challenge in procedure-local shape analysis is the handling of *cutpoints*, objects that separate the input heap of an invoked procedure from the rest of the heap, which—from the view-point of that invocation—is non-accessible and immutable. In this paper, we limit our attention to *effectively cutpoint-free* programs—programs in which the only objects that separate the callee's heap from the rest of the heap, when considering *live* reference fields, are the ones pointed to by the actual parameters of the invocation. This limitation (and certain variations of it, which we also describe) simplifies the local-reasoning about procedure calls because the analysis needs not track cutpoints. Furthermore, our analysis (conservatively) verifies that a program is effectively cutpoint-free.

Our work extends the state-of-the-art in interprocedural shape analysis using the three-valued-logic framework (TVLA) of Sagiv et al. We also present a variation of our technique which allows procedure invocations to have up to a bounded number of live references to cutpoint objects. This extension provides a new perspective on the relation between TVLA-based analyses and ones based on separation logic, where, intuitively, such references are treated as additional parameters to the procedure. Most importantly, our work shows that most of the benchmarks used for evaluating shape analysis algorithms are comprised of effectively cutpoint-

free programs. Thus opening a new research direction: verifying heap-manipulating programs with an unbounded number of live references to cutpoint objects.

4 Extensions of VeriFast

VeriFast [9, 8, 7] is a prototype sound modular program verification tool being developed at KUL. It accepts as input a C or Java program annotated with preconditions, postconditions, loop invariants and other assertions expressed in a variant of separation logic, and it verifies each function/method against its contract through symbolic execution, using a separation logic representation of memory, very much like Smallfoot [1]. In contrast to Smallfoot, VeriFast is optimized for verification of deep functional properties instead of automation.

The following results of this work package have been developed in the context of VeriFast. The I/O, TSO, and shared boxes extensions have been incorporated into the latest VeriFast distribution, and example annotated programs are included in the directory `examples/{io,tso,shared_boxes}`.

Interactive Behavior [13] ***(attached)*** We have developed an approach for modularly specifying and verifying the interactive behavior of C programs, through a form of spatio-temporal reasoning: the specification is given in the form of a spatial assertion over a resource algebra describing a Petri net which expresses the temporal constraints on the program's behavior. The approach abstracts over the precise alphabet of the program's interaction, thus supporting compositionality both in the form of modules defining composite I/O actions on top of other modules and in the form of programs performing I/O actions implemented by multiple independent modules.

The approach is best understood by example. Here is a specification in the annotation language of VeriFast of a C program that prints "Hi":

```
void putchar(char c);
    //@ requires time(?t0) &*& putchar(t0, c, ?t1);
    //@ ensures time(t1);

void main()
    //@ requires time(?t0) &*& putchar(t0, 'H', ?t1) &*& putchar(t1, 'i', ?t2);
    //@ ensures time(t2);
{
    putchar('H');
    putchar('i');
}
```

The precondition of `main` describes a Petri net with three places `t0`, `t1`, and `t2`, and two transitions, corresponding to printing "H" and "i". It also describes a marking: there is a single token at place `t0`. The postcondition describes a marking where the token is at place `t2`, and furthermore, the two transitions have been consumed. The only way for the program to arrive at the state described in the postcondition is by printing "H" and "i".

A more complex example, which illustrates the compositionality of the approach, is the following:

```
/*@
predicate cat(time t0, list<char> cs, time t1) =
    getchar(t0, ?c, ?t2) &*&
    c < 0 ?
        cs == nil &*& t1 == t2
    :
        putchar(t2, c, ?t3) &*& cat(t3, ?cs1, t1) &*&
        cs == cons(c, cs1);
```

```
@*/
void cat()
    //@ requires time(?t0) &*& cat(t0, ?cs, ?t1);
    //@ ensures time(t1);
{
    for (;;)
        //@ invariant time(?t2) &*& cat(t2, _, t1);
    {
        int r = getchar();
        if (r < 0) break;
        putchar(r);
    }
}
```

The function `cat` reads characters from standard input until it reaches end-of-file, and writes them to standard output as it reads them. Notice the two forms of compositionality: firstly, the contract of `cat` is of the same form as that of `putchar` above; it is not possible to tell whether a function is a primitive I/O operation. Secondly, the functions `putchar` and `getchar` could be defined by independently developed libraries; there is no need to define the alphabet of I/O actions in a single place.

The technical report [13] formalizes the programming language and the proof system, defines the soundness property, and shows a detailed proof outline for an example program.

Relaxed Memory Consistency: TSO [6] ***(attached to D3.1)*** We have developed a preliminary encoding of the abstract reachability-based approach for modular verification of TSO relaxed memory programs developed in WP3 into the annotation language of VeriFast, and used it to verify three interesting patterns: a C TSO lock implementation, a C program that uses TSO operations to achieve zero-overhead unrestricted safe field access as would be required in a Java virtual machine, and a C program using TSO operations to implement a zero-overhead producer-consumer pattern.

In the development of our encoding, we needed to take special care to make sure that the abstract updates associated with TSO updates and the abstract lower bounds associated with TSO reads do not depend on ghost variables. Indeed, the naive approach of specifying the abstract updates and abstract lower bounds as ghost arguments of the TSO operations is unsound, since VeriFast allows ghost arguments to depend on ghost variables.

In our current encoding, we work around this issue by requiring the list of all abstract updates and abstract lower bounds to be used by operations on a given TSO space to be specified when the TSO space is created. The specific abstract update or lower bound for a particular operation is then selected by passing an index into this list as a non-ghost argument to the TSO operation (which appears in the program as a C function call). Furthermore, to allow the abstract updates and lower bounds to depend on the (non-ghost) state of the thread, we allow a variable number of additional non-ghost arguments to be passed to the TSO operations. These are passed on as extra arguments to the abstract updates and lower bounds.

This encoding is sound but it has the downside that it requires modifications to the C program: extra arguments to the TSO operations, and extra variables to track the thread state.

A better approach which we envision for future work is to extend VeriFast with support for additional ghost-levels of code and variables, beyond the level of reality (the lowest ghost-level) and the single existing ghost-level. Information flow from higher to lower ghost-levels would be disallowed. For the TSO encoding, we would use three ghost-levels: the real level, the semi-ghost level, and the full ghost level. Full ghost variables can be modified as part of TSO updates; abstract updates and lower bounds are specified as semi-ghost arguments.

Rely-Guarantee Reasoning [14] ***(attached)*** In plain separation logic, a thread either has full ownership of a memory location and knows the value at the location, or it has no ownership and no knowledge of the value of the location. Existing work proposes a marriage of rely-guarantee reasoning [10] and separation logic to address this. We describe the shared boxes mechanism, which marries separation logic and rely-guarantee reasoning in VeriFast.

To allow multiple threads to access a set of memory locations concurrently, while at the same time allowing each thread to retain partial knowledge of the values of the memory locations, the proof author can create a *shared box*. Upon creation, the shared box takes ownership of the memory locations described by its *box invariant*. Threads may mutate the memory held by a shared box only if the mutation complies with one of the *action specifications* declared in the *box class*. An action specification consists of a precondition and a postcondition over the variables bound by the box invariant. Furthermore, threads may retain partial knowledge about a shared box in the form of an instance of a *handle predicate* declared in the box class, and verified to be stable with respect to the box class' actions.

Shared boxes are not the only means available in VeriFast to retain partial knowledge about shared data. The other means is through *ghost variables* and *fractional permissions* [2]. In principle, any program can be verified using ghost variables and fractional permissions. However, expressing arbitrary history constraints in this way can be very cumbersome; shared boxes offer a more direct, more convenient mechanism.

We introduce and motivate the shared boxes mechanism using a minimalistic example and a realistic example. The minimalistic example is a *counter* program where one thread continuously increments a counter and other threads check that the counter does not decrease. For the realistic example, we verify functional correctness of the Michael-Scott queue [12], a lock-free concurrent data structure. We define the syntax and semantics of a simple C-like programming language, and we define a separation logic with shared boxes and prove its soundness. We discuss the implementation in VeriFast and the examples we verified using our VeriFast implementation.

Invariant Inference: Detailed Proof [15] ***(attached)*** Currently, VeriFast provides only limited automation: in general, predicate definitions must be folded and unfolded explicitly; any inductive properties must be applied explicitly through lemma calls; and loop invariants must be provided by the user.

In [4], the authors propose a local shape analysis based on separation logic: given a precondition for a function that manipulates linked lists, the algorithm proposed by the authors attempts to infer loop invariants and postconditions fully automatically.

To alleviate the annotation burden when using VeriFast, we are working to integrate the algorithm from [4] into VeriFast. However, the description, soundness argument, and termination argument of the algorithm in [4] are somewhat lacking in clarity and detail, especially to readers new to the field. Therefore, as a first step, we have elaborated a detailed formalisation and soundness proof of the algorithm.

In particular, we formalized the syntax of programs, the concrete semantics, the symbolic execution, and the abstract execution, and we proved that symbolic execution soundly approximates concrete execution, and that abstract execution soundly approximates symbolic execution. Furthermore, we offer a detailed proof of the fact that the abstraction function maps any symbolic state into one of a finite number of canonical symbolic states, thus proving termination of the analysis.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS 2005*, volume 3780 of *LNCS*, pages 52–68, Heidelberg, 2005. Springer.

[2] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[3] G. Castelnuovo, M. Naik, N. Rinetzky, M. Sagiv, and H. Yang. Modular lattices for compositional interprocedural analysis. Technical Report TR-103/13, School of Computer Science, Tel Aviv University, 2013.

[4] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 3920 in LNCS, pages 287–302. Springer, 2006.

[5] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. D'Argenio and H. Melgratti, editors, *CONCUR 2013 - Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 242–256. Springer Berlin Heidelberg, 2013.

[6] B. Jacobs. Verifying TSO programs. Technical Report CW-660, Department of Computer Science, KU Leuven, Belgium, 2014.

[7] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*, 46(1):271–282, 2011.

[8] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311, Heidelberg, 2010. Springer.

[9] B. Jacobs, J. Smans, and F. Piessens. VeriFast. http://www.cs.kuleuven.be/~bartj/verifast/, 2014.

[10] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[11] J. Kreiker, T. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, and E. Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In A. Voronkov and C. Weidenbach, editors, *Programming Logics*, volume 7797 of *Lecture Notes in Computer Science*, pages 414–445. Springer Berlin Heidelberg, 2013.

[12] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[13] W. Penninckx, B. Jacobs, and F. Piessens. Modular, compositional and sound verification of the input/output behavior of programs. Technical Report CW-663, Department of Computer Science, KU Leuven, Belgium, 2014.

[14] J. Smans, D. Vanoverberghe, D. Devriese, B. Jacobs, and F. Piessens. Shared boxes: rely-guarantee reasoning in VeriFast. Technical Report CW-662, Department of Computer Science, KU Leuven, Belgium, 2014.

[15] A. Timany and B. Jacobs. A local shape analysis based on separation logic: detailed presentation and soundness proof. Technical Report CW-661, Department of Computer Science, KU Leuven, Belgium, 2014.

List of Attached Papers

[3] Ghila Castelnuovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. Modular lattices for compositional interprocedural analysis. Technical report TR-103/13, School of Computer Science, Tel Aviv University, 2013.

[13] Willem Penninckx, Bart Jacobs, and Frank Piessens. Modular, compositional and sound verification of the input/output behavior of programs. Technical report CW-663, Department of Computer Science, KU Leuven, Belgium, 2014.

[14] Jan Smans, Dries Vanoverberghe, Dominique Devriese, Bart Jacobs, and Frank Piessens. Shared boxes: rely-guarantee reasoning in VeriFast. Technical report CW-662, Department of Computer Science, KU Leuven, Belgium, 2014.

[11] Kreiker, J. and Reps, T. and Rinetzky, N. and Sagiv, M. and Wilhelm, Reinhard and Yahav, E. Interprocedural Shape Analysis for Effectively Cutpoint-Free Programs. In Voronkov, Andrei and Weidenbach, Christoph (Eds.), Programming Logics, LNCS 7797, Springer-Verlag, 2013.

[15] Amin Timany and Bart Jacobs. A local shape analysis based on separation logic: detailed presentation and soundness proof. Technical report CW-661, Department of Computer Science, KU Leuven, Belgium.

Modular Lattices for Compositional Interprocedural Analysis

Ghila Castelnuovo, Tel-Aviv University
Mayur Naik, Georgia Institute of Technology
Noam Rinetzky, Mooly Sagiv, Tel-Aviv University
Hongseok Yang, University of Oxford

Abstract

Interprocedural analyses are *compositional* when they compute over-approximations of procedures in a bottom-up fashion. These analyses are usually more scalable than top-down analyses which compute a different procedure summary for every calling context. However, compositional analyses are rare in practice, because it is difficult to develop such analyses with enough precision. In this paper, we establish a connection between a restricted class of compositional analyses and so called *modular lattices*, which require certain associativity between the lattice join and meet operations. Our connection provides sufficient conditions for building a compositional analysis that is as precise as a top-down analysis.

We developed a compositional version of the connection pointer analysis by Ghiya and Hendren which is slightly more conservative than the original top-down analysis in order to meet our modularity requirement. We implemented and applied our compositional connection analysis to real-world Java programs. As expected, the compositional analysis scales much better than the original top-down version. The top-down analysis times out in the largest two of our five programs, and the loss of precision due to the modularity requirement in the remaining programs ranges only between 2-5%.

1. Introduction

Scaling program analysis to large programs is an ongoing challenge for program verification. Typical programs include many relatively small procedures. Therefore, a promising direction for scalability is analyzing each procedure in isolation, using pre-computed summaries for called procedures and computing a summary for the analyzed procedure. Such analyses are called *bottom-up interprocedural analysis* or *compositional analysis*. Notice that the analysis of the procedure itself need not be compositional and can be costly. Indeed, bottom-up interprocedural analyses have been found to scale well [3, 5, 8, 14, 21].

The theory of bottom-up interprocedural analysis has been studied in [7]. In practice, designing and implementing a bottom-up interprocedural analysis is challenging for several reasons: it requires accounting for all potential calling contexts of a procedure in a sound and precise way; the summary of the procedures can be quite large leading to infeasible analyzers; and it may be costly to instantiate procedure summaries. An example of the challenges underlying bottom-up interprocedural analysis is the unsound original formulation of the compositional pointer analysis algorithm in [21]. A corrected version of the algorithm was subsequently proposed in [19] and recently proven sound in [15] using abstract interpretation. In contrast, top-down interprocedural analysis [6, 17, 20] is much better understood and has been integrated into existing tools such as SLAM [1], Soot [2], WALA [9], and Chord [16].

This paper contributes to a better understanding of bottom-up interprocedural analysis. Specifically, we attempt to characterize the cases under which bottom-up and top-down interprocedural analyses yield the same results. To guarantee scalability, we limit the discussion to cases in which bottom-up and top-down analyses use the same underlying abstract domains.

We use *connection analysis* [11], which was developed in the context of parallelizing sequential code, as a motivating example of our approach. Connection analysis is a kind of pointer analysis that aims to prove that two references can never point to the same weakly-connected heap component, and thus ignores the direction of pointers. Despite its conceptual simplicity, connection analysis is flow- and context-sensitive, and the effect of program statements is non-distributive. In fact, the top-down interprocedural connection analysis is exponential, and indeed our experiments indicate that this analysis scales poorly.

Main Contributions. The main results of this paper can be summarized as follows:

- We formulate a sufficient condition on the effect of commands on abstract states that guarantees bottom-up and top-down interprocedural analyses will yield the same results. The condition is based on lattice theory. Roughly speaking, the idea is that the abstract semantics of primitive commands and procedure calls and returns can only be expressed using meet and join operations with constant elements, and that elements used in the meet must be *modular* in a lattice theoretical sense [13].
- We formulate a variant of the connection analysis in a way that satisfies the above requirements. The main idea is to over-approximate the treatment of variables that point to null in all program states that occur at a program point.
- We implemented two versions of the top-down interprocedural connection analysis for Java programs in order to measure the extra loss of precision of our over-approximation. We also implemented the bottom-up interprocedural analysis for Java programs. We report empirical results for five benchmarks of sizes 15K–310K bytecodes. The original top-down analysis times out in over six hours on the largest two benchmarks. For the remaining three benchmarks, only 2-5% of precision was lost by our bottom-up analysis due to the modularity requirement compared to the original top-down version.

This work is based on the master thesis of [4] which contains additional experiments, elaborations, and proofs.

2. Informal Explanation

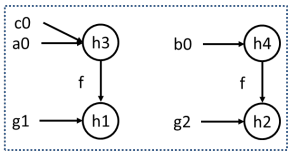
This section presents the use of modular lattices for compositional interprocedural program analyses in an informal manner.

2.1 A Motivating Example

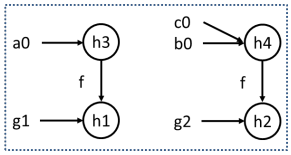
Fig. 1 shows a schematic artificial program illustrating the potential complexity of interprocedural analysis. The main procedure invokes procedure $\mathtt{p}_0$, which invokes $\mathtt{p}_1$ with an actual parameter $\mathtt{a}_0$ or $\mathtt{b}_0$. For every $1 \leq i \leq n$, procedure $\mathtt{p}_i$ either assigns $\mathtt{a}_i$ with formal parameter $\mathtt{c}_{i-1}$ and invokes procedure $\mathtt{p}_{i+1}$ with an actual parameter $\mathtt{a}_i$ or assigns $\mathtt{b}_i$ with formal parameter $\mathtt{c}_{i-1}$ and invokes procedure $\mathtt{p}_{i+1}$ with an actual parameter $\mathtt{b}_i$. Procedure $\mathtt{p}_n$ either assigns $\mathtt{a}_n$ or $\mathtt{b}_n$ with formal parameter $\mathtt{c}_{n-1}$. Fig. 2 depicts the two

```
// a0,...,an,b0,...,bn,g1, and g2 are static variables

static main() {
  g1 = new h1; g2 = new h2; a0 = new h3; b0 = new h4;
  a0.f = g1; b0.f = g2;
  p0();
}

p0() {if(*) p1(a0) else p1(b0)}

p1(c0) {
  if(*) {a1 = c0; p2(a1)} else  {b1 = c0; p2(b1)} }

p2(c1) {
  if(*) {a2 = c1; p3(a2)} else  {b2 = c1; p3(b2)} }
...
pn-1(cn-2) {
   if(*) {an-1 = cn-2; pn(an-1)}
   else  {bn-1 = cn-2; pn(bn-1)}
}

pn(cn-1) if(*) {an = cn-1 else bn = cn-1}
```

Figure 1. Example program.



$d_{conn} = \{\{g_1, a_0, c_0\}, \{g_2, b_0\}, \{a_1\}, \{b_1\}, \ldots, \{a_n\}, \{b_n\}\}$
$d'_{point} = \{\langle a_0, h_3\rangle, \langle b_0, h_4\rangle, \langle c_0, h_3\rangle, \langle h_3, f, h_1\rangle, \langle h_4, f, h_2\rangle, \langle g_1, h_1\rangle, \langle g_2, h_2\rangle\}$



$d_{conn} = \{\{g_1, a_0\}, \{g_2, b_0, c_0\}, \{a_1\}, \{b_1\}, \ldots, \{a_n\}, \{b_n\}\}$
$d'_{point} = \{\langle a_0, h_3\rangle, \langle b_0, h_4\rangle, \langle c_0, h_4\rangle, \langle h_3, f, h_1\rangle, \langle h_4, f, h_2\rangle, \langle g_1, h_1\rangle, \langle g_2, h_2\rangle\}$

Figure 2. Concrete states at the entry of procedure p_1 (see Fig. 1) and the corresponding connection and points-to abstractions.

concrete states that can occur when the procedure p_1 is invoked. There are two different concrete states corresponding to the then- and the else-branch in p_0.

2.2 Connection Analysis and Points-to Analysis

Connection Analysis. We say that two heap objects are *connected* in a state when we can reach from one object to the other by following fields forward or backward. Two variables are *connected* when they point to connected heap objects.

The goal of the connection analysis is to soundly estimate connection relationships between variables. The abstract states d of the analysis are families $\{X_i\}_{i\in I}$ of disjoint sets of variables. Two variables x, y are in the same set X_i, which we call a *connection set*, when x and y may be connected. Fig. 2 depicts two abstract states at the entry of procedure p_1. There are two calling contexts for the procedure. In the first one, a_0 and c_0 point to the same heap object, whose f field goes to the object pointed to by g_1. In addition to these two objects, there are two further ones, pointed to by b_0 and g_2 respectively, where the f field of the object pointed to by b_0 points to the object pointed to by g_2. As a result, there are two connection sets $\{a_0, g_1, c_0\}$ and $\{b_0, g_2\}$. The second calling context is similar to the first, except that c_0 is aliased with b_0 instead of a_0. The connection sets are changed accordingly, and they are $\{a_0, g_1\}$ and $\{b_0, g_2, c_0\}$. In both cases, the other variables are pointing to null, and thus are not connected to any variable.

$$
\begin{aligned}
d_1 &= \{\{g_1, a_0, a_1, a_2, \ldots, a_{n-1}, c_{n-1}\}, \{g_2, b_0\}, \\
&\qquad \{a_n\}, \{b_1\}, \ldots, \{b_{n-1}\}, \{b_n\}\} \\
d_2 &= \{\{g_1, a_0, a_1, a_2, \ldots, b_{n-1}, c_{n-1}\}, \{g_2, b_0\}, \\
&\qquad \{a_{n-1}\}, \{a_n\}, \{b_1\}, \ldots, \{b_n\}\} \\
&\ldots \\
d_{(2^{n-1})} &= \{\{g_1, a_0, b_1, b_2, \ldots, b_{n-1}, c_{n-1}\}, \{g_2, b_0\}, \\
&\qquad \{a_1\}, \ldots, \{a_{n-1}\}, \{a_n\}, \{b_n\}\} \\
d_{(2^{n-1}+1)} &= \{\{g_1, a_0\}, \{g_2, b_0, a_1, a_2, \ldots, a_{n-1}, c_{n-1}\}, \\
&\qquad \{a_n\}, \{b_1\}, \ldots, \{b_{n-1}\}, \{b_n\}\} \\
&\ldots \\
d_{2^n} &= \{\{g_1, a_0\}, \{g_2, b_0, b_1, b_2, \ldots, b_{n-1}, c_{n-1}\}, \\
&\qquad \{a_1\}, \ldots, \{a_{n-1}\}, \{a_n\}, \{b_n\}\}
\end{aligned}
$$

Figure 3. Connection abstraction at the entry of procedure p_n of the program in Fig. 1.

Points-to Analysis. The purpose of the points-to analysis is to compute points-to relations between variables and objects (which are represented by allocation sites). The analysis expresses points-to relations as a set of tuples of the form $\langle x, h\rangle$ or $\langle h_1, f, h_2\rangle$. The pair $\langle x, h\rangle$ means that variable x may point to an object allocated at the site h, and the tuple $\langle h_1, f, h_2\rangle$ means that the f field of an object allocated at h_1 may point to an object allocated at h_2. Fig. 2 depicts the abstract states at the entry to procedure p_1. Also in this case, there are two calling abstract contexts for p_1. In one of them, c_0 may point to h_3, and in the other, c_0 may point to h_4.

2.3 Top-Down Interprocedural Analysis

A standard approach for the top-down interprocedural analysis is to analyze each procedure once for each different calling context. This approach often has scalability problems. One of the reasons is the large number of different calling contexts that arise. In the program shown in Fig. 1, for instance, for each procedure p_i there are two calls to procedure p_{i+1}, where for each one of them, the connection and the points-to analyses compute two different calling contexts for procedure p_{i+1}. Therefore, in both the analyses, the number of calling contexts at the entry of procedure p_i is 2^i.

Fig. 3 shows the connection-abstraction at the entry of procedure p_n. Each abstract state in the abstraction corresponds to one path to p_n. For example, the first state corresponds to selecting the then-branch in all p_0,...,p_{n-1}, while the second state corresponds to selecting the then-branch in all p_0,...,p_{n-2} , and the else-branch in p_{n-1}. Finally, the last state corresponds to selecting the else-branch in all p_0,...,p_{n-1}.

2.4 Bottom-Up Compositional Interprocedural Analysis

Bottom-up compositional analyses avoid the explosion of calling context by computing for each procedure a summary which is independent of the input, and instantiating as a function of particular calling contexts. Unfortunately, it is hard to analyze a procedure independently of its calling contexts and at the same time compute a summary that is sound and precise enough. One of the reasons is that the abstract transfer functions may depend on the input abstract state, which is often unavailable for the compositional analysis. For example, in the program in Fig. 1, the abstract transformer for the assignment a_i = c_{i-1} in the points-to analysis is

$$[\![a_i = c_{i-1}]\!]^\sharp(d) = (d \backslash \{\langle a_i, z\rangle | z \in \mathsf{Var}\}) \cup \{\langle a_i, w\rangle | \langle c_{i-1}, w\rangle \in d\}\,.$$

Note that the rightmost set depends on the input abstract state d.

2.5 Modular Lattices for Compositional Interprocedural Analysis

This paper formulates a sufficient condition for performing compositional interprocedural analysis using lattices theory. Our condition requires that the abstract domain be a lattice with a so-called *modularity* property, and that the effects of primitive commands (such as assignments) on abstract elements be expressed by applying the $\sqcap$ and $\sqcup$ operations to the input states. If this condition is met, we can construct a bottom-up compositional analysis that summarizes each procedure independently of particular inputs.

DEFINITION 1. *Let $\mathcal{D}$ be a lattice. A pair of elements $\langle d_0, d_1\rangle$ is called* **modular***, denoted by $d_0 M d_1$, iff*

$$d \sqsubseteq d_1 \text{ implies that } (d \sqcup d_0) \sqcap d_1 = d \sqcup (d_0 \sqcap d_1)$$

An element d_1 is called **right-modular** *if $d_0 M d_1$ holds for all $d_0 \in \mathcal{D}$. $\mathcal{D}$ is called* **modular** *if $d_0 M d_1$ holds for all $d_0, d_1 \in \mathcal{D}$.*

Intuitively, a lattice is **modular** when it satisfies a restricted form of associativity between its $\sqcup$ and $\sqcap$ operations [13]. (Note, for example, that every distributive lattice is modular, but not all modular lattices are distributive.) In our application to the interprocedural analysis, the left-hand side of the equality in Def. 1 represents the top-down computation and the right-hand side corresponds to the bottom-up computation. Therefore, modularity ensures that the results coincide.

Our approach requires that transfer functions of primitive commands be defined by the combination of $- \sqcap d_0$ and $- \sqcup d_1$ for some constant abstract elements d_0 and d_1, independent of the input abstract state where d_0 elements are right-modular. Our encoding of points-to analysis described in Sec. 2.2 does not meet this requirement on transfer functions, because it does not use $\sqcup$ with a constant element to define the meaning of the statement $x = y$. In contrast, in connection analysis the transfer function of the statement $x = y$ is defined by

$$[\![x = y]\!]^\sharp = \lambda d.\, (d \sqcap S_x) \sqcup U_{xy}$$

where S_x, U_{xy} are fixed abstract elements and do not depend on the input abstract state d. In Sec. 4, we formally prove that the connection analysis satisfies both the modularity requirement and the requirement on the transfer functions.

We complete this informal description by illustrating how the two requirements lead to the coincidence between top-down and bottom-up analyses. Consider again the assignment $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp$, in the body of some procedure $\mathtt{p_i}$. Let $\{d_k\}_k$ denote abstract states at the entry of $\mathtt{p_i}$, and suppose there is some d such that

$$\forall k : \exists d'_k \sqsubseteq S_{\mathtt{a_i}} : d_k = d \sqcup d'_k.$$

The compositional approach first chooses the input state d, and computes $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d)$. This result is then adapted to any d_k by being joined with d'_k, whenever this procedure is invoked with the abstract state d_k. This adaptation of the bottom-up approach gives the same result as the top-down approach, which applies $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp$ on d_k directly, as shown below:

$$\begin{aligned}
[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d) \sqcup d'_k &= ((d \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}}) \sqcup d'_k \\
&= ((d \sqcap S_{\mathtt{a_i}}) \sqcup d'_k) \sqcup U_{\mathtt{a_i c_{i-1}}} \\
&= ((d \sqcup d'_k) \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}} \\
&= (d_k \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}} \\
&= [\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d_k).
\end{aligned}$$

The second equality uses the associativity and commutativity of the $\sqcup$ operator, and the third holds due to the modularity requirement.

3. Programming Language

Let PComm, G, L, and PName be sets of primitive commands, global variables, local variables, and procedure names, respectively. We use the following symbols to range over these sets:

$$a, b \in \mathsf{PComm}, \quad g \in \mathsf{G}, \quad x, y, z \in \mathsf{G} \cup \mathsf{L}, \quad p \in \mathsf{PName}.$$

We formalize our results for a simple imperative programming language with procedures:

$$\begin{aligned}
\textit{Commands}\ \ C &::= \mathtt{skip} \mid a \mid C; C \mid C + C \mid C^* \mid p() \\
\textit{Declarations}\ \ D &::= \mathsf{proc}\ p() = \{\mathsf{var}\ \vec{x}; C\} \\
\textit{Programs}\ \ P &::= \mathsf{var}\ \vec{g}; C \mid D; P
\end{aligned}$$

A program P in our language is a sequence of procedure declarations, followed by a sequence of declarations of global variables and a main command. Commands contain primitive commands $a \in \mathsf{PComm}$, left unspecified, sequential composition $C; C'$, nondeterministic choice $C + C'$, iteration C^*, and procedure calls $p()$. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands, conditionals and loops can be easily defined.

Declarations D give the definitions of procedures. A procedure is comprised of a sequence of local variables declarations $\vec{x}$ and a command, which we refer to as the procedure's *body*. Procedures do not take any parameters or return any values explicitly; values can instead be passed to and from procedures using global variables. To simplify presentation, we do not consider mutually recursive procedures in our language; direct recursion is allowed. We denote by C_{body_p} and L_p the body of procedure p and the set of its local variables, respectively.

We assume that L and G are fixed arbitrary finite sets. Also, we consider only well-defined programs where all the called procedures are defined.

Standard Semantics. The standard semantics propagates every caller's context to the callee's entry point and computes the effect of the procedure on each one of them. Formally,

$$[\![p()]\!]^\sharp(d) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d), d)$$

where C_{body_p} is the body of the procedure p, and

$$[\![\mathsf{entry}]\!]^\sharp : \mathcal{D} \to \mathcal{D} \quad \text{and} \quad [\![\mathsf{return}]\!]^\sharp : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$$

are the functions which represent, respectively, entering and returning from a procedure.

Relational Collecting Semantics. The semantics of our programming language tracks pairs of memory states $\langle \overline{\sigma}, \sigma' \rangle$ coming from some unspecified set Σ of memory states. $\overline{\sigma}$ is the *entry* memory state to the procedure of the executing command (or if we are executing the main command, the memory state at the start of the program execution), and σ' is the *current* memory state. We assume that we are given the meaning $[\![a]\!] : \Sigma \to 2^\Sigma$ of every primitive command, and lift it to sets of pairs $\rho \subseteq \mathcal{R} = 2^{\Sigma \times \Sigma}$ of memory states by applying it in a pointwise manner to the current states:

$$[\![c]\!](\rho) = \{\langle \overline{\sigma}, \sigma' \rangle \mid \langle \overline{\sigma}, \sigma \rangle \in \rho \wedge \sigma' \in [\![c]\!](\sigma)\}.$$

The meaning of composed commands is standard:

$$\begin{aligned}
[\![C_1 + C_2]\!](\rho) &= [\![C_1]\!](\rho) \cup [\![C_2]\!](\rho) \\
[\![C_1; C_2]\!](\rho) &= [\![C_2]\!]([\![C_1]\!](\rho)) \\
[\![C^*]\!](\rho) &= \mathsf{leastFix}\, \lambda\rho'.\, \rho \cup [\![C]\!](\rho').
\end{aligned}$$

The effect of procedure invocations is computed using the auxiliary functions entry, return, combine, and $\cdot|_G$, which we explain below.

$$[\![p()]\!](\rho_c) = [\![\mathsf{return}]\!]([\![C_{\mathsf{body}_p}]\!] \circ [\![\mathsf{entry}]\!](\rho_c), \rho_c), \quad \text{where}$$

$$[\![\mathsf{entry}]\!] \;:\; \mathcal{R} \to \mathcal{R}$$
$$[\![\mathsf{entry}]\!](\rho_c) = \{\langle \sigma_e, \sigma_e\rangle \mid \sigma_e = \sigma_c|_G \wedge \langle \overline{\sigma_c}, \sigma_c\rangle \in \rho_c\}$$

$$[\![\mathsf{return}]\!] \;:\; \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$
$$[\![\mathsf{return}]\!](\rho_x, \rho_c) = \{\mathsf{combine}(\overline{\sigma_c}, \sigma_c, \sigma_x|_G) \mid \langle \overline{\sigma_c}, \sigma_c\rangle \in \rho_c \wedge \langle \overline{\sigma_x}, \sigma_x\rangle \in \rho_x \wedge \sigma_c|_G = \sigma_e|_G\}$$

$$[\![\mathsf{combine}]\!] \;:\; \Sigma \times \Sigma \times \Sigma \to \mathcal{R} \text{ (assumed to be given)}$$
$$(\,\cdot\,|_G) \;:\; \Sigma \to \Sigma \text{ (assumed to be given)}$$

Function entry computes the relation ρ_e at the entry to the invoked procedure. It removes the information regarding the caller's local variables from the current states σ_c coming from the caller's relation at the call-site ρ_c using function $(\,\cdot\,|_G)$, which is assumed to be given. Note that in the computed relation, the entry state and the calling state of the callee are identical.

Function return computes relation ρ_r, which updates the caller's current state with the effect of the callee. The function computes triples $\langle \overline{\sigma_c}, \sigma_c, \sigma_x|_G\rangle$ out of relations $\langle \overline{\sigma_c}, \sigma_c\rangle$ and $\langle \overline{\sigma_x}, \sigma_x\rangle$ coming from the caller at the calls site and to the callee at the return site. return considers only relations where the global part of the caller's current state matches that of the callee's entry state. Note that at the triple, the middle state, σ_c, contains the values of the caller's local variables, which the callee cannot modify, and the last state, $\sigma_x|_G$, contains the updated state of the global parts of the memory state. Procedure combine combines these two kinds of information and generates the updated relation at the return site.

EXAMPLE 2. *For memory states $\langle s_g, s_l, h\rangle \in \Sigma$ comprised of environments s_g and s_l, giving values to global and local variables, respectively, and a heap h, $(\,\cdot\,|_G)$ and* combine *are defined as*

$$\langle s_g, s_l, h\rangle|_G = \langle s_g, \bot, h\rangle,$$
$$[\![\mathsf{combine}]\!](\langle \overline{s_g}, \overline{s_l}, \overline{h}\rangle, \langle s_g, s_l, h\rangle, \langle s'_g, \bot, h'\rangle) = (\langle \overline{s_g}, \overline{s_l}, \overline{h}\rangle, \langle s'_g, s_l, h'\rangle).$$

4. Intraprocedural Analysis using Modularity

In this section we show how the modularity properties of lattice elements can help in the analysis of programs without procedures. (Programs with procedures are handled in Sec. 5.) The main idea is to require that only meet and join operators are used to define the abstract semantics of primitive commands and that the argument of the meet is right-modular. We begin with the connection analysis example, and then describe the general case.

4.1 Intraprocedural Connection Analysis

Abstract Domain. The abstract domain consists of equivalence relations on the variables from $\mathsf{L} \cup \mathsf{G}$ and a minimal element $\bot$. Intuitively, variables belong to different partitions if they never point to connected heap objects (i.e., those that are not connected by any chain of pointers even when the directions of these pointers are ignored). For instance, if there is a program state occurring at a program point pt in which $x.f$ and y denote the same heap object, then it must be that x and y belong to the same equivalence class of the analysis result at pt. We denote by $\mathsf{Equiv}(\Upsilon)$ the set of equivalence relations over a set Υ. Every equivalence relation on Υ induces a unique partitioning of Υ into its equivalence classes and vice versa. Thus, we use these binary-relation and partition views of an equivalence relation interchangeably throughout this paper.

DEFINITION 3. *A* **partition lattice** *over a set Υ is a 6-tuple $\mathcal{D}_{\mathsf{part}}(\Upsilon) = \langle \mathsf{Equiv}(\Upsilon), \sqsubseteq, \bot_{\mathsf{part}}, \top_{\mathsf{part}}, \sqcup, \sqcap\rangle$.*

- *For any equivalence relations d_1, d_2 in $\mathsf{Equiv}(\Upsilon)$,*

$$d_1 \sqsubseteq d_2 \Leftrightarrow \forall v_1, v_2 \in \Upsilon, v_1 \overset{d_1}{\cong} v_2 \Rightarrow v_1 \overset{d_2}{\cong} v_2,$$

  where $v_1 \overset{d_i}{\cong} v_2$ means that v_1 and v_2 are related by d_i.
- *The minimal element $\bot_{\mathsf{part}} = \{\{a\} \mid a \in \Upsilon\}$ is the identity relation, relating each element only with itself.*

$$\begin{array}{lcccl}
[\![x = \texttt{null}]\!]^\sharp(d) & = & [\![y = \texttt{new}]\!]^\sharp(d) & = & d \sqcap S_{\widehat{x}} \\
[\![x = y]\!]^\sharp(d) & = & [\![x = y.f]\!]^\sharp(d) & = & (d \sqcap S_{\widehat{x}}) \sqcup U_{\widehat{x}\widehat{y}} \\
[\![x.f = y]\!]^\sharp(d) & = & d \sqcup U_{\widehat{x}\widehat{y}} & &
\end{array}$$

$$\text{where} \quad \begin{array}{lcl} S_{\widehat{x}} & = & \{\{\widehat{x}\}\} \cup \{\{z \mid z \in \Upsilon \setminus \{\widehat{x}\}\} \\ U_{\widehat{x}\widehat{y}} & = & \{\{\widehat{x}, \widehat{y}\}\} \cup \{\{z\} \mid z \in \Upsilon \setminus \{\widehat{x}, \widehat{y}\}\} \end{array}$$

Table 1. Abstract semantics of primitive commands in the connection analysis for $d \neq \bot$. $[\![a]\!]^\sharp(\bot) = \bot$ for any command a. $U_{\widehat{x}\widehat{y}}$ is used to merge the connection sets of $\widehat{x}$ and $\widehat{y}$. $S_{\widehat{x}}$ is used to separate $\widehat{x}$ from its current connection set. In Sec. 4.1, $\widehat{x}$ is x and $\widehat{y}$ is y. In Sec. 5, $\widehat{x}$ denotes x' and $\widehat{y}$ denotes y'.

- *The maximum element $\top_{\mathsf{part}} = \{\Upsilon\}$ is the complete relation, relating each element to every element in Υ. It defines the partition with only one equivalence class:*
- *The join is defined by $d_1 \sqcup d_2 = (d_1 \cup d_2)^+$, where we take the binary-relation view of equivalence relations d_1 and d_2 and $-^+$ is the transitive closure operation.*
- *The meet is defined by $d_1 \sqcap d_2 = d_1 \cap d_2$. Here again we take the binary-relation view of d_i's.*

For an element $x \in \Upsilon$, the **connection set** of x in $d \in \mathsf{Equiv}(\Upsilon)$, denoted $[x]$, is the equivalence class of x in d.

Throughout the paper, we refer to an extended partition domain $\mathcal{D} = \mathcal{D}_{\mathsf{part}} \cup \{\bot\}$, which is the result of adding a bottom element $\bot$ to the original partition lattice, where for every $d \in \mathcal{D}_{\mathsf{part}}$, $\bot \sqsubset d$.

Abstract Semantics. Table 1 shows the abstract semantics of primitive commands for the connection analysis.

Assigning `null` or a newly allocated object to a variable x separates x from its connection set. Therefore, the analysis takes the meet of the current abstract state with S_x — the partition with two connection sets $\{x\}$ and the rest of the variables.

The effect of the statement $x = y$ is to separate the variable x from its connection set and to add x to the connection set of y. This is realized by performing a meet with S_x, and then a join with U_{xy} — a partition with $\{x, y\}$ as a connection set and singleton connection sets for the rest of the variables.

The abstraction does not distinguish between the objects pointed to by y and $y.f$. Thus, following [11], we set x to be in the same connection set as y after the assignment $x = y.f$. As a result, the same abstract semantics is used for both $x = y.f$ and $x = y$.

The concrete semantics of $x.f = y$ redirects the f field of the object pointed to by x to the object pointed to by y. The abstract semantics treats this statement in a rather conservative way, performing "weak updates": We merge the connection sets of x and y by joining the current abstract state with U_{xy}.

4.2 Conditionally Compositional Intraprocedural Analysis

DEFINITION 4 (Conditionally Adaptable Functions). *Let $\mathcal{D}$ be a lattice. A function $f : \mathcal{D} \to \mathcal{D}$ is* ***conditionally adaptable*** *if it has the form $f = \lambda d.((d \sqcap d_p) \sqcup d_g)$ for some $d_p, d_g \in \mathcal{D}$ and the element d_p is right-modular. We refer to d_p as f's* ***meet element*** *and to d_g as f's* ***join element****.*

We focus on static analyses where the transfer function for every atomic command a is some conditionally adaptable function $[\![a]\!]^\sharp$. We denote the meet elements of $[\![a]\!]^\sharp$ by $P[\![a]\!]^\sharp$. For a command C, we denote by $P[\![C]\!]^\sharp$ the set of meet elements of primitive subcommands occurring in C.

LEMMA 5. *Let $\mathcal{D}$ be a lattice. Let C be a command which does not contain procedure calls. For every $d_1, d_2 \in \mathcal{D}$ if $d_2 \sqsubseteq d_p$ for every $d_p \in P[\![C]\!]^\sharp$, then $[\![C]\!]^\sharp(d_1 \sqcup d_2) = [\![C]\!]^\sharp(d_1) \sqcup d_2$.*

Lem. 5 can be used to justify compositional summary-based intraprocedural analyses in the following way: Take a command C and an abstract value d_2 such that the conditions of the lemma hold. Computing the abstract value $[\![C]\!]^\sharp(d_1 \sqcup d_2)$ can be done by computing $d = [\![C]\!]^\sharp(d_1)$, possibly caching (d_1, d) in a summary for C, and then adapting the result by joining d with d_2.[1]

LEMMA 6. *The transfer functions of primitive commands in the intraprocedural connection analysis are conditionally adaptable.*

In contrast, and perhaps counter-intuitively, our framework for the interprocedural analysis has non-conditional summaries, which do not have a proviso like $d_2 \sqsubseteq P[\![C]\!]^\sharp$. It achieves this by requiring certain properties of the abstract domain used to record procedures summaries, which we now describe.

5. Compositional Analysis using Modularity

In this section, we define an abstract framework for compositional interprocedural analysis using modularity and illustrate the framework using the connection analysis. To make the material more accessible, we formulate some of the definitions specifically for the connection analysis and defer the general definitions to [4].

The main message is that the meet elements of atomic commands are right-modular and greater than or equal to all the elements in a sublattice of the domain which is used to record the effect of the caller on the callee's entry state. This allows to summarize the effects of procedures in a bottom-up manner, and to get the coincidence between the results of the bottom-up and top-down analyses.

5.1 Partition Domains for Ternary Relations

We first generalize the abstract domain for the *intra*procedural connection analysis described in Sec. 4.1 to the *inter*procedural setting.

Recall that the return operation defined in Sec. 3 operates on triplets of states. For this reason, we use an abstract domain that allows representing ternary relations between program states. We now formulate this for the connection analysis. For every global variable $g \in \mathsf{G}$, $\overline{g}$ denotes the value of g at the entry to a procedure and g' denotes its current value. The analysis computes at every program point a relation between the objects pointed to by global variables at the entry to the procedure (represented by $\overline{\mathsf{G}}$) and the ones pointed to by global variables and local variables at the current state (represented by G' and L', respectively).

For technical reasons, described later, we also use the set $\dot{\mathsf{G}}$ to compute the effect of procedure calls. These sets are used to represent partitions over variables in the same way as in Sec. 4.1. Formally, we define $\mathcal{D} = \mathsf{Equiv}(\Upsilon) \cup \{\bot\}$ in the same way as in Def. 3 of Sec. 4.1 where $\Upsilon = \overline{\mathsf{G}} \cup \mathsf{G}' \cup \dot{\mathsf{G}} \cup \mathsf{L}'$ and

$$\mathsf{G}' = \{g' \mid g \in \mathsf{G}\} \qquad \overline{\mathsf{G}} = \{\bar{g} \mid g \in \mathsf{G}\}$$
$$\dot{\mathsf{G}} = \{\dot{g} \mid g \in \mathsf{G}\} \qquad \mathsf{L}' = \{x' \mid x \in \mathsf{L}\}$$

[1] Interestingly, the notion of condensation in [12] is similar to the implications of Lem. 5 (and to the frame rule in separation logic) in the sense that the join (or $*$ in separation logic) distributes over the transfer functions. However, [12] requires the distribution $S(a+b) = a + S(b)$ hold for every two elements a and b in the domain. Our requirements are less restrictive: In Lem. 5, we require such equality only for elements smaller than or equal to the meet elements of the transfer functions. This is important for handling the connection analysis in which condensation property does not hold. (In addition, the method of [12] is developed for domains for logical programs using completion and requires the refined domain to be compatible with the projection operator, which is specific to logic programs. and be finitely generated [12, Cor. 4.9].)

$$\begin{array}{rcl} R_X & = & \{\{x \mid x \in X\}\} \cup \{\{x\} \mid x \in \Upsilon \setminus X\} \\ \mathcal{D}_{\mathsf{in}} & = & \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}}}\} \\ \mathcal{D}_{\mathsf{out}} & = & \{d \in \mathcal{D} \mid d \sqsubseteq R_{\mathsf{G}'}\} \\ \mathcal{D}_{\mathsf{inout}} & = & \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}} \cup \mathsf{G}'}\} \\ \mathcal{D}_{\mathsf{inoutloc}} & = & \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}'}\} \end{array}$$

Table 2. Constant projection element R_X for an arbitrary set X and the sublattices of $\mathcal{D}$ used by the interprocedural relational analysis. R_X is the partition that contains a connection set for all the variables in X and singletons for all the variables in $\Upsilon \setminus X$. Each one of the sublattices represents connection relations in the current state between objects which were pointed to by local or global variables at different stages during the execution.

REMARK 1. *Formally, the interprocedural connection analysis computes an over-approximation of the relational concrete semantics defined in Sec. 3. A Galois connection between the $\mathcal{D}$ and a standard (concrete collecting relational) domain for heap manipulating programs is defined in [4].*

5.2 Triad Partition Domains

We first informally introduce the concept of *Triad Domain*. Triad domains are used to perform abstract interpretation to represent concrete domains and their concrete semantics as defined in Sec. 3. A triad domain $\mathcal{D}$ is a complete lattice which conservatively represents binary and ternary relations (hence the name "triad") between memory states arising at different program points such as the entry point to the caller procedure, the call-site, and the current program point. The analysis uses elements $d \in \mathcal{D}$ to represent ternary relations when computing procedure returns. For all other purposes binary relations are used. More specifically, the analysis makes special use of the *triad sublattices* of $\mathcal{D}$ defined in Table 2, which we now explain.

Each sublattice is used to abstract binary relations between sets of program states arising at different program points. We construct these sublattices by first choosing *projection elements* d_{proj_i} from the abstract domain $\mathcal{D}$, and then defining the sublattice $\mathcal{D}_i$ to be the closed interval $[\bot, d_{\mathsf{proj}_i}]$, which consists of all the elements between $\bot$ and d_{proj_i} according to the $\sqsubseteq$ order (including $\bot$ and d_{proj_i}). Moreover, for every $i \in \{\mathsf{in}, \mathsf{out}, \mathsf{inout}, \mathsf{inoutloc}\}$, we define the projection operation $(\cdot|_i)$ as follows: $d|_i = d \sqcap d_{\mathsf{proj}_i}$. Note that $d|_i$ is always in $\mathcal{D}_i$.

In the connection analysis, projection elements d_{proj_i} are defined in terms of R_X's in Table 2:

$$d_{\mathsf{proj}_{\mathsf{in}}} = R_{\overline{\mathsf{G}}}, d_{\mathsf{proj}_{\mathsf{out}}} = R_{\mathsf{G}'}, d_{\mathsf{proj}_{\mathsf{inout}}} = R_{\overline{\mathsf{G}} \cup \mathsf{G}'}, d_{\mathsf{proj}_{\mathsf{inoutloc}}} = R_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}'}.$$

R_X is the partition that contains a connection set containing all the variables in X and singleton sets for all the variables in $\Upsilon \setminus X$.

Each abstract state in the sublattice $\mathcal{D}_{\mathsf{out}}$ represents a partition on heap objects pointed to by global variables in the current state, such that two such heap objects are grouped together in this partition when they are *weakly connected*, i.e., we can reach from one object to the other by following pointers forward or backward. For example, suppose that a global variable g_1 points to an object o_1 and a global variable g_2 points to an object o_2 at a program point pt, and that o_1 and o_2 are weakly connected. Then, the analysis result will be an equivalence relation that puts g_1' and g_2' in the same equivalence class.

Each abstract state in $\mathcal{D}_{\mathsf{in}}$ represents a partition of objects pointed to by global variables upon the procedure entry where the partition is done according to weakly-connected components.

The sublattice $\mathcal{D}_{\mathsf{inout}}$ is used to abstract relations in the current heap between objects pointed to by global variables upon procedure entry and those pointed to by global variables in the current program point. For example, if at point pt in a procedure p an ob-

$$\iota_{\mathsf{entry}} = \bigsqcup_{g \in \mathsf{G}} U_{g'\overline{g}} = \{\{g', \overline{g}\}, \{\dot{g}\} \mid g \in \mathsf{G}\}$$
$$[\![\mathsf{entry}]\!]^\sharp(d) = (d \sqcap R_{\mathsf{G}'}) \sqcup \iota_{\mathsf{entry}}$$
$$[\![\mathsf{return}]\!]^\sharp(d_{\mathsf{exit}}, d_{\mathsf{call}}) = (f_{\mathsf{call}}(d_{\mathsf{call}}) \sqcup f_{\mathsf{exit}}(d_{\mathsf{exit}} \sqcap R_{\overline{\mathsf{G}} \cup \mathsf{G}'})) \sqcap R_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}}$$
$$[\![p()]\!]^\sharp(d) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d), d)$$
$$[\![p()]\!]^\sharp{}_{\mathsf{BU}}(d) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d)$$

Table 3. The definition of ι_{entry} and the interprocedural abstract semantics for the top-down and bottom-up connection analyses. ι_{entry} is the element that represents the identity relation between input and output, and C_{body_p} is the body of procedure p.

ject is currently pointed to by a global variable g_1 and it belongs to the same weakly connected component as an object that was pointed to by a global variable g_2 at the entry point of p, then the partition at pt will include a connection set with $\overline{g_1}$ and g_2'.

Similarly, the sublattice $\mathcal{D}_{\mathsf{inoutloc}}$ is used to abstract relations in the current heap between objects pointed to by global variables upon procedure entry and global and local variables in the current program point.

5.3 Interprocedural Top Down Triad Connection Analysis

We describe here the abstract semantics for the top-down interprocedural connection analysis. The intraprocedural semantics is shown in Table 1. Notice that there is a minor difference between the semantics of primitive commands for the intraprocedural connection analysis defined in Sec. 4.1 and for the analysis in this section. In the analysis without procedures we use x, whereas in the analysis of this section we use x'.

The abstract meaning of procedure calls in the connection analysis is defined in Table 3. Again, we refer to the auxiliary constant elements R_X for a set X defined in Table 2.

When a procedure is entered, local variables of the procedure and all the global variables $\overline{g}$ at the entry to the procedure are initialized to `null`. This is realized by applying the meet operation with auxiliary variable $R_{\mathsf{G}'}$. Then, each of the $\overline{g}$ is initialized with the current variable value g' using ι_{entry}. The ι_{entry} element denotes a particular state that abstracts the identity relation between input and output states. In the connection analysis, it is defined by a partition containing $\{\overline{g}, g'\}$ connection sets for all global variables g. Intuitively, this stores the current value of variable g into $\overline{g}$, by representing the case where the object currently pointed to by g is in the same weakly connected component as the object that was pointed to by g at the entry point of the procedure.

The effect of returning from a procedure is more complex. It takes two inputs: d_{call}, which represents the partition at the call-site, and d_{exit}, which represents the partition at the exit from the procedure. The meet operation of d_{exit} with $R_{\overline{\mathsf{G}} \cup \mathsf{G}'}$ emulates the nullification of local variables of the procedure. The computed abstract values emulate the composition of the input-output relation of the call-site with that of the return-site. Variables of the form $\dot{g}$ are used to implement a natural join operation for composing these relations. $f_{\mathsf{call}}(d_{\mathsf{call}})$ renames global variables from g' to $\dot{g}$ and $f_{\mathsf{exit}}(d_{\mathsf{exit}})$ renames global variables from $\overline{g}$ to $\dot{g}$ to allow natural join. Intuitively, the old values $\overline{g}$ of the callee at the exit-site are matched with the current values g' of the caller at the call-site. The last meet operation represents the nullification of the temporary values $\dot{g}$ of the global variables.

In [4] we generalize these definitions to generic *triad analyses*.

5.4 Bottom Up Triad Connection Analysis

In this section, we introduce a bottom-up semantics for the connection analysis. Primitive commands are interpreted in the same way as in the top-down analysis. The effect of procedure calls is computed using the function $[\![p()]\!]^\sharp{}_{\mathsf{BU}}(d)$, defined in Table 3, instead of $[\![p()]\!]^\sharp(d)$. The two functions differ in the first argument they use when applying $[\![\mathsf{return}]\!]^\sharp$: $[\![p()]\!]^\sharp{}_{\mathsf{BU}}(d)$ uses a *constant* value, which is the abstract state at the procedure exit computed when analyzing $p()$ with ι_{entry}. In contrast, $[\![p()]\!]^\sharp(d)$ uses the abstract state resulting at the procedure exit when analyzing the call to $p()$ with d.

5.5 Coincidence Result in Connection Analysis

We are interested in finding a sufficient condition on an analysis, for the following equality to hold:

$$\forall d \in \mathcal{D}.\, [\![p()]\!]^\sharp{}_{\mathsf{BU}}(d) = [\![p()]\!]^\sharp(d)\,.$$

We sketch the main arguments of the proof, substantiating their validity using examples from the interprocedural connection analysis in lieu of more formal mathematical arguments, given in [4].

5.5.1 Uniform Representation of Entry Abstract States

Any abstract state d arising at the entry to a procedure in the top-down analysis is **uniform**, i.e., it is a partition such that for every global variable g, variables $\overline{g}$ and g' are always in the same connection set. This is a result of the definition of function `entry`, which projects the abstract element at the call-site into the sublattice $\mathcal{D}_{\mathsf{out}}$ and the successive join with the ι_{entry} element. The projection results in an abstract state where all connection sets containing more than a single element are comprised only of primed variables. Then, after joining $d|_{\mathsf{out}}$ with ι_{entry}, each old variable $\overline{g}$ resides in the same partition as its corresponding current primed variable g'.

We point out that the uniformity of the entry states is due to the property of ι_{entry} that its connection sets are comprised of pairs of variables of the form $\{x', \overline{x}\}$. One important implication of this uniformity is that every entry abstract state d_0 to any procedure has a dual representation. In one representation, d is the join of ι_{entry} with some elements $U_{x'y'} \in \mathcal{D}_{\mathsf{out}}$. In the other representation, d is expressed as the join of ι_{entry} with some elements $U_{\overline{x}\overline{y}} \in \mathcal{D}_{\mathsf{in}}$. In the following, we use the function o that replaces relationships among current variables by those among old ones: $o(U_{x'y'}) = U_{\overline{x}\overline{y}}$; and $o(d)$ is the least upper bounds of ι_{entry} and elements $U_{\overline{x}\overline{y}}$ for all x, y such that x' and y' are in the same connection set of d.

5.5.2 Delayed Evaluation of the Effect of Calling Contexts

Elements of the form $U_{\overline{x}\overline{y}}$, coming from $\mathcal{D}_{\mathsf{in}}$, are smaller than or equal to the meet elements of intraprocedural statements. In Lem. 6 of Sec. 4 we proved that the semantics of the connection analysis is conditionally adaptable. Thus, computing the composed effect of any sequence τ of intraprocedural transformers on an entry state of the form $d_0 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$ results in an element of the form $d_0' \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$, where d_0' results from applying the transformers in τ on d_0. Using the observation we made in Sec. 5.5.1, this means that we can represent any abstract element d resulting at a call-site as $d = d_1 \sqcup d_2$, where d_1 is the effect of τ on ι_{entry} and $d_2 \in \mathcal{D}_{\mathsf{in}}$ is a join of elements of the form $U_{\overline{x}\overline{y}} \in \mathcal{D}_{\mathsf{in}}$:

$$d = d_1 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}\,. \tag{1}$$

5.5.3 Counterpart Representation for Calling Contexts

Because of the previous reasoning, we can now assume that any abstract value at the call-site to a procedure $p()$ is of the form $d_1 \sqcup d_3$, where $d_3 \in \mathcal{D}_{\mathsf{in}}$ and it is a join of elements of form $U_{\overline{x}\overline{y}}$.

For each $U_{\overline{x}\overline{y}}$, the entry state resulting from analyzing $p()$ when the calling context is $d_1 \sqcup U_{\overline{x}\overline{y}}$ is either identical to the one resulting from d_1 or can be obtained from d_1 by merging two of its connection sets. Furthermore, the need to merge occurs only if there are variables w' and z' such that w' and $\overline{x}$ are in one of the connection sets of d_1 and z' and $\overline{y}$ are in another. This means that the effect of

$U_{\overline{x}\overline{y}}$ on the entry state can be expressed via primed variables:

$$d_1 \sqcup U_{\overline{x}\overline{y}} = d_1 \sqcup U_{w'z'} \,.$$

This implies that if the abstract state at the call-site is $d_1 \sqcup d_3$, then there is an element $d'_3 \in \mathcal{D}_{\mathsf{out}}$ such that

$$(d_1 \sqcup d_3)|_{\mathsf{out}} = d_1|_{\mathsf{out}} \sqcup d'_3 \tag{2}$$

We refer to the element $d'_3 \in \mathcal{D}_{\mathsf{out}}$, which can be used to represent the effect of $d_3 \in \mathcal{D}_{\mathsf{in}}$ at the call-site as d_3's *counterpart*, and denote it by $\widehat{d_3}$.

5.5.4 Representing Entry States with Counterparts

The above facts imply that we can represent an abstract state d at the call-site as

$$d = d_1 \sqcup d_3 \sqcup d_4, \tag{3}$$

where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$. d_3 is a join of the elements of the form $U_{\overline{x}\overline{y}}$ such that $\overline{x}$ and $\overline{y}$ reside in d_1 in different partitions, which also contain current (primed) variables, and thus possibly affect the entry state; d_4 is a join of all the other elements $U_{\overline{x}\overline{y}} \in \mathcal{D}_{\mathsf{in}}$, which are needed to represent d in this form, but either $\bar{x}$ or $\bar{y}$ resides in the same partition in d_1 or one of them is in a partition containing only old variables. As explained in the previous paragraph, there is an element $d'_3 = \widehat{d_3}$ that joins elements of the form $U_{x'y'}$ such that

$$(d_1 \sqcup d_3)|_{\mathsf{out}} = (d_1 \sqcup d'_3)|_{\mathsf{out}} \tag{4}$$

and

$$d = d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d'_3 \sqcup d_4 \,. \tag{5}$$

Thus, after applying the entry's semantics, we get that abstract states at the entry point of procedure are always of the form

$$[\![\mathsf{entry}]\!]^\sharp(d) = (d_1 \sqcup d'_3)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} \tag{6}$$

where d'_3 represents the effect of $d_3 \sqcup d_4$ on partitions containing current variables g' in d_1. Because $U_{x'y'} \sqsubseteq R_{G'}$ and d'_3 joins elements of form $U_{x'y'}$, the *modularity of the lattice* gives that

$$(d_1 \sqcup d'_3)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} = (d_1|_{\mathsf{out}} \sqcup d'_3) \sqcup \iota_{\mathsf{entry}}$$

This implies that every state d_0 at an entry point to a procedure is of the following form:

$$d_0 = \iota_{\mathsf{entry}} \sqcup \underbrace{(U_{x'y'} \ldots \sqcup U_{x'_l y'_l})}_{d_1|_{\mathsf{out}}} \sqcup \underbrace{(U_{x'_{l+1} y'_{l+1}} \ldots \sqcup U_{x'_n y'_n})}_{d'_3} \,.$$

Using the dual representation of entry state, we get that

$$\iota_{\mathsf{entry}} \sqcup U_{x'_1 y'_1} \ldots \sqcup U_{x'_n y'_n} = \iota_{\mathsf{entry}} \sqcup o(U_{x'_1 y'_1} \ldots \sqcup U_{x'_n y'_n})$$

and thus the form of a state d_0 at an entry point to a procedure is

$$d_0 = \iota_{\mathsf{entry}} \sqcup U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n} \tag{7}$$

5.5.5 Putting It All Together

We now show that the interprocedural connection analysis can be done compositionally. Intuitively, the effect of the caller's calling context can be carried over procedure invocations. Alternatively, the effect of the callee on the caller's context can be adapted unconditionally for different caller's calling contexts.

We sketch here an outline of the proof for case $C = p()$ using the connection analysis domain. The proof goes by induction on the structure of the program. In Eq.3 we showed that every abstract value that arises at the call-site is of the form $d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$. Thus, we show that

$$[\![C]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![C]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4 \,. \tag{8}$$

Say we want to compute the effect of invoking $p()$ on abstract state d according to the top-down abstract semantics.

$$[\![p()]\!]^\sharp(d) \quad = \quad [\![\mathsf{return}]\!]^\sharp\big(\big(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d)\big), d\big)$$

First, let's compute the first argument to $[\![\mathsf{return}]\!]^\sharp$.

$$\begin{aligned}
&([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4)) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3 \sqcup d_4)|_{\mathsf{out}}) \sqcup \iota_{\mathsf{entry}}) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3)|_{\mathsf{out}}) \sqcup \iota_{\mathsf{entry}}) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup d'_3 \sqcup \iota_{\mathsf{entry}}) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup o(d'_3) \sqcup \iota_{\mathsf{entry}}) \\
&\quad = \quad [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)
\end{aligned} \tag{9}$$

The first equalities are mere substitutions based on observations we made before. The last one comes from the induction assumption.

When applying the return semantics, we first compute the natural join and then remove the temporary variables. Hence, we get

$$(f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)))|_{\mathsf{inoutloc}}$$

Let's first compute the result of the inner parentheses.

$$\begin{aligned}
&f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\
&\quad = \quad f_{\mathsf{call}}(d_1 \sqcup d'_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\
&\quad = \quad f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup \\
&\qquad f_{\mathsf{exit}}(o(d'_3)) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}))
\end{aligned} \tag{10}$$

The first equality is by the definition of d'_3 and the last equality is by the isomorphism of the renaming operations f_{call} and f_{exit}.

Note, among the join arguments, $f_{\mathsf{exit}}(o(d'_3))$ and $f_{\mathsf{call}}(d'_3)$. Let's look at the first element. $o(d'_3)$ replaces all the occurrences of $U_{x'y'}$ in d'_3 with $U_{\overline{x}\overline{y}}$. f_{exit} replaces all the occurrences of $U_{\overline{x}\overline{y}}$ in $o(d'_3)$ with $U_{\dot{x}\dot{y}}$. Thus, the first element is

$$U_{\dot{x}_1 \dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n \dot{y}_n}$$

which is the result of replacing in d'_3 all the occurrences of $U_{x'y'}$ with $U_{\dot{x}\dot{y}}$. Consider the second element. f_{exit} replaces all occurrences of $U_{x'y'}$ in d'_3 with $U_{\dot{x}\dot{y}}$. Thus, also the second element is

$$U_{\dot{x}_1 \dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n \dot{y}_n}$$

Thus, we get that

$$(10) = f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}))$$

Moreover, f_{call} is isomorphic and by Eq.5

$$= f_{\mathsf{call}}(d_3 \sqcup d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}))$$

Remember (Eq.1) that d_3 and d_4 are both of form

$$U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}$$

and that $f_{\mathsf{call}}(d)$ only replaces g' occurrences in d; thus

$$f_{\mathsf{call}}(U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}) = U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}$$

Finally, we get

$$\begin{aligned}
&= f_{\mathsf{call}}(d_1) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup (d_3 \sqcup d_4) \\
&= [\![p()]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4
\end{aligned}$$

5.5.6 Precision Coincidence

We combine the observations we made to informally show the coincidence result between the top-down and the bottom-up semantics. According to Eq.3, every state d at a call-site can be represented as

$d = d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$

$$\begin{aligned}[\![p()]\!]^\sharp(d) &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d)), d) \\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} \sqcup d_3'), d) \\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}} \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3')), d) \quad (11)\end{aligned}$$

The second equivalence is by Eq.6, and the second equivalence is because $d_1|_{\mathsf{out}}, d_3' \in \mathcal{D}_{\mathsf{out}}$ and

$$\iota_{\mathsf{entry}} \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3') = \iota_{\mathsf{entry}} \sqcup d_1|_{\mathsf{out}} \sqcup d_3'$$

We showed that for every $d = d_1 \sqcup d_3 \sqcup d_4$, such that $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$,

$$[\![C]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![C]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4$$

for any command C. Therefore, since $o(d_3'), o(d_1|_{\mathsf{out}}) \in \mathcal{D}_{\mathsf{in}}$,

$$(11) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}) \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3'), d_1 \sqcup d_3 \sqcup d_4)\,.$$

By Eq.5, $d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d_3' \sqcup d_4$ Thus, we can remove $o(d_3')$ because $f_{\mathsf{exit}}(o(d_3'))$ will be redundant in the natural join of the $[\![\mathsf{return}]\!]^\sharp$ operator. Using a similar reasoning, we can remove $f_{\mathsf{exit}}(o(d_1|_{\mathsf{out}}))$, since $f_{\mathsf{call}}(d_1|_{\mathsf{out}}) \sqsubseteq f_{\mathsf{call}}(d_1)$. Hence, finally,

$$(11) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d_1 \sqcup d_3 \sqcup d_4) = [\![p()]\!]^\sharp_{\mathsf{BU}}(d)\,.$$

6. Experimental evaluation

In this section, we evaluate the effectiveness of our approach in practice using the connection analysis for concreteness. We implemented three versions of this analysis: the original top-down version from [11], our modified top-down version, and our modular bottom-up version that coincides in precision with the modified top-down version. We next briefly describe these three versions.

The original top-down connection analysis does not meet the requirements described in Sec. 5, because the abstract transformer for destructive update statements $x.f = y$ depends on the abstract state; the connection sets of x and y are not merged if x or y points to null in all the executions leading to this statement. We therefore conservatively modified the analysis to satisfy our requirements, by changing the abstract transformer to always merge x's and y's connection sets. Our bottom-up modular analysis that coincides with this modified top-down analysis operates in two phases. The first phase computes a summary for every procedure by analyzing it with an input state ι_{entry}. The summary over-approximates relations between all possible inputs of this procedure and each program point in the body of the procedure. The second phase is a chaotic iteration algorithm which propagates values from callers to callees using the precomputed summaries, and is similar to the second phase of the interprocedural functional algorithm of [18, Figure 7].

We implemented all three versions of connection analysis described above using Chord [16] and applied them to five Java benchmark programs whose characteristics are summarized in Table 4. They include two programs (grande2 and grande3) from the Java Grande benchmark suite and two (antlr and bloat) from the DaCapo benchmark suite. We excluded programs from these suites that use multi-threading, since our analyses are sequential. Our larger three benchmark programs are commonly used in evaluating pointer analyses. All our experiments were performed using Oracle HotSpot JRE 1.6.0 on a Linux machine with Intel Xeon 2.13 GHz processors and 128 Gb RAM.

We next compare the top-down and bottom-up approaches in terms of precision (Sec. 6.1) and scalability (Sec. 6.2). We omit the modified top-down version of connection analysis from further evaluation, as we found its performance difference from the original top-down version to be negligible, and since its precision is identical to our bottom-up version (in principle, due to our coincidence result, as well as confirmed in our experiments).

6.1 Precision

We measure the precision of connection analysis by the size of the connection sets of pointer variables at program points of interest. Each such pair of variable and program point can be viewed as a separate *query* to the connection analysis. To obtain such queries, we chose the parallelism client proposed in the original work on connection analysis [11], which demands the connection set of each dereferenced pointer variable in the program. In Java, this corresponds to variables of reference type that are dereferenced to access instance fields or array elements. More specifically, our queries constitute the base variable in each occurrence of a getfield, putfield, aload, or astore bytecode instruction in the program. The number of such queries for our five benchmarks are shown in the "# of queries" column of Table 5. To avoid counting the same set of queries across benchmarks, we only consider queries in application code, ignoring those in JDK library code. This number of queries ranges from around 0.6K to over 10K for our benchmarks.

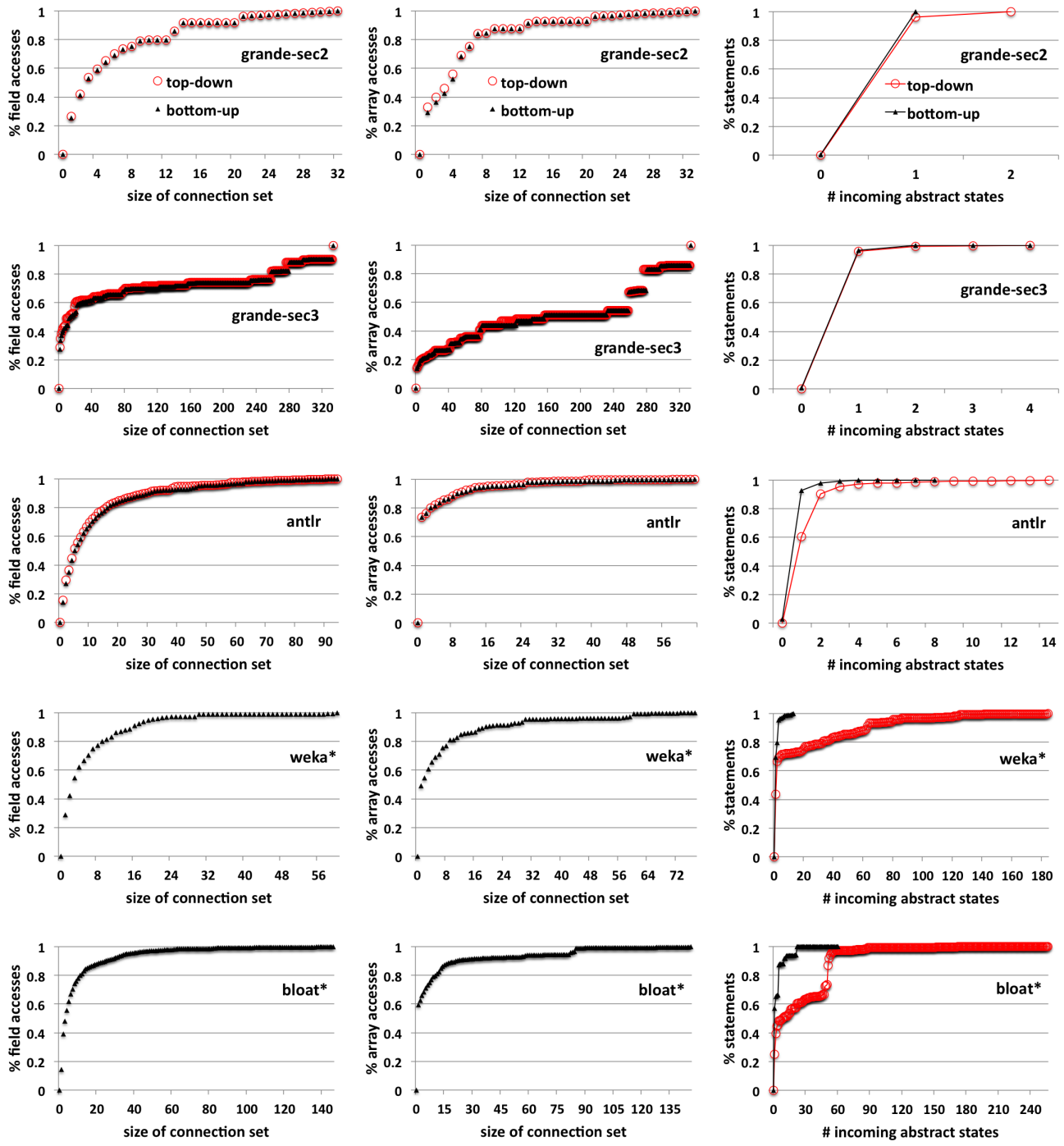
A precise answer to a query $x.f$ (a field access) or *x[i]* (an array access) is one that is able to disambiguate the object pointed to by x from objects pointed to by another variable y. In the connection analysis abstraction, x and y are disambiguated if they are not connected. We thereby measure the precision of connection analysis in terms of the size of the connection set of variable x, where a more precise abstraction is one where the number of other variables connected to x is small. To avoid gratuitously inflating this size, we perform intra-procedural copy propagation on the intermediate representation of the benchmarks in Chord.

Fig. 4 provides a detailed comparison of precision, based on the above metric, of the top-down and bottom-up versions of connection analysis, separately for field access queries (column (a)) and array access queries (column (b)). Each graph in columns (a) and (b) plots, for each distinct connection set size (on the X axis), the fraction of queries (on the Y axis) for which each analysis computed connection sets of equal or smaller size. Graphs marked (*) indicate where the sizes of connection sets computed by the top-down analysis are not plotted because the analysis timed out after six hours. This happens for our two largest benchmarks (weka and bloat). The graphs for the remaining three benchmarks (grande2, grande3, and antlr) show that the precision of our modular bottom-up analysis closely tracks that of the original top-down analysis: the points for the bottom-up and top-down analyses, denoted ▲ and ○, respectively, overlap almost perfectly in each of the six graphs. The ratio of the connection set size computed by the top-down analysis to that computed by the bottom-up analysis on average across all queries is 0.977 for grande2, 0.977 for grande3, and 0.952 for antlr. While we do not measure the impact of this precision loss of 2-5% on a real client, we note that for our largest two benchmarks, the top-down analysis does not produce any useful result.

6.2 Scalability

Table 5 compares the scalability of the top-down and bottom-up analyses in terms of three different metrics: running time, memory consumption, and the total number of computed abstract states. As noted earlier, the bottom-up analysis runs in two phases: a summary computation phase followed by a summary instantiation phase. The above data for these phases is reported in separate columns of the table. On our largest benchmark (bloat), the bottom-up analysis takes around 50 minutes and 873 Mb memory, whereas the top-down analysis times out after six hours, not only on this benchmark but also on the second largest one (weka).

The "# of abstract states" columns provide the sum of the sizes of the computed abstractions in terms of the number of abstract states, including only incoming states at program points of queries (in the "queries" sub-column), and incoming states at all program points, including the JDK library (in the "total" sub-column). Col-



(a) Precision comparison for field accesses. (b) Precision comparison for array accesses. (c) Scalability comparison.

Figure 4. Comparison of the precision and scalability of the original top-down and our modular bottom-up versions of connection analysis. Each graph in columns (a) and (b) shows, for each distinct connection set size (on the X axis), the fraction of queries (on the Y axis) for which the analyses computed connection sets of equal or smaller size. This data is missing for the top-down analysis in the graphs marked (*) because this analysis timed out after six hours on those benchmarks. For the remaining benchmarks, the near perfect overlap in the points plotted for the two analyses indicates very minor loss in precision of the bottom-up analysis over the top-down analysis. Column (c) compares scalability of the two analyses in terms of the total number of abstract states computed by them. Each graph in this column shows, for each distinct number of incoming abstract states computed at each program point (on the X axis), the fraction of program points (on the Y axis) with equal or smaller number of such states. The numbers for the top-down analysis in the graphs marked (*) were obtained at the instant of timeout. These graphs clearly show the blow-up in the number of states computed by the top-down analysis over the bottom-up analysis.

	description	# classes		# methods		# bytecodes	
		app only	total	app only	total	app only	total
grande2	Java Grande kernels	17	61	112	237	8,146	13,724
grande3	Java Grande large-scale applications	42	241	231	1,162	27,812	75,139
antlr	Parser and translator generator	116	358	1,167	2,400	128,684	186,377
weka	Machine-learning library for data-mining tasks	62	530	575	3,391	40,767	223,291
bloat	Java bytecode optimization and analysis tool	277	611	2,651	4,699	194,725	311,727

Table 4. Benchmark characteristics. The "# of classes" column is the number of classes containing reachable methods. The "# of methods" column is the number of reachable methods computed by a static 0-CFA call-graph analysis. The "# of bytecodes" column is the number of bytecodes of reachable methods. The "total" columns report numbers for *all* reachable code, whereas the "app only" columns report numbers for only application code (excluding JDK library code).

	# of queries	Bottom-Up analysis						Top-Down analysis			
		summary computation		summary instantiation							
		time	memory	time	memory	# of abstract states		time	memory	# of abstract states	
						queries	total			queries	total
grande2	616	0.6 sec	78 Mb	0.9 sec	61 Mb	616	1,318	1 sec	37 Mb	616	3,959
grande3	4,236	43 sec	224 Mb	1:21 min	137 Mb	4,373	8,258	1:11 min	506 Mb	4,354	27,232
antlr	5,838	16 sec	339 Mb	30 sec	149 Mb	6,207	21,437	1:23 min	1.1 Gb	8,388	79,710
weka	2,205	46 sec	503 Mb	2:48 min	228 Mb	2,523	25,147	$>$ 6 hrs	26 Gb	5,694	688,957
bloat	10,237	3:03 min	573 Mb	30 min	704 Mb	36,779	131,665	$>$ 6 hrs	24 Gb	139,551	962,376

Table 5. The number of queries to connection analysis and three metrics comparing the scalability of the original top-down and our modular bottom-up versions of the analysis on those queries: running time, memory consumption, and number of incoming abstract states computed at program points of interest. These points include only query points in the "query" sub-columns and all points in the "total" sub-columns. All three metrics show that the top-down analysis scales much more poorly than the bottom-up analysis.

umn (c) of Fig. 4 provides more detailed measurements of the latter numbers. The graphs there show, for each distinct number of incoming states computed at each program point (on the X axis), the fraction of program points (on the Y axis) with equal or smaller number of incoming states. The numbers for the top-down analysis in the graphs marked (*) were obtained at the instant of timeout. The graphs clearly show the blow-up in the number of states computed by the top-down analysis over the bottom-up analysis.

7. Conclusions

We show using lattice theory that when an abstract domain has enough right-modular elements to allow transfer functions to be expressed as joins and meets with constant elements—and the elements used in the meet are right-modular—a compositional (bottom-up) interprocedural analysis can be as precise as a top-down analysis. Using the above, we developed a new bottom-up interprocedural algorithm for connection pointer analysis of Java programs. Our experiments indicate that, in practice, our algorithm is nearly as precise as the existing algorithm, while scaling significantly better. In [4] we apply the same technique to derive a new bottom-up analysis for a variant of the copy-constant propagation problem [10]. The algorithm utilizes a sophisticated join to compute the effect of copy statements of the form `x:=y`. Notice that this is not simple under our restrictions since constant values of y are propagated into x. Indeed, we found that designing the right join operator is the key step when using our approach.

Acknowledgments. Noam Rinetzky was supported by the EU project ADVENT, grant number: 308830.

References

[1] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103, 2001.

[2] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *ACM SIGPLAN SOAP Workshop*, pages 3–8, 2012.

[3] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[4] G. Castelnuovo. Modular lattices for compositional interprocedural analysis. Master's thesis, School of Computer Science, Tel Aviv University, 2012.

[5] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland, 1978.

[7] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

[8] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–5–77, 2011.

[9] J. Dolby, S. Fink, and M. Sridharan. T. J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`, 2006.

[10] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[11] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *IJPP*, 24(6):547–578, 1996.

[12] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract domains condensing. *ACM Trans. Comput. Log*, 6(1):33–60, 2005.

[13] G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, 1978.

[14] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis using lisf. *ACM TOPLAS*, 33(5):17, 2011.

[15] R. Madhavan, G. Ramalingam, and K. Vaswani. Purity analysis: An abstract interpretation formulation. In *SAS*, pages 7–24, 2011.

[16] M. Naik. Chord: A program analysis platform for Java. Available at `http://pag.gatech.edu/chord/`, 2006.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[18] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167, 1996.

[19] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data

flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. 1981.

[21] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

8. Compositional Constant Propagation Analysis

In this section we describe an encoding of a bottom-up interprocedural copy constant propagation analysis as a triad analysis.

8.1 Programming Language

We define a simple programming language which manipulates integer variables. The language is defined according to the requirements of our general framework. (See Sec. 3.) In this section, we assume that programs have only global integer variables $g \in G$ which are initialized to 0. We also assume that the primitive commands $a \in \mathsf{PComm}$ are of the form

$$\texttt{x := c} \quad , \quad \texttt{x := y} \quad , \text{and} \quad \texttt{x := *},$$

pertaining to assignments to a variable `x` of a constant value `c`, of the value of a variable `y`, or of an unknown value, respectively. We denote by $\mathsf{K}_P \subset_{fin} \mathcal{N}$ the finite set of constants which appear in a program P. We assume K_P contains 0. We denote by $\mathsf{G}_P \subset_{fin} G$ the finite set of global variables which appear in a program P.

In the following, we assume a fixed arbitrary program P and denote by $\mathsf{K} = \mathsf{K}_P$ the (fixed finite) set of constants that appear in P, and by $\mathsf{G} = \mathsf{G}_P$ the (fixed finite) set of global variables of P.

For technical reasons, explained in Sec. 8.5, we assume that the analyzed program contains a special global variable `t` which is not used directly by the program, but is used only to implement copy assignments of the form `x := y` using the following sequence of assignments

```
t:=y; y:=0; x:=0; y:=t; x:=t; t:=0.
```

(Note that, in particular, we assume that there are no statements of the form `x := x`.)

8.2 Concrete Semantics

8.2.1 Standard Intraprocedural Concrete Semantics

A *standard memory state* $s \in \mathcal{S} = \mathsf{G} \mapsto \mathcal{N}$ maps variables to their integer values. The meaning of primitive commands $a \in \mathsf{AComm}$ is standard, and defined below.

$$\begin{array}{lcl} [\![x := c]\!](s) & = & \{(s[x \mapsto [\![c]\!]])\} \\ [\![x := y]\!](s) & = & \{(s[x \mapsto s(y)])\} \\ [\![x := *]\!](s) & = & \{(s[x \mapsto n]) \mid n \in \mathcal{N}\} \end{array}$$

Note that $[\![a]\!] : \mathcal{S} \to 2^{\mathcal{S}}$ for a primitive a.

8.2.2 Relational Concrete Semantics

An *input-output pair of standard memory states* $r = (s, s') \in \mathcal{R} = \mathcal{S} \times \mathcal{S}$ records the values of variables at the entry to the procedure (s) and at the current state (s'). The meaning of intraprocedural statements in lifted to input-output pairs as described in Sec. 3. The interprocedural semantics is defined, as described in Sec. 3, using the functions $\cdot|_G : \mathcal{S} \to \mathcal{S}$ and $[\![\mathsf{combine}]\!] : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{R}$, whose meaning is defined below:

$$\begin{array}{lcl} s|_G & = & s \\ [\![\mathsf{combine}]\!](s_1, s_2, s_3) & = & (s_1, s_3) \end{array}$$

Informally, the projection of the state on its global part does not modify the state due to our assumption a state is a mapping of *global* variables to values. For a similar reason, the combination of the caller's input-output pair at the call-site with that of the callee at the exit-site results in a pair of memory states where the first one records the memory at the entry-site of the caller and the second one records the state at the exit-site of the callee.

8.3 Abstract Semantics

Notations. For every global variable $g \in \mathsf{G}$, $\overline{g}$ denotes the value of g at the entry to a procedure and g' denotes its current value. Similarly to the connection analysis, we use an additional set $\dot{\mathsf{G}}$ of variables to compute the effect of procedure calls. We denote by $\upsilon \in \Upsilon = \mathsf{G}' \cup \overline{G} \cup \dot{\mathsf{G}}$ the set of all *annotated variables* which are ranged by a meta variable υ.

We denote by $\zeta \in \mathsf{VAL} = \Upsilon \cup \mathsf{K} \cup \{*\}$ the set of *abstract values* ranged over by ζ. VAL is comprised of annotated variables, constants which appear in the program, and the special value $*$.

8.3.1 Abstract Domain

Let $\mathcal{D}_{map}$ be the set of all maps from variables $\upsilon \in \Upsilon$ to 2^{VAL}

$$\mathcal{D}_{map} = \Upsilon \mapsto 2^{\mathsf{VAL}}.$$

We denote the set $\mathcal{D}_{trans} \subseteq \mathcal{D}_{map}$ of *transitively closed maps* by

$$\mathcal{D}_{trans} = \{\lceil d \rceil \mid d \in \mathcal{D}_{map}\},$$

where

$$\lceil d \rceil = \lambda \upsilon \in \Upsilon.\{\upsilon\} \cup \left\{ \zeta \in d(\upsilon_n) \;\middle|\; \begin{array}{c} \exists \upsilon_0, \ldots, \upsilon_n.\, \upsilon_0 = \upsilon \wedge \\ \forall 0 \le i < n.\, \upsilon_{i+1} \in d(\upsilon_i) \end{array} \right\}.$$

Note that a map $d \in \mathcal{D}_{map}$ is *transitively closed*, i.e., $d \in \mathcal{D}_{trans}$, if and only if it associates υ to a set containing υ, i.e., $\upsilon \in d(\upsilon)$, and for any $\upsilon' \in d(\upsilon)$ it holds that $d(\upsilon') \subseteq d(\upsilon)$.

The abstract domain $\mathcal{D}$ of the copy constant propagation analysis is an augmentation of $\mathcal{D}_{trans}$ with an explicit bottom element.

$$\mathcal{D} = \langle \mathcal{D}_{const}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle, \text{ where}$$

$$\mathcal{D}_{const} = \mathcal{D}_{trans} \cup \{\bot\}$$

$$d_1 \sqsubseteq d_2 \Leftrightarrow d_1 = \bot \vee \forall \upsilon \in \Upsilon.\, d_1(\upsilon) \subseteq d_2(\upsilon)$$

$$\top = \lambda \upsilon \in \Upsilon.\mathsf{VAL}$$

$$d_1 \sqcup d_2 = \begin{cases} d_1 & d_2 = \bot \\ d_2 & d_1 = \bot \\ \lceil d_1 \cup d_2 \rceil & \text{otherwise} \end{cases}$$

$$d_1 \sqcap d_2 = \begin{cases} \bot & d_1 = \bot \vee d_2 = \bot \\ d_1 \cap d_2 & \text{otherwise} \end{cases}$$

8.3.2 Abstract Intraprocedural Transformers

The abstract meaning of the primitive intraprocedural statements is defined as follows:

$$\begin{array}{lcl} [\![x := c]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'c} \\ [\![x := y]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'y'} \\ [\![x := *]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'*} \end{array}$$

where

$$S_{x'}(\upsilon) = \lambda \upsilon \in \Upsilon. \begin{cases} \{x'\} & \upsilon = x' \\ \mathsf{VAL} \setminus \{x'\} & \upsilon \neq x' \end{cases}$$

$$U_{x'\zeta} = \lambda \upsilon \in \Upsilon. \begin{cases} \{x', \zeta\} & \upsilon = x' \\ \{\upsilon\} & \upsilon \neq x' \end{cases}$$

In the following, we show that the abstract transfer functions of the copy constant propagation analysis are conditionally adaptable. We first prove a simple lemma that holds for every lattice.

LEMMA 7. *For any lattice $(D, \sqsubseteq)$ and elements $d, d', d_s \in D$ such that $d' \sqsubseteq d_s$ it holds that*

$$d' \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s \,.$$

Proof By the definition of $\sqcup$ it holds that

$$d \sqsubseteq (d' \sqcup d) \quad \text{and} \quad d' \sqsubseteq (d' \sqcup d) \,.$$

By the monotonicity of $\sqcap$, we get that

$$d \sqcap d_s \sqsubseteq (d' \sqcup d) \sqcap d_s \quad \text{and} \quad d' \sqcap d_s \sqsubseteq (d' \sqcup d) \sqcap d_s \,.$$

By the monotonicity and the idempotence of $\sqcup$, we get that

$$(d' \sqcap d_s) \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s \,.$$

By the assumption $d' \sqsubseteq d_s$. Hence, $d' \sqcap d_s = d'$, and it follows that

$$d' \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s \,.$$

□

DEFINITION 8 (General projection and separation elements). *Let $X \subseteq \Upsilon$. We denote by*

$$S_X(v) = \lambda v \in \Upsilon. \begin{cases} \{v\} & v \in X \\ \mathsf{VAL} \setminus X & v \notin X \end{cases}$$

the separation element *of X and by*

$$R_X = S_{\Upsilon \setminus X}$$

the projection element *of X.*

LEMMA 9. *For every $X \subseteq \Upsilon$, S_X is right-modular.*

Proof We need to prove that for all $d, d' \in \mathcal{D}$ such that $d' \sqsubseteq S_X$ it holds that

$$(d' \sqcup d) \sqcap S_X = d' \sqcup (d \sqcap S_X) \,.$$

By Lem. 7, it holds that

$$(d' \sqcup d) \sqcap S_X \sqsupseteq d' \sqcup (d \sqcap S_X) \,.$$

Thus it suffices to show that

$$d_1 = (d' \sqcup d) \sqcap S_X \sqsubseteq d' \sqcup (d \sqcap S_X) = d_2 \,. \qquad (*)$$

We prove (*) by induction on the size of X.

Base Case. For $|X| = 1$, we get that $S_X = S_{\{v\}}$ for some $v \in \Upsilon$. Pick $v_0 \in \Upsilon$. We need to show that $d_1(v_0) \subseteq d_2(v_0)$. There can be two case: either $v_0 = v$ or not.

If $v_0 = v$, then $d_1(v_0) \subseteq S_{\{v'\}}(v_0) = \{v_0\}$. By definition of the domain, which includes only transitively closed maps, $v_0 \in d'(v_0)$. Hence, $d_1(v_0) \subseteq \{v_0\} \subseteq d_2(v_0)$.

Otherwise, $v_0 \neq v$. Pick $v_1 \in d_1(v_0)$. By the definition of $S_{\{v\}}$ and the meet operation, $v_1 \neq v$, and again by the definition of the meet operation, $v_1 \in (d' \sqcup d)(v_0)$. Thus, there exists a minimal sequence $\zeta_0, \ldots, \zeta_n$ such that $\zeta_0 = v_0 \wedge \zeta_n = v_1$ and for all $0 \leq i < n$, $\zeta_{i+1} \in d'(\zeta_i) \cup d(\zeta_i)$.

CLAIM 10. *For all $0 \leq j < n$, $\zeta_j \neq v$.*

Proof $\zeta_0 = v' \neq v$, by the assumption.

Assume that there exists some $0 < j < n$ such that $\zeta_j = v$. By the minimality of the sequence, $\zeta_{j-1} \neq v$ and $\zeta_{j+1} \neq v$, and since $d' \sqsubseteq S_{\{v\}}$, $\zeta_j = v \notin d'(\zeta_{j-1})$ and $\zeta_{j+1} \notin d'(\zeta_j) = d'(v)$ and thus $\zeta_j \in d(\zeta_{j-1})$ and $\zeta_{j+1} \in d(\zeta_j)$. d is transitively closed and from this, $\zeta_{j+1} \in d(\zeta_{j-1})$. Thus, the sequence $\zeta_0, \ldots, \zeta_{j-1}, \zeta_{j+1}, \ldots, \zeta_n$ is a valid sequence for v', v'', and this is a contradiction to the minimality of the original sequence.

□

By the claim, we got that for all $0 \leq j < n$,

$$\begin{array}{rlll} & \zeta_{i+1} & \in & (d'(\zeta_i) \cup d(\zeta_i)) \setminus \{v\} \quad \text{By the previous claim} \\ \Rightarrow & \zeta_{i+1} & \in & (d'(\zeta_i) \cup d(\zeta_i)) \cap S_{\{v\}}(\zeta_i) \quad \text{By the definition of } S_{\{v\}} \\ & & = & (d'(\zeta_i) \cap S_{\{v\}}(\zeta_i)) \cup (d(\zeta_i) \cap S_{\{v\}}(\zeta_i)) \\ & & = & d'(\zeta_i) \cup (d(\zeta_i) \cap S_{\{v\}}(\zeta_i)) \quad \text{By } d'(\zeta_i) \subseteq S_{\{v\}}(\zeta_i) \\ & & = & d'(\zeta_i) \cup (d \sqcap S_{\{v\}})(\zeta_i) \,. \end{array}$$

Therefore we can create a sequence for v', v'', by taking elements only from $d' \cup (d \sqcap S_{\{v\}})$, and hence $v'' \in (d' \sqcup (d \sqcap S_{\{v\}}))(v')$.

Induction Step. Assume that the induction assumption holds for sets X such that $|X| = n$, we will prove for sets X such that $|X| = n + 1$. Let $v \in X$ be an arbitrary element. Notice that by its definition

$$S_X = S_{X \setminus \{v\}} \sqcap S_{\{v\}} \,.$$

Therefore,

$$\begin{array}{rcl} (d' \sqcup d) \sqcap S_X & = & (d' \sqcup d) \sqcap (S_{X \setminus \{v\}} \sqcap S_{\{v\}}) \\ & = & ((d' \sqcup d) \sqcap S_{X \setminus \{v\}}) \sqcap S_{\{v\}} \\ & \sqsubseteq & (d' \sqcup (d \sqcap S_{X \setminus \{v\}})) \sqcap S_{\{v\}} \\ & = & d' \sqcup ((d \sqcap S_{X \setminus \{v\}}) \sqcap S_{\{v\}}) \\ & = & d' \sqcup (d \sqcap (S_{X \setminus \{v\}} \sqcap S_{\{v\}})) \\ & = & d' \sqcup (d \sqcap S_X) \end{array}$$

□

LEMMA 11. *The abstract transfer functions of the atomic commands are conditionally adaptable.*

Proof By Lem. 9, $S'_x = S_{\{x'\}}$ is right-modular and all the transfer functions are of form

$$f = \lambda d.((d \sqcap d_p) \sqcup d_g) \,.$$

where $d_p = S_{x'}$ for some $x' \in \Upsilon$.

□

8.4 Soundness of the Top Down Analysis

The soundness of the copy constant propagation analysis is formalized by the concretization function $\gamma : \mathcal{D} \to 2^{\mathcal{S} \times \mathcal{S}}$, where

$$\begin{array}{l} (s, s') \in \gamma(d) \iff \\ \quad \big(\forall \overline{x} \in \overline{\mathsf{G}}. (s(x) \in d(\overline{x}) \cap \mathsf{K}) \vee (* \in d(\overline{x}))\big) \bigwedge \\ \quad \big(\forall x' \in \mathsf{G}'. s'(x) \in ((d(\overline{x}) \cap \mathsf{K}) \cup \{s(y) \mid \overline{y} \in d(\overline{x}))\}) \vee (* \in d(\overline{x}))\big) \,. \end{array}$$

Intuitively, an input-output pair (s, s') is *conservatively represented* by an abstract element d if and only if (a) the input state maps a variable g to n if n is one of the constants mapped to $\overline{g}$ by d or if $* \in d(\overline{g})$ and (b) the output state maps a variable g to n if n is one of the constants mapped to g' by d, the value of global variable y at the entry state that is mapped to g' by d, or if $* \in d(g')$.

LEMMA 12 (Soundness). *The abstract transformers pertaining to intraprocedural primitive commands, $|_G$, and* combine *overapproximate the concrete ones.*

8.5 Precision Improving Transformations

In Sec. 8.1, we place certain restrictions on the analyzed programs. Specifically, we forbid copy assignments of the form x:=y between arbitrary global variables `x` and `y`, and, instead, require that the value of `y` be copied to `x` through a sequence of assignments that use a temporary variable `t`. In the concrete semantics, our requirements do not affect the values of the program's variables outside of the sequences of intermediate assignments. In the abstract semantics, however, adhering to our requirements can improve the precision of the analysis, as we explain below.

Consider the execution of the sequence of abstract transformers pertaining to the (non deterministic) command x:=y + y:=3 on an abstract state d, in which $3 \notin d(y')$. Applying the abstract transformer $[\![\texttt{x := y}]\!]^\sharp$ to d results in an abstract element d', where $y' \in d(x')$. Applying $[\![\texttt{y := 3}]\!]^\sharp$ to d results in an abstract state d'', where $3 \in d''(y')$. Perhaps surprisingly, in the abstract state $d''' = d' \sqcup d''$, which conservatively represent the possible states after the non-deterministic choice (+), we get that $3 \in d'''(x')$. This is sound, but imprecise. The reason for the imprecision is that our domain includes only transitively closed maps and having $y' \in d'(x')$ results in an undesired correlation between the possible values of x in d''' and that of y in d''. In particular, the assignment of 3 to y is propagated to x in a flow-insensitive manner.

Rewriting the copy assignment using t according to our restrictions breaks such undesired correlations. Consider, for example, the sequence of abstract transformers pertaining to the aforementioned command: t:=y; y:=0; x:=0; y:=t; x:=t; t:=0, and apply this sequence to d. In the abstract state $\hat{d}$ arising just before t is assigned 0 we get that $t' \in \hat{d}(x')$ and $t' \in \hat{d}(y')$ but $y' \notin \hat{d}(x')$ and $x' \notin \hat{d}(y')$. Assigning 0 to t breaks the correlation between t and x and y.

8.6 Copy Constant Propagation as a Triad Analysis

8.6.1 Triad Domain

LEMMA 13. *$\mathcal{D}$ is a triad domain.*

Projection Elements

Proof We define the projection elements

$$\begin{array}{rcl} d_{\mathsf{proj}_{\mathsf{in}}} & = & R_{\overline{\mathsf{G}}} \\ d_{\mathsf{proj}_{\mathsf{tmp}}} & = & R_{\dot{\mathsf{G}}} \\ d_{\mathsf{proj}_{\mathsf{out}}} & = & R_{\mathsf{G}'} \end{array}$$

By Lem. 9, $d_{\mathsf{proj}_{\mathsf{in}}}, d_{\mathsf{proj}_{\mathsf{tmp}}}$ and $d_{\mathsf{proj}_{\mathsf{out}}}$ are right-modular.

Isomorphism functions We define the renaming functions

$$f^\Upsilon_{\mathsf{call}}, f^\Upsilon_{\mathsf{exit}}, f^\Upsilon_{\mathsf{inout}} \colon \Upsilon \to \Upsilon$$

$$f^\Upsilon_{\mathsf{call}}(\tilde{\upsilon}) = \begin{cases} \dot{\upsilon} & \tilde{\upsilon} = \upsilon' \\ \upsilon' & \tilde{\upsilon} = \dot{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

$$f^\Upsilon_{\mathsf{exit}}(\tilde{\upsilon}) = \begin{cases} \dot{\upsilon} & \tilde{\upsilon} = \overline{\upsilon} \\ \overline{\upsilon} & \tilde{\upsilon} = \dot{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

$$f^\Upsilon_{\mathsf{inout}}(\tilde{\upsilon}) = \begin{cases} \overline{\upsilon} & \tilde{\upsilon} = \upsilon' \\ \upsilon' & \tilde{\upsilon} = \overline{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

Let $f^{\mathsf{VAL}}_{\mathsf{call}}$, $f^{\mathsf{VAL}}_{\mathsf{exit}}$, $f^{\mathsf{VAL}}_{\mathsf{inout}}$ be the renaming function induced on 2^{VAL} and finally let f_{call}, f_{exit}, f_{inout} be the renaming functions induced on $\mathcal{D}$,

$$f_i(d) = \begin{cases} \bot & d = \bot \\ \lambda \upsilon \in \Upsilon.\, f_i^{\mathsf{VAL}}(d(f_i^{\Upsilon\,-1}(\upsilon))) & \text{otherwise} \end{cases} \qquad (12)$$

where $i \in [\mathsf{call}, \mathsf{inout}, \mathsf{exit}]$.

CLAIM 14.

$$\begin{array}{llllll} f_{\mathsf{call}}(R_{\mathsf{G}'}) = R_{\dot{\mathsf{G}}}, & f_{\mathsf{call}}(R_{\overline{\mathsf{G}}}) = R_{\overline{\mathsf{G}}}, & f_{\mathsf{call}}(R_{\dot{\mathsf{G}}}) = R_{\mathsf{G}'} \\ f_{\mathsf{exit}}(R_{\mathsf{G}'}) = R_{\mathsf{G}'}, & f_{\mathsf{exit}}(R_{\overline{\mathsf{G}}}) = R_{\dot{\mathsf{G}}}, & f_{\mathsf{exit}}(R_{\dot{\mathsf{G}}}) = R_{\overline{\mathsf{G}}} \\ f_{\mathsf{inout}}(R_{\mathsf{G}'}) = R_{\overline{\mathsf{G}}}, & f_{\mathsf{inout}}(R_{\overline{\mathsf{G}}}) = R_{\mathsf{G}'}, & f_{\mathsf{inout}}(R_{\dot{\mathsf{G}}}) = R_{\dot{\mathsf{G}}} \end{array}$$

Proof We prove the claim on f_{call} and $R_{\mathsf{G}'}$. The other cases are symmetric.

$$\begin{array}{rcl} f_{\mathsf{call}}(R_{\mathsf{G}'}) & = & f_{\mathsf{call}}\left(\lambda \upsilon \in \Upsilon. \begin{cases} \{\upsilon\} & \upsilon \in \Upsilon \setminus \mathsf{G}' \\ \mathsf{K} \cup \mathsf{G}' & \upsilon \in \mathsf{G}' \end{cases}\right) \\ & = & \lambda \upsilon \in \Upsilon. \begin{cases} \{\upsilon\} & \upsilon \in \Upsilon \setminus \dot{\mathsf{G}} \\ \mathsf{K} \cup \dot{\mathsf{G}} & \upsilon \in \dot{\mathsf{G}} \end{cases} \\ & = & R_{\dot{\mathsf{G}}} \end{array}$$

□

CLAIM 15. *For all $d \in \mathcal{D}_{\mathsf{out}}$*

$$f_{\mathsf{exit}}(f_{\mathsf{inout}}(d)) = f_{\mathsf{call}}(d)$$

Proof Let $d \in \mathcal{D}_{\mathsf{out}}$ and let $\upsilon \in \Upsilon$. If $d = \bot$ then

$$f_{\mathsf{call}}(d) = f_{\mathsf{exit}}(f_{\mathsf{inout}}(d)) = \bot\,.$$

Otherwise, by Eq.12,

$$f_{\mathsf{call}}(d)(\upsilon) = f^{\mathsf{VAL}}_{\mathsf{call}}(d(f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)))$$

and

$$\begin{array}{rcl} ((f_{\mathsf{exit}} \circ f_{\mathsf{inout}})(d))(\upsilon) & = & f_{\mathsf{exit}}(f_{\mathsf{inout}}(d))(\upsilon) \\ & = & f^{\mathsf{VAL}}_{\mathsf{exit}}(f_{\mathsf{inout}}(d)(f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon)) \\ & = & f^{\mathsf{VAL}}_{\mathsf{exit}}(f^{\mathsf{VAL}}_{\mathsf{inout}}(d(f^{\Upsilon\,-1}_{\mathsf{inout}}(f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon)) \\ & = & (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(d(f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}})(\upsilon)) \end{array}$$

If $\upsilon \notin \dot{\mathsf{G}}$, then

$$f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon) \notin \mathsf{G}'$$

and

$$f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon) \notin \mathsf{G}'$$

and therefore

$$d(f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)) = \{f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)\}$$

and

$$d(f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon)) = \{f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon)\}$$

Hence,

$$f_{\mathsf{call}}(d)(\upsilon) = f^{\mathsf{VAL}}_{\mathsf{call}}(\{f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)\}) = \{\upsilon\}$$

and

$$(f_{\mathsf{exit}} \circ f_{\mathsf{inout}})(d)(\upsilon) = (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(\{f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}}(\upsilon)\}) = \{\upsilon\}$$

Otherwise, if $\upsilon \in \dot{\mathsf{G}}$, by the definition of the renaming functions

$$f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon) = (f^{\Upsilon\,-1}_{\mathsf{inout}} \circ f^{\Upsilon\,-1}_{\mathsf{exit}})(\upsilon)$$

and by the definition of $\mathcal{D}_{\mathsf{out}}$,

$$d(f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)) \subseteq G'$$

and therefore again by the definition of the renaming functions

$$f^{\mathsf{VAL}}_{\mathsf{call}}(d(f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon))) = (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(d(f^{\Upsilon\,-1}_{\mathsf{call}}(\upsilon)))$$

□

CLAIM 16. *For all $d \in \mathcal{D}_{\mathsf{in}}$*

$$f_{\mathsf{call}}(d) = d$$

Proof By the definition of $\mathcal{D}_{\mathsf{in}}$ and of f_{call}.

□

ι_{entry} ***element*** We define

$$\iota_{\mathsf{entry}} = [v' \mapsto \{v', \overline{v}\} \mid v \in \mathsf{G}] \cup [\overline{v} \mapsto \{v', \overline{v}\} \mid v \in \mathsf{G}] \cup [\dot{v} \mapsto \{\dot{v}\} \mid v \in \mathsf{G}]$$

CLAIM 17. *For every* $d \in \mathcal{D}_{\mathsf{out}}$, $d \sqcup \iota_{\mathsf{entry}} = f_{\mathsf{inout}}(d) \sqcup \iota_{\mathsf{entry}}$.

Proof

$$\begin{aligned}
d \sqcup \iota_{\mathsf{entry}} &= \lceil d \cup \iota_{\mathsf{entry}} \rceil \\
&= \left\lceil \lambda v \in \Upsilon. \begin{cases} d(g') \cup \{g', \bar{g}\} & v = g' \in \mathsf{G}' \\ \{\bar{g}\} \cup \{g', \bar{g}\} & v = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & v = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= \lambda v \in \Upsilon. \begin{cases} \{\bar{h}, h' \mid h' \in d(g')\} & v = g' \in \mathsf{G}' \vee v = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} & v = \dot{g} \in \dot{\mathsf{G}} \end{cases} \\
&= \left\lceil \lambda v \in \Upsilon. \begin{cases} \{g'\} \cup \{g', \bar{g}\} & v = g' \in \mathsf{G}' \\ \{\bar{h} \mid h' \in d(g')\} \cup \{g', \bar{g}\} & v = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & v = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= \left\lceil \lambda v \in \Upsilon. \begin{cases} \{g'\} \cup \{g', \bar{g}\} & v = g' \in \mathsf{G}' \\ f_{\mathsf{inout}}(d)(\bar{g}) \cup \{g', \bar{g}\} & v = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & v = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= f_{\mathsf{inout}}(d) \sqcup \iota_{\mathsf{entry}}
\end{aligned}$$

□

□

Modular, compositional and sound verification of the input/output behavior of programs

Willem Penninckx, Bart Jacobs, Frank Piessens

Department of Computer Science, KU Leuven, Belgium

Report CW663, April 2014

Abstract

We present a sound verification approach for verifying input/output properties of programs. Our approach supports compositionality (you can build high-level I/O actions on top of low-level ones), modularity (you can define and implement input/output actions without taking into account which other actions exist) and other features.

1 Introduction

Many software verification approaches are based on Hoare logic. A Hoare triple consists of a precondition, a program, and a postcondition. If a Hoare triple is true, then every execution of the program starting from any state satisfying the precondition results in a state satisfying the postcondition. Hoare logic has been extended to support e.g. aliasing, concurrency, and so forth. But a certain limitation is often left untackled. Indeed, the pre- and postcondition of a Hoare triple typically constrain the state of a program by only looking at the initial and final state of memory. This makes it possible to prove e.g. that a quicksort implementation sorts properly, but it does not state that this result is e.g. printed on the screen. For the user of a program, the proofs about the state of memory of a program are useless if the user never sees the result on his screen. In the end, the program is supposed to correctly perform input/output, a problem typically left untouched. However, this is not the challenge itself. The interesting part of the challenge lays in the side constraints such as compositionality and modularity:

Modularity A programmer of a library typically does not consider all possible other libraries that might exist. Still, a programmer of an application can use multiple libraries in his program, even though these libraries do not know of each other's existence. Similarly, we want to write specifications of a library without keeping in mind existence of other libraries.

Compositionality In regular software development, a programmer typically does not call the low-level system calls. Instead, he calls high-level libraries, which might be implemented in terms of other libraries, implemented on top of other libraries, and so on. This is the concept of compositionality. The verification approach for I/O should support programs written in a compositional manner. Furthermore, it should be possible to write the formal I/O specifications themselves in a compositional manner, i.e. in terms of other libraries' I/O actions instead of in terms of the low-level system calls.

Other Besides compositionality and modularity, the I/O verification should also

- be sound (i.e. not searching for bugs but proving absence of bugs) and static (i.e. not detecting errors at runtime, but proving that such errors will never occur)
- blend in well with existing verification techniques that solve other problems like aliasing (or solve them itself)

- support non-deterministic behaviour (e.g. operations can fail, or return unspecified values, like reading user input)
- support underspecified specifications (e.g. the specifications describe two possibilities and the implementor can choose freely.
- support arguments for operations (e.g. when writing to a file, the content and the filename are arguments that should be part of the specifications)
- support concurrency
- support unspecified ordering of operations. If the order is unimportant, the specification should not fix them such that the implementor can choose freely.
- support specifying ordering of operations, also if the operations are specified and implemented by independent teams. e.g. it might be necessary that the put-shield-on operation happens before the start-explosion operation.
- support non-terminating programs: a non-terminating program can still only do the allowed I/O operations in the allowed order.
- support terminating programs: a program is only allowed to terminate if the I/O operations done are as specified and invoked.
- support operations that depend on the outcome of the previous operation, e.g. a specification like "read a number, and then print a number that is one higher than the read number".

This technical report provides an elegant way to perform sound modular compositional input/output verification based on separation logic and supports all the requirements explained above.

Section 2 describes the approach in a tutorial-style fashion, so it does not explain how it works but only how to use it. Section 3 formalizes the approach. Section 4 gives a proof outline of an example.

2 How to use the approach: a tutorial

We describe the input/output verification approach in a tutorial-style fashion. So this section does not explain how it works but only how to use it.

To get a better understanding, it might help to experiment. You can do so by using the VeriFast verifier, available for free from `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`. The VeriFast distribution contains some examples of verifying I/O properties of programs. At the moment of writing, they are in the directory `examples/io`.

2.1 Warming up

When writing $\{P\}\, C\, \{Q\}$, we call P the precondition, Q the postcondition, and C a program (here, a function is also considered a program). A pre- and a postcondition both consist of an assertion. An assertion states some properties about the state of the program. Usually, the state of the program is the state of memory of the program. A precondition expresses some properties the program is allowed to assume to be true before execution; the postcondition are properties the program is supposed to make true after every possible execution starting from a state where the precondition is true. If this is the case, we say $\{P\}\, C\, \{Q\}$ is true.

Formal verification consists of proving that, starting from any state where the precondition is true, every execution of the program will result in a state for which the postcondition is true.

The pre- and postcondition together is what we call a contract. In this tutorial, we will learn only how to write contracts that specify the input-output behavior of programs. The formal semantics are out of scope of the tutorial and studied in section 3.

2.2 Time

We give a small input-output contract, namely one where no input-output is allowed.

```
{}
    // any implementation that doesn't crash or race and does no input-output
{}
```

Indeed, the empty pre- and postcondition do not allow input/output. Let's keep disallowing I/O while making the contract bigger:

```
{time(t1)}
  // any implementation that doesn't crash or race and does no input-output
{time(t1)}
```

The precondition `time(t_1)` can conceptually be considered as stating that the current time is t_1. For starters, you can think of a time as a point in time you are used to, e.g 8 AM or noon. Performing an input-output action would increase the time. The postcondition is the same as the precondition, so the program must make sure all constraints on the list of performed (and future) input-output actions still hold after execution. The only way to do this, is to not perform any input-output.

2.3 Actions

```
{time(t1) * print_char_io(t1, 'h', t2)}
  print_char('h');
{time(t2)}
```

The above contract states the only input-output behavior the program is allowed to perform, is writing the character 'h', and that, after execution of the program, the program must have done this input-output behavior.

`print_char_io(t_1, 'h', t_2)` states it is possible to go from time t_1 to time t_2 by writing the character 'h'. You can consider `print_char_io(t_1, 'h', t_2)` as a permission[1] to print 'h' provided the current time is t_1, and a promise that the time will increase to t_2 when doing so.

The postcondition says the time must become t_2, and because the only "permission" to obtain t_2 is by performing the input-output action of writing 'h', the program must have written 'h' to satisfy the postcondition. It cannot print 'h' twice, because it has no permission to print 'h' starting from a time t_2.

2.4 Choice

In the previous contract, there was only one permission provided to obtain t_2. In the following contract, two permissions are provided. As a result, the implementation can choose freely which of the two permissions to use.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t1, 'w', t2)}
  print_char('w');
{time(t2)}
```

So, the implementation can either print 'h' or 'w'. It can even choose at runtime what to do (e.g. using a (non I/O performing) random generator). It, however, can not do both.

2.5 Sequence

The following contract states the program must print "hi" and should be self-explaining by now.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t2, 'i', t3)}
  print_char('h');
  print_char('i')
{time(t3)}
```

Note that you can combine sequencing and choice, e.g.

[1] technically, it is not always entirely correct to do so, but it most often works to reason about it as such.

```
{time(t1) * print_char_io(t1, 'h', t2) * print_char_io(t2, 'i', t4)
 * print_char_io(t1, 'l', t3) * print_char_io(t3, 'o', t4)
 * print_char_io(t4, '!', t5)}
  // ... (implementation here)
{time(t5)}
```

The above program prints either "hi!" or "lo!".

2.6 Defining actions

So far, we treated `print_char_io` as an action coming from nowhere. You can define your own action in terms of other actions:

```
predicate print_string_io(t1, str, t2) =
  if str = nil then
    t1 = t2
  else (
    print_char_io(t1, head(str), t_between)
    * print_string_io(t_between, tail(str), t2)
  )
```

`print_string_io(t1, ''hi!'', t2)` can conceptually be considered a "shorthand" to

```
print_char_io(t1, 'h', t_between0)
* print_char_io(t_between0, 'i', t_between1)
* print_char_io(t_between1, '!', t2)
```

Technically, it's not really a shorthand but a predicate in the sense of [1]. When using a proof checker, one can write predicates without understanding the underlying theory because errors will be spotted by the proof checker.

Now we can finally write a clean hello world:

```
function print_string(str) =
{time(t1) * print_string_io(t1, str, t2)}
  while str != nil
    print_char(head(str));
    str := tail(str)
{time(t2)}

function helloworld() =
{time(t1) * print_string_io(t1, ''hello world!'', t2)}
  print_string(''hello world!'')
{time(t2)}
```

Of course, the implementation (and contract) of `print_string` is usually not considered part of the hello-world program, but part of a (standard) library, just like C's `helloworld` typically does not contain the implementation of `printf`.

Note that we sneaked in compositionality: we defined an I/O action `print_string_io` in terms of lower-level I/O actions (here `print_char_io`). You can now also define higher-level actions in terms of `print_string_io`.

2.7 Interleaved actions

Quite often, the order in which things happen matters. For example, you might want to put on the goggles before turning on the laser beam. But sometimes order does not matter, and in these cases it would be annoying if the specifications restrict the order in which the implementor can write his program. Sometimes the specification should leave space for the implementor by not being too restrictive.

An example: consider Unix' `cat`, a small program that just writes what it reads. The following contract would be annoying (`read_string_io` is similar to `print_string_io` and reads until e.g. end of file).

```
{time(t1) * read_string_io(t1, str, t2) * print_string_io(t2, str, t3)}
  // ...
{time(t3)}
```

Indeed: the programmer's implementation would be forced to first read everything, and then write everything. This would be impractical, certainly in case of limited memory.

Let's try again:

```
predicate readwrite_io(t1, str, t2) =
  read_char_io(t1, c, t_between0)
  * if c < 0 then (
      t2 = t_between0
      * str = nil
    ) else (
      c == head(str)
      * print_char_io(t1, head(str), t_between1)
      * readwrite_io(t_between1, tail(str), t2)
    )
{time(t1) * readwrite_io(t1, str, t2)}
  // ...
{time(t2)}
```

This would certainly solve the memory problem. But it introduces another one. What if the implementor wants to keep a buffer? For example, he might want to read 10 bytes, and then write 10 bytes, then read 10, and so forth. The contract disallows this. If we want to disallow this, we're ready. But if we want to allow the programmer to use any buffer size he wants, we can use a feature called split-join.

This is a solution:

```
{time(t1)
 * split(t1, t2, t3)
 * read_string_io(t2, str, t4)
 * print_string_io(t3, str, t5)
 * join(t4, t5, t6)
}
  // ...
{time(t6)}
```

Here, the implementor can interleave reads and writes as much or as little as he wants. Technically, he is even allowed to write everything before reading everything, if he is able to prove that in every execution the written output will be the same as the read input (which he will not be able to do).

You should now be able to tell the difference with:

```
{time(t1) * read_string_io(t1, str, t2) * print_string_io(t1, str, t2)}
  // ...
{time(t2)}
```

(Spoiler: with split you must both read and print, but the order between reading and printing is unspecified. Without split, you must or read or print, but not both).

We're not ready yet. We saw that the precondition of `print_string` is

```
  time(t1) * print_string_io(t1, str, t2)
```

So, in order to call this function, one must have a "time" t_1 that is equal to the first argument of `print_string_io`. Which is, right after the precondition of `cat`, not the case.

One can obtain `time(t2) * time(t3)` out of `time(t1) * split(t1, t2, t3)`. Hence the name "split": it splits time. Similarly, one can obtain `time(t3)` out of `join(t1, t2, t3) * time(t1) * time(t2)`. Join joins times together.

Note that without split-join, the approach would be completely unusable in practice.

2.8 Time revisited

Earlier, we said you can consider a time like t_1 as a point in the real-world time, like 8 AM or noon. This might be a confusing way to reason since time is not always before or after another time. Indeed, when splitting time, we obtain two times which do not have much relative ordering.

2.9 Other properties

The above part of the tutorial might give a misleading impression on the expressiveness of the approach. Since the approach blends in with separation logic-based verification, we have all the expressiveness of separation logic-based verification.

For example, if we want to express "No single program shall ever write to a file without opening it", we can simply do this by putting a permission in the postcondition of the function to open a file that expresses that the file has been opened, and require this permission in the precondition of the function to read from files. This was already possible before and the approach presented in this technical report is compatible with it.

Also, the approach can be combined with concurrency. In case two threads can do actions where there is no locking required, one can simply use split-join to allow any interleaving of these actions. If locking is required, the permissions to perform the action can be included among the permissions protected by the lock (the invariant of the lock).

3 Formalisation

We present a simple programming language and a verification approach.

$v \in \text{VarNames}$, $v^l \in \text{ListVarNames}$, $n, r \in \mathbb{Z}$, $bio \in \text{BioNames}$, $l \in \text{Lists}$, $f \in \text{FuncNames}$

$l ::= \textbf{nil} \mid n :: l$

$e ::= n \mid v \mid e + e \mid e - e \mid \text{head}(e^l)$

$e^l ::= l \mid v^l \mid e^l +\!\!+ e^l \mid \text{tail}(e^l)$

$b ::= \text{true} \mid \neg b \mid e = e \mid b \wedge b \mid e < e \mid e^l = e^l$

$c ::= \textbf{skip} \mid v := e \mid c \,;\, c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b\ c \mid v := bio(\overline{e}) \mid v := f((\overline{e}), (\overline{e^l}))$

The language is standard except it supports doing Basic Input Output actions (BIOs). A BIO can be thought of as a system call, but for readability we use names ($bio \in \text{BioNames}$) as their identifiers instead of numbers. The arguments of a BIO can be considered as data written to the outside world, while the return-value can be considered as data that is read from the outside world. This way, a BIO allows doing both input and output.

We define Commands as the set of commands creatable by the grammar symbol "c" and quantify over it with c. Stores $= \{s_v \cup s_l \mid s_v \in \text{VarNames} \rightharpoonup \mathbb{Z} \wedge s_l \in \text{ListVarNames} \rightharpoonup \text{Lists}\}$, quantified over by s.

We assume a set FuncDefs $\subset \{(f, (\overline{v}), (\overline{v^l}), c) \mid f \in \text{FuncNames} \wedge \overline{v} \in \overline{\text{VarNames}} \wedge \overline{v^l} \in \overline{\text{ListVarNames}} \wedge c \in \text{Commands} \wedge \text{mod}(c) \cap (\overline{v} \cup \overline{v^l}) = \emptyset\}$. Here, $\text{mod}(c)$ returns the set of variables that command c writes to. FuncDefs represents the functions of the program under consideration. Note that we disallow functions for which the body assigns to a parameter of the functions. We also disallow overlap in parameter names.

Evaluation of expressions

$[\![n]\!]_s$	$= n$
$[\![e_1 + e_2]\!]_s$	$= [\![e_1]\!]_s + [\![e_2]\!]_s$
$[\![e_1 - e_2]\!]_s$	$= [\![e_1]\!]_s - [\![e_2]\!]_s$
$[\![v]\!]_s$	$= s(v)$ if v defined in s, otherwise 0.
$[\![\text{head}(e^l)]\!]_s$	$= \text{head}([\![e^l]\!]^l_s)$.
$[\![l]\!]^l_s$	$= l$
$[\![v^l]\!]^l_s$	$= s(v^l)$ if v^l defined in s, otherwise nil.
$[\![e^l_1 +\!+ e^l_2]\!]^l_s$	$= [\![e^l_1]\!]^l_s +\!+ [\![e^l_1]\!]^l_s$.
$[\![\text{tail}(e^l)]\!]^l_s$	$= \text{tail}([\![e^l]\!]^l_s)$.
$[\![\text{true}]\!]^b_s$	$= \text{true}$
$[\![\neg b]\!]^b_s$	$= \neg[\![b]\!]^b_s$
$[\![b_1 \wedge b_2]\!]^b_s$	$= [\![b_1]\!]^b_s \wedge [\![b_2]\!]^b_s$
$[\![e_1 = e_2]\!]^b_s$	$= ([\![e_1]\!]_s = [\![e_2]\!]_s)$
$[\![e_1 < e_2]\!]^b_s$	$= [\![e_1]\!]_s < [\![e_2]\!]_s$
$[\![e^l_1 = e^l_2]\!]^b_s$	$= ([\![e^l_1]\!]_s = [\![e^l_2]\!]_s)$

Notation For a (partial) function f, $f[a := b]$ is the (partial) function $\{(x, y) | (x \neq a \wedge x \in dom(f) \wedge y = f(x)) \vee (x = a \wedge y = b)\}$.

For lists, we use the infix functions $+\!+$ for concatenation and :: for cons. We write the empty list as nil.

We frequently notate lists with overline, e.g. $\overline{e}$ denotes a (implicitly quantified) list of expressions. We leave the technical parts implicit, e.g. when two lists are expected to have the same length. Sometimes we use such a list as a set. We abbreviate $\overline{[\![e]\!]}$ as $[\![\overline{e}]\!]$. We overline sets to obtain the set of lists of elements of this set, e.g. $\overline{\text{VarNames}}$ denotes the set of lists of variable names.

For an assertion P, $P[e/v]$ is the formula obtained by replacing all free occurrences of the variable v with e. We write multiple such replacements as $P[\overline{e}/\overline{v}]$. We also use this notation for replacing logical variables, logical expressions, etc.

For multisets A and B, $A - B$ denotes the multiset obtained by removing the occurrences of B in A, e.g. $\{\!\{1, 1, 1, 2, 2, 3\}\!\} - \{\!\{1, 2, 2\}\!\} = \{\!\{1, 1, 3\}\!\}$. $A + B$ yields the multiset such that for every x, the number of occurrences of x in $A + B$ (which can be zero) equals the number of occurrences of x in A plus the number of occurrences of x in B.

Step semantics We define Traces as the set of lists over the set $\{bio(\overline{n}) \mid bio \in \text{BioNames} \wedge \overline{n} \in \bigcup_{m>0} \mathbb{Z}^m\}$. An element of the list, $bio(\overline{n}, r)$, expresses the BIO bio has happened with arguments $\overline{n}$ and return value r. The order of the items in the list expresses the order in time in which they happened. We quantify over Traces with τ.

Continuations = Commands $\cup$ {**partial**, **done**}, quantified over by κ.

$$\textsc{Assign}\ \frac{}{s, v := e \Downarrow s[v := [\![e]\!]_s], \text{nil}, \textbf{done}} \qquad \textsc{AssignList}\ \frac{}{s, v^l := e^l \Downarrow s[v^l := [\![e^l]\!]^l_s], \text{nil}, \textbf{done}}$$

$$\textsc{IfThen}\ \frac{[\![b]\!]^b_s = \text{true} \qquad s, c_{then} \Downarrow s', \tau, \kappa}{s, \textbf{if}\ b\ \textbf{then}\ c_{then}\ \textbf{else}\ c_{else} \Downarrow s', \tau, \kappa} \qquad \textsc{IfElse}\ \frac{[\![b]\!]^b_s = \text{false} \qquad s, c_{else} \Downarrow s', \tau, \kappa}{s, \textbf{if}\ b\ \textbf{then}\ c_{then}\ \textbf{else}\ c_{else} \Downarrow s', \tau, \kappa}$$

$$\textsc{WhileIn}\ \frac{[\![b]\!]^b_s = \text{true} \qquad s, c\,; \textbf{while}\ b\ c \Downarrow s', \tau, \kappa}{s, \textbf{while}\ b\ c \Downarrow s', \tau, \kappa} \qquad \textsc{WhileOut}\ \frac{[\![b]\!]^b_s = \text{false}}{s, \textbf{while}\ b\ c \Downarrow s, \text{nil}, \textbf{done}} \qquad \textsc{Skip}\ \frac{}{s, \textbf{skip} \Downarrow s, \text{nil}, \textbf{done}}$$

$$\textsc{Seq2}\ \frac{s_1, c_1 \Downarrow s_2, \tau_2, \textbf{done} \qquad s_2, c_2 \Downarrow s_3, \tau_3, \kappa}{s_1, c_1; c_2 \Downarrow s_3, \tau_2 +\!\!+ \tau_3, \kappa} \qquad \textsc{Seq}\ \frac{s_1, c_1 \Downarrow s_2, \tau_2, \textbf{partial}}{s_1, c_1; c_2 \Downarrow s_2, \tau_2, \textbf{partial}} \qquad \textsc{Empty}\ \frac{}{s, c \Downarrow s, \text{nil}, \textbf{partial}}$$

$$\textsc{Bio}\ \frac{i \in \mathbb{Z}}{s, v := bio(\overline{e}) \Downarrow s[v := i], bio([\![\overline{e}]\!]_s, i) :: \text{nil}, \textbf{done}}$$

$$\textsc{FuncCall}\ \frac{\emptyset[\overline{v}, \overline{v^l} := [\![\overline{e}]\!]_s, [\![\overline{e^l}]\!]_s], c \Downarrow s_f, \tau, \kappa \qquad (f, (\overline{v}), (\overline{v^l}), c) \in \text{FuncDefs}}{s, v := f((\overline{e}), (\overline{e^l})) \Downarrow s[v := [\![\text{result}]\!]_{s_f}], \tau, \kappa}$$

Note that the step semantics do not only support terminating runs, but also partial runs. This allows us to verify the input/output behavior of programs that do not terminate.

Assertions We define Times as the set of times and quantify over it with t. Intuitively, a time t can be considered as a name for a timestamp with unknown value, or as a set of constraints on a timestamp.

We define Chunks as $\{\text{time}(t) \mid t \in \text{Times}\} \cup \{bio(t_1, \overline{n}, r, t_2) \mid bio \in \text{BioNames} \wedge \overline{n} \in \bigcup_{m \geq 0} \mathbb{Z}^m \wedge r \in \mathbb{Z} \wedge t_1, t_2 \in \text{Times}\}$ and Heaps as Chunks $\rightharpoonup \mathbb{N}$. We will use the heap for permissions (rather than memory footprint) and treat it like separation logic [2].

$V^n \in \text{IntegerLogicalVarNames}$, $V^t \in \text{TimeLogicalVarNames}$, $V^l \in \text{ListLogicalVarNames}$, $p \in \text{PredNames}$, $V \in \text{IntegerLogicalVarNames} \cup \text{TimeLogicalVarNames} \cup \text{ListLogicalVarNames}$.

$$E^n ::= n \mid v \mid V^n \mid E^n + E^n \mid E^n - E^n \mid \text{head}(E^l)$$
$$E^t ::= t \mid V^t$$
$$E^l ::= l \mid v^l \mid V^l \mid E^l +\!\!+ E^l \mid \text{tail}(E^l)$$
$$B ::= \text{true} \mid E^n = E^n \mid E^t = E^t \mid E^l = E^l \mid \neg B \mid E^n < E^n$$
$$P, Q, R ::= B \mid \textbf{emp} \mid P \star P \mid bio(E^t, \overline{E^n}, E^n, E^t) \mid \textbf{split}(E^t, E^t, E^t) \mid \textbf{join}(E^t, E^t, E^t) \mid \textbf{time}(E^t) \mid p(\overline{E^n}, \overline{E^l}, \overline{E^t}) \mid P \vee P \mid \exists V.\ P$$

We assume all sets of variable names (IntegerLogicalVarNames, VarNames, ...) to be disjoint.

We use P, Q and R to quantify over the assertions formed by grammar symbol P.

We define Interpretations $= \{i_n \cup i_t \cup i_l \mid i_n \in \text{IntegerLogicalVarNames} \rightharpoonup \mathbb{Z} \wedge i_t \in \text{TimeLogicalVarNames} \rightharpoonup \text{Times} \wedge i_l \in \text{ListLogicalVarNames} \rightharpoonup \text{Lists}\}$ and quantify over it with i. An interpretation maps logical variables to values.

emp denotes the heap is empty and $\star$ is the separating conjunction [2]. The existential quantors for integers, times, and lists are necessary such that they can be used in definitions of inductive predicates.

We use predicates based on and similar to [1]. A predicate can be considered as a named assertion, but the assertion can contain the predicate name to allow recursion. A predicate definition consists of a predicate name, a number of integer argument names, a number of list argument names, a number of time argument names (all argument names distinct), and an assertion. We disallow mutual recursion. We write PredDefs for the set of predicate definitions for the program under consideration. This is

the set of definitions for (the contracts of) a particular program, not the set of all possible definitions. PredDefs $\subset$ PredNames $\times$ $(\overline{V^n})$ $\times$ $(\overline{V^l})$ $\times$ $(\overline{V^t})$ $\times$ P.

An assertion $bio(t_1, \overline{e}, e_r, t_2)$ expresses the permission to perform the BIO bio with arguments $\overline{e}$ at timestamp t_1 and includes the prediction that performing that BIO at the given time will return e_r and finish at time t_2.

The **split** and **join** assertions allow interleaved actions and choosing freely which actions happen first. We refer to the tutorial for better insight in the expressiveness of these assertions.

In the tutorial, we wrote assertions of the form **if** b **then** P **else** Q. This is shorthand notation for $(b \star P) \vee (\neg b \star Q)$.

Evaluation of assertion expressions

$[\![n]\!]_{s,i}$ $= n$

$[\![v]\!]_{s,i}$ $= s(v)$ if v defined in s, otherwise 0.

$[\![V^n]\!]_{s,i}$ $= i(V^n)$ if V^n defined in i, otherwise 0.

$[\![V^t]\!]_{s,i}$ $= i(V^t)$ if V^t defined in i, otherwise undefined.

$[\![E_1^n + E_2^n]\!]_{s,i}$ $= [\![E_1^n]\!]_{s,i} + [\![E_2^n]\!]_{s,i}$

$[\![E_1^n - E_2^n]\!]_{s,i}$ $= [\![E_1^n]\!]_{s,i} - [\![E_2^n]\!]_{s,i}$

$[\![l]\!]_{s,i}$ $= l$.

$[\![v^l]\!]_{s,i}$ $= s(v^l)$ if v^l defined in s, otherwise nil.

$[\![V^l]\!]_{s,i}$ $= i(V^l)$ if V^l defined in i, otherwise nil.

$[\![E_1^l +\!+ E_2^l]\!]_{s,i}$ $= [\![E_1^l]\!]_{s,i} +\!+ [\![E_2^l]\!]_{s,i}$.

$[\![\text{head}(E^l)]\!]_{s,i}$ $= \text{head}([\![E^l]\!]_{s,i})$

$[\![\text{tail}(E^l)]\!]_{s,i}$ $= \text{tail}([\![E^l]\!]_{s,i})$

$[\![E_1^n = E_2^n]\!]_{s,i}$ $= ([\![E_1^n]\!]_{s,i} = [\![E_2^n]\!]_{s,i})$

$[\![E_1^t = E_2^t]\!]_{s,i}$ $= ([\![E_1^t]\!]_{s,i} = [\![E_2^t]\!]_{s,i})$

$[\![E_1^l = E_2^l]\!]_{s,i}$ $= ([\![E_1^l]\!]_{s,i} = [\![E_2^l]\!]_{s,i})$

$[\![\neg B]\!]_{s,i}$ $= \neg [\![B]\!]_{s,i}$

$[\![E_1^n < E_2^n]\!]_{s,i}$ $= [\![E_1^n]\!]_{s,i} < [\![E_2^n]\!]_{s,i}$

$[\![\text{true}]\!]_{s,i}$ $= \text{true}$

Satisfaction relation of formulae For predicates, we assume a context I which we will define later. $I \subseteq$ PredNames $\times$ $\overline{\mathbb{Z}}$ $\times$ $\overline{\text{Lists}}$ $\times$ $\overline{\text{Times}}$ $\times$ Heaps. I expresses, for a given predicate name and argument values, the heap chunks a predicate assertion covers.

$$
\begin{array}{lcl}
I, s, h, i \models B & \iff & [\![B]\!]_{s,i} = true \wedge h = \emptyset \\
I, s, h, i \models bio(t_1, \overline{E^n}, E_r^n, t_2) & \iff & h = \{\!\{bio(t_1, [\![\overline{E^n}, E_r^n]\!]_{s,i}, t_2)\}\!\} \\
I, s, h, i \models \mathbf{join}(t_1, t_2, t_3) & \iff & h = \{\!\{\mathbf{join}(t_1, t_2, t_3)\}\!\} \\
I, s, h, i \models \mathbf{split}(t_1, t_2, t_3) & \iff & h = \{\!\{\mathbf{split}(t_1, t_2, t_3)\}\!\} \\
I, s, h, i \models \mathbf{time}(t) & \iff & h = \{\!\{\mathbf{time}(t)\}\!\} \\
I, s, h, i \models \mathbf{emp} & \iff & h = \{\!\{\}\!\} \\
I, s, h, i \models P \star Q & \iff & \exists h_1, h_2 \,.\; h_1 + h_2 = h \;\wedge\; I, s, h_1 \models P \;\wedge\; I, s, h_2 \models Q \\
I, s, h, i \models p(\overline{E^n}, \overline{E^l}, \overline{t}) & \iff & (p, ([\![\overline{E^n}]\!]_{s,i}), ([\![\overline{E^l}]\!]), (\overline{t}), h) \in I \\
I, s, h, i \models P \vee Q & \iff & (I, s, h, i \models P) \vee (I, s, h, i \models Q) \\
I, s, h, i \models \exists V^n.\; P & \iff & \exists n \in \mathbb{Z}.\; I, s, h, i \models P[V^n := n] \\
I, s, h, i \models \exists V^l.\; P & \iff & \exists l.\; I, s, h, i \models P[V^l := l] \\
I, s, h, i \models \exists V^t.\; P & \iff & \exists t.\; I, s, h, i \models P[V^t := t]
\end{array}
$$

Note that the satisfaction relation is undefined for formulae with unbound variables of V^n, V^l and V^l.

We assumed I so far but did not define it yet.

$$
\begin{aligned}
&I_0 = \emptyset \\
&I_{n+1} = \{\ (p, (\overline{n}), (\overline{l}), (\overline{t}), h) \mid \exists \overline{V^n}, \overline{V^l}, \overline{V^t}, P.\ (p, (\overline{V^n}), (\overline{V^l}), (\overline{V^t}), P) \in \text{PredDefs} \wedge \\
&\qquad I_n, \emptyset, h, \emptyset \models P[\overline{n}, \overline{l}, \overline{t} / \overline{V^n}, \overline{V^l}, \overline{V^t}]\ \}
\end{aligned}
$$

We define I as $\bigcup_{n \in \mathbb{N}} I_n$.

Validity of Hoare triples Intuitively, the Hoare triple $\{P\}\, c\, \{Q\}$ expresses that the program c satisfies the contract with precondition P and postcondition Q.

For examples of Hoare triples and their meaning, we refer to the tutorial (section 2).

Note that a program that satisfies the contract cannot perform any other I/O operations, cannot do them in another order, cannot do them more than once, etc.

We define a relation traces $\subset$ (Heaps $\times$ Traces $\times$ Heaps). $\{h_1\}\, \tau\, \{h_2\}$ denotes $(h_1, \tau, h_2) \in$ traces. It expresses that τ is allowed by h_1. An implementation is thus allowed to produce the trace τ. Thus, a heap is mapped to a set of allowed traces. Remember that an element in the heap can make a prediction about the world, e.g. $bio(t_1, \overline{n}, r, t_2)$ predicts performing the BIO bio with arguments $\overline{n}$ (at a certain time) will have return-value r. Thus, a heap can contradict itself, e.g. $\{\!|\mathbf{time}(\mathrm{t}_1), \mathrm{some\_bio}(\mathrm{t}_1, 1, 2, \mathrm{t}_2), \mathrm{some\_bio}(\mathrm{t}_1, 1, 3, \mathrm{t}_2)|\!\}$ contradicts itself because it says performing the BIO some_bio (at time t_1 with argument 1) will return 2 and will return 3. After a BIO that violates a prediction, all further BIOs are allowed.

$$
\textsc{TraceBio}\ \frac{r = r' \Rightarrow \{h + \{\!|\mathbf{time}(t_2)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), bio(t_1, \overline{n}, r, t_2)|\!\}\}\, bio(\overline{n}, r') :: \tau\, \{h'\}} \qquad \textsc{TraceNil}\ \frac{}{\{h\}\, \mathrm{nil}\, \{h\}}
$$

$$
\textsc{TraceSplit}\ \frac{\{h + \{\!|\mathbf{time}(t_2), \mathbf{time}(t_3)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), \mathbf{split}(t_1, t_2, t_3)|\!\}\}\, \tau\, \{h'\}} \qquad \textsc{TraceJoin}\ \frac{\{h + \{\!|\mathbf{time}(t_3)|\!\}\}\, \tau\, \{h'\}}{\{h + \{\!|\mathbf{time}(t_1), \mathbf{time}(t_2), \mathbf{join}(t_1, t_2, t_3)|\!\}\}\, \tau\, \{h'\}}
$$

We define validity of a Hoare triple. Intuitively, it expresses that any execution starting from a state (a store, a heap and an interpretation) that satisfies the precondition, results in a trace that is allowed by the heap. In case the execution is a finished one (i.e. the program terminated), the state at termination must satisfy the postcondition.

$$
\begin{aligned}
&\forall P, c, Q.\ \models \{P\}\, c\, \{Q\} \iff \\
&\quad \forall s, h, i.\ I, s, h, i \models P \Rightarrow \\
&\qquad \forall s', \tau', \kappa'.\ s, c \Downarrow s', \tau', \kappa' \Rightarrow \\
&\qquad\quad \exists h'.\ \{h\}\, \tau'\, \{h'\} \wedge \\
&\qquad\qquad (\kappa' = \mathbf{done} \Rightarrow I, s', h', i \models Q)
\end{aligned}
$$

In case you expected a universal quantifier for h', note that the concrete execution does not use a heap. If it would use a heap, h' would be introduced in the universal quantification together with s', τ' and κ'.

Proof rules

$$\textsc{Assignment}\ \frac{}{\{P[e/v]\}\, v := e\, \{P\}} \qquad \textsc{AssignmentList}\ \frac{}{\{P[e^l/v^l]\}\, v^l := e^l\, \{P\}} \qquad \textsc{Composition}\ \frac{\{P_1\}\, c_1\, \{P_2\} \qquad \{P_2\}\, c_2\, \{P_3\}}{\{P_1\}\, c_1; c_2\, \{P_3\}}$$

$$\textsc{Consequence}\ \frac{P_1 \Rightarrow P_2 \qquad \{P_2\}\, c\, \{P_3\} \qquad P_3 \Rightarrow P_4}{\{P_1\}\, c\, \{P_4\}} \qquad \textsc{While}\ \frac{\{b \star P\}\, c\, \{P\}}{\{P\}\, \textbf{while}\ b\ c\, \{\neg b \star P\}} \qquad \textsc{Skip}\ \frac{}{\{P\}\, \textbf{skip}\, \{P\}}$$

$$\textsc{If}\ \frac{\{P \star b\}\, c_{\text{then}}\, \{Q\} \qquad \{P \star \neg b\}\, c_{\text{else}}\, \{Q\}}{\{P\}\, \textbf{if}\ b\ \textbf{then}\ c_{\text{then}}\ \textbf{else}\ c_{\text{else}}\, \{Q\}} \qquad \textsc{Disjunction}\ \frac{\{P_1\}c\{Q\} \qquad \{P_2\}c\{Q\}}{\{P_1 \vee P_2\}\, c\, \{Q\}}$$

$$\textsc{Bio}\ \frac{}{\{P[e_r/v] \star bio(t_1, \overline{e}, e_r, t_2) \star \textbf{time}(t_1)\}\, v := bio(\overline{e})\, \{P \star \textbf{time}(t_2)\}}$$

$$\textsc{Split}\ \frac{\{\textbf{time}(t_2) \star \textbf{time}(t_3)\}\, c\, \{\textbf{time}(t_4)\}}{\{\textbf{time}(t_1) \star \textbf{split}(t_1, t_2, t_3)\}\, c\, \{\textbf{time}(t_4)\}} \qquad \textsc{Join}\ \frac{\{\textbf{time}(t_3)\}\, c\, \{\textbf{time}(t_4)\}}{\{\textbf{time}(t_1) \star \textbf{time}(t_2) \star \textbf{join}(t_1, t_2, t_3)\}\, c\, \{\textbf{time}(t_4)\}}$$

$$\textsc{Frame}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fv}(R) \cap \text{mod}(c) = \emptyset}{\{P \star R\}\, c\, \{Q \star R\}}$$

$$\textsc{FuncCall}\ \frac{\begin{array}{c}\{P\}\, c\, \{Q\} \qquad \text{fv}(P) \subseteq (\overline{v} \cup \overline{v^l}) \\ \text{fv}(Q) \subseteq (\overline{v} \cup \overline{v^l} \cup \{\text{result}\}) \qquad \{\text{result}, v\} \cap \text{fv}(\overline{e} \cup \overline{e^l}) = \emptyset \qquad (f, (\overline{v}), (\overline{v^l}), c) \in \text{FuncDefs}\end{array}}{\{P[\overline{e}, \overline{e^l}/\overline{v}, \overline{v^l}]\}\, v := f((\overline{e}), (\overline{e^l}))\, \{Q[\overline{e}, \overline{e^l}, v/\overline{v}, \overline{v^l}, \text{result}]\}}$$

$$\textsc{Exists}\ \frac{\{P\}\, c\, \{Q\}}{\{\exists V.\ P\}\, c\, \{\exists V.\ Q\}} \qquad \textsc{Substitution}\ \frac{\{P\}\, c\, \{Q\} \qquad \text{fv}(\overline{E^n}, \overline{E^l}) \cap \text{mod}(c) = \emptyset}{\{P[\overline{E^n}, \overline{E^l}, \overline{E^t}/\overline{V^n}, \overline{V^l}, \overline{V^t}]\}\, c\, \{Q[\overline{E^n}, \overline{E^l}, \overline{E^t}/\overline{V^n}, \overline{V^l}, \overline{V^t}]\}}$$

Here, fv of an expression or formula returns the set of free variables of the expression or formula. The frame rule is studied in [2].

We say a Hoare triple $\{P\}\, c\, \{Q\}$ is derivable, written $\vdash \{P\}\, c\, \{Q\}$, if it can be derived using the above proof rules.

Theorem 3.1 (Soundness). $\forall P, c, Q.\ \vdash \{P\}\, c\, \{Q\} \Rightarrow \models \{P\}\, c\, \{Q\}$.

A proof is future work.

4 Example

Consider the example below. The contract specifies it reads from standard input (until end-of-file), and writes what it reads to both standard output and standard error. It is written in a compositional manner: an action `tee_out` represents the action of writing to both standard output and standard error. The action that represents the whole program is built upon the `tee_out` action. The specifications allow a read-buffer of any size. The implementation chooses a read-buffer of size 2. This example is also shipped with VeriFast. At the moment of writing you can find it in `examples/io/tee/tee_buffered.c` in the VeriFast release ZIP or tar.

```
predicate tee_out(t1, c, t2) =
  split(t1, t_stdout1, t_stderr1)
  * print_char(t_stdout1, c, t_stdout2)
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
```

```
  * join(t_stdout2, t_stderr2, t2)

function tee_out(c) =
{
  time(t1) * tee_out(t1, c, t2)
}
  tmp := write_char(c);
  tmp := write_char_stderr(c)
{
  time(t2)
}

predicate reads(t1, contents, t2) =
  read(t1, c, t_read)
  * if c >= 0 then
      length(contents) > 0 * c == head(contents)
      * reads(t_read, tail(contents), t2)
    else
      t2 == t_read * contents == nil

predicate tee_out_string(t1, contents, t2) =
  if contents == nil then
    t2 == t1
  else
    tee_out(t1, head(contents), t_out)
    * tee_out_string(t_out, tail(contents), t2)

predicate tee(t1, text, t2) =
  split(t1, t_read1, t_write1)
  * reads(t_read1, text, t_read2)
  * tee_out_string(t_write1, text, t_write2)
  * join(t_read2, t_write2, t2)

function main() =
{
  time(t1) * tee(t1, text, t2)
}
  c2 := 0;
  while (c2 >= 0) (
    c1 := read_char();
    if (c1 >= 0) (
      c2 := read_char();
      tmp := tee_out(c1);
      if (c2 >= 0)(
        tmp := tee_out(c2)
      )
    ) else (
      c2 := -1
    )
  )
{
  time(t2)
}
```

We give a proof outline. We start with the function `tee_out`.

```
{time(t1) * tee_out(t1, c, t2)}
```

```
{time(t_stdout1) * time(t_stderr1))
  * print_char_io(t_stdout1, c, t_stdout2)
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
  * join(t_stdout2, t_stderr2, t2)
}
write_char(c);
{time(t_stdout2) * time(t_stderr1))
  * print_char_stderr_io(t_stderr1, c, t_stderr2)
  * join(t_stdout2, t_stderr2, t2)
}
write_char_stderr(c)
{time(t_stdout2) * time(t_stderr2)) * join(t_stdout2, t_stderr2, t2)}
{time(t2)}
```

Before looking at the main function, we define a helper predicate:

```
predicate invariant(c2, t2) =
 if c2 >= 0 then
   time(t_r1) * reads(t_r1, text, t_r2)
   * time(t_w1) * tee_out_string(t_w1, text, t_w2)
   * join(t_r2, t_w2, t2)
 else
   time(t2)
```

Next, we give a proof outline for the main function:

```
{ time(t1) * tee(t1, text, t2) }
{ 0 = 0 * time(t1) * tee(t1, text, t2)}
c2 := 0;
{ invariant(c2, t2) }
while (c2 >= 0) (
  {
    c1' = c1' * time(t_r1) * read(t_r1, c1', t_rb1) *
    if c1' >= 0 then ( length(text') > 0 * c1' = head(text') * reads(t_rb1, tail(text'), t_r2) )
        else ( t_r2 = t_rb1 * text' = nil )
    * time(t_w1) * tee_out_string(t_w1, text', t_w2)
    * join(t_r2, t_w2, t2)
  }
  c1 := read_char();
  {
    time(t_rb1) *
    if c1 >= 0 then ( length(text') > 0 * c1 = head(text') * reads(t_rb1, tail(text'), t_r2) )
      else ( t_r2 = t_rb1 * text' = nil )
    * time(t_w1) * tee_out_string(t_w1, text', t_w2)
    * join(t_r2, t_w2, t2)
  }
  if (c1 >= 0) (
    {
      c2' = c2' * time(t_rb1) * read(t_rb1, c2', t_rb2)
      * length(text') > 0 * c1 = head(text')
      * if c2' >= 0 then ( length(tail(text')) > 0 * c2' = head(tail(text'))
          * reads(t_rb2, tail(tail(text')), t_r2) )
        else ( t_r2 = t_rb2 * tail(text') = nil )
      * time(t_w1) * tee_out_string(t_w1, text', t_w2)
      * join(t_r2, t_w2, t2)
    }
    c2 := read_char();
    {
```

```
    time(t_rb2) * length(text') > 0 * c1 = head(text')
    * if c2 >= 0 then ( length(tail(text')) > 0 * c2 = head(tail(text'))
        * reads(t_rb2, tail(tail(text')), t_r2) )
      else t_r2 = t_rb2 * tail(text') = nil
    * time(t_w1) * tee_out(t_w1, head(text'), t_wb1)
    * tee_out_string(t_wb1, tail(text'), tw2)
    * join(t_r2, t_w2, t2)
  }
  tmp := tee_out(c1);
  {
    time(t_rb2) * length(text') > 0 * c1 = head(text')
    * if c2 >= 0 then ( length(tail(text')) > 0 * c2 = head(tail(text'))
        * reads(t_rb2, tail(tail(text')), t_r2) )
      else ( t_r2 = t_rb2 * tail(text') = nil )
    * time(t_wb1) * tee_out_string(t_wb1, tail(text'), tw2)
    * join(t_r2, t_w2, t2)
  }
  if (c2 >= 0)(
    {
      * time(t_rb2) * reads(t_rb2, tail(tail(text')), t_r2)
      * length(text') > 1 * c1 = head(text') * c2 = head(tail(text'))
      * time(t_wb1) * tee_out(t_wb1, head(tail(text')), t_wb2)
      * tee_out_string(t_wb2, tail(tail(text')), t_w2)
      * join(t_r2, t_w2, t2)
    }
    tmp := tee_out(c2);
    {
      * time(t_rb2) * reads(t_rb2, tail(tail(text')), t_r2)
      * length(text') > 1 * c1 = head(text') * c2 = head(tail(text'))
      * time(t_wb2) * tee_out_string(t_wb2, tail(tail(text')), t_w2)
      * join(t_r2, t_w2, t2)
    }
  ) else (
    { c2 < 0 * time(t2) }
  )
  { invariant(c2, t2) }
) else (
  { -1 = -1 * time(t2) }
  c2 := -1;
  { invariant(c2, t2) }
)
{ invariant(c2, t2) }
)
{ time(t2) }
```


Acknowledgements

We would like to thank Amin Timany for many useful discussions. This work was partially funded by EU FP7 FET-Open project ADVENT under grant number 308830.

References

[1] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.

[2] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, Washington, 2002. IEEE.

Shared Boxes: Rely-Guarantee Reasoning in VeriFast

Jan Smans Dries Vanoverberghe Dominique Devriese
Bart Jacobs Frank Piessens
iMinds-DistriNet, Dept. Comp. Sci., KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

VeriFast is a verifier for single-threaded and multithreaded C and Java programs. It takes a C or Java program annotated with preconditions and postconditions in a separation logic notation, and verifies statically that these preconditions and postconditions hold, using symbolic execution. In plain separation logic, a thread either has full ownership of a memory location and knows the value at the location, or it has no ownership and no knowledge of the value of the location. Existing work proposes a marriage of rely-guarantee reasoning and separation logic to address this. In this document, we describe the shared boxes mechanism, which marries separation logic and rely-guarantee reasoning in VeriFast.

We introduce and motivate the shared boxes mechanism using a minimalistic example and a realistic example. The minimalistic example is a *counter* program where one thread continuously increments a counter and other threads check that the counter does not decrease. For the realistic example, we verify functional correctness of the Michael-Scott queue, a lock-free concurrent data structure. We define the syntax and semantics of a simple C-like programming language, and we define a separation logic with shared boxes and prove its soundness. We discuss the implementation in VeriFast and the examples we verified using our VeriFast implementation.

1 A Minimalistic Example

Consider the following program:

$$\begin{array}{l}\mathsf{c} := \mathbf{cons}(0);\\ \mathbf{fork}\ (\mathbf{while\ true\ do}\ \langle \mathsf{n} := [\mathsf{c}]; [\mathsf{c}] := \mathsf{n}+1\rangle);\\ \mathbf{while\ true\ do\ fork}\ (\langle \mathsf{m} := [\mathsf{c}]\rangle; \langle \mathsf{m}' := [\mathsf{c}]\rangle; \mathbf{assert}\ \mathsf{m} \leq \mathsf{m}')\end{array}$$

We use the usual notation for heap-manipulating programs from the separation logic literature. The command $x := \mathbf{cons}(\overline{e})$ allocates a sequence of consecutive heap locations and initializes them with the values of the expressions $\overline{e}$; the example allocates a single cell (which we will refer to as the *counter cell*), and initializes it to zero. We use the following notation for concurrent programming: command **fork** c executes command c in a new thread; $\langle c \rangle$ denotes atomic execution of command c.

After allocating the counter cell and storing its address in variable `c`, the program forks one thread that repeatedly atomically increments the cell, and an unbounded number of threads that inspect the cell twice and assert that its value increases monotonically.

We wish to prove that the **assert** command never fails. Notice that this program cannot be verified in plain separation logic, since no single thread can be the exclusive owner of the counter cell. Concurrent Separation Logic (CSL), which extends separation logic with support for critical sections accessing shared resources with resource invariants, also does not support this example directly, since a resource invariant could only state that the cell's value is nonnegative and could not describe the *evolution* of the counter cell's value.

A combination of CSL and *ghost cells* with *fractional permissions* [1, 6] could verify this example, using a dynamic form of the resource invariant-based Owicki-Gries method [8, 6] for reasoning about concurrent programs: the shared resource containing the counter cell could be extended with a *ghost linked list* built from ghost cells. Each inspector thread, during its first inspection of the counter cell value, would add a ghost node to the end of this list, containing as its value the observed counter cell value. The thread would retain in its local state a fraction of the linked list, from the head up to the thread's node. The resource invariant would state that each node value is a lower bound for the current counter cell value. Upon the second inspection, the thread could match up the linked list in the resource with its local knowledge and conclude that the new counter cell value must be at least the previously observed value.

Such building of *ghost objects* probably yields a complete proof system. Still, in the present document, we present an alternative approach, which allows the proof author to express his insights more conveniently and more directly: shared boxes.

A shared box can be thought of at a high level as a shared resource from CSL equipped with a *two-state invariant* (or, equivalently, a *rely condition*) instead of a regular single-state invariant. This enables the proof author to express directly any desired constraints on the evolution of the shared resource. Furthermore, we allow assertions in thread proof outlines to include *shared box assertions*, assertions about the state of the shared resource that are checked to be *stable* with respect to the shared box's rely condition.

To verify the example, we put the counter cell in a shared box whose rely condition states that the cell's value may not decrease. Each inspector thread's proof outline, between the two inspections, includes a shared box assertion stating that the counter cell's value is bounded below by the first observed value.

This general idea is very similar to what has been proposed before (e.g. [9, 4, 3, 2]). However, in order to integrate this mechanism conveniently into our VeriFast verification tool, we have made a number of design decisions:

- Shared boxes can be created dynamically, but each shared box must be an instance of a statically declared *box class*.
- A box class has a *name* and a *parameter list*.
- A box class's rely condition is specified in the form of the combination of a *box invariant* and a set of *action specifications*.
- A box invariant is a VeriFast separation logic assertion that may use the

box class parameters and may bind additional logical variables, together with the box class parameters called the *box state variables*.

- An action specification consists of an action name, a parameter list, a precondition, and a postcondition. The precondition is a boolean (i.e. pure, non-spatial) expression over the action parameters and the box state variables. The postcondition is a boolean expression over the action parameters and two versions of the box state variables: the old versions and the new versions.
- Whenever a thread mutates the resources held by a box, it must specify an action name and action arguments, and VeriFast checks that the operation complies with the action precondition and postcondition.
- Shared box assertions are expressed as *box handle predicate assertions*, referring to one of a set of *box handle predicates* (or *handle predicates* for short) declared as part of the box class. A handle predicate declaration consists of a name, a parameter list, and a *handle predicate invariant*, which is a boolean expression over the handle parameters and the box state variables.
- Each handle predicate declaration must include a *preserved-by clause* for each box class action, which may state any ghost commands (such as lemma invocations) required to establish that the handle predicate invariant is preserved by the action.

We formalize the syntax of our proof system. Let $B \in \mathcal{B}$ range over box class names, $A \in \mathcal{A}$ over action names, $P \in \mathcal{P}$ over handle predicate names, $x \in \mathcal{X}$ over program variable names, and $X \in \mathcal{L}$ over logical variable names. The syntax of box classes, assertions a, and program commands c is as follows:

$$
\begin{array}{rl}
\mathit{boxClass} ::= & \textbf{boxclass}\ B(\overline{X})\ \{\ \textbf{inv}\ \exists \overline{X}.\ a\ \overline{\mathit{actionSpec}}\ \overline{\mathit{handlePred}}\ \} \\
\mathit{actionSpec} ::= & \textbf{action}\ A(\overline{X})\ \textbf{req}\ b\ \textbf{ens}\ b \\
\mathit{handlePred} ::= & \textbf{handlePred}\ P(\overline{X})\ \{\ \textbf{inv}\ b\ \} \\
e ::= & z \mid x \mid X \mid e + e \mid e - e \\
b ::= & e = e \mid e < e \mid b \wedge b \mid \neg b \\
f ::= & e/e \\
a ::= & b \mid a \vee a \mid \exists X.\ a \mid e \overset{f}{\mapsto} e \mid a * a \mid [f]B(e, \overline{e}) \mid P(e, e, \overline{e}) \\
c ::= & x := e \mid c; c \mid \textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c \mid \textbf{while}\ b\ \textbf{do}\ c \\
& \mid x := \textbf{cons}(\overline{e}) \mid x := [e] \mid [e] := e \mid \langle c \rangle \mid \textbf{fork}\ c
\end{array}
$$

We assume that logical variables X do not appear inside program commands c. Notice that we do not formalize preserved-by clauses; we will formalize the stability constraints but leave the mechanism for proving them unspecified. Our proof system supports fractional permissions f on points-to assertions $e \overset{f}{\mapsto} e$ and *box assertions* $[f]B(e, \overline{e})$. The first argument of a box assertion is the *box identifier*; the first argument of a *handle predicate assertion* $P(e, e, \overline{e})$ is the *handle identifier*, and the second argument is the box identifier.

The box class declaration and proof outline for the example program are shown in Figure 1. Notice that in the formal system, in an action postcondition, we use unprimed versions of the box invariant variables to denote the old values, and primed versions to denote the new values.

```
boxclass incrbox(c) {
  inv ∃v. c ↦ v
  action incr() req true ens v ≤ v′
  handlePred observed(val) { inv val ≤ v }
}

{true}
c := cons(0);
{c ↦ 0}
{[1]incrbox(_, c)}   CreateBox
fork (
  {[_]incrbox(_, c)}
  while true do
    ⟨n := [c]; [c] := n + 1⟩   Action incr()
);
{[_]incrbox(_, c)}
while true do (
  ⟨m := c⟩;
  {∃b. [_]incrbox(b, c) ∗ observed(_, b, m)}
  ⟨m′ := c⟩;
  {[_]incrbox(_, c) ∗ m ≤ m′}
  assert m ≤ m′
)
```

Figure 1: Proof of the minimalistic example program

$$\textsc{StablePred}\ \frac{\begin{array}{l}\textbf{boxclass}\ B(\overline{X})\ \{\\ \quad \textbf{inv}\ \exists\overline{Y}.\ I\\ \quad \cdots\\ \quad \textbf{action}\ A(\overline{Z})\ \textbf{req}\ b\ \textbf{ens}\ b'\\ \quad \cdots\\ \quad \textbf{handlePred}\ P(\overline{U})\ \{\ \textbf{inv}\ b''\ \}\\ \quad \cdots\\ \}\end{array} \qquad \begin{array}{l}\forall \overline{v}, \overline{w}, \overline{w'}, \overline{z}, \overline{u}, r, r'.\\ \quad (r \vDash (I \wedge b'')[\overline{vwu}/\overline{XYU}]) \wedge\\ \quad (r' \vDash I[\overline{v}\overline{w'}/\overline{XY}]) \wedge\\ \quad (b \wedge b')[\overline{vwz}\overline{w'}/\overline{XYZY'}]\\ \quad \Rightarrow\ b''[\overline{v}\overline{w'}\overline{u}/\overline{XYU}]\end{array}}{A \vdash \mathsf{stable}\ P}$$

$$\textsc{Stable}\ \frac{\textbf{boxclass}\ B(\overline{X})\ \{\ \cdots\ \overline{\textbf{action}\ A_{1..n}\ \cdots}\ \overline{\textbf{handlePred}\ P_{1..m}\ \cdots}\ \}\qquad \forall i, j.\ A_i \vdash \mathsf{stable}\ P_j}{\mathsf{stable}\ B}$$

$$\textsc{CreateBox}\ \frac{\textbf{boxclass}\ B(\overline{X})\ \{\ \textbf{inv}\ \exists\overline{Y}.\ I\ \cdots\ \}\qquad \{[1]B(\_, \overline{v}) * R\}\ c\ \{Q\}}{\{I[\overline{vw}/\overline{XY}] * R\}\ c\ \{Q\}}$$

$$\textsc{Action}\ \frac{\begin{array}{l}\textbf{boxclass}\ B(\overline{X})\ \{\\ \quad \textbf{inv}\ \exists\overline{Y}.\ I\\ \quad \cdots\\ \quad \textbf{action}\ A(\overline{Z})\ \textbf{req}\ b\ \textbf{ens}\ b'\\ \quad \cdots\\ \quad \textbf{handlePred}\ P_i(\overline{U}_i)\ \{\ \textbf{inv}\ b_i\ \}\\ \}\\ \{I[\overline{vw}/\overline{XY}] * \Pi_i b_i[\overline{vwu}_i/\overline{XYU}_i] * R\}\\ \forall \overline{w}.\ \ c\\ \quad \{\exists\overline{w'}.\ I[\overline{v}\overline{w'}/\overline{XY}] * \Pi_j(b_j'' \vee b_j'[\overline{v}\overline{w'}\overline{u}_j'/\overline{XYU}_j'\,]) * (b \wedge b')[\overline{vw}\overline{w'}\overline{z}/\overline{XYY'Z}] * R'\}\end{array}}{\{[\pi]B(\beta, \overline{v}) * \Pi_i P_i(\_, \beta, \overline{u}_i) * R\}\ \langle c\rangle\ \{[\pi]B(\beta, \overline{v}) * \Pi_j(b_j'' \vee P_j'(\_, \beta, \overline{u}_j')) * R'\}}$$

Figure 2: Proof rules

Our proof system extends separation logic with extra rules for box class stability checking, box creation, and shared box mutation. The extra rules are shown in Figure 2. Note: there are restrictions on nested applications of the ACTION rule. For now, we assume that no such nested applications occur.

We assume that each declared box class B is stable: stable B. This means that each of the class' handle predicates is stable with respect to each of its actions. Stability of a handle predicate with respect to an action means that given arbitrary values $\overline{v}$ of the box class parameters, old values $\overline{w}$ and new values $\overline{w'}$ of the box invariant variables, values $\overline{z}$ of the action parameters, and values $\overline{u}$ of the handle predicate parameters, and arbitrary old and new resource bundles r and r' representing the old and new contents of the shared box, if the box invariant holds in the pre- and post-state, the action pre- and postcondition hold, and the handle predicate invariant holds in the pre-state, then the handle predicate invariant holds in the post-state.

Rule CREATEBOX allows a shared box instance to be created at any time provided its invariant holds for some part of the current locally held resources. Those resources are then consumed and a *box chunk* (a resource representing the existence of a shared box instance) is produced.

Rule ACTION allows the verification of an atomic command $\langle c \rangle$ that accesses the resources held by a shared box instance of class B, with identifier β, and with arguments $\overline{v}$. This requires that the thread own a fraction π of the box chunk. During verification of command c, the box invariant becomes available. It must be re-established before the atomic command is exited. Furthermore, local resources R may be passed into the atomic command and resources R' may be extracted and retained locally. Also, handle predicates may be consumed and produced. Any number of handle predicate chunks P_i for the box instance may be consumed on entry; their invariants b_i are assumed to hold in the pre-state. Any other number of handle predicates P'_j may be produced on exit, provided their invariants b'_j are established in the post-state. They may be produced conditionally under conditions $\neg b''_j$. It is checked that there is some action A and argument list $\overline{z}$ such that the action's precondition and postcondition are satisfied by the command.

In rule ACTION as presented in Figure 2, handle identifiers are ignored. Handle identifiers are important in advanced scenarios which will be discussed in a later section.

We show the example proof in the form of an annotated C program, as accepted and successfully verified by VeriFast, in Figure 3. This example is included in the VeriFast distribution in file `examples/shared_boxes/incrbox.c`.

2 Soundness Proof

In this section, we formalize the soundness property targeted by our proof system and then we prove it.

2.1 Operational Semantics

We first formalize a small-step operational semantics for our programming language.

```
#include <threading.h>
#include "atomics.h"
/*@ box_class incr_box(int *x) {
    invariant *x |-> ?value;
    action increase();
        requires true;
        ensures old_value <= value;
    handle_predicate observed(int v) {
        invariant v <= value;
        preserved_by increase() {}
    }
} @*/
//@ predicate_family_instance thread_run_data(inc)(int* x) = [_]incr_box(_, x);
void inc(int *x) /*@ : thread_run @*/
    /*@ requires thread_run_data(inc)(x); @*/ /*@ ensures true; @*/ {
    //@ open thread_run_data(inc)(x);
    while(true) /*@ invariant [_]incr_box(_, x); @*/ {
        ;
        /*@
        consuming_box_predicate incr_box(_, x)
        perform_action increase()
        { @*/ atomic_increment(x); /*@ };
        @*/
    }
}
void reader(int *x) /*@ requires [_]incr_box(_, x); @*/ /*@ ensures false; @*/ {
    for (;;) /*@ invariant [_]incr_box(_, x); @*/ {
        ;
        /*@
        consuming_box_predicate incr_box(_, x)
        perform_action increase()
        { @*/ int m0 = atomic_load_int(x); /*@ }
        producing_fresh_handle_predicate observed(m0);
        @*/
        /*@
        consuming_box_predicate incr_box(_, x)
        consuming_handle_predicate observed(_, m0)
        perform_action increase()
        { @*/ int m1 = atomic_load_int(x); /*@ };
        @*/
        assert(m0 <= m1);
    }
}
int main() /*@ requires true; @*/ /*@ ensures true; @*/ {
    int x;
    //@ create_box id = incr_box(&x);
    //@ leak incr_box(id, &x);
    //@ close thread_run_data(inc)(&x);
    thread_start(inc, &x);
    reader(&x);
}
```

Figure 3: The minimalistic example as an annotated C program accepted by VeriFast.

The set of machine configurations $\gamma \in \mathit{Configs}$ is defined as follows:

$$\begin{aligned} s \in \mathit{Stores} &= \mathcal{X} \to \mathbb{Z} \\ R \in \mathit{Heaps} &= \mathbb{Z} \rightharpoonup \mathbb{Z} \\ \kappa \in \mathit{Continuations} &::= \mathbf{done} \mid c; \kappa \\ \theta \in \mathit{ThreadConfigs} &= \mathit{Stores} \times \mathit{Continuations} \\ \gamma \in \mathit{Configs} &= \mathit{Heaps} \times (\mathit{ThreadConfigs} \to \mathbb{N}) \end{aligned}$$

A configuration consists of a heap and a multiset[1] of thread configurations. A thread configuration θ consists of a store and a continuation. A continuation is either **done**, indicating that the thread has finished, or a command followed by another continuation.

We use the notation $\{\!\{a, b, c\}\!\}$ to represent a multiset: $\{\!\{a_1, \ldots, a_n\}\!\} = \mathbf{0} + \{\!\{a_1\}\!\} + \cdots + \{\!\{a_n\}\!\}$ where $\mathbf{0} = \lambda x.\, 0$ represents the empty multiset and $M + \{\!\{a\}\!\} = M[a := M(a) + 1]$. We use the notations $+$ and $\uplus$ interchangeably for multiset addition.

We define a big-step relation $\Downarrow \subseteq (\mathit{Heaps} \times \mathit{Stores} \times \mathit{Commands}) \times (\mathit{Heaps} \times \mathit{Stores} \cup \{\mathbf{abort}\})$ for commands that may appear inside atomic commands:

Assign
$$(R, s, x := e) \Downarrow (R, s[x := s(e)])$$

Lookup
$$\frac{s(e) \in \operatorname{dom} R}{(R, s, x := [e]) \Downarrow (R, s[x := R(s(e))])}$$

LookupAbort
$$\frac{s(e) \notin \operatorname{dom} R}{(R, s, x := [e]) \Downarrow \mathbf{abort}}$$

Mutate
$$\frac{s(e) \in \operatorname{dom} R}{(R, s, [e] := e') \Downarrow (R[s(e) := s(e')], s)}$$

MutateAbort
$$\frac{s(e) \notin \operatorname{dom} R}{(R, s, [e] := e') \Downarrow \mathbf{abort}}$$

SeqAtomic
$$\frac{(R, s, c) \Downarrow (R', s') \qquad (R', s', c') \Downarrow o}{(R, s, c; c') \Downarrow o}$$

SeqAbort
$$\frac{(R, s, c) \Downarrow \mathbf{abort}}{(R, s, c; c') \Downarrow \mathbf{abort}}$$

IfTrueAtomic
$$\frac{s(b) \qquad (R, s, c) \Downarrow o}{(R, s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') \Downarrow o}$$

IfFalseAtomic
$$\frac{\neg s(b) \qquad (R, s, c') \Downarrow o}{(R, s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') \Downarrow o}$$

AtomicAtomic
$$\frac{(R, s, c) \Downarrow o}{(R, s, \langle c \rangle) \Downarrow o}$$

[1] A multiset over elements of a set A is a function $M : A \to \mathbb{N}$ where $M(a)$ is the number of occurrences of a in M.

We define a small-step relation $\rightsquigarrow \subseteq \mathit{Configs} \times (\mathit{Configs} \cup \{\mathbf{abort}\})$ as follows:

Cons
$$\frac{0 < \ell \qquad \{\ell, \ldots, \ell + n - 1\} \cap \mathrm{dom}(R) = \emptyset \qquad R' = R[\ell := s(e_1), \ldots, \ell + n - 1 := s(e_n)]}{(R, \{\!|(s, x := \mathbf{cons}(e_1, \ldots, e_n); \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R', \{\!|(s[x := \ell], \kappa)|\!\} \uplus \Theta)}$$

Seq
$$(R, \{\!|(s, (c; c'); \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, c; (c'; \kappa))|\!\} \uplus \Theta)$$

IfTrue
$$\frac{s(b)}{(R, \{\!|(s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, c; \kappa)|\!\} \uplus \Theta)}$$

IfFalse
$$\frac{\neg s(b)}{(R, \{\!|(s, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, c'; \kappa)|\!\} \uplus \Theta)}$$

WhileTrue
$$\frac{s(b)}{(R, \{\!|(s, \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, c; \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)|\!\} \uplus \Theta)}$$

WhileFalse
$$\frac{\neg s(b)}{(R, \{\!|(s, \mathbf{while}\ b\ \mathbf{do}\ c; \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, \kappa)|\!\} \uplus \Theta)}$$

Fork
$$\frac{}{(R, \{\!|(s, \mathbf{fork}\ c; \kappa)|\!\} \uplus \Theta) \rightsquigarrow (R, \{\!|(s, c; \mathbf{done}), (s, \kappa)|\!\} \uplus \Theta)}$$

Atomic
$$\frac{(R, s, c) \Downarrow (R', s')}{(R, \{\!|(s, \langle c \rangle; \kappa)\rangle|\!\} \uplus \Theta) \rightsquigarrow (R', \{\!|(s', \kappa)|\!\} \uplus \Theta)}$$

AtomicAbort
$$\frac{(R, s, c) \Downarrow \mathbf{abort}}{(R, \{\!|(s, \langle c \rangle; \kappa)\rangle|\!\} \uplus \Theta) \rightsquigarrow \mathbf{abort}}$$

2.2 Meaning of Assertions

The assertions of our proof system denote *resource bundles* $r \in \mathcal{R}$:

$$\begin{array}{rl} & \ell, v, \beta, h, u \in \mathbb{Z} \\ \alpha \in \mathit{Chunks} ::= & \ell \mapsto v \mid B(\beta, \overline{v}) \mid P(h, \beta, \overline{u}) \\ r \in \mathcal{R} = & \mathit{Chunks} \to [0, 1] \end{array}$$

A *chunk* is either a points-to chunk $\ell \mapsto v$, a box chunk $B(\beta, \overline{v})$, or a handle predicate chunk $P(h, \beta, \overline{u})$. A resource bundle is a function from chunks to real numbers between 0 and 1, inclusive. Note that the heaps $R \in \mathit{Heaps}$ can be identified with a subset of the resource bundles: we will identify heap R with the resource bundle $\{(\ell \mapsto v) \mapsto 1 \mid (\ell \mapsto v) \in R\}$.

We define *consistency* of a resource bundle as follows:

$$\frac{\begin{array}{c} \forall \ell, v_1, v_2.\ r(\ell \mapsto v_1) > 0 \wedge r(\ell \mapsto v_2) > 0 \Rightarrow v_1 = v_2 \\ \forall B, \beta, \overline{v}_1, \overline{v}_2.\ r(B(\beta, \overline{v}_1)) > 0 \wedge r(B(\beta, \overline{v}_2)) > 0 \Rightarrow \overline{v}_1 = \overline{v}_2 \\ \forall P, h, \beta_1, \overline{u}_1, \beta_2, \overline{u}_2.\ r(P(h, \beta_1, \overline{u}_1)) > 0 \wedge r(P(h, \beta_2, \overline{u}_2)) > 0 \Rightarrow \beta_1 = \beta_2 \wedge \overline{u}_1 = \overline{u}_2 \end{array}}{\mathsf{consistent}\ r}$$

We define satisfaction $r \vDash a$ of a closed assertion in the obvious way. We say a implies a' iff $\forall r.\ r \vDash a \Rightarrow r \vDash a'$.

2.3 Soundness Property

The soundness property that we target with our proof system is that if $\{\mathbf{true}\}\ c\ \{\mathbf{true}\}$ then $(\emptyset, \{\!|(\mathbf{0}, c; \mathbf{done})|\!\}) \not\rightsquigarrow^* \mathbf{abort}$.

2.4 Soundness with respect to the Big-Step Semantics

We extend the big-step semantics to operate on resource bundles as follows:

$$\frac{r + r_{\mathrm{F}} = R + r_{\mathrm{R}} \qquad (R, s, c) \Downarrow (R', s') \qquad R' + r_{\mathrm{R}} = r' + r_{\mathrm{F}}}{(r, s, c) \Downarrow (r', s')}$$

$$\frac{r + r_{\mathrm{F}} = R + r_{\mathrm{R}} \qquad (R, s, c) \Downarrow \mathbf{abort}}{(r, s, c) \Downarrow \mathbf{abort}}$$

Separation logic is sound with respect to the big-step semantics:

Lemma 1. *If $\{a\}\ c\ \{a'\}$ was derived without using the* CreateBox *or* Action *rules, and $r, s \vDash a$ and $(r, s, c) \Downarrow o$, then $\exists r', s'.\ o = (r', s') \wedge r', s' \vDash a'$.*

Proof. By induction on the derivation of the Hoare triple. □

2.5 Safety Relation

We define the *semantic assertions* $SemAsns = 2^{\mathcal{R} \times Stores}$.

We define a safety relation $\mathsf{safe} \subseteq Commands \times SemAsns \times \mathcal{R} \times Stores$ in Figure 4.

We prove a correspondence between correctness and safety of a command. (We prove a generalized property, involving a frame r and a weakened postcondition Q; this makes the induction hypothesis strong enough for the frame rule and the rule of consequence.)

Lemma 2. *If $\{a\}\ c\ \{a'\}$ and $a' * r \Rightarrow Q$, then $a * r \Rightarrow \mathsf{safe}(c, Q)$.*

Proof. By induction on the derivation. □

We define safety of a continuation:

$$\mathsf{safe}(\mathbf{done}, r, s) \qquad \frac{\mathsf{safe}(c, \mathsf{safe}(\kappa), r, s)}{\mathsf{safe}(c; \kappa, r, s)}$$

We define safety of a machine configuration:

$$\frac{\begin{array}{c}\iota_{\mathrm{B}} : \mathbb{Z} \rightharpoonup_{\mathrm{fin}} \mathcal{B} \times \mathbb{Z}^* \\ \iota_{\mathrm{P}} : \mathbb{Z} \rightharpoonup_{\mathrm{fin}} \mathcal{P} \times \mathbb{Z} \times \mathbb{Z}^* \qquad \rho : \mathrm{dom}\ \iota_{\mathrm{B}} \to \mathcal{R} \qquad r = R \uplus \iota_{\mathrm{B}} \uplus \iota_{\mathrm{P}} \\ r = \Sigma_\beta \rho(\beta) + \Sigma_i r_i \qquad \forall (\beta \mapsto (B, \overline{v}_\beta)) \in \iota_{\mathrm{B}}.\ \rho(\beta) \vDash I_B[\overline{v}_\beta \overline{w}_\beta / \overline{X}_B \overline{Y}_B] \\ \forall (h \mapsto (P, \beta, \overline{u})) \in \iota_{\mathrm{P}}.\ b_P[\overline{v}_\beta \overline{w}_\beta \overline{u} / \overline{X}_\beta \overline{Y}_\beta \overline{U}_P] \qquad \forall i.\ r_i, s_i \vDash \mathsf{safe}(\kappa_i, \mathbf{true})\end{array}}{\mathsf{safe}\,(R, \Sigma_i \{\!|(s_i, \kappa_i)|\!\})}$$

$$\frac{Q(r, s[x := s(e)])}{\mathsf{safe}(x := e, Q, r, s)}$$

LOOKUP
$$\frac{r \vDash s(e) \overset{\pi}{\mapsto} v \qquad Q(r, s[x := v])}{\mathsf{safe}(x := [e], Q, r, s)}$$

MUTATE
$$\frac{r, s \vDash e \mapsto v * (e \mapsto e' \mathrel{-\!\!*} Q)}{\mathsf{safe}([e] := e', Q, r, s)}$$

ATOMICNOBOX
$$\frac{\forall o.\ (r, s, c) \Downarrow o \Rightarrow \exists r', s'.\ o = (r', s') \wedge Q(r', s')}{\mathsf{safe}(c, Q, r, s)}$$

ATOMICBOX
$$\frac{\begin{array}{l}\textbf{boxclass } B(\overline{X}) \ \{ \\ \quad \textbf{inv } \exists \overline{Y}.\ I \\ \quad \cdots \\ \quad \textbf{action } A(\overline{Z}) \textbf{ req } b \textbf{ ens } b' \\ \quad \cdots \\ \quad \textbf{handlePred } P_i(\overline{U}_i) \ \{ \textbf{ inv } b_i \ \} \\ \} \\ r, s \vDash [\pi] B(\beta, \overline{v}) * \Pi_i P_i(\_, \beta, \overline{u}_i) * \\ \quad (\forall \overline{w}. \\ \quad\quad I[\overline{vw}/\overline{XY}] * \Pi_i b_i[\overline{vwu}_i/\overline{XYU}_i] \ \mathrel{-\!\!*} \\ \quad\quad \mathsf{safe}(c, \exists \overline{w'}. \\ \quad\quad\quad I[\overline{vw'}/\overline{XY}] * \Pi_j (b''_j \vee b'_j[\overline{vw'u}'_j/\overline{XYU}'_j]) \\ \quad\quad\quad * (b \wedge b')[\overline{vww'z}/\overline{XYY'Z}] \\ \quad\quad\quad * ([\pi] B(\beta, \overline{v}) * \Pi_j (b''_j \vee P'_j(\_, \beta, \overline{u}'_j)) \mathrel{-\!\!*} Q)))\end{array}}{\mathsf{safe}(\langle c \rangle, Q, r, s)}$$

CREATEBOX
$$\frac{r, s \vDash I[\overline{vw}/\overline{XY}] * ([1] B(\_, \overline{v}) \mathrel{-\!\!*} Q)}{\mathsf{safe}(c, Q, r, s)}$$

Figure 4: Safety of a command

In words: a machine configuration is safe if there exists a set of box instances (with a box identifier, a box class, values for the box parameters, and values for the box invariant variables) and a set of handle predicate instances (with a handle predicate name, a handle identifier, a box identifier, and a set of handle predicate arguments) such that there exists a partitioning of the available resources (i.e. one points-to chunk for each heap cell plus one box chunk for each box instance plus one handle chunk for each handle) into one bundle for each box and one bundle for each thread, such that each box's bundle satisfies the box invariant and all handle predicate invariants pertaining to it, and each thread's bundle ensures the safety of that thread's continuation.

Lemma 3. *Safety of a machine configuration is preserved by machine steps:*

$$\mathsf{safe}\,\gamma \wedge \gamma \rightsquigarrow o \Rightarrow \exists \gamma'.\ o = \gamma' \wedge \mathsf{safe}\,\gamma'$$

Proof. By induction on the derivation of $\mathsf{safe}(\kappa_i, \mathbf{true}, r_i, s_i)$ for the thread i that performs the step. We elaborate an illustrative case.

- **Case** CreateBox. The thread's bundle r_i can be split into a part r_I that satisfies the resource invariant of some box class B, and a residue r'. We pick a new box identifier β and extend ι_{B} with the new box instance. We define the new bundle for thread i as $r_i' = r' + \{\!\!\{B(\beta, \overline{v})\}\!\!\}$. Since no handles have β as their box identifier, all constraints are satisfied. We finish by applying the induction hypothesis.

□

Theorem 1 (Soundness)**.** *If* $\{\mathbf{true}\}\ c\ \{\mathbf{true}\}$ *then* $(\emptyset, \{\!\!\{(\mathbf{0}, \kappa)\}\!\!\}) \not\rightsquigarrow^* \mathbf{abort}$.

Proof. The initial configuration is safe. We can derive by induction on the number of steps that any reachable outcome is a safe configuration. □

3 Verifying the Michael-Scott Queue

We show an encoding of the Michael-Scott queue concurrent data structure (for a garbage-collected language) into our formal syntax in Figure 5.

We wish to verify this implementation against the following specification:

$$\{I(\epsilon)\}\ q := \mathsf{create}()\ \{\mathsf{queue}(1, q, I)\}$$

$$\frac{\forall \alpha.\ \{I(\alpha) * P\}\ \rho\ \{I(\alpha \cdot v) * Q\}}{\{\mathsf{queue}(\pi, q, I) * P\}\ \mathsf{enqueue}(q, v, \rho)\ \{\mathsf{queue}(\pi, q, I) * Q\}}$$

$$\frac{\{I(\epsilon) * P\}\ \rho\ \{I(\epsilon) * Q(0)\} \qquad \forall v, \alpha.\ \{I(v \cdot \alpha) * P\}\ \rho'\ \{I(\alpha) * Q(v)\}}{\{\mathsf{queue}(\pi, q, I) * P\}\ x := \mathsf{dequeue}(q, \rho, \rho')\ \{\mathsf{queue}(\pi, q, I) * Q(x)\}}$$

These specifications are similar to the specification style of [6], but with some differences. When a queue is created, a *queue invariant* I, an assertion parameterized by a sequence of values, is associated with it. Upon creation of the queue, the invariant, instantiated with the empty sequence, is consumed. The client may include fractional ownership of ghost cells in this invariant to track

```
procedure create() returns (result){
  n := cons(next := 0, value := 0);
  q := cons(head := n, tail := n);
  result := q
}
procedure enqueue(q, x) {
  new := cons(next := 0, value := x);
  done := 0;
  while done = 0 do (
    ⟨t := [q.tail]⟩;
    ⟨n := [t.next]; if n = 0 then [t.next] := new⟩;
    if n = 0 then
      done := 1
    else
      ⟨t′ := [q.tail]; if t′ = t then [q.tail] := n⟩
  )
}
procedure dequeue(q) returns (result){
  done := 0;
  while done = 0 do (
    ⟨h := [q.head]⟩;
    ⟨n := [h.next]⟩;
    if n = 0 then (
      result := 0; done := 1
    ) else (
      ⟨t := [q.tail]; if t = h then [q.tail] := n⟩;
      ⟨h′ := [q.head]; if h′ = h then [q.head] := n⟩;
      if h′ = h then (
        ⟨result := [n.value]⟩;
        done := true
      )
    )
  )
}
```

Figure 5: The Michael-Scott queue

information about the contents of the queue. Therefore, when the queue is updated, these ghost cells may also need to be updated. This is made possible by allowing the proof author to associate ghost commands ρ and ρ' which update these ghost cells with calls of enqueue and dequeue. the Hoare triples for enqueue and dequeue have premises specifying the behavior of these ghost commands.

To verify the data structure, we declare the box class msqueue_box, as follows:

```
boxclass msqueue_box(q, l) {
  inv ∃i, nodes, vs, h, t.
    lseg(i, 0, nodes, vs) ∗ q.head ↦ nodes_h ∗ q.tail ↦ nodes_t
    ∗ h ≤ t ∗ |nodes| − 1 ≤ t ∗ l(vs_{h+1..|vs|})
  action enqueue(n, v)
    ens nodes′ = nodes · n ∧ vs′ = vs · v
  action dequeue()
    ens h′ = h + 1
  action move_tail()
    ens t′ = t + 1
  handlePred was_head(hd) { inv ∃j ≤ h. hd = nodes_j }
  handlePred was_head_with_succ(hd, nn) {
    inv ∃j ≤ h. hd = nodes_j ∧ nn = nodes_{j+1}
  }
  handlePred was_head_with_succ_not_tail(hd, nn) {
    inv ∃j ≤ h. hd = nodes_j ∧ nn = nodes_{j+1} ∧ j < t
  }
  handlePred node_has_value(n, v) { inv ∃j. n = nodes_j ∧ v = vs_j }
  handlePred was_tail(tn) { inv ∃j ≤ t. tn = nodes_j }
  handlePred was_tail_with_succ(tn, nn) {
    inv ∃j ≤ t. tn = nodes_j ∧ nn = nodes_{j+1}
  }
}
```

Here, we adopt three notational conventions (which have not yet been implemented in VeriFast): firstly, for each box invariant variable Y whose primed version is not mentioned in an action postcondition, that postcondition gets an additional conjunct saying $Y' = Y$; secondly, each action postcondition implicitly gets an additional disjunct saying that nothing has changed; thirdly, an action precondition that is not declared explicitly defaults to **true**.

A proof outline for the queue is shown in Figures 6 and 7.

4 Additional Features

In this section we briefly describe additional features of VeriFast's shared boxes.

4.1 Handle Identifiers

The examples we used in the preceding sections had the property that at no point in time did any thread perform a distinguished role in the protocol: all threads were subject to the same restrictions, or, in other words still, the rely condition did not mention thread identities.

```
predicate queue(f, q, I) = [f]msqueue_box(_, q, I)
procedure create() returns (result){
  {I(ϵ)}
  n := cons(next := 0, value := 0);
  q := cons(head := n, tail := n);
  result := q
  {lseg(n, 0, n, 0) * q.head ↦ n * q.tail ↦ n * I(ϵ)}
  {[1]msqueue_box(_, q, I)}   CreateBox
}
procedure enqueue(q, x) {
  {queue(f, q, I) * P}
  new := cons(next := 0, value := x);
  done := 0;
  {queue(f, q, I) * (done = 0 * new.next ↦ 0 * new.value ↦ x * P ∨ done = 1 * Q)}
  while done = 0 do (
    {[f]msqueue_box(β, q, I) * new.next ↦ 0 * new.value ↦ x * P}
    ⟨t := [q.tail]⟩;
    {[f]msqueue_box(β, q, I) * new.next ↦ 0 * new.value ↦ x * P * was_tail(_, β, t)}
    ⟨n := [t.next]; if n = 0 then ([t.next] := new; ρ)⟩;
    { [f]msqueue_box(β, q, I) * (n = 0 * Q ∨
      n ≠ 0 * new.next ↦ 0 * new.value ↦ x * P * was_tail_with_succ(_, β, t, n)) }
    if n = 0 then
      done := 1
    else
      ⟨t' := [q.tail]; if t' = t then [q.tail] := n⟩
  )
  {queue(f, q, I) * Q}
}
```

Figure 6: Proof of the Michael-Scott queue, part 1 of 2

```
procedure dequeue(q) returns (result){
  {queue(f, q, I) * P}
  done := 0;
  {queue(f, q, I) * (done = 0 * P ∨ done = 1 * Q(result))}
  while done = 0 do (
    {[f]msqueue_box(_, q, I) * P}
    ⟨h := [q.head]⟩;
    {[f]msqueue_box(β, q, I) * P * was_head(_, β, h)}
    ⟨n := [h.next]; if n = 0 then ρ⟩;
    {[f]msqueue_box(β, q, I) * (n = 0 * Q ∨ n ≠ 0 * P * was_head_with_succ(_, β, h, n))}
    if n = 0 then (
      result := 0; done := 1
    ) else (
      ⟨t := [q.tail]; if t = h then [q.tail] := n⟩;
      {[f]msqueue_box(β, q, I) * P * was_head_with_succ_not_tail(_, β, h, n)}
      ⟨h' := [q.head]; if h' = h then ([q.head] := n; ρ')⟩;
      { [f]msqueue_box(β, q, I) *
        (h = h' * (∃v. Q(v) * is_good_node(_, β, n, v)) ∨ h ≠ h' * P) }
      if h' = h then (
        ⟨result := [n.value]⟩;
        done := true
      )
    )
  )
  {queue(f, q, I) * Q(result)}
}
```

Figure 7: Proof of the Michael-Scott queue, part 2 of 2

In many other concurrent algorithms, thread identities do play a role in a rely condition. For example, in the case of a spin lock, only the thread that acquired the lock may release it (except if the thread explicitly yielded ownership of the lock to some other thread). As another example, in an algorithm that uses hazard pointers for memory reclamation, such as the Treiber stack [7], only the thread that removed a node from the data structure may deallocate it.

VeriFast supports these scenarios by associating a *handle identifier* with each handle predicate. Action specifications may specify which participants may perform the action by constraining the special variable actionHandles, which denotes the list of the handle identifiers of the handles consumed by the action. As a result, only those threads which own particular handle predicate chunks can perform certain actions.

Similarly, a handle predicate invariant may mention the handle predicate's identifier using the special variable predicateHandle.

To support stable and unique identities, for each handle predicate that is produced by an action, the proof author must specify the handle identifier (which must be the identity of one of the handles that was consumed) or else that the handle identifier should be a fresh one.

The following examples that ship with VeriFast in the `examples/shared_boxes` directory use handle identifiers:

Example	Description
`spinlock.c`	Spinlock
`ticket_lock.c`	Ticketed lock
`concurrentstack.c`	Treiber stack with hazard pointers
`cowl.c`	Copy-on-write list

4.2 Nested actions

In order to build fine-grained concurrent data structures on top of other fine-grained concurrent data structures, it is useful to be able to nest actions. Note, however, that care must be taken to deal correctly with box re-entry, i.e. performing multiple nested actions on the same box. Obviously, it would be unsound to produce the box invariant multiple times.

VeriFast supports nested actions. Box re-entry is ruled out by assigning a unique *box level* to each box (whose relationship to existing box levels may be specified by the proof author), and checking that an inner action is on a higher-level box than its outer action.

The following examples use nested actions:

Example	Description
`gotsmanlock.c`	Gotsman lock [5]
`atomic_integer.c`	Atomic integer
`spinlock_with_atomic_integer`	Spinlock on top of atomic integer
`ticketlock_with_atomic_integer`	Ticketed lock on top of atomic integer

It is important to note, however, that composing fine-grained concurrent data structures each verified using shared boxes does not always require nested actions. For example, the examples `cell_refcounted.c`, `cowl.c`, and `lcl_set.c` (a set implementation using a lock-coupling list) are built on top of `gotsmanlock.c` without the need for nested actions.

4.3 Action permissions

An alternative way to deal with algorithms where participants have distinct roles, is using *action permissions*, first introduced in CAP [3]. VeriFast supports action permissions: an action may be declared as `permbased`. As in CAP, an action permission chunk is produced when the box is created. Performing a `permbased` action requires (a fraction of) the action permission chunk.

The examples `ticketlock_cap.c` and `ticketlock_with_atomic_integer.c` use action permissions.

Both of these examples use `permbased` actions that have parameters. In this case, upon creation of the box, conceptually a distinct chunk is produced for each value of the parameter. To represent this finitely in VeriFast's symbolic heap, a *dispenser* chunk is produced which represents the action permission chunks for all parameter values except for a given list of values for which a separate action permission has been split off.

4.4 Spatial handle predicate invariants

In CAP [3], stability of a shared region assertion may depend on chunks locally held by the thread. VeriFast supports this as well by allowing spatial handle predicate invariants. One example that illustrates this is `ticketlock_cap.c`.

5 Conclusion

Through the mechanism of shared boxes, VeriFast integrates rely-guarantee reasoning into its separation logic-based program logic. We introduced the mechanism through the motivating examples of a monotonic counter and the Michael-Scott queue, formalized the proof system and sketched a soundness proof, and briefly discussed additional features and additional examples available in the VeriFast distribution. Perhaps most notably, we achieved a reasonably clean proof of a Treiber stack with hazard pointers.

Acknowledgements

This work was supported by the European Commission under EU FP7 FET-Open project ADVENT (grant number 308830).

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[2] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.

[3] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[4] Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[5] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[6] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[7] M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), 2004.

[8] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, May 1976.

[9] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

Interprocedural Shape Analysis for Effectively Cutpoint-Free Programs

J. Kreiker[1], T. Reps[2⋆], N. Rinetzky[3⋆⋆], M. Sagiv[4], R. Wilhelm[5], and E. Yahav[6]

[1] Technical University of Munich joba@model.in.tum.de
[2] University of Wisconsin reps@cs.wisc.edu
[3] Queen Mary University of London maon@eecs.qmul.ac.uk
[4] Tel Aviv University msagiv@tau.ac.il
[5] University des Saarlandes wilhelm@cs.uni-sb.de
[6] IBM T.J. Watson Research Center eyahav@us.ibm.com

Abstract. We present a framework for local interprocedural shape analysis that computes procedure summaries as transformers of procedure-local heaps (the parts of the heap that the procedure may reach). A main challenge in procedure-local shape analysis is the handling of *cutpoints*, objects that separate the input heap of an invoked procedure from the rest of the heap, which—from the viewpoint of that invocation—is non-accessible and immutable.

In this paper, we limit our attention to *effectively cutpoint-free* programs—programs in which the only objects that separate the callee's heap from the rest of the heap, when considering *live* reference fields, are the ones pointed to by the actual parameters of the invocation. This limitation (and certain variations of it, which we also describe) simplifies the local-reasoning about procedure calls because the analysis needs not track cutpoints. Furthermore, our analysis (conservatively) verifies that a program is effectively cutpoint-free,

1 Introduction

Shape-analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. The algorithms are *conservative* (sound), i.e., the discovered information is true for every input. Handling the heap in a precise manner requires strong pointer updates [3]. However, performing strong pointer updates requires a flow-sensitive and context-sensitive analysis and expensive heap abstractions, which may be doubly-exponential in the program size [25]. The presence of procedures escalates the problem because of interactions between the program stack and the heap [22] and because recursive calls may introduce additional exponential factors in an analysis. This makes interprocedural shape analysis a challenging problem.

This paper introduces a new approach for *local* [10,18] interprocedural shape analysis for a class of imperative programs. The main idea is to restrict the aliasing between

⋆ Supported by NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371, by ONR under grant N00014-09-1-0510, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.

⋆⋆ Supported by EPSRC.

live access paths at procedure calls. This allows procedure invocations to be analyzed ignoring *non-relevant* parts of the heap, more specifically, the parts of the heap not reachable from actual parameters. Moreover, shape analysis verifies that the above restrictions are satisfied.

The restricted class of programs is chosen based on observations made in [20]. There, Rinetzky et al. present a non-standard semantics in which procedures operate on procedure-local heaps containing only the objects reachable from actual parameters. The most complicated aspect of [20] is the treatment of sharing from the global heap and local variables of pending calls into the procedure-local heap. The problem is that the local heap can be accessed via access paths that bypass actual parameters. Therefore, objects in the local heap are treated differently when they separate the local heap (accessible by a procedure) from the rest of the heap (which—from the viewpoint of that procedure—is non-accessible and immutable). These objects are referred to as *cutpoints* [20].

Example 1. Fig. 1 illustrates the notions of local heaps and cutpoints. To gain intuition, Fig. 1 shows these notions using the familiar *store-based* semantics. (See, e.g., [18]). The figure depicts a memory state of a program comprised of four procedures: `main`, `foo`, `bar`, and `zoo`. The figure depicts a memory state that may occur at the entry to `zoo`. The stack of activation records is depicted on the left side of the diagram. Each activation record is labeled with the name of the procedure it is associated with. Thus, as we can see, `zoo` was invoked by `bar`; procedure `bar` was invoked by `foo`; and `foo` was invoked by the `main` procedure. The activation record at the top of the stack pertains to the *current* procedure (`zoo`). All other activation records pertain to *pending* procedure calls. Thus, for example, the access paths `z1.f1.f1`, `y9`, and `x5.f2` are pending access paths.

Heap-allocated objects are depicted as rectangles labeled with their location. The value of a reference variable (resp. field) is depicted by an edge labeled with the name of the variable (resp. field). The shaded cloud marks the part of the heap that `zoo` can access (i.e., the part of the heap containing the relevant objects for the invocation). The cutpoints for the invocation of `zoo` ($u8$ and $u9$) are heavily shaded. Note that $u7$ is not a cutpoint because it is also pointed to by $h7$, `zoo`'s formal parameter.

Cutpoints present a major challenge for shape abstractions: Procedure-local heaps together with special handling of cutpoints was found to be key in obtaining efficient and precise interprocedural shape-analysis algorithms [28]. Thus, the shape abstraction cannot abstract away the sharing patterns induced by cutpoints between the procedure-local heap of the procedure and the rest of the heap. These sharing patterns may lack any regular shape. However, the regularity of the sharing pattern is, in fact, what enables the effective shape abstraction of unbounded linked data structures.

We observe that cutpoints need special treatment in the analysis of a procedure because the caller may use its direct references to the cutpoint after the procedure returns. We develop an interprocedural shape analysis in which such direct usages are forbidden. We refer to a reference that, at the time when a procedure is invoked, points to a cutpoint and does not come from an object in the callee's local heap as a *piercing reference* for that invocation. An execution is *effectively cutpoint-free* if in every invocation that occurs during the execution, all the piercing references for that invocation are not *live* [26] at the time of the invocation, i.e., their r-values are not used later on in the execution before being set. A program is effectively cutpoint-free if all its executions

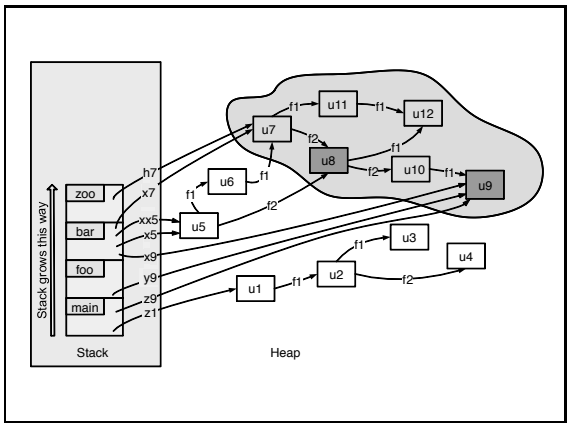


Fig. 1. An illustration of the cutpoints for an invocation in a store-based small-step (stack-based) operational semantics at the entry to zoo. We assume that $h7$ is zoo's formal parameter.

are. When analyzing effectively cutpoint-free programs, there is no need to give special care to cutpoint objects. However, to verify that a program is effectively cutpoint-free, special care needs to be taken regarding future usages of piercing references.

In this paper we present $\mathcal{ECPF}$, a small-step operational semantics [16] that handles *effectively cutpoint-free* programs. This semantics is interesting because procedures operate on local heaps, i.e., every procedure invocation starts executing on a memory state in which *parts of the heap not relevant to the invocation are ignored*. Thus, $\mathcal{ECPF}$ supports the notion of *heap-locality* [10,18] while permitting the usage of a global heap and destructive updates. Moreover, the absence of cutpoints drastically simplifies the meaning of procedure calls. $\mathcal{ECPF}$ tracks the set of piercing references and checks that their values are never used, thus dynamically verifying that the program execution is indeed effectively cutpoint-free. As a result, $\mathcal{ECPF}$ is applicable to arbitrary programs, and does not require an a priori classification of a program as effectively cutpoint-free. We show that for effectively cutpoint-free programs, $\mathcal{ECPF}$ is observationally equivalent to the standard global heap semantics.

$\mathcal{ECPF}$ gives rise to a functional [6,27] interprocedural shape analysis for effectively cutpoint-free programs. The analysis tabulates abstractions of memory states before and after procedure calls. Mimicking the semantics, memory states are represented in a procedure-local way *ignoring parts of the heap not relevant to the procedure with no special abstraction for cutpoints*. This reduces the complexity of the analysis because the analysis of procedures does not represent information about references and the heap from calling contexts. Indeed, this makes the analysis local in the heap and thus allows reusing the summarized effect of a procedure at different calling contexts.

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [13,25]. Thus, it is parametric in the heap abstraction and in the concrete effects of program statements, which allows experimenting with different instances of interprocedural shape analyzers. For example, we can employ different ab-

stractions for singly-, doubly-linked lists, and trees. Also, a combination of theorems in Appendix A.2 and [25] guarantees that every instance of our *interprocedural* framework is sound (see Sec. 5).

Main results. The contributions of this paper can be summarized as follows:

1. We define the notion of effectively cutpoint-free programs, in which the context not reachable from a procedure's actual parameters can be ignored when reasoning about the procedure's possible effect.
2. We define an operational semantics for a simple imperative language with references and procedures. The semantics dynamically checks that a program execution is effectively cutpoint-free. Procedures operate on procedure-local heaps, thus supporting the notion of heap-locality while permitting the usage of a global heap and destructive updates.
3. We present an interprocedural shape analysis for effectively cutpoint-free programs. The analysis is local in the heap and thus allows reusing the effect of a procedure at different calling contexts and at different call-sites.
4. We describe several extensions to our approach that allow its efficiency, precision, and applicability to be improved by utilizing a limited form of user-supplied annotations.

Outline. The rest of the paper is organized as follows. Sec. 2 presents an informal overview of our approach. Sec. 3 introduces our programming model. Sec. 4 defines our new local heap semantics, which checks whether a program is effectively cutpoint-free. Sec. 5 conservatively abstracts this semantics and provides the semantic foundation of the local interprocedural shape analysis algorithm described in Sec. 6. Sec. 7 describes certain efficiency-oriented extensions of our approach and certain relaxations of our restrictions aimed at increasing the class of effectively cutpoint-free programs. Sec. 8 describes related work, and Sec. 9 concludes.

2 Overview

This section provides an overview of our framework for interprocedural shape analysis using procedure-local heaps. The presentation is at an intuitive level; a more detailed treatment of this material is presented in the later sections of the paper.

2.1 Motivating Example

Fig. 2 shows a simple Java program that splices three non-shared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. This program serves as a running example in this paper.

2.2 Procedure-Local Heaps

In our semantics, procedures operate on local heaps. The local heap contains only the part of the program's heap accessible to the procedure. Thus, procedures are invoked on local heaps containing only objects reachable from actual parameters. We refer to these objects as the *relevant* objects for the invocation.

```
public class List{
  List n = null;
  int data;

  public List(int d){
     this.data = d;
  }

  static public List create3(int k) {
     List t1 = new List(k);
     List t2 = new List(k+1);
     List t3 = new List(k+2);
     t1.n = t2; t2.n = t3;
     return t1;
  }

  static public int getData(List w) {
     assert(w != null);
     int d = w.data;
     return d;
  }
```

```
  public static List splice(List p, List q) {
     List w = q;
     if (p != null) {
        List pn = p.n;
        p.n = null;
        p.n = splice(q, pn);
        w = p;
     }
     return w;
  }

  public static void main(String[] argv) {
     List x = create3(1);
     List y = create3(4);
     List z = create3(7);
     List t = splice(x, y);
     List s = splice(t, z);
     int i = 0;
     ℓ0 : // if (y == null) i++;
     ℓ1 : // if (y == x) i++;
     ℓ2 : // int i = getData(y);
     print(i);
  }
}
```

Fig. 2. An effectively-cutpoint-free program written in Java

Example 2. Fig. 3 shows the concrete memory states that occur at the call `t=splice(x,y)`. S_3^c shows the state at the point of the call, and S_3^e shows the state on entry to `splice`. Here, `splice` is invoked on local heaps containing the (relevant) objects reachable from either `x` or `y`.

The fact that the local heap of the invocation `t=splice(x,y)` contains only the lists referenced by `x` and `y` guarantees that destructive updates performed by `splice` can only affect access paths that pass through an object referenced by either `x` or `y`.

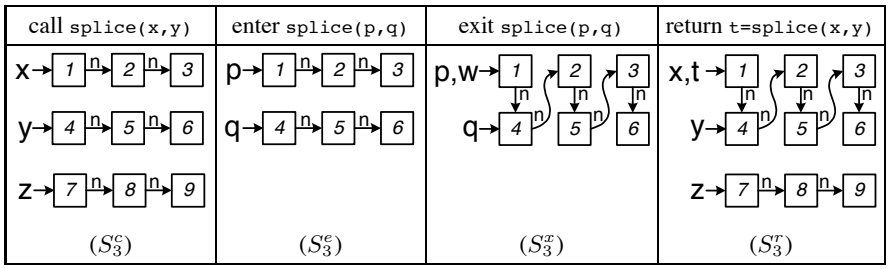


Fig. 3. Concrete states for the invocation `t = splice(x, y)` in the running example.

2.3 Cutpoints and Cutpoint-Freedom

Obviously, this is not always the case. In particular, consider the second call in the example program, `s=splice(t,z)`. Fig. 4(a) shows the concrete states when

`s=splice(t,z)` is invoked. $S_4^{c_{cp}}$ shows the state on invocation, and $S_4^{r_{cp}}$ the state when the call returns. As shown in the figure, the destructive updates of the `splice` procedure change not only paths from `t` and `z`, but also change the access paths from `y`.

To emphasize the effect of this invocation, consider a variant of the example program in which the invocation `s=splice(t,z)` has been replaced with an invocation `s=splice(y,z)`, as shown in Fig. 4(b). In this variant, the invocation can only affect access paths that pass through an object referenced by either `y` or `z`.

We capture the difference between these invocations by introducing the notion of a cutpoint [20]. A cutpoint for an invocation is an object that is: (i) reachable from an actual parameter, (ii) not pointed-to by an actual parameter, and (iii) reachable without going through an object that is pointed-to by an actual parameter (that is, it is either pointed-to by a variable or by an object not reachable from the parameters). In other words, a cutpoint is a relevant object that separates the part of the heap that is reachable for the invocation from the rest of the heap, but not pointed-to by a parameter.

For example, the object pointed-to by `y` at the call `s=splice(t,z)` (Fig. 4(a)) is a *cutpoint*, thus this invocation is not *cutpoint-free* [23]. In contrast, in the invocation `s=splice(y,z)` (Fig. 4(b)) no object is a cutpoint, and thus this invocation is *cutpoint-free* [23].

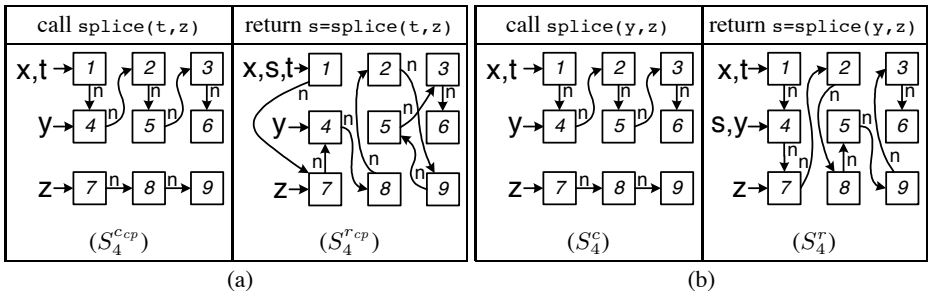


Fig. 4. Concrete states for: (a) the invocation `s=splice(t,z)` in the program of Fig. 2; (b) a variant of this program with an invocation `s=splice(y,z)`.

2.4 Effective Cutpoint-Freedom

The importance of cutpoints is that they allow the analysis to handle more precisely the notion of procedure local variables: No invocation of `splice` can modify the local variables of `main`. Thus, when control returns to `main`, it is guaranteed that the local variable `y` points to the same object that it pointed to before the invocation, and the `main` procedure can use the `y` reference to access directly that object. In general, it is very challenging to design a shape analysis that can track relations between arbitrary objects across the execution of procedure calls. However, if the caller does not use its direct references to the cutpoints after the procedure returns, the analysis does not need to track this relation.

For example, note that after `main` regains control, it does not use the value of the `y` variable. Thus, although the invocation `s=splice(t,z)` has a cutpoint, and is thus not cutpoint-free, in the context of the whole execution this invocation is *effectively cutpoint free*.

The semantics utilizes the above observation and instead of giving special treatment to the cutpoint objects, it assigns a special *inaccessible* value to all piercing references. The inaccessible value is used to track references which should not be used. It is a simple mechanism which the semantics uses to check (in runtime) whether a piercing pointer is used , e.g., in a dereference operation or during the evaluation of a condition, and if such a usage occurs to abort the execution and report that the program is not effectively cutpoint-free. (See Sec. 4).

Example 3. Fig. 5 shows the concrete memory states that occur at the call `s=splice(t,z)`. S_5^c shows the state at the point of the call, in which the object pointed to by `y` is a cutpoint. In S_5^r, the return state of that call, `y` no longer points to an object, instead it has the inaccessible value, depicted by a black bullet. The semantics intentionally does not utilize the information it has regarding the identity of objects. It acts as if it "forgets" that the object referenced by `y` at the call state is the third node in the returned list, mimicking in the concrete semantics the loss of information that occurs in the analysis. Note that the cutpoint object is not treated differently during the execution of `splice`, e.g., S_5^e and S_5^x show the states on entry to `splice` of the call and at its exit, respectively.

Also note that if any of the statements in lines $\ell_0 - \ell_2$ was to be uncommented, variable `y` would have been live at the time of the call `s=splice(t,z)`, and thus the execution would not have been effectively cutpoint-free.

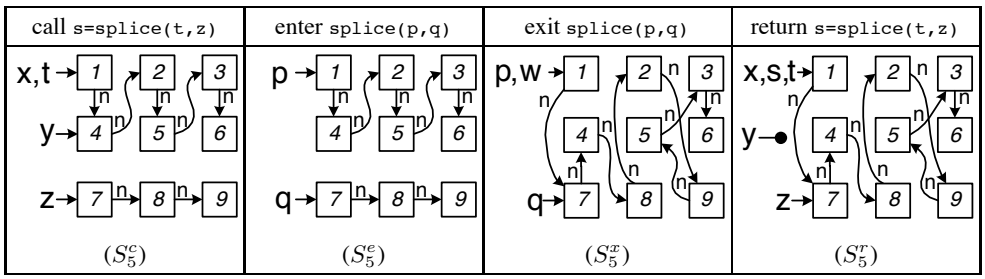


Fig. 5. Concrete states for the invocation `s = splice(t, z)` in the running example.

2.5 Interprocedural Shape Analysis

The algorithm computes procedure summaries by tabulating pairs of abstract input memory-states and abstract output memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The tabulated abstract memory-states represent procedure-local heaps, but do not keep track of cutpoints. However, they do record the inaccessible values. Therefore, these abstract states are

independent of the context in which a procedure is invoked. As a result, the summary computed for a procedure could be used at different calling contexts and at different call-sites while sustaining enough information to verify effective cutpoint freedom.

3 Programming Model

For expository reasons we limit our attention to a small imperative programming language. It has references to objects. Objects have fields, which can be either references to other objects or integers. The analyses developed here can be applied to Java-like languages and other imperative pointer languages alike (unless pointer arithmetic is used).

We abstract from specific control-flow statements and simply assume the presence of one control-flow graph per procedure. Control-flow graph edges are annotated with any one of the following statements below, where `x.f` denotes the `f` field of the object referenced by `x`. The statement `x = alloc()` returns a reference to a newly created object. Conditionals are implemented using `assume` statements.

$$\begin{aligned} \mathit{stms} ::= {} & \mathtt{x} = \mathtt{null} \mid \mathtt{x} = \mathtt{y} \mid \mathtt{x} = \mathtt{y.f} \mid \\ & \mathtt{x.f} = \mathtt{y} \mid \mathtt{x} = \mathtt{alloc}() \mid \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}) \mid \\ & \mathtt{y} = \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_k) \mid \mathtt{return} \end{aligned}$$

In our running example we take the liberty to use integer variables and fields as well.

In the rest of the paper, we assume that we are working with a fixed arbitrary program P. For a procedure p, V_p denotes the set of its local variables and $F_p \subseteq V_p$ denotes the set of its formal parameters. A procedure returns a value by assigning it to a designated variable `ret`. We assume that parameters are passed by value and that formal parameters cannot be assigned to. The set of all local variables of P is written $\mathcal{V}$. We write $\mathcal{F}$ to denote the set of all field names in P.

We assume a standard store-based operational semantics for our language, very much like $\mathcal{GSB}$ defined previously in [19,20]. $\mathcal{GSB}$ treats live cutpoints properly.

4 Concrete Semantics

In this section, we define $\mathcal{ECPF}$ (*effectively cutpoint-free*), a non-standard semantics that checks whether a program execution is effectively cutpoint-free. $\mathcal{ECPF}$ defines the execution traces that are the foundation of our analysis.

$\mathcal{ECPF}$ is a *store-based* semantics (see, e.g., [18]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{ECPF}$, on the other hand, is a *procedure-local heap* semantics [20]: A memory state that occurs during the execution of a procedure does not represent objects that, at the time of the invocation, are not reachable from the actual parameters.

$\mathcal{ECPF}$ is a small-step operational semantics [16]. Instead of encoding a stack of activation records inside the memory state, as is traditionally done, $\mathcal{ECPF}$ maintains a *stack of program states* [12,21]: Every program state consists of a program point and a memory state. The program state of the *current procedure* is stored at the top of the

stack, and it is the only one that can be manipulated by intraprocedural statements. We refer to this memory state as the *current memory state*. When a procedure is invoked, the *entry memory state* of the callee is computed by a $Call$ operation according to the caller's current memory state, and pushed onto the stack. When a procedure returns, the stack is popped, and the caller's *return memory state* is updated using a Ret operation according to its memory state before the invocation (the *call memory state*) and the callee's (popped) *exit memory state*. The $Call$ and Ret operations of $\mathcal{ECPF}$ are defined in Fig. 8.

The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the local heap of just one procedure. The *lifting* of an intraprocedural semantics to an interprocedural semantics, that uses a stack of program states, is formally defined in [19].

An execution trace of a program P always begins with P's `main` executing on an *initial memory state* in which all its reference variables have the value *null* and the heap is empty. We say that a memory state is *reachable* in a program P if it occurs as the current memory state in an execution trace of P.

$\mathcal{ECPF}$ is a procedure-local heap semantics [20]: when a procedure is invoked, it starts executing on an *input heap* containing only the set of *relevant objects for the invocation*. An object is *relevant for an invocation* if it is a *parameter object*, *i.e.*, either referenced by an actual parameter or reachable from one.

A procedure-local heap semantics and its abstractions benefit from not having to represent irrelevant objects. However, in general, the semantics needs to take special care of cutpoints. In this paper, we avoid the need to take special care of cutpoint objects by assuming and verifying that a program is *effectively cutpoint free*: We refer to a reference that at invocation time points to a cutpoint and does not come from an object in the callee's local heap as a *piercing reference* for that invocation. An execution is *effectively cutpoint-free* if in every of its invocations during an execution all the piercing references for that invocation are *dead* at the time of the invocation, i.e., their r-values are not used before being set. A program is *effectively cutpoint-free* if all of its executions are.

For effectively cutpoint-free programs, there is no need to give special care to cutpoint objects. However, to verify that a program is effectively cutpoint-free, special care needs to be taken regarding the piercing references. In this section, we describe the way $\mathcal{ECPF}$ validates at runtime that an execution is effectively cutpoint-free.

4.1 Memory States

Fig. 6 defines the concrete semantic domains and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. A value $v \in Val$ is either a location, *null*, or $\bullet$, the inaccessible value used to represent references to locations that should not be accessed.

A memory state in the $\mathcal{ECPF}$ semantics is, essentially, a 2-level store. Formally, a memory state is a 3-tuple $\sigma = \langle \rho, L, h \rangle$: $\rho \in \mathcal{E}$ is an environment assigning values for the variables of the *current* procedure. $L \subset Loc$ is the set of allocated locations. (A dynamically allocated object is identified by its location. We interchangeably use the terms object and location.) $h \in \mathcal{H}$ assigns values to fields of allocated objects.

$l \in \mathit{Loc}$
$v \in \mathit{Val} = \mathit{Loc} \cup \{\mathit{null}\} \cup \{\bullet\}$
$\rho \in \mathcal{E} = \mathcal{V} \rightharpoonup \mathit{Val}$
$h \in \mathcal{H} = \mathit{Loc} \rightharpoonup \mathcal{F} \rightharpoonup \mathit{Val}$
$\sigma \in \Sigma = \mathcal{E} \times 2^{\mathit{Loc}} \times \mathcal{H}$

Fig. 6. Semantic domains.

In $\mathcal{ECPF}$, reachability is defined with respect to relevant objects: Informally, an object l_2 is *reachable from* an object l_1 in a memory state σ if there is a directed path in the heap of σ from l_1 to l_2. An object l is *reachable* in σ if it is reachable from a location that is pointed-to by some variable. Note that $\bullet$-valued references do not point to any object.

4.2 Operational Semantics of Intraprocedural Statements

The meaning of atomic statements is described by a transition relation $\overset{i}{\rightsquigarrow} \subseteq (\Sigma \times \mathit{stms}) \times \Sigma \uplus \{\sigma_\bullet\}$, where $\sigma_\bullet$ is a special error state indicating a forbidden usage of the inaccessible value.

Fig. 7 defines the axioms for atomic intraprocedural statements. These are handled as in a standard 2-level store semantics like $\mathcal{GSB}$.[7] The main difference between the $\mathcal{ECPF}$ semantics and $\mathcal{GSB}$ with respect to the meaning of intraprocedural statements is captured by the side-conditions of the form $\rho(x) = \bullet$ or $\rho(y) = \bullet$, which prevent usage of the inaccessible locations.

$\langle \mathtt{x} = \mathtt{null}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto \mathit{null}], L, h\rangle$	
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto \rho(y)], L, h\rangle$	
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto h(\rho(y), f)], L, h\rangle$	$\rho(y) \in \mathit{Loc}$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho, L, h[(\rho(y), f) \mapsto \rho(x)]\rangle$	$\rho(y) \in \mathit{Loc}$
$\langle \mathtt{x} = \mathtt{alloc}(), \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto l], L \cup \{l\}, h[l \mapsto I]\rangle$	$l \in \mathit{Loc} \setminus L$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{i}{\rightsquigarrow} \sigma$	$\rho(x) \bowtie \rho(y)$
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$ or $h(\rho(y)) = \bullet$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$ or $\rho(x) = \bullet$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(x) = \bullet$ or $\rho(y) = \bullet$

Fig. 7. Axioms for intraprocedural statements, where in each line σ is understood as a shorthand for $\langle \rho, L, h\rangle$. I denotes the function $\lambda f \in \mathcal{F}.\mathit{null}$. $\bowtie$ stands for either $=$ or $\neq$. When convenient, we sometimes treat h as an uncurried function, i.e., as a function from $\mathit{Loc} \times \mathcal{F}$ to Val.

4.3 Operational Semantics of Interprocedural Statements

Fig. 8 defines the meaning of the $Call$ and Ret operations pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ assuming p's formal parameters are $z_1, \ldots, z_k$, the memory state at the call site is $\sigma_c = \langle \rho_c, L_c, h_c \rangle$, and the memory state at the exit of p is $\sigma_x = \langle \rho_x, L_x, h_x \rangle$. The $Call$ operation is used to compute the state update along a call edge in the control-flow graph; the Ret operation computes the state update along a return edge. As defined in Sec. 3, variable `ret` is used to communicate the return value. We use the function $R_h(L)$ to compute the locations that are reachable in heap h from the set of locations L. This function is formally defined in Appendix A.1.

$$
\begin{array}{ll}
Call_{y=p(x_1,\ldots,x_k)}(\sigma_c) = \sigma_e & Ret_{y=p(x_1,\ldots,x_k)}(\sigma_c, \sigma_x) = \sigma_r \\
\quad \sigma_e = \langle \rho_e, L_c, h_c|_{L_{rel}} \rangle & \quad \sigma_r = \langle \rho_r, L_x, h_r \rangle \\
\quad\quad \rho_e = [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k] & \quad\quad \rho_r = (block \circ \rho_c)[y \mapsto \rho_x(ret)] \\
 & \quad\quad h_r = (block \circ h_c|_{L_c \setminus L_{rel}}) \cup h_x
\end{array}
$$

where:

$$
\begin{array}{l}
L_{parameters} = \{\rho_c(x_i) \in Loc \mid 1 \leq i \leq k\} \\
L_{rel} = R_{h_c}(L_{parameters}) \\
L_{cutpoints} = (L_{rel} \setminus L_{parameters}) \cap \\
\qquad\qquad (\{\rho_c(z) \mid z \in V_q\} \cup \{h_c(l)f \in Loc \mid l \in L_c \setminus L_{rel}, f \in \mathcal{F}) \\
block = \lambda v \in Val. \begin{cases} \bullet & v \in L_{cutpoints} \\ v & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
Call_{y=p(x_1,\ldots,x_k)}(\sigma_c) = \sigma_\bullet & \rho_c(x_1) = \bullet \text{ or } \cdots \text{ or } \rho_c(x_k) = \bullet \\
Ret_{y=p(x_1,\ldots,x_k)}(\sigma_c, \sigma_x) = \sigma_\bullet & \rho_x(ret) = \bullet
\end{array}
$$

Fig. 8. $Call$ and Ret operations for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q, where it is understood that $\sigma_c = \langle \rho_c, L_c, h_c \rangle$, $\sigma_x = \langle \rho_x, L_x, h_x \rangle$, and V_q denotes the set of local variables of procedure q.

Procedure calls The $Call$ operation computes the callee's *entry memory state* (σ_e) from the state at the call-site (σ_c). The entry memory state is computed by binding the values of the formal parameters in the callee's environment to the values of the corresponding actual parameters (ρ_e) and restricting the caller's heap to the relevant objects for the invocation (L_{rel}).

Example 4. Fig. 3 shows the entry state S_3^e that results from applying the $Call$ operation pertaining to the invocation `t=splice(x, y)` to the call memory state S_3^c. Fig. 5 shows the entry state S_5^e that results from applying the $Call$ operation pertaining to the invocation `s=splice(t, z)` to the call memory state S_5^c.

Procedure returns The Ret operation maps the memory state at the exit of a procedure (σ_x) together with the state at call-site (σ_c) to the return state σ_r from which the caller resumes its computation. Ret updates the caller's memory state by carving out the input heap passed to the callee from the caller's heap ($h_c|_{L_c \setminus L_{rel}}$) and replacing it with the callee's (possibly) mutated heap (h_x).

In $\mathcal{ECPF}$, an object never changes its location, and locations are never reallocated. Thus, any pointer to a relevant object in the caller's memory state (either by a field of an irrelevant object or a variable) points after the replacement to an up-to-date version of the object.

Blocking piercing references. $\mathcal{ECPF}$ detects forbidden accesses that violate the effective-cutpoint-freedom condition, and aborts the program in an error state if such an access is detected. Technically, when a procedure invocation returns, $\mathcal{ECPF}$ assigns the special value $\bullet$ to all piercing references, an operation which we refer to as *blocking*, and uses this special value to detect forbidden accesses. (Recall that in an effectively cutpoint-free execution, every live reference that points to an object which separate the callee's heap from the caller's heap should point to a parameter object, i.e., to one of the objects in $L_{parameters}$.)

Example 5. Fig. 3 shows the return state S_3^r, that results from applying the Ret operation pertaining to the invocation `t=splice(x, y)` to the call memory state S_3^c and the exit memory state S_3^x. Fig. 5 shows the return state S_5^r, that results from applying the Ret operation pertaining to the invocation `s=splice(t, z)` to the call memory state S_5^c and the exit memory state S_5^x. The second node in the list pointed to by `t` at the call state S_5^c is a cutpoint. Thus, variable `y` gets blocked when computing S_5^r.

4.4 Observational Soundness

We say that two values are *comparable* in $\mathcal{ECPF}$ if neither one is $\bullet$. We say that a $\mathcal{ECPF}$ memory state σ is *observationally sound* with respect to a standard semantics σ_G if for every pair of access paths that have comparable values in σ, they have equal values in σ iff they have equal values in σ_G. $\mathcal{ECPF}$ *simulates* the standard 2-level store semantics: Executing the same sequence of statements in the $\mathcal{ECPF}$ semantics and in the standard semantics either results in a $\mathcal{ECPF}$ memory states that is observationally sound with respect to the resulting standard memory state, or the $\mathcal{ECPF}$ execution gets to an *error state* due to a constraint breach (detected by $\mathcal{ECPF}$). A program is *effectively cutpoint-free* if it does not have an execution trace that gets to an error state. (Note that the initial state of an execution in $\mathcal{ECPF}$ is observationally sound with respect to its standard counterpart).

Our goal is to detect structural invariants that are true according to the *standard semantics*. $\mathcal{ECPF}$ acts like the standard semantics as long as the program's execution satisfies certain constraints. $\mathcal{ECPF}$ enforces these restrictions by blocking references that a program should not access. Similarly, our analysis reports an invariant concerning equality of access paths only when these access paths have comparable values.

An invariant concerning equality of access paths in $\mathcal{ECPF}$ for an effectively cutpoint-free program is also an invariant in the standard semantics. This makes abstract

interpretations of $\mathcal{ECPF}$ suitable for verifying data-structure invariants, for detecting memory access violations, and for performing compile-time garbage collection.

5 Abstract Interpretation

In this section, we present $\mathcal{ECPF}^{\#}$, an abstract interpretation [5] of the $\mathcal{ECPF}$ semantics. $\mathcal{ECPF}^{\#}$ is the basis of our static-analysis algorithm which uses the 3-valued logic-based framework of [25]. The soundness of the abstract semantics with respect to $\mathcal{GSB}$[7] is guaranteed by the combination of the theorems in Appendix A.2 and [25]:

- In Appendix A.2, we show that for effectively cutpoint-free programs, $\mathcal{ECPF}$ is observationally equivalent to $\mathcal{GSB}$.
- In [25], it is shown that every program-analyzer that is an instance of the 3-valued logic-based framework is sound with respect to the concrete semantics it is based on.

5.1 Abstract States

We conservatively represent unbounded sets of unbounded memory states using a bounded set of bounded 3-valued logical structures, which we refer to as *abstract states*. Note that there are actually three different notions of *concrete states*. The most concrete states are those in $\mathcal{GSB}$, containing full information including integer variables and fields. Integers are already abstracted away when we talk about $\mathcal{ECPF}$, which, on top of that, also yields errors when cutpoint references are illegally used. $\mathcal{ECPF}$ states are equivalently encoded into *two-valued* logical structures by viewing objects as individuals in a logical structure and references as binary predicates (see below). Note, however, that location identifiers play no role in the logical structure encoding. Indeed, the semantics does not distinguish between isomorphic structures.

We use the term *concrete state* whenever we talk about a state that is not a 3-valued logical structure. We believe that, despite the resulting imprecision, our intentions are clear. In drawings, we use the same graphical notations to depict concrete states in all of the aforementioned semantics. (Integer values, when drawn, should be ignored when considering a figure to be a graphical depiction of a state in $\mathcal{ECPF}$ or of a logical structure.)

3-valued logical structures. A 3-valued logical structure is a logical structure with an extra truth-value $\frac{1}{2}$, which denotes values that may be 1 or may be 0. The information partial order on the set $\{0, \frac{1}{2}, 1\}$ is defined as $0 \sqsubseteq \frac{1}{2} \sqsupseteq 1$, and $0 \sqcup 1 = \frac{1}{2}$. Formally, a 3-valued logical structure is $S^\sharp = \langle U^{S^\sharp}, \iota^{S^\sharp} \rangle$ where:

- $U^{S^\sharp}$ is the universe of the structure.
- $\iota^{S^\sharp}$ is an interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) \colon {U^{S^\sharp}}^k \to \{0, \frac{1}{2}, 1\}$.

[7] $\mathcal{GSB}$ is a standard two-level store semantics for heap-manipulating programs. It is formally defined in [20].

A 2-valued logical structure is a 3-valued logical structure where the truth-values of predicates are either 0 or 1. The set of *3-valued* logical structures is denoted by $3Struct$. The set of *2-valued* logical structures is denoted by $2Struct$.

Abstraction function. We abstract sets of $\mathcal{ECPF}$ memory states by a point-wise application of an *extraction function* $\beta : \Sigma \rightharpoonup 3Struct$ mapping an $\mathcal{ECPF}$ memory state to its *best representation* by an *abstract state*. The extraction function β is defined as a composition of two functions: (i) $\beta_{shape} : \Sigma \rightharpoonup 2Struct$, which maps an $\mathcal{ECPF}$ memory state to a 2-valued logical structure and (ii) *canonical abstraction* [25], which maps 2-valued logical structures to a bounded number of 3-valued logical structures.

Representing Memory States using 2-Valued Logical Structures We represent $\mathcal{ECPF}$ memory states using 2-valued logical structures. Every individual in the structure corresponds to a heap-allocated object. Predicates of the structure correspond to properties of heap-allocated objects.

Core predicates. Tab. 1 shows the core predicates used in this paper. A binary predicate $f(v_1, v_2)$ holds when the $\texttt{f} \in \mathcal{F}$ field of v_1 points to v_2. The designated binary predicate $eq(v_1, v_2)$ is the equality predicate, which records equality between v_1 and v_2. A unary predicate $x(v)$ holds for an object that is referenced by the reference variable $\texttt{x} \in \mathcal{V}$ of the *current* procedure.[8] The predicate ia holds only for a unique individual, which represents the inaccessible locations. The role of the predicates $inUc$ and $inUx$ is explained in Sec. 5.2.

Instrumentation predicates. Instrumentation predicates record derived properties of individuals, and are defined using a logical formula over core predicates. Instrumentation predicates are stored in the logical structures like core predicates. They are used to refine the abstract semantics, as we shall shortly see. Tab. 2 lists the instrumentation predicates used in this paper. We use $F(v_1, v_2)$ as a shorthand to denote that v_1 has a field $f \in \mathcal{F}$ which points to v_2 and $F^*(v_1, v_2)$ as the reflexive transitive closure of F. (For a formal definition, see Appendix B).

2-valued logical structures are depicted as directed graphs. We draw individuals as boxes. We depict the value of a reference variable `x` by drawing an edge from `x` to the individual representing the object that `x` references. For all other unary predicates p, we draw p inside a node u when $\iota^S(p)(u) = 1$; conversely, when $\iota^S(p)(u) = 0$ we do not draw p in u. A directed edge between nodes u_1 and u_2 that is labeled with a binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. For clarity, we do not draw the binary equality predicate eq. The inaccessible value is depicted as a line ending with $\bullet$.

Example 6. The structure S_3^c of Fig. 3 shows a *2-valued* logical structure that represents the memory state of the program at the call `t=splice(x, y)`. The depicted numerical values are only shown for presentation reasons, and have no meaning in the logical representation.

The structure S_5^r of Fig. 5 shows a *2-valued* logical structure that represents the memory state of the program at the return of `s=splice(t, y)`. Note that the value of `y` is the inaccessible value.

[8] For simplicity, we use the same set of predicates for all procedures. Thus, our semantics ensures that $\iota^S(x) = \lambda u.0$ for every local variable `x` that does not belong to the current call.

Table 1. Predicates used to represent (concrete) memory states.

Predicate	Intended Meaning
$f(v_1, v_2)$	the `f`-field of object v_1 points to object v_2
$eq(v_1, v_2)$	v_1 and v_2 are the same object
$x(v)$	reference variable `x` points to the object v
$ia(v)$	v is an inaccessible location
$inUc(v)$	v originates from the caller's memory state at the call site
$inUx(v)$	v originated from the callee's memory state at the exit site

Table 2. The instrumentation predicates used in this paper.

Predicate	Intended Meaning	Defining Formula
$r_{obj}(v_1, v_2)$	v_2 is reachable from v_1 by some field path	$\neg ia(v_1) \wedge \neg ia(v_2) \wedge F^*(v_1, v_2)$
$ils(v)$	v is *locally* shared. i.e., v is pointed-to by a field of more than one object in the *local heap*	$\exists v_1, v_2 \colon \neg ia(v) \wedge \neg eq(v_1, v_2) \wedge F(v_1, v) \wedge F(v_2, v)$
$c(v)$	v resides on a directed cycle of fields	$\exists v_1 \colon F(v, v_1) \wedge F^*(v_1, v)$
$r_x(v)$	v is reachable from variable `x`	$\neg ia(v) \wedge \exists v_x \colon x(v_x) \wedge F^*(v_x, v)$

Bounded Abstraction We now formally define how memory states are represented using abstract memory states. The idea is that each object from the (concrete) state is mapped to an individual in the abstract state. An abstract memory state may include *summary nodes*, i.e., individuals that correspond to one or more concrete nodes in one of the concrete states represented by the abstract state. For a summary node $u \in U^\sharp$ in abstract state $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ it holds that $\iota(eq)(u, u) = \frac{1}{2}$.

Canonical abstraction. A *3-valued* logical structure $S^\sharp$ is a **canonical abstraction** of a *2-valued* logical structure S if there exists a surjective function $\upsilon \colon U^S \to U^{S^\sharp}$ satisfying the following conditions: (i) For all $u_1, u_2 \in U^S$, $\upsilon(u_1) = \upsilon(u_2)$ iff for all unary predicates $p \in \mathcal{P}$, $\iota^S(p)(u_1) = \iota^S(p)(u_2)$, and (ii) for all predicates $p \in \mathcal{P}$ of arity k and for all k-tuples $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

$$\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \bigsqcup_{\substack{u_1, \ldots, u_k \in U^S \\ \upsilon(u_i) = u_i^\sharp}} \iota^S(p)(u_1, u_2, \ldots, u_k).$$

3-valued logical structures are also drawn as directed graphs. Definite values (0 and 1) are drawn as for 2-valued structures. Binary indefinite predicate values ($\frac{1}{2}$) are drawn as dotted directed edges. Summary nodes are depicted by a double frame.

Example 7. Fig. 9 shows the abstract states (as *3-valued* logical structures) representing the concrete states of Fig. 3. Note that only the local variables `p` and `q` are represented inside the call to `splice(p,q)`. Representing only the local variables inside a call ensures that the number

of unary predicates to be considered when analyzing the procedure is proportional to the number of its local variables. This reduces the overall complexity of our algorithm to be worst-case doubly-exponential in the maximal number of local variables rather than doubly-exponential in their total number (as in e.g., [22]).

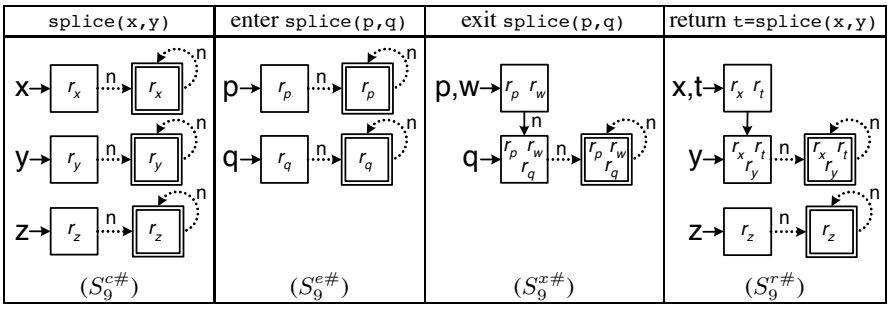


Fig. 9. Abstract states for the invocation `t = splice(x, y);` in the running example.

The Importance of Reachability Recording derived properties by means of *instrumentation predicates* may provide additional information that would have been otherwise lost under abstraction. In particular, because canonical abstraction is directed by unary predicates, adding unary instrumentation predicates may further refine the abstraction. This is called the *instrumentation principle* in [25]. In our framework, the predicates that record reachability from variables play a central role. They enable us to identify the individuals representing objects that are reachable from actual parameters. For example, in the *3-valued* logical structure $S_9^{c\#}$ depicted in Fig. 9, we can detect that the top two lists represent objects that are reachable from the actual parameters because either r_x or r_y holds for these individuals. None of these predicates holds for the individuals at the (irrelevant) list referenced by `z`. We believe that these predicates should be incorporated in any instance of our framework.

5.2 Abstract Operational Semantics

The meaning of statements is described by a transition relation $\overset{\sharp}{\rightsquigarrow} \subseteq (3Struct \times \mathit{stms}) \times 3Struct$. Because our framework is based on [25], the encoding of the meaning of statements in $\mathcal{ECPF}$ (as transformers of 2-valued structures), also defines the corresponding abstract semantics (as transformers of 3-valued structures). This abstract semantics is obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for our static analysis. In particular, reinterpreting the side conditions of intraprocedural statements conservatively verifies that the *program* is effectively cutpoint-free.

For brevity, we omit the aforementioned encoding from the body of the paper and provide it in Appendix B. We wish to note that all the transformers, including the inter-

procedural operations $Call$ and Ret are specified using predicate-update formulae [9] in first-order logic with transitive closure.

6 Interprocedural Static Analysis

Abstract interpretation of the $\mathcal{ECPF}$ semantics provides the semantic foundations for an interprocedural static-analysis algorithm that computes procedure summaries by tabulating abstract input memory-states to abstract output memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The interprocedural tabulation algorithm is the variant of the IFDS-framework [17] presented in [23], adapted to assume and verify effective cutpoint freedom.

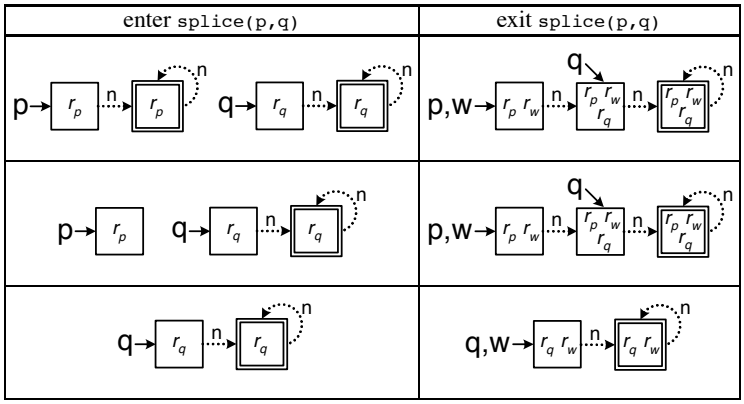


Fig. 10. Partial tabulation of abstract states for the splice procedure.

Example 8. Fig. 10 shows a partial tabulation of abstract local heaps for the `splice` procedure of the running example. The figure shows 3 possible input states of the list pointed-to by `p`. Identical possible input states of the list pointed-to by `q`, and their combinations are not shown. As mentioned in Sec. 1, the splice procedure is only analyzed 9 times before its tabulation is complete, producing a summary that is then reused whenever the effect of `splice(p, q)` is needed.

Note that this tabulation represents the input/output relation for any call to `splice`, including ones with cutpoints, e.g., the call `s=splice(t, y)` and all recursive calls to `splice` in our running example.

[9] Predicate-update formulae express the semantics of statements: Suppose that σ is a memory state that arises before statement st, that σ' is the store that arises after st is evaluated on σ, and that S is the 2-valued logical structure that encodes σ. A collection of predicate-update formulae–one for each predicate p in the vocabulary of S–allows one to obtain the structure S' that encodes σ'. When evaluated in structure S, the predicate-update formula for a predicate p indicates what the value of p should be in S'. See [25, Observation 2.6]. Evaluation of the predicate-update formulae in 3-valued logic captures the transfer function for st of the abstract semantics. See [25, Observation 2.9].

7 Extensions and Relaxations

In this section, we describe several extensions that use a limited form of annotations on procedures to improve the analysis algorithmic's efficiency, precision, and applicability.

7.1 Blindspots

$\mathcal{ECPF}$ records in every state the value of every formal parameter at the entry to the procedure. This is done to allow the caller to observe the (possibly mutated) part of the heap that was relevant to the callee after the callee returns. However, in certain cases, such observations are not needed or even desirable.

For example, in the program of Fig. 2, the variable `y` is not used after the call `t=splice(x, y)`. Thus, the effort invested to restore its value when the call returns is, for all practical purposes, wasted. Furthermore, direct access to the list returned by `splice` through one of the actual parameters might be considered a form of bad programming. (A clearer example might be a `merge` procedure that merges two sorted lists. When an invocation of `merge` returns, one actual parameter references the head of the list and the other one references one of the list elements. Using the actual parameters at this point makes the code less readable and more sensitive to the implementation details of `merge`. Thus, it is reasonable to expect that the caller uses the returned value, but not the actual parameters.)

Blindspots (for a procedure invocation) are parameter objects for which all the variables and fields pointing to them at the time of the call, excluding fields of relevant objects for the invocation, are *dead* when the procedure returns.[10] $\mathcal{ECPF}$, and its abstract interpretations, can utilize an annotation (e.g., in the form of a subset of the actual/formal parameters) that states which of the parameter objects are blindspots. Such information can improve the efficiency of the analysis algorithim by allowing it to avoid tracking unnecessary information. It also allows verifying good programming style.

For example, Fig. 11 shows the call, entry, exit, and return states that occur in the $\mathcal{ECPF}$ during the invocation `t=splice(x,y)` when both parameter objects are annotated as a *blindspots*. Based on this annotation, the exit state does not record the value of the formal parameters, allowing for more compact summaries. Note that at the return state, `x` and `y` are blocked. As a result, the returned list can be accessed only through `t`.

7.2 Tolerance for a Bounded Number of Cutpoints

$\mathcal{ECPF}$, and its abstract interpretations, *can* allow for procedure invocations to have up to a bounded number of live cutpoints, i.e., cutpoints that are accessed directly by a piercing reference after the procedure returns. The main idea is to treat cutpoints as additional parameters: Every procedure is modified to have k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, the (modified) *semantics* binds the additional parameters with references to the cutpoints.

[10] Note that a blindspot for a procedure invocation is not necessarily a dead *object*.

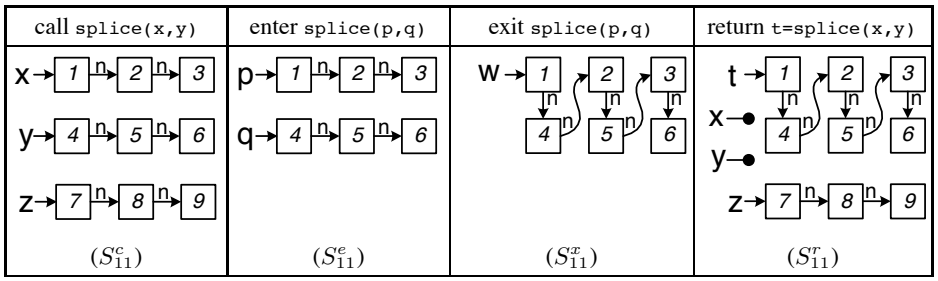


Fig. 11. Concrete states for the invocation `t = splice(x, y)` when both parameter objects are annotated as a *blindspots*.

We can allow for a bounded number of cutpoints by having an annotation regarding the maximal number of allowed cutpoints [11] or by having the user provide a specification (using first-order formulae with transitive closure) of a distinguished set of explicitly-allowed cutpoints. For example, a cutpoint at the last element of a list can be treated differently then other cutpoints.

Fig. 12 depicts the call, entry, exit, and return states that occur in the $\mathcal{ECPF}$ during the invocation `s = splice(t, z)` when procedures are allowed to have at least one cutpoint, or, alternatively, when the second element of the first list is specified as an explicitly-allowed cutpoint. The hidden parameter X_1 gets bound to the cutpoint at the entry state and used to restore the value of `y` at the return state.

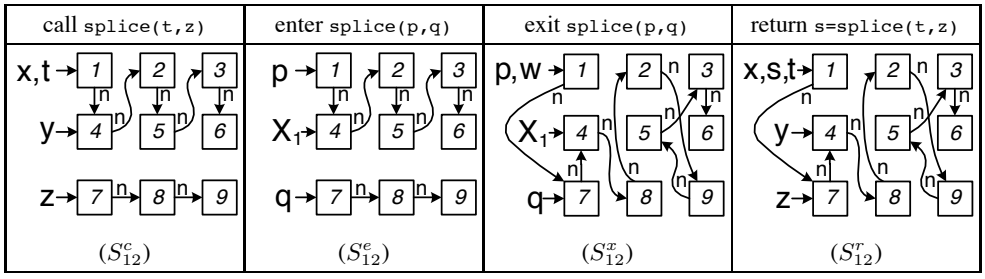


Fig. 12. Concrete states for the invocation `s = splice(t, z)` when one cutpoint is allowed or alternatively, when access path `t.n` is specified as an *explicitly-allowed* cutpoint.

7.3 Restricted Access to the Inaccessible Value

For a program to be effectively cutpoint-free, every piercing reference must not be live at the time of the actual invocation. The reason behind this requirement is to allow the semantics/analysis to avoid maintaining certain aliasing relations, yet still maintain a certain notion of observational soundness with respect to the standard semantics.

[11] This is the essence of the treatment of cutpoints by Gotsman et al. [8].

However, certain *usages* of piercing references are innocuous, i.e., our notion of observational soundness is still maintained as long as programs use piercing references in certain restricted ways. For example, statements such as `x = y`, as well as conditions involving comparisons between •-valued references and `null`, are innocuous. In the former case, the assignment neither affects the control flow of the program nor may lead to a memory fault. In the latter case, it always holds that a •-valued reference is not *null*-valued; thus the condition of the `assume` statement always evaluates to the same value in both semantics.

Effectively, the above observation allows us to relax the requirements of executions to be effectively cutpoint-free: Instead of forbidding all future usages of piercing references (i.e., requiring that they are not live when the invocation of the callee returns), we need only to forbid "effective" future usages of this pointers, i.e., we need only to forbid them from being dereferenced or compared with non-null values both in `assume` statements and in assertions.

7.4 Arbitrary Cutpoints in Pure Procedures

An additional relaxation regarding the requirements of a procedure invocation to be effectively cutpoint-free is possible when a procedure invocation is found to be *pure*. A pure invocation does not modify the shared state. Thus, the abstract representation of the heap at the call site can be reused at the return site. As a result, for reconstructing the layout of the heap, the number of cutpoints is irrelevant, and piercing references do not need to be blocked.

The above approach has one rather significant complication: In case the procedure's return value is a pointer to a heap-allocated object, figuring out which object in the call state corresponds to the one returned by the procedure is not simple. (This complication arises because the abstract semantics does not retain the identity of locations.)

One possible remedy is not to use this relaxation when the return value of the invoked procedure is a (non-null) reference. Another possible remedy is to apply a *meet* operator between the call state and the exit state (after certain renaming operations, similar to the ones used in [11]). We note that the framework of [25] provides an algorithmic meet operator [1]. We also note that (some) information regarding cutpoints can (potentially) make the results of the meet operator more precise.

8 Discussion and Related Work

In this section, we review closely related work.

Rinetzky and Sagiv [22] explicitly represent the runtime stack and abstract it as a linked-list. In this approach, the entire heap, and the runtime stack are represented at every program point. As a result, the abstraction may lose information about properties of the heap *for parts of the heap that cannot be affected by the procedure at all*.

Jeannet et al. [11] consider procedures as transformers from the (entire) heap before the call to the (entire) heap after the call. Irrelevant objects are summarized into a single summary node. Relevant objects are summarized using a two-store vocabulary. One vocabulary records the current properties of the object. The other vocabulary encodes

the properties that the object had when the procedure was invoked. The latter vocabulary allows to match objects at the call-site and at the exit-site. Note that this scheme never summarizes together objects that were not summarized together when the procedure was invoked. For cutpoint-free programs, this may lead to needlessly large summaries. Consider for example a procedure that operates on several lists and nondeterministically replaces elements between the list tails. The method of [11] will not summarize list elements that originated from different input lists. Thus, it will generate exponentially more mappings in the procedure summary than the ones produced by our method. On the other hand, the method of [11] can establish properties of called procedures that our method cannot establish (e.g., that a procedure to reverse a list actually reverses all elements of the list).

Rinetzky et al. [20] present a procedure-local storeless concrete semantics and describe an abstract interpretation of their semantics that can be used for interprocedural shape-analysis for programs manipulating singly linked lists. Their abstract interpretation algorithm explicitly records cutpoint objects in the local heap, and may become imprecise when there is more than one cutpoint. Our algorithm can be seen as a specialization of [20] that provides a partial answer to this problem. In addition, because we restricted our attention to effectively cutpoint-free programs, our semantics and analysis are much simpler than the ones in [20].

In [23], the problem of abstracting cutpoint-induced sharing patterns is addressed by forbidding cutpoints: We developed an interprocedural shape analysis for the class of *cutpoint-free* programs, in which program invocations never generate cutpoints. In the present paper, we extend the framework developed in [23] to a larger class of programs: *effectively cutpoint-free* programs. One can see [23] as an eager form of enforcing effective cutpoint-freedom, while the present paper takes a more lazy approach.

Hackett and Rugina [9] develop a staged analysis to obtain a relatively scalable interprocedural shape analysis. Their approach uses a scalable imprecise pointer-analysis to decompose the heap into a collection of independent locations. The precision of this approach might be limited because it relies on pointer-expressions that appear in the program's text. The analysis tabulates global heaps, potentially leading to a low reuse of procedure summaries.

For the special case of singly-linked lists, another approach for modular shape analysis is presented by Chong and Rugina [4] without an implementation. The main idea there is to record for every object both its current properties and the properties it had at that time the procedure was invoked.

Gotsman et al. [8] describe a heap-modular interprocedural shape analysis for singly linked lists that can handle a bounded numbers of cutpoints. The main idea is to treat a bounded number of cutpoint-labels as, essentially, additional parameters: Every procedure can be seen as having k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, their analysis (non-deterministically) binds these additional parameters with references to the cutpoints. If the procedure has more than k cutpoint, they turn every piercing reference to a dangling pointer, which, essentially, makes the reference inaccessible. Thus, their analysis does not differentiate between dangling references and piercing references.

However, every program that it manages to analyze is a k-cutpoint-tolerant effectively cutpoint-free program.

Yang et al. [28] present a heap-modular interprocedural shape analysis that, similar to [8], is based on a domain of separation-logic formulae. Their experimental results indicate that the use of local heaps provides a speedup of $2-3\times$ in the analysis compared to a global heap analysis. Furthermore, the use of an interprocedural analysis that passes only the reachable portion of the heap was found to be one of the three key reasons for the scalability of their analysis. (The other two key reasons being an efficient join operator and the discard of intermediate states.) In this analysis, cutpoints are passed as additional (hidden) parameters to called procedures, but their number is not bounded. This is one of the possible reasons that their analysis may not terminate (although in many interesting cases it does). In later work [2], the problem of cutpoint abstraction is reduced because the compositional nature of the analysis allows to represent only a subset of the reachable heap.

Marron et al. [14] present a context-sensitive shape analysis that is employed for automatic parallelization of sequential heap manipulating programs. The interprocedural analysis is based on an abstraction of local heaps with cutpoints. The analysis employs an abstraction of cutpoint-labels that uses two main ideas: (i) avoid summarizing cutpoints that are generated by the local variables of the *immediate* caller and (ii) abstract all other cutpoints by recording the set of roots of access paths. The analysis also uses liveness information to avoid recording as cutpoints objects that are only pointed to by dead references.

Rubinstein [24] provides a preliminary study regarding the classification of cutpoints that occur in real-life Java programs. The study is conducted by monitoring program executions. Algorithms for detecting usages of piercing references [12] are presented but not implemented. While the experimental results are non-conclusive, they do indicate that in several interesting cases the unbounded number of cutpoints occur when the program manipulates shared *immutable* data structures. This can motivate special treatment for pure (i.e., readonly) methods (see Sec. 7.4).

A local interprocedural may-alias analysis is given in [7]. The key observation there is that a procedure operates uniformly on all aliasing relationships involving variables of pending calls. This method applies to programs with cutpoints. However, the lack of *must*-alias information may lead to a loss of precision in the analysis of destructive updates. For more details on the relation between [7] and local heap shape analysis see [19].

Local reasoning [10, 18] provides a way of proving properties of a procedure independently of its calling contexts by using the "frame rule". In some sense, the approach used in this paper is in the spirit of local reasoning. The $\mathcal{ECPF}$ semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap. Local reasoning allows for an arbitrary partitioning of the heap based on user-supplied specifications. In contrast, in our work, the partitioning of the heap is built into the concrete semantics, and abstract interpretation is used to establish properties in the absence of user-supplied specifications.

[12] The term a *live cutpoint* is used in [24] to refer to an object which gets dereferenced using a piercing reference.

Another relevant body of work is that concerning *encapsulation*, also known as *confinement* or *ownership*. (A review about different encapsulation models can be found in [15]). These works allow modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of these works, as described in [15], is that they all place various restrictions on the kind of sharing allowed in the heap, while pointers from the stack are generally left unrestricted. In our work, the semantics allows for arbitrary heap sharing within the same procedure, but restricts both the heap sharing and the stack *live* sharing across procedure calls.

9 Conclusions and Future Work

In this paper, we presented an interprocedural shape analysis for effectively cutpoint-free programs. The analysis is local in the heap and thus allows reusing the effect of a procedure at different calling contexts. We presented the first non-trivial solution for procedure calls with an unbounded number of cutpoints. The solution is limited because it applies only to pure (read-only) procedures; however, we believe that it opens the door for future work to address the important, and still open, problem of handling an unbounded number of live cutpoints under abstraction.

In general, we believe that the distinction between live piercing references and dead ones can benefit analyses that abstract an unbounded number of cutpoints by allowing them to focus on only abstracting cutpoints that are pointed to by live piercing references. We consider this issue to be future work.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 33–48, 2006.
2. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Symp. on Princ. of Prog. Lang. (POPL)*, pages 289–300. ACM, 2009.
3. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1990.
4. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *International Static Analysis Symposium (SAS)*, 2003.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang. (POPL)*, pages 238–252, New York, NY, 1977. ACM Press.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.
7. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1994.
8. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *International Static Analysis Symposium (SAS)*, 2006.
9. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.

10. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2001.
11. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *International Static Analysis Symposium (SAS)*, 2004.
12. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct. (CC)*, 1992.
13. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *International Static Analysis Symposium (SAS)*, 2000. Available at http://www.math.tau.ac.il/∼ tvla.
14. M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Int. Conf. on Comp. Construct. (CC)*, pages 245–259, 2008.
15. J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *The First International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003.
16. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
17. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang. (POPL)*, 1995.
18. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Symp. on Logic in Computer Science (LICS)*, 2002.
19. N. Rinetzky. *Interprocedural and Modular Local Heap Shape Analysis*. PhD thesis, Tel Aviv University, June 2008.
20. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.
21. N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *16th European Symposium on Programming (ESOP)*, pages 220–236, 2007.
22. N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct. (CC)*, 2001.
23. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *International Static Analysis Symposium (SAS)*, 2005.
24. S. Rubinstein. On the utility of cutpoints for monitoring program execution. Master's thesis, Tel Aviv University, Tel Aviv, Israel, 2006.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst. (TOPLAS)*, 24(3):217–298, 2002.
26. R. Shaham, E. Yahav, E.K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *International Static Analysis Symposium (SAS)*, 2003.
27. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
28. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Conf. on Computer Aided Verification (CAV)*, pages 385–398. Springer-Verlag, 2008.

A Formal Details Pertaining to the $\mathcal{ECPF}$ Semantics

In this section, we provide the technical details that were glanced over in Sec. 4.

A.1 Reachability

In this section, we give formal definitions for the notions of *reachability*. These definitions are based on the corresponding standard notions in 2-level stores. Intuitively, location l_2 is *reachable from* a location l_1 in a memory state σ if there is a directed path in the heap of σ from l_1 to l_2. A locations l is *reachable* in σ if it is reachable from a location which is referenced by some variable. Note that the inaccessible value, similarly to the *null* value, is not a location.

Definition 1 (Heap path). *A sequence of locations* $\zeta : \{0, \ldots, n \mid n \in \mathcal{N}\} \to Loc$ *is a* directed heap path *in a heap* $h \in \mathcal{H}$*, if for every* $0 \leq i < |\zeta| - 1$ *there exists* $f_i \in \mathcal{F}$ *such that* $h(\zeta(i), f_i) = \zeta(i+1)$*. A directed heap path* ζ goes from l_1*, if* $\zeta(0) = l_1$*, it* goes to l_2 *if* $\zeta(|\zeta| - 1) = l_2$*. A heap path* ζ traverses through l *if there exists* i *such that* $0 \leq i < |\zeta|$ *and* $l = \zeta(i)$.

Definition 2 (Reachability). *A location* l_2 *is* reachable from *a location* l_1 *in a memory state* $\sigma = \langle \rho, L, h \rangle$*, if there is a directed heap path in* h *going from* l_1 *to* l_2.

Definition 3 (Reachable locations). *A locations* l *is* reachable *in* σ *if it is reachable from a location which is referenced by some variable. We denote the set of* reachable locations *in* $\sigma \in \Sigma$ *by* $\mathcal{R}(\sigma)$*, i.e.,* $\mathcal{R}(\sigma) = \{l \in L | x \in \mathcal{V}$ *and* l *is reachable in* σ *from* $\rho(x) \in Loc\}$.

A.2 Properties of the $\mathcal{ECPF}$ Semantics

In this section, we formally define the notions of *observational soundness* and of *simulation* between the $\mathcal{ECPF}$ semantics and the standard semantics. To be precise, when referring to the standard semantics we refer to the standard store-based semantics $\mathcal{GSB}$ defined in [19,20]. In short, memory states in $\mathcal{GSB}$ are represented in the same way as memory states in $\mathcal{ECPF}$. The main difference between $\mathcal{GSB}$ and $\mathcal{ECPF}$ is that the operational semantics never blocks references in $\mathcal{GSB}$, and thus $\bullet$ is not a possible value.

Access paths We introduce access paths, which are the only means by which a program can observe a state. Note that the program cannot observe location names.

Definition 4 (Field Paths). *A* field path $\delta \in \Delta = \mathcal{F}^*$ *is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by* ϵ.

Definition 5 (Access path). *An* access path $\alpha = \langle x, \delta \rangle \in AccPath = \mathcal{V} \times \Delta$ *is a pair consisting of a local variable and a field path.*

Definition 6 (Access path value in the $\mathcal{ECPF}$ semantics). *The value of an access path* $\alpha = \langle x, \delta \rangle$ *in state* $\sigma = \langle \rho, L, h \rangle$ *of the* $\mathcal{ECPF}$ *semantics, denoted by* $[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma)$*, is defined to be* $\hat{h}(\rho(x), \delta)$*, where*

$$\hat{h}\colon Val \times \Delta \rightharpoonup Val \text{ such that}$$

$$\hat{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \quad \text{(note that } v \text{ might be } \bullet\text{)} \\ \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in Loc \\ undefined & \text{otherwise} \quad \text{(note that } v \text{ might be } \bullet\text{)} \end{cases}$$

Note that an access to a field of the inaccessible value is not defined.

Definition 7 (Comparable values). *A pair of values of the $\mathcal{ECPF}$ semantics $v_1, v_2 \in Val$ are* comparable, *denoted by $v_1 \overset{?}{\bowtie} v_2$, if $v_1 \neq \bullet$ and $v_2 \neq \bullet$.*

Definition 8 (Access path value in the $\mathcal{GSB}$ semantics). *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\sigma_G = \langle \rho, L, h \rangle$ of the $\mathcal{GSB}$ semantics, denoted by $[\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G)$, is defined to be $\overline{h}(\rho(x), \delta)$, where $Val_G = Val \setminus \{\bullet\}$ and*

$$\overline{h}\colon Val_G \times \Delta \rightharpoonup Val_G \text{ such that}$$
$$\overline{h}(v,\delta) = \begin{cases} v & \text{if } \delta = \epsilon \\ \overline{h}(h(v,f),\delta') & \text{if } \delta = f\delta',\ v \in Loc \\ undefined & \text{otherwise} \end{cases}$$

Observational soundness We define the notion of observational soundness between a $\mathcal{ECPF}$ memory state σ and a standard 2-level store σ_G of the $\mathcal{GSB}$ semantics as the preservations in σ_G of all equalities and inequalities which hold in σ.[7] Note that the preservation in the other direction is not required. Also note that an equality resp. inequality of values of access paths holds in σ only when the two access paths have comparable values. For simplicity, we define $[\![\texttt{null}]\!]_{\mathcal{ECPF}}(\sigma) = [\![\texttt{null}]\!]_{\mathcal{GSB}}(\sigma) = \mathit{null}$.

Definition 9 (Observational soundness). *The memory state $\sigma \in \Sigma$ is* observationally sound *with respect to memory state $\sigma_G \in \Sigma_G$, denoted by $\sigma_G \leqq \sigma$, if for every $\alpha, \beta \in AccPath \cup \{\texttt{null}\}$ it holds that*

if $[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma) \overset{?}{\bowtie} [\![\beta]\!]_{\mathcal{ECPF}}(\sigma)$ *then*
$[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma)) = [\![\beta]\!]_{\mathcal{ECPF}}(\sigma) \Leftrightarrow [\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G) = [\![\beta]\!]_{\mathcal{GSB}}(\sigma_G)$

We define the notion of observational soundness between two $\mathcal{ECPF}$ memory states (resp. two standard memory states) in a similar manner.

Simulation Before we define the notion of simulation we briefly review some execution traces accessing-functions (formally defined in [19]). Given an execution trace π, the initial resp. final memory state of an execution trace π, denoted by $in(\pi)$ resp. $out(\pi)$, is the current memory state in the first resp. last stack of program states. $\pi(i)$ returns the stack at the ith step of the execution and $|\pi(i)|$ returns its height. $path(\pi)$ is the sequence of program points which the execution traverses. *i.e.*, $path(\pi)(i)$ is the program point in the ith step of the execution. (We assume that every statement is labeled by a program point.)

The following theorem shows that $\mathcal{ECPF}$ simulates the standard semantics. In the lemma, we denote by $path(\pi)$ the sequence of intraprocedural statements and $Call$ and $Return$ operations executed in π. We also use $[\pi]_k$ to denote the memory state of the current procedure at $\pi(k)$, the kth program state of π

Theorem 1 (Simulation). *Let P be an effectively cutpoint-free program according to the $\mathcal{ECPF}$ semantics. Let π_S be a trace of a program P according to the standard semantics. There exists a trace π_E of P according to the $\mathcal{ECPF}$ semantics such that the folowing holds (i) $|\pi_S| = |\pi_E|$, (ii) $path(\pi_S) = path(\pi_E)$, and (iii) $[\pi_S]_k \leqq [\pi_E]_k$ for every $0 \leq k < |\pi_S|$.*

Sketch of Proof: The proof is done by induction on the length of the execution. We look at memory states as graphs. The graph nodes are the allocated objects and the graph edges are the object fields. The graph nodes may be labeled by variables. The graph edges are labeled by field names.

We prove that observational equivalence is preserved by showing a stronger property: every memory state $[\pi_E]_k$ produced by the $\mathcal{ECPF}$ can be seen as a subgraph of $[\pi_S]_k$, the corresponding memory state of the $\mathcal{GSB}$ semantics. Furthermore, that two graphs agree on the values of live references.

We maintain an injective and a surjective function ϱ from the set of objects that are reachable from the variables of the current procedure in a memory state of the $\mathcal{GSB}$ to the set of objects in the corresponding memory state of the $\mathcal{ECPF}$ semantics. Clearly when a program starts, and prior to the allocation of any object, the two memory states are isomorphic. It is easy to verify that atomic statement preserves the isomorphism: ϱ remains unchanged, except that object allocation maps the new location to the new individual.

When a procedure is invoked, the mapping ϱ is projected on the set of objects passed to the invoked procedure. When a procedure returns, the mapping of locations that were irrelevant for the invocation remains as in the call site. The mapping for locations that were relevant for the invocation, as well as those that were allocated during the invocation, are taken from the exit site. Note that the induction assumption ensures that the above scheme is well defined.

To show that the return memory state produced by the $\mathcal{ECPF}$ semantics is a subgraph of the corresponding return memory state of the $\mathcal{GSB}$ semantics agrees with it on the values of live references, we make the following argument: The computation of return states in the $\mathcal{ECPF}$ semantics blocks piercing references. The computation of the return states in the $\mathcal{GSB}$ semantics does not. Thus, it remains to show that all the references that gets blocked by the $\mathcal{ECPF}$ semantics are not live in the $\mathcal{GSB}$ semantics.

The computation of return states in the $\mathcal{ECPF}$ semantics restores all references from the caller's local heap to parameter objects which, by the induction assumption, must be in the ϱ relation. It only blocks the value of piercing references (i.e., it changes the value of every pointer field or variable pointing to a cutpoint). The execution π_S never uses a a field f of an object o such that the f-field in $\varrho(o)$ at the corresponding $\mathcal{ECPF}$ points to the inaccessible location. Otherwise, π_E is a non effectively cutpoint-free execution of P in $\mathcal{ECPF}$ which is a contradiction to the assumption that P is effectively cutpoint-free. For similar reasons, the value of a variable which gets blocked by the $\mathcal{ECPF}$ semantics does not get used by the $\mathcal{GSB}$ semantics.

Lemma 1. *Let P be an effectively cutpoint-free program. The following holds:*

[Invariants] *An invariant concerning equality of values of access paths in the $\mathcal{ECPF}$ semantics is an invariant in the standard semantics*

[Cleanness] *P does not dereferences null references in the standard semantics.*

Lemma 2. *Let P be an effectively cutpoint-free program. A reference, that at a given program point always has the inaccessible value, is not live at that program point in the standard semantics.*

Definition 10 (Observational equivalence). *The $\mathcal{ECPF}$ memory states $\sigma_1, \sigma_2 \in \Sigma$ are* observationally equivalent, *denoted by $\sigma_1 \lessapprox \sigma_2$, if $\sigma_1 \leqq \sigma_2$ and $\sigma_2 \leqq \sigma_1$.*

The following lemma shows that $\mathcal{ECPF}$ is indifferent to location names.

Theorem 2 (Indifference to location names). *Let π_1, π_2 be execution traces of a program P according to the $\mathcal{ECPF}$ semantics. If $|\pi_1(1)| = |\pi_2(1)| = 1, in(\pi_1) \lessapprox in(\pi_2)$ and $path(\pi_1) = path(\pi_2)$ then $out(\pi_1) \lessapprox out(\pi_2)$.*

B Update Formulae

In this section, we encode the abstract transformers using the notations of [25].

B.1 Intraprocedural Statements

The meaning of assignments is specified by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [25]. The inference rules for assignments are rather straightforward. We encode conditional using `assume()` statements.

The operational semantics for assignments is specified by *predicate-update formulae*: for every predicate p and for every statement st, the value of p in the 2-valued structure which results by applying st to S, is defined in terms of a formula evaluated over S.

The predicate-update formulae of the core-predicates for assignment is given in Fig. 13. The table also specifies the side condition which enables that application of the statement. These conditions check that null-dereference is not performed and that the *inaccessible value is not used*. The value of every core-predicate p after the statement executes, denoted by p', is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates whose update formula is not specified, are assumed to be unchanged, i.e., $p'(v_1, \ldots) = p(v_1, \ldots)$.

None of the assignments, except for object allocation, modifies the underlying universe. Object allocation is handled as in [25]: A new individual is added to the universe to represent the allocated object; the auxiliary predicate new is set to hold *only* at that individual; only then, the predicate-update formulae is evaluated.

The semantics transitions into the error state ($\sigma_\bullet$) under the same conditions as the $\mathcal{ECPF}$ semantics, i.e., when an inaccessible-valued variable or field are accessed. (See Fig. 7). The following side condition trigers such a transition when a variable `x` points to an inaccessible location $\exists v\colon x(v) \wedge ia(v_2)$. Similarly, the following side condition trigers such a transition when the `f`-field of the object pointed to by a variable `x` points to an inaccessible location $\exists v_1, v_2\colon x(v_1) \wedge f(v_1, v_2) \wedge ia(v_2)$.

B.2 Interprocedural Statements

The treatment of procedure call and return could be briefly described as follows: (i) constructing the memory state at the callee's entry site (S_e) and (ii) the caller's memory state at the call site (S_c) and the callee's memory state at the exit site (S_x) are used to

Statement	Predicate-update formulae	$\mathbf{side-condition}$
`y` $=$ `null`	$y'(v) = 0$	
`y` $=$ `x`	$y'(v) = x(v)$	$\forall v_1 \colon \neg(x(v_1) \wedge ia(v_1))$
`y` $=$ `x.f`	$y'(v) = \exists v_1 \colon x(v_1) \wedge f(v_1, v)$	$\exists v_1 \colon x(v_1) \wedge \neg ia(v_1) \wedge$ $\forall v_2 \colon \neg(x(v_1) \wedge f(v_1, v_2) \wedge ia(v_2))$
`y.f` $=$ `null`	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$	$\exists v_1 \colon y(v_1) \wedge \neg ia(v_1)$
`y.f` $=$ `x`	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$	$\exists v_1 \colon y(v_1) \wedge \neg ia(v_1) \wedge$ $\forall v_2 \colon \neg(x(v_2) \wedge ia(v_2))$
`y` $=$ `alloc`	$eq'(v_1, v_2) = eq(v_1, v_2) \vee new(v_1) \wedge new(v_2)$ $new'(v) = 0$	

Fig. 13. The predicate-update formulae defining the operational semantics of assignments. Note that we always assume that a reference variable is nullified before re-assigned.

construct the caller's memory state at the return site (S_r). We now formally define and explain these steps.

Fig. 14 specifies the procedure call rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. The rule is instantiated for each call statement in the program.

Computing The Memory State at the Entry Site. S_e, the memory state at the entry site to p, represents the local heap passed to p. It contains only these individuals in S_c that represent objects that are relevant for the invocation. It also contains the individual representing the inaccessible value. The formal parameters are initialized by $updCall_q^{y=p(x_1,\ldots,x_k)}$, defined in Fig. 15(a). The latter, specifies the value of the predicates in S_e using a predicate-update formulae evaluated over S_c. We use the convention that the updated value of x is denoted by x'. Predicates whose update formula is not specified, are assumed to be unchanged, i.e., $x'(v_1, \ldots) = x(v_1, \ldots)$. Note that only the predicates that represent variable values are modified. In particular, field values, represented by binary predicates, remain in p's local heap as in S_c.

Computing The Memory State at the Return Site. The memory state at the return-site (S_r) is constructed as a combination of the memory state in which p was invoked (S_c) and the memory state at p's exit-site (S_x). Informally, S_c provides the information about the (unmodified) irrelevant objects and S_x contributes the information about the destructive updates and allocations made during the invocation.

The main challenge in computing the effect of a procedure is relating the objects at the call-site to the corresponding objects at the return site. The fact that the invocation is effectively cutpoint-free guarantees that the only live references into the local heap are references to objects referenced by an actual parameter. This allows us to reflect the effect of p into the local heap of q by: (i) replacing the relevant objects in S_c with S_x,

Table 3. Formulae shorthands and their intended meaning.

Shorthand	**Formula**	**Intended Meaning**
$F(v_1, v_2)$	$\bigvee_{f \in \mathcal{F}} f(v_1, v_2)$	v_1 has a field that points to v_2
$\varphi^*(v_1, v_2)$	$(eq(v_1, v_2) \vee$ $(TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2))$	the reflexive transitive closure of φ
$R_{\{x_1,\ldots,x_k\}}(v)$	$\neg ia(v) \wedge$ $\bigvee_{x \in \{x_1,\ldots,x_k\}} \exists v_1 : x(v_1) \wedge F^*(v_1, v)$	v is reachable from $\mathtt{x}_1$ or $\mathtt{x}_2$ or ... or $\mathtt{x}_k$
$isCP_{q,\{x_1,\ldots,x_k\}}(v)$	$R_{\{x_1,\ldots,x_k\}}(v) \wedge$ $(\neg x_1(v) \wedge \ldots \wedge \neg x_k(v)) \wedge$ $(\bigvee_{y \in V_q} y(v) \vee$ $\exists v_1 : \neg R_{\{x_1,\ldots,x_k\}}(v_1) \wedge F(v_1, v))$	v is a cutpoint

$$Call_{y=p(x_1,\ldots,x_k)}(S_c) = S_e \qquad Ret_{y=p(x_1,\ldots,x_k)}(S_c, S_x) = S_r$$

where

$S_e = \langle U_e, \iota_e \rangle$ where
$U_e = \{u \in U^{S_c} \mid S_c \models R_{\{x_1,\ldots,x_k\}}(u) \vee ia(v)\}$
$\iota_e = updCall_q^{y=p(x_1,\ldots,x_k)}(S_c)$

$S_r = \langle U_r, \iota_r \rangle$ where
$Let\ U' = \{u.c \mid u \in U_c\} \cup \{u.x \mid u \in U_x\}$

$$\iota' = \lambda p \in \mathcal{P}. \begin{cases} \iota_c[inUc \mapsto \lambda v.1](p)(u_1, \ldots, u_m) & : \quad u_1 = w_1.c, \ldots, u_m = w_m.c \\ \iota_x[inUx \mapsto \lambda v.1](p)(u_1, \ldots, u_m) & : \quad u_1 = w_1.x, \ldots, u_m = w_m.x \\ 0 & : \quad otherwise \end{cases}$$

$in\ \ U_r = \{u \in U' \mid \langle U', \iota' \rangle \models inUx(u) \vee (inUc(u) \wedge \neg ia(u) \wedge \neg R_{\{x_1,\ldots,x_k\}}(u))$
$\iota_r = updRet_q^{y=p(x_1,\ldots,x_k)}(\langle U', \iota' \rangle)$

$Call_{y=p(x_1,\ldots,x_k)}(S_c) = \sigma_\bullet \qquad S_c \models \exists v\colon ia(v) \wedge (x_1(v) \vee \cdots \vee x_k(v))$
$Ret_{y=p(x_1,\ldots,x_k)}(S_c, S_x) = \sigma_\bullet \qquad S_x \models \exists v\colon ia(v) \wedge ret(v)$

Fig. 14. The inference rule for a procedure call $y = p(x_1, \ldots, x_k)$ by a procedure q. The functions $updCall_q^{y=p(x_1,\ldots,x_k)}$ and $updRet_q^{y=p(x_1,\ldots,x_k)}$ are defined in Fig. 15.

the local heap at the exit from p; (ii) redirecting all references to an object referenced by an actual parameter to the object referenced by the corresponding formal parameter in S_x; (iii) block every piercing reference.

Technically, S_c and S_x are *combined* into an intermediate structure $\langle U', \iota' \rangle$. The latter contains a copy of the memory states at the call site and at the exit site. To distinguish between the copies, the auxiliary predicates $inUc$ and $inUx$ are set to hold for individuals that originate from S_c and S_x, respectively.

Pointer redirection is specified by means of predicate update formulae, as defined in Fig. 15(b). The most interesting aspect of these update-formulae is the formula

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} x_i(v) & : \quad z = h_i \\ 0 & : \quad z \in \mathcal{V} \setminus \{h_1, \ldots, h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} ret_p(v) & : \quad z = y \\ inUc(v) \wedge z(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee & : \quad z \in V_q \setminus \{y\} \\ \quad \exists v_1 \colon z(v_1) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \vee & \\ \quad\quad \exists v_1 \colon z(v_1) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_1) \wedge inUx(v) \wedge ia(v) & \\ 0 & : \quad z \in \mathcal{V} \setminus V_q \end{cases}$$

$$f'(v_1, v_2) = inUx(v_1) \wedge inUx(v_2) \wedge f(v_1, v_2) \vee$$
$$inUc(v_1) \wedge inUc(v_2) \wedge f(v_1, v_2) \wedge \neg ia(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee$$
$$inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} \colon f(v_1, v_{sep}) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{sep}, v_2) \vee$$
$$inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} \colon f(v_1, v_{sep}) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_{sep}) \wedge ia(v_2)$$
$$inUc'(v) = inUx'(v) = 0$$

Fig. 15. Predicate-update formulae for the core predicates used in the procedure call rule. We assume that the p's formal parameters are $h_1, \ldots, h_k$. There is a separate update formula for every local variable $\mathtt{z} \in \mathcal{V}$ and for every field $\mathtt{f} \in \mathcal{F}$.

$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}$, defined below:

$$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v_2) \stackrel{\text{def}}{=} \begin{array}{l} inUc(v_1) \wedge ia(v_1) \wedge inUx(v_2) \wedge ia(v_2) \vee \\ \bigvee_{i=1}^{k} inUc(v_1) \wedge x_i(v_1) \wedge inUx(v_2) \wedge h_i(v_2) \end{array}$$

This formula matches an individual that represents a (parameter) object which is referenced by an actual parameter at the call-site, with the individual that represents the object which is referenced by the corresponding formal parameter at the exit-site. The assumption that formal parameters are not modified allows us to match these two individuals as representing the same object. Once pointer redirection is complete, all individuals originating from S_c and representing relevant objects are removed, resulting with the updated memory state of the caller. In addition, the formula matches the individual representing the inaccessible value at the call site with the one representing the inaccessible value at the return site, thus preserving the value of inaccessible references from before the call.

We block piercing references using formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, defined in Tab. 3. The formula holds when v is a cutpoint object. It is comprised of three conjuncts. The first conjunct, requires that v be reachable from an actual parameter. The second conjunct, requires that v not be pointed-to by an actual parameter. The third conjunct, requires that v be an entry point into p's local heap, i.e., is pointed-to by a local variable of q (the caller procedure) or by a field of an object not passed to p.

Predicate update formulae for instrumentation predicates. Fig. 16 provides the update formulae for instrumentation predicates used by the procedure call rule. We use

$PT_X(v)$ as a shorthand for $\bigvee_{x \in X} x(v)$. The intended meaning of this formula is to specify that v is pointed to by some variable from $X \subseteq \mathcal{V}$. We use $bypass_X(v_1, v_2)$ as a shorthand for $(F(v_1, v_2) \wedge \neg R_X(v_1))^*$. The intended meaning of this formula is to specify that v_2 is reachable from v_1 by a path that does not traverse any object which is reachable from any variable in $X \subseteq \mathcal{V}$. Note that, again, formula $match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_1, v_2)$ again plays a central role.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$ils'(v) = ils(v) \wedge \neg(PT_{x_1,\ldots,x_k}(v) \vee isCP_{q,\{x_1,\ldots,x_k\}}(v)) \vee \exists v_1, v_2 \colon R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_1, v) \wedge F(v_2, v) \wedge \neg eq(v_1, v_2))$$

$$r'_y(v) = \begin{cases} r_{x_i}(v) & : \quad y = h_i \\ 0 & : \quad y \in \mathcal{V} \setminus \{h_1, \ldots, h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$\begin{aligned}
ils'(v) &= ils(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v)) \vee \\
&\quad PT_{x_1,\ldots,x_k}(v) \wedge \exists v_1, v_2, v_3 \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \wedge \neg eq(v_2, v_3) \wedge \\
&\quad\quad inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_2, v_1) \wedge \\
&\quad\quad\quad (inUc(v_3) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_3) \wedge F(v_3, v_1) \vee inUx(v_3) \wedge F(v_3, v)) \\
r'_{obj}(v_1, v_2) &= r_{obj}(v_1, v_2) \wedge inUx(v_1) \wedge inUx(v_2) \vee \\
&\quad r_{obj}(v_1, v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\
&\quad\quad inUc(v_1) \wedge inUx(v_2) \wedge \exists v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\quad\quad\quad bypass_{\{x_1,\ldots,x_k\}}(v_1, v_a) \wedge r_{obj}(v_f, v_2) \\
r'_x(v) &= inUc(v) \wedge r_x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee \\
&\quad\quad inUx(v) \wedge \exists v_x, v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\quad\quad\quad x(v_x) \wedge bypass_{\{x_1,\ldots,x_k\}}(v_x, v_a) \wedge r_{obj}(v_f, v)
\end{aligned}$$

Fig. 16. The predicate update formulae for the instrumentation predicates used in the procedure call rule. We give the semantics for an arbitrary function call $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. We assume that the p's formal parameters are $h_1, \ldots, h_k$.

A Local Shape Analysis Based on Separation Logic : Detailed Presentation and Soundness Proof *

Amin Timany
amin.timany@cs.kuleuven.be

Bart Jacobs
bart.jacobs@cs.kuleuven.be

Computer Science Department
K. U. Leuven

1 Introduction

Shape analysis is a static analysis of the source code of a program to determine shapes and manipulations of the dynamically allocated data structures at each point which that program can reach in an execution.

To do so, the analysis presented in this report computes (an overapproximation of) the set of states that the program can possibly be in, at each point in the program. This is done by computing all possible states that the program can be in, before execution of each statement of the program, i.e., that statement's *invariant*.

Here, we use a denotational style semantics to define semantics of programs. In denotational semantics, we consider a domain set $\mathcal{D}$. The semantics of statements (also that of the whole program) is then defined as a function on that domain; i.e., the semantics of a statement c is $[\![c]\!] : \mathcal{D} \to \mathcal{D}$. Here, we use the power set, $2^{\mathfrak{S}}$, of a set of states, $\mathfrak{S}$, as the domain. As a result, given the invariant of a statement, computing the invariant of the next statement is straightforward. We simply apply the statement's semantics to its invariant and get the invariant of the next statement [1]. We assume that the set of all states that the program can possibly be in before execution of the program begins (program's precondition) is provided together with the program being analyzed. The analysis then computes invariants of all program statements and also the postcondition of the whole program.

As the set of states that the program can be in is generally not computable, accurate computation of invariants is impossible. Therefore, we compute a sound overapproximation of them. We compute an overapproximation so that we have a guarantee that whenever a state is part of an invariant, it is also represented as part of the overapproximation of that invariant computed by the analysis. Here, the word *sound* is put to emphasize the fact that if a program has a reachable faulty state, so does the overapproximation. This assures that whenever the analysis indicates that a program is free of memory faults it is indeed the case [2].

*Based on [DOY06], a paper with the same title by Dino Distefano, Peter W O'Hearn, and Hongseok Yang. Here, we give a rigorous presentation and soundness proof of the shape analysis approach they presented.

[1] In case the next statement is a loop, the result is not the invariant of the next statement (the loop) but, as we shall see, can be used to compute the invariant of that loop

[2] The other direction, of course, does not hold (the analysis can indicate that a program is unsafe while it is not) as a sound and complete analysis is non-computable as mentioned above

The term "program state", as repeatedly referred to above, is generic and can cover a wide range of possibilities. The states of a program may simply be representations of its store (assigning values to program variables) and heap or they can be describing properties about data structures residing in the heap such as sortedness of a linked list, it being the reverse of another linked list, etc. In this regard, there are shape analyses that can be used to obtain a wide range of information about the programs they analyze, e.g., those presented in [LRS04, OCDY06, CR08, CRN07]. In this report, we consider a very simple programming language, *SimpleLang*, in which all program variables are assumed to be pointers pointing to heap cells. Heap cells, then, are themselves considered as pointers pointing to other heap cells. In other words, this programming language can be used to write programs manipulating singly-linked lists where the data field(s) of the linked lists are abstracted away, i.e., each linked list cell is considered to only point to the next cell in the linked list.

As we will discuss shortly, we will define three semantics for this language. One is a low-level semantics considered as the reference semantics. This semantics simply represents states as stores and heaps, respectively, a mapping from program variables to heap cells and a partial mapping from heap cells to where they point to (in case allocated). The other two are high level semantics, representing states with a variant of symbolic heaps. Symbolic heaps are a special form of separation logic formulas defined for symbolic execution [BCO05]. This representation uses (separation logic) predicates to indicate fragments of heap containing acyclic linked list segments of length at least one. Choosing to represent states in this symbolic way, not only does it allow to confirm lack of memory faults from a successful analysis but also gives information about different linked lists residing in the heap and their cyclicities.

In the rest of this report, we describe the syntax of *SimpleLang* and define three different semantics for programs expressed in this language, namely, *concrete* semantics, *symbolic* semantics and *abstract* semantics.

In concrete semantics (Sem_κ), we use a low level representation of a store and a heap to describe states. In other words, a state is a pair of a heap and a store, or an error state representing a state where a memory fault has occurred. Heap cells are represented by natural numbers, their memory address. A store is represented by a mapping that maps program variables to heap cells or nil (the null pointer), and a heap is represented by a partial mapping that maps heap cells to heap cells or nil if they are allocated. Concrete semantics is the basic semantics defined to correspond to a low level interpretation of the language.

In symbolic semantics (Sem_σ), program states are defined with help of symbols. We use nil and program variables as symbols to stand for themselves, i.e., the null pointer and where program variables point respectively. We, in addition, introduce primed variables, to represent heap cells that are (possibly) not pointed to by program variables, or nil. Primed variables are by definition assumed to be existentially quantified in a state. A symbolic state then, is either an error state or consists of a pair of a spatial part and a pure part. The spatial part expresses spatial assertions, e.g., $Ls(x, y')$, which asserts that there is a linked list segment starting from the heap cell pointed to by program variable x to some other heap cell or nil as represented by primed variable y'. The pure part, on the other hand, expresses equalities between symbols, e.g., asserting y' equals nil.

In abstract semantics (Sem_α), on the other hand, we introduce an abstraction that is a mapping from symbolic states to (abstract) symbolic states. The abstract semantics, then, is symbolic semantics after which the abstraction is applied to the resulting states.

We will establish a modeling relation between concrete and symbolic states and show

	Sem_κ	Sem_σ	Sem_α
States	pairs of heaps and stores or error	pairs of spatial and pure (equality) assertions or error	Abstracted symbolic states
Overapproximates	–	Sem_κ	Sem_κ
$\lvert$ States $\rvert$	Infinite	Infinite	Finite
Computing Invariants	Non-computational	Non-computational	Computational

Table 1: Summary of properties of concrete, symbolic and abstract semantics

$$
\begin{array}{rcl}
x & \in & \mathit{Vars} \\
\mathit{stm} & ::= & \textbf{new}(x) \mid \textbf{free}(x) \mid \mathit{exp} := \mathit{exp} \mid \textbf{while}(\mathit{bexp})\ \{\mathit{stm}\} \mid \mathit{stm}; \mathit{stm} \\
\mathit{exp\text{-}s} & ::= & x \mid \textbf{nil} \\
\mathit{exp} & ::= & \mathit{exp\text{-}s} \mid x.\textbf{next} \\
\mathit{bexp} & ::= & \mathit{exp\text{-}s} == \mathit{exp\text{-}s} \mid \mathit{exp\text{-}s} \mathrel{!}= \mathit{exp\text{-}s}
\end{array}
$$

Figure 1: The grammar of the programming language

that symbolic semantics and abstract semantics are both sound overapproximations of concrete semantics. Furthermore, we show that the set of abstract states is finite which means abstract semantics can be used to compute invariants and ultimately for shape analysis. Table 1 shows a summary of properties of concrete, symbolic and abstract semantics and their relation.

In the rest of this report we assume that the reader is familiar with basic notions of lattice and domain theory, e.g., complete lattices, the fact that the power set of any set with subset relation forms a complete lattice, (Scott-)continuous functions, etc.

2 The Programming Language

Here, we define the simple language (*SimpleLang*) that we are going to use throughout the rest of the report. This language is minimally designed for the purpose of formalizing concepts presented in this report. *SimpleLang* can express programs that manipulate singly-linked lists where data field(s) are abstracted away, i.e., linked lists only have a "next" field. Here, we give the syntax of *SimpleLang* and discuss its general semantics.

2.1 Syntax

The BNF grammar of *SimpleLang* is given in Figure 1. In this figure, the words in bold are keywords and Vars is the set of program variables. A program is simply a statements (represented as stm in the grammar). In the sequel, we use $x, y, z, \ldots$ to refer to program variables. In addition, we use e, E, b and c (possibly indexed) to represent simple expressions ($\mathit{exp\text{-}s}$ in grammar), expressions (exp in grammar), boolean expressions (bexp in grammar) and statements (stm in grammar), respectively.

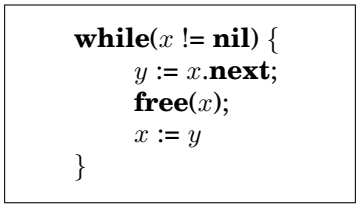


Figure 2: Dispose program: a program disposing a list

All program variables are assumed to be pointers that can be **nil** (the null pointer) or pointing to some memory location corresponding to a structure (in the sense of C structures) that has a next field. A program variable can be followed by a dot and the **next** keyword which indicates accessing the next field of the structure pointed to by that variable. Keywords **new** and **free** respectively allocate and deallocate memory cells. The **new**(x) statement changes the value of variable x so that it points to the newly allocated memory cell and the **free**(x) statement requires variable x to be pointing to some allocated memory cell which will be deallocated after its execution.

Effectively, this language can express programs that represent manipulations of singly linked data structures, e.g., cyclic or acyclic linked lists. As an example, Figure 2 depicts a program that disposes a list whose head is pointed to by x.

2.2 General Semantics

Here, we give a general account of semantics, we discuss how a semantics for *SimpleLang*, independent of the set of states, is defined. We use denotational semantics to define the semantics of programs. Assuming $\mathfrak{S}$ is the set of states, we define the domain of interpretation as $2^{\mathfrak{S}}$ (the power set of $\mathfrak{S}$). Hence, for a statement c, the semantics of c is a function $[\![c]\!] : 2^{\mathfrak{S}} \to 2^{\mathfrak{S}}$. In defining the semantics for a basic statement c (allocation, deallocation and assignment), we define a preliminary semantics $[\![c]\!]^\dagger : \mathfrak{S} \to 2^{\mathfrak{S}}$ and define the actual semantics of c based on that.

$$[\![c]\!](S) = \bigcup_{st \in S} [\![c]\!]^\dagger(st)$$

The semantics of the composition of statements is then defined as the composition of the semantics functions of those statements. In other words,

$$[\![c_1; c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$$

Given the semantics of the basic statements and the composition of statements, we define the semantics of the while loops, independent of the set of states chosen. To do so, we use the function

$$\mathit{filter} : \mathit{bexp} \to \mathfrak{S} \to \mathfrak{S}$$

which, given a boolean expression (bexp) and a set of states, filters out all states that are not compatible with the given boolean expression.

Assuming $S_0 \subseteq \mathfrak{S}$ is the set of states before a while loop **while**(b)$\{c\}$, we define the function $f_{S_0}(S)$ as follows:

$$f_{S_0}(S) = S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S)$$

f_{S_0}, is the function that assuming that states in S are some states that the program can be in before the while loop (e.g., resulting from previous iterations of the loop), gets the new set

of states that the program can be in before the loop. In other words, considering S_0 as a set of states that we already know that the program can be in before entering the while loop, it evaluates the effect of one iteration of the while loop on members of S and considers the resulting states to also be part of the states that the program can be in before entering the while loop. In particular, assuming S_0 is the set of states that the program can be in before entering the while loop for the first time, $f^{n+1}_{S_0}(\varnothing)$ is the set of states that the program can be in before the while loop assuming the loop has already been evaluated up to n times. Given the definitions above, we can see that:

$$\begin{aligned} f_{S_0}(\varnothing) &= S_0 \\ f^2_{S_0}(\varnothing) &= S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S_0) \\ f^3_{S_0}(\varnothing) &= S_0 \cup [\![c]\!] \circ \mathit{filter}(b)(S_0) \cup ([\![c]\!] \circ \mathit{filter}(b))^2(S_0) \\ &\vdots \end{aligned}$$

Hence, the set of all states that the program can be in before the while loop is:

$$\varnothing \cup f_{S_0}(\varnothing) \cup f^2_{S_0}(\varnothing) \cup f^3_{S_0}(\varnothing) \cup \ldots \tag{1}$$

On the other hand, our domain, $2^{\mathfrak{S}}$, together with $\subseteq$ relation forms a complete lattice. Moreover, it is obvious, according to the definition of semantics as given above, that $[\![c]\!]$ for any basic statement c is continuous. For composition of statements, we will shortly discuss that the semantics defined below for the while loops (Formula 2, below) is also continuous, which shows continuousness of the semantics of the body of while loops; even if they contain other while loops. Hence, according to Kleene's fixpoint theorem, the Formula 1, above, is the least fixpoint of the function f_{S_0}. Therefore, we define the semantics of while loops as follows:

$$[\![\textbf{while}(b)\{c\}]\!](S) = \mathit{filter}(\neg b)(lfix(\lambda S'.S \cup [\![c]\!] \circ \mathit{filter}(b)(S'))) \tag{2}$$

where, $\neg b$ is simply the negation of b, i.e., the result of swapping "==" and "! =" in b.

Moreover, as alluded to earlier, the semantics definition of the while loops given above, is continuous. This we can see from the fact that:

$$\forall \mathcal{X} \subseteq 2^{\mathfrak{S}}.\ [\![\textbf{while}(b)\{c\}]\!](\bigcup_{X \in \mathcal{X}} X) = \bigcup_{X \in \mathcal{X}} [\![\textbf{while}(b)\{c\}]\!](X)$$

This means that the definition above for the semantics of the while loops can be used to give an inductively defined semantics for all of *SimpleLang*.

It is worth noting that the $f^n_{S_0}(\varnothing)$ is increasing with respect to $\subseteq$, i.e.,

$$\forall n \in \mathbb{N}.\ f^n_{S_0}(\varnothing) \subseteq f^{n+1}_{S_0}(\varnothing)$$

This can be easily shown with an induction over n. Hence, one can try to compute the fixpoint of the semantics of the while loop by computing $f_{S_0}(\varnothing), f^2_{S_0}(\varnothing), \ldots$ until reaching the least fixpoint, i.e., some n for which $f^{n+1}_{S_0}(\varnothing) = f^n_{S_0}(\varnothing)$. Although, in the general case, there is no guarantee that there is such a finite n for which $f^n_{S_0}(\varnothing)$ is the least fixpoint. However, for the case of abstract semantics, as we will discuss, since the set of states (and hence the domain which is its power set) is finite, such a finite n always exists. Throughout the rest of this report, we show example programs for which we compute the fixpoint using this approach and also discuss some examples for which no finite n exists such that $f^n_{S_0}(\varnothing)$ is the least fixpoint.

The way the semantics of the while loops are defined here is not the conventional way in denotational semantics. There, usually the fixpoint is applied to the semantics function itself; i.e., a lattice of (partial) semantics functions is defined for the while loop together with a completion operation on those partial functions (adding to the domain of the definition of the partial semantics functions). The semantics of the loops, then, is defined as the least fixpoint of this completion operation over the lattice of partial functions. Here, the choice to define the semantics of the while loops differently, is so that the fixpoint is applied to sets of states rather than (partial) semantics functions.

Here, we have shown a way to define the semantics of the language using the preliminary semantics for basic statements and the filter function, independent of the set of states chosen. Hence, in the rest of the report, in order to define a semantics, we simply define the set of states, the preliminary semantics for basic statements and the filter function for that semantics. Furthermore, to define the preliminary semantics of basic statements, we define a relation of the form $st, c \Rightarrow^{\dagger} st'$ where c is a basic statement and $st, st' \in \mathfrak{S}$. We then derive the preliminary semantics as follows:

$$[\![c]\!](S) = \{st' \mid st \in \mathfrak{S} \wedge st, c \Rightarrow^{\dagger} st'\}$$

3 Concrete Semantics

In order to give a basic definition of the behavior of programs written in our simple language, we define a denotational semantics called concrete semantics (Sem_{κ}). A state of Sem_{κ} is either an error state $\top_{\kappa}$ or a simple representation of program memory, i.e., a mapping from program variables to heap cells and a representation of the program heap.

Definition 3.1 (Concrete States)**.** Let Vars be the set of program variables. Then, we define concrete program heaps, concrete stores and concrete states as follows:

- A heap $h \in \mathbb{H}_{\kappa}$ is a partial function $h : \mathbb{N} \rightharpoonup \mathit{Values}$
- A store $s \in \mathbb{ST}_{\kappa}$ is a function $s : \mathit{Vars} \to \mathit{Values}$
- The set of concrete states is $\mathfrak{S}_{\kappa} = (\mathbb{H}_{\kappa} \times \mathbb{ST}_{\kappa}) \cup \{\top_{\kappa}\}$

Where, $\mathit{Values} = \mathbb{N} \cup \{\mathit{nil}\}$. Furthermore, for a heap $h \in \mathbb{H}_{\kappa}$, we use $\mathit{dom}(h)$ to refer to all natural numbers for which h is defined. Where $a \in \mathbb{N}$ is a natural number for which $h(a)$ is undefined, we write $h(a) = \bot$. ■

Note that, nil and memory location 0 are different; the former is the null pointer while the latter is a valid memory address that can be allocated to and accessed, just like any other memory address.

To depict concrete states, we use a table to represent heap and store. As an example, the following table represents a concrete state $(h, s) \in \mathfrak{S}_{\kappa}$ of the program depicted in Figure 2. In this state, program variables x and y are respectively pointing to heap cell 10 and nil; while heap cell 10 points to heap cell 13 which in turn points to nil, i.e., x points to a linked list of length 2.

h		s	
address	value	variable	value
10	13	x	10
13	nil	y	nil

As we explained earlier, we only need to define the preliminary semantics for basic statements and the filter function and we do this by establishing a relation $\Rightarrow^{\dagger}_{\kappa}$ on statements and states.

In the sequel, we use $f[a \mapsto b]$ to denote the result of updating function f such that it maps a to b. In particular, $f[a \mapsto \bot]$ denotes the updating of partial function f such that it has no value for a.

$$f[a \mapsto b](d) \quad = \quad \begin{cases} b & \text{if } d = a \\ f(d) & \text{otherwise} \end{cases}$$

Definition 3.2. The rules defining the relation $\Rightarrow^{\dagger}_{\kappa}$ used to define the preliminary concrete semantics for basic program statements are as follows:

$$\frac{\text{any basic statement } c}{\top_\kappa, c \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{l \in \mathbb{N} \setminus dom(h) \text{ and } n \in Values}{(h,s), \textbf{new}(x); \Rightarrow^{\dagger}_{\kappa} (h[l \mapsto n], s[x \mapsto l])}$$

$$\frac{s(x) \in dom(h)}{(h,s), \textbf{free}(x) \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto \bot], s)} \qquad \frac{s(x) \notin dom(h)}{(h,s), \textbf{free}(x) \Rightarrow^{\dagger}_{\kappa} \top_\kappa}$$

$$\frac{}{(h,s), x := e \Rightarrow^{\dagger}_{\kappa} (h, s[x \mapsto val_\kappa(s,e)])} \qquad \frac{s(y) \in dom(h)}{(h,s), x := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} (h, s[x \mapsto h(s(y))])}$$

$$\frac{s(y) \notin dom(h)}{(h,s), x := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{s(x) \in dom(h)}{(h,s), x.\textbf{next} := e \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto val_\kappa(s,e)], s)}$$

$$\frac{s(x) \notin dom(h)}{(h,s), x.\textbf{next} := e \Rightarrow^{\dagger}_{\kappa} \top_\kappa} \qquad \frac{s(x), s(y) \in dom(h)}{(h,s), x.\textbf{next} := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} (h[s(x) \mapsto h(s(y))], s)}$$

$$\frac{s(x) \notin dom(h) \text{ or } s(y) \notin dom(h)}{(h,s), x.\textbf{next} := y.\textbf{next} \Rightarrow^{\dagger}_{\kappa} \top_\kappa}$$

■

Where $val_\kappa(s,e)$ for e, a simple expression, and $s \in \mathbb{ST}_\kappa$ is the concrete valuation of e under s. It is defined as follows:

$$val_\kappa(s,e) = \begin{cases} s(e) & \text{if } e \in Vars \\ nil & \text{if } e = nil \end{cases}$$

Definition 3.3. The concrete filter function $filter_\kappa$ is defined as follows:

$$\begin{array}{lcl} filter_\kappa(e_1 == e_2)(S) & = & \{(h,s) \in S \mid val_\kappa(s,e_1) = val_\kappa(s,e_2)\} \cup \{\top_\kappa \mid \top_\kappa \in S\} \\ filter_\kappa(e_1 != e_2)(S) & = & \{(h,s) \in S \mid val_\kappa(s,e_1) \neq val_\kappa(s,e_2)\} \cup \{\top_\kappa \mid \top_\kappa \in S\} \end{array}$$

■

This concludes the definition of concrete semantics. As an example, Figure 4 depicts a computation of the least fixpoint for the while loop of the dispose program depicted in Figure 2 by assuming an initial set S_0 of states that the program can be in before entering the loop. It also shows the result of the semantics of the while loop of this program applied to S_0.

On the other hand, Figure 5 shows that the least fixpoint of the semantics of the while loop in the program depicted in Figure 3 is not obtainable with iterations of computing $f^n_{S_0}(\varnothing)$.

```
while(nil == nil) {
    new(x);
}
```

Figure 3: Infinite allocation: a program that allocates heap cells indefinitely

$S_0 =$

h		s	
addr.	val.	var.	val.
10	13	x	10
13	nil	y	nil

$[\![dispose]\!]_\kappa(S_0) =$

h		s	
addr.	val.	var.	val.
		x	nil
		y	nil

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$

1

h		s	
addr.	val.	var.	val.
10	13	x	10
13	nil	y	nil

2

h		s		,	h		s	
addr.	val.	var.	val.		addr.	val.	var.	val.
10	13	x	10	,	13	nil	x	13
13	nil	y	nil				y	13

3

h		s		,	h		s		,	h		s	
addr.	val.	var.	val.		addr.	val.	var.	val.		addr.	val.	var.	val.
10	13	x	10	,	13	nil	x	13	,			x	nil
13	nil	y	nil				y	13				y	nil

Figure 4: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the concrete semantics. In this case, $f^3_{S_0}$ is the least fixpoint. In addition, $[\![dispose]\!]_\kappa$ is the concrete semantics of the whole dispose program (while loop).

$S_0 =$

h		s	
addr.	val.	var.	val.
		x	10

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{$ h: addr. | val. — ; s: var. | val. — x | 10 $\}$
2	$\{$ h: addr. | val. — 1 | nil ; s: var. | val. — x | 1 , h: addr. | val. — 1 | 0 ; s: var. | val. — x | 1 , h: addr. | val. — 2 | nil ; s: var. | val. — x | 2 , $\ldots$ $\}$
3	$\{$ h: addr. | val. — 1 | nil ; 2 | 0 ; s: var. | val. — x | 2 , h: addr. | val. — 1 | 0 ; 10 | nil ; s: var. | val. — x | 10 , h: addr. | val. — 2 | nil ; 10 | 13 ; s: var. | val. — x | 10 , $\ldots$ $\}$
$\vdots$	$\vdots$

Figure 5: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the concrete semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

4 Symbolic Semantics

In this section we introduce symbolic semantics for programs of *SimpleLang*. Symbolic states are high-level representations of program memory where actual addresses of cells are represented by symbols. Since we are using a symbolic representation of memory, we do not need stores and need only to represent heaps. To do so, we use a variant of symbolic heaps. Symbolic heaps are special separation logic formulas presented in [BCO05] to be used for symbolic execution. A symbolic heap is a separation logic formula consisting of conjunctions of a spatial formulas and pure assertions (assertions on equalities of symbolic expressions). Here, we define symbolic states to be pairs of separation logic formulas of spatial assertions and pure parts which represent equalities by explicit representation of equivalence classes over symbols.

The spatial assertions we use for the spatial part of symbolic states express facts of the form $x \mapsto y$ which signifies that the memory location indicated by symbol x represents a heap cell pointing to symbol y or $Ls(x, y)$ which signifies the fact that there is a linked list segment in the heap starting from memory location represented by symbol x ending in symbol y. We denote the set of symbols with Sym and it is defined as follows:

$$\mathsf{Sym} = \mathit{Vars} \cup \mathit{Vars}' \cup \{\mathit{nil}\}$$

where, Vars is the set of programs, they stand for memory locations pointed to by them, Vars' is a countably infinite set of primed variables $x', y', \ldots$ that represent some memory location (potentially) not pointed to by program variables and nil stands for the null pointer.

Definition 4.1 (Symbolic States)**.** Symbolic states are defined as follows:

– A spatial part $\Sigma \in \mathbb{SP}_\sigma$ is a separation logic formula defined below:

$$\begin{array}{lll} \varsigma_1, \varsigma_2 & \in & \mathsf{Sym} \\ \Phi & ::= & Ls(\varsigma_1, \varsigma_2) \mid \varsigma_1 \mapsto \varsigma_2 \mid \mathit{Junk} \mid \mathit{emp} \mid \Phi * \Phi \end{array}$$

– A pure part $\Pi \in \mathbb{PR}_\sigma$ is a set $\Pi : 2^{2^{\mathsf{Sym}}}$ of equivalence classes of symbols

– The set of symbolic states is $\mathfrak{S}_\sigma = (\mathbb{SP}_\sigma \times \mathbb{PR}_\sigma) \cup \{\top_\sigma\}$

Furthermore, we use $\varsigma_1 =_\Pi \varsigma_2$ to stand for the fact that $\varsigma_1 = \varsigma_2$ or ς_1 and ς_2 belong to the same equivalence class in Π and use $\varsigma_1 \neq_\Pi \varsigma_2$ as its negation. In cases where we do not distinguish between Ls and $\mapsto$, we use capital letter P (possibly indexed) to denote them, i.e., we implicitly assume that $P \in \{Ls, \mapsto\}$. ■

We assume that the primed variables that appear in a symbolic state are all existentially quantified. That is to say, by a symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ in which primed variables $x'_1, \ldots, x'_n$ appear, we mean:

$$\exists x'_1, \ldots, x'_n.\ (\Sigma, \Pi)$$

In the above definition, $*$ is the separating conjunction. We later discuss the relation between concrete and symbolic semantics and give the exact interpretation of separating conjunction. For now, suffice it to say that a separation logic formula $A * B$ means that the heap of the program can be divided into two disjoint subheaps (no address is given a value in both subheaps) such that one is represented by A and the other is represented by B. A separation logic formula with no separating conjunction (a single instance of a predicate) is called a (heap or memory) chunk. As the intuitive meaning of separating conjunction explained here suggests, the separating conjunction (similarly to classical logic's conjunction connective) is commutative and associative. Therefore, in the sequel we consider two spatial parts of symbolic states equal if they only differ in the order of chunks. For further reading on separation logic, refer to the seminal paper [Rey] by J. C. Reynolds, where he introduces separation logic.

Intuitively, the heap chunk $Ls(\varsigma_1, \varsigma_2)$ represents (part of) a heap where a non-empty (i.e., $\varsigma_1 \neq \varsigma_2$) acyclic linked list in the heap starting in the memory location represented by ς_1 ending in the memory location (if $\varsigma_2 \neq \mathit{nil}$) represented by ς_2. The heap chunk $\varsigma_1 \mapsto \varsigma_2$ represents (part of) a heap where memory location represented by ς_1 points to memory location (if $\varsigma_2 \neq \mathit{nil}$) represented by ς_2. The heap chunk Junk, on the other hand, represents (a part of) heap that contains at least one allocated memory location. This chunk simply signifies memory leaks. Finally, the emp chunk represents (a part of) heap that is empty, i.e., no memory location is allocated. As emp chunk represents an empty (part of) heap, in the sequel, we assume two spatial parts Σ and $\Sigma * \mathit{emp}$ equal.

As we discussed earlier, all primed variables are implicitly assumed to be existentially quantified. Therefore, we consider two symbolic states equal if they are equal up to renaming of primed variables. That is, in addition to the fact that spatial parts are equal up to reordering of their chunks and addition of emp chunks.

Definition 4.2 (Symbolic State Equality)**.** Let $st_1, st_2 \in \mathfrak{S}_\sigma$ be two symbolic states. Then, st_1 and st_2 are equal ($st_1 = st_2$) if and only if

$$st_1 = st_2 = \top_\sigma$$

or

$$st_1 = (\Sigma_1, \Pi_1) \text{ and } st_2 = (\Sigma_2, \Pi_2) \text{ such that } st_1 = \hat{r}(st_2)$$

for some bijection $r : Vars' \to Vars'$. Where $\hat{r}(\Sigma, \Pi)$ is the simultaneous renaming of primed variables in Σ and Π according to r and the equality between spatial parts is considered up to reordering of their chunks and addition of emp chunks.

In the sequel, we will call such a renaming that makes two symbolic states equal their *equalizer renaming*. ■

Note that the definition above is obviously reflexive, symmetric and transitive, i.e., it is an equivalence relation.

Definition 4.3 (Consistency of Symbolic States)**.** A non-error symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ is consistent (written as $(\Sigma, \Pi) \not\models false$) if *none* of the following holds:

I. There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ or $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ and $\varsigma_1 =_\Pi nil$

II. There a Σ' such that $\Sigma = P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4) * \Sigma'$ and $\varsigma_1 =_\Pi \varsigma_3$

III. There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ such that $\varsigma_1 =_\Pi \varsigma_2$

■

Moreover, for $e_1, e_2 \in Vars \cup \{nil\}$, we need to be able to determine whether a non-error symbolic state $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ entails $e_1 = e_2$ or $e_1 \neq e_2$. For the case of equality, $e_1 = e_2$, we simply see if $e_1 =_\Pi e_2$. On the other hand, for the case of inequality, the entailment might be drawn from the spatial part. In particular, we say (Σ, Π) entails $e_1 \neq e_2$ if adding equality of e_1 and e_2 to it makes it inconsistent. This is formally defined in the following.

Definition 4.4 (Equalities and Inequalities with Respect to Symbolic States)**.** Let $(\Sigma, \Pi) \in \mathfrak{S}_\sigma$ be a symbolic state and $e_1, e_2 \in Vars \cup \{nil\}$. Then, the entailment of equality and inequality of e_1 and e_2 under (Σ, Π), written as $(\Sigma, \Pi) \vdash e_1 = e_2$ and $(\Sigma, \Pi) \vdash e_1 \neq e_2$ respectively, are defined as follows.

$$(\Sigma, \Pi) \vdash e_1 = e_2 \quad \text{if} \quad e_1 =_\Pi e_2$$
$$(\Sigma, \Pi) \vdash e_1 \neq e_2 \quad \text{if} \quad (\Sigma, \Pi[e_1 :=: e_2]) \vdash false$$

where, $\Pi[\varsigma_1 :=: \varsigma_2]$, for two symbols ς_1 and ς_2, is obtained by uniting the equivalence classes of ς_1 and ς_2 if they exist or otherwise adding a new equivalence class $\{\varsigma_1, \varsigma_2\}$. ■

In order to evaluate statements that access pointers x (e.g., **free**(x);), we need to explicitly have a points-to chunk, $x \mapsto \varsigma$ for some symbol ς. Although, this is not always the case when x represents an allocated heap cell; e.g., we have an $Ls(x, \varsigma)$ chunk instead. Therefore, we define a rearrangement function which, given a symbolic state, $st \in \mathfrak{S}_\sigma$, and a program variable x, gives a set of symbolic states all of which are of the form $(\Sigma' * x \mapsto \varsigma, \Pi')$ for some ς if x represents an allocated heap cell in st or gives $\{\top_\sigma\}$, otherwise. The rearrangement function is formally defined as follows.

Definition 4.5 (Rearrangement)**.** Let $st \in \mathfrak{S}_\sigma$ be a symbolic heap and $x \in Vars$ be a program variable. Then, the rearrangement of st to reveal x (written as $Rearr(st, x)$) is defined as follows:

$$Rearr(st, x) = \begin{cases} \{(\Sigma * x \mapsto \varsigma_2, \Pi)\} & \text{if } st = (\Sigma * \varsigma_1 \mapsto \varsigma_2, \Pi) \text{ for some } \varsigma_1, \varsigma_2 \in \mathsf{Sym} \\ & \quad \text{and } \varsigma_1 =_\Pi x \\ \{(\Sigma * x \mapsto \varsigma_2, \Pi), & \text{if } st = (\Sigma * Ls(\varsigma_1, \varsigma_2), \Pi) \text{ for } \varsigma_1, \varsigma_2 \in \mathsf{Sym}, \\ \quad (\Sigma * x \mapsto x' * Ls(x', \varsigma_2), \Pi)\} & \quad \varsigma_1 =_\Pi x \text{ and } x' \text{ is a fresh primed variable} \\ \{\top_\sigma\} & \text{otherwise} \end{cases}$$

■

Next, we define the preliminary symbolic semantics for basic statements and define the filter function for symbolic semantics.

Definition 4.6 (Preliminary Symbolic Semantics for Basic Statements)**.** The preliminary symbolic semantics for basic statements is defined as follows:

$$\frac{\text{any basic statement } c}{\top_\sigma, c \Rightarrow^\dagger_\sigma \top_\sigma} \qquad \frac{x', y' \text{ fresh primed variables}}{(\Sigma, \Pi), \textbf{new}(x) \Rightarrow^\dagger_\sigma (\Sigma[x'/x] * x \mapsto y', \Pi[x'/x])}$$

$$\frac{(\Sigma' * x \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), \textbf{free}(x) \Rightarrow^\dagger_\sigma (\Sigma', \Pi')} \qquad \frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), \textbf{free}(x) \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{x' \text{ fresh primed variable}}{(\Sigma, \Pi), x := e \Rightarrow^\dagger_\sigma (\Sigma[x'/x], (\Pi[x'/x])[x :=: e[x'/x]])}$$

$$\frac{x' \text{ fresh primed variable}, (\Sigma' * y \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), y)}{(\Sigma, \Pi), x := y.\textbf{next} \Rightarrow^\dagger_\sigma ((\Sigma' * y \mapsto \varsigma)[x'/x], (\Pi'[x'/x])[x :=: \varsigma[x'/x]])}$$

$$\frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), y)}{(\Sigma, \Pi), x := y.\textbf{next} \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{(\Sigma' * x \mapsto \varsigma, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), x.\textbf{next} := e \Rightarrow^\dagger_\sigma (\Sigma' * x \mapsto e, \Pi')} \qquad \frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x)}{(\Sigma, \Pi), x.\textbf{next} := e \Rightarrow^\dagger_\sigma \top_\sigma}$$

$$\frac{(\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi') \in \mathit{Rearr}((\Sigma, \Pi), x, y)}{(\Sigma, \Pi), x.\textbf{next} := y.\textbf{next} \Rightarrow^\dagger_\sigma (\Sigma' * x \mapsto \varsigma_2 * y \mapsto \varsigma_2, \Pi')}$$

$$\frac{\top_\sigma \in \mathit{Rearr}((\Sigma, \Pi), x, y)}{(\Sigma, \Pi), x.\textbf{next} := y.\textbf{next} \Rightarrow^\dagger_\sigma \top_\sigma}$$

where $A[\varsigma_1/\varsigma_2]$ is replacement of ς_2 with ς_1 in A and $\mathit{Rearr}(st, x, y)$ is short for

$$\bigcup_{st' \in \mathit{Rearr}(st, x)} \mathit{Rearr}(st', y)$$

■

Definition 4.7 (Symbolic Filter Function)**.** The symbolic filter function is defined as follows:

$$\mathit{filter}_\sigma(e_1 == e_2)(S) = \{(\Sigma, \Pi[e_1 :=: e_2]) \mid (\Sigma, \Pi) \in S \wedge (\Sigma, \Pi) \not\models e_1 \neq e_2\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

$$\mathit{filter}_\sigma(e_1 \mathrel{!}= e_2)(S) = \{(\Sigma, \Pi) \mid (\Sigma, \Pi) \in S \wedge (\Sigma, \Pi) \not\models \mathit{false} \wedge (\Sigma, \Pi) \not\models e_1 = e_2\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

■

Figure 6 shows the symbolic computation of the least fixpoint of the semantics of the while loop of the dispose program, depicted in Figure 2, starting in a set of states consisting of a single symbolic state where x points to a linked list ending in nil. In this case, there does not exist an $n \in \mathbb{N}$ such that $f^n_{S_0}(\varnothing)$ is the least fixpoint. This is due to the introduction of new primed variables which causes the pure part to grow indefinitely.

On the other hand, Figure 7 shows the symbolic computation of the least fixpoint of the semantics of the while loop of the infinite allocation program, depicted in Figure 3, starting in a set of states consisting of a single symbolic state where the heap is empty and there are no pure assertions. Evidently, in the case of symbolic semantics, similarly to the case of concrete semantics, the least fixpoint of the while loop in this program is not obtainable by iterative computation of $f^n_{S_0}(\varnothing)$.

$$S_0 = \{(Ls(l, nil), \varnothing)\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(Ls(x, nil), \varnothing)\}$
2	$\{(Ls(x, nil), \varnothing), (emp, \{\{y, nil, x\}\}), (Ls(x'_1, nil), \{\{y, x'_1, x\}\})\}$
3	$\{(Ls(l, nil), \varnothing), (emp, \{\{n, nil, l\}\}), (Ls(x'_1, nil), \{\{x, x'_1, y\}\}),$ $(emp, \{\{x'_3, x'_2, x'_5\}, \{y, nil, x\}\}), (Ls(x'_4, nil), \{\{x, x'_4, x\}, \{x'_3, x'_2, x'_5\}\})\}$
$\vdots$	$\vdots$

Figure 6: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the symbolic semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

$$S_0 = \{(emp, \varnothing)\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(emp, \varnothing)\}$
2	$\{(emp * x \mapsto x'_1, \varnothing)\}$
3	$\{(emp * x \mapsto x'_3 * x'_2 \mapsto x'_1, \varnothing)\}$
4	$\{(emp * x \mapsto x'_5 * x'_4 \mapsto x'_3 * x'_2 \mapsto x'_1, \varnothing)\}$
$\vdots$	$\vdots$

Figure 7: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the symbolic semantics. In this case, there *does not* exist an $n \in \mathbb{N}$ such that $f^n_{S_0}$ is the least fixpoint.

5 Overapproximation of Concrete Semantics by Symbolic Semantics

In this section, we show that the symbolic semantics is a sound overapproximation of the concrete semantics. Before we can establish this result though, we must define what constitutes a symbolic state representing a concrete state.

We say a concrete state (h, s) models a symbolic state (Σ, Π), if there is a $s' : \mathit{Vars}' \to \mathit{Values}$ such that the values assigned by s and s' are compatible with the equalities of Π and s and s' can be used to translate the symbols in Σ such that Σ holds in h.

Definition 5.1 (Modeling Relation)**.** Let (h, s) and (Σ, Π) be a concrete and symbolic state respectively. Then, we say (h, s) models (Σ, Π) (written as $(h, s) \vDash (\Sigma, \Pi)$) if there exists a $s' : \mathit{Vars}' \to \mathit{Values}$ such that:

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_\Pi \varsigma_2 \to \mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \varsigma_2)$$

and

$$(h, s) \overset{s'}{\vDash} \Sigma$$

where $\mathit{val}_\sigma(s, s', \varsigma)$ is defined as follows:

$$\mathit{val}_\sigma(s, s', \varsigma) = \begin{cases} s(\varsigma) & \text{if } \varsigma \in \mathit{Vars} \\ s'(\varsigma) & \text{if } \varsigma \in \mathit{Vars}' \\ \mathit{nil} & \text{if } \varsigma = \mathit{nil} \end{cases}$$

Furthermore, $(h, s) \overset{s'}{\vDash} \Sigma$ is defined inductively as follows:

$(h, s) \overset{s'}{\vDash} \mathit{emp}$ if and only if $\mathit{dom}(h) = \varnothing$

$(h, s) \overset{s'}{\vDash} \mathit{Junk}$ if and only if $\mathit{dom}(h) \neq \varnothing$

$(h, s) \overset{s'}{\vDash} \varsigma_1 \mapsto \varsigma_2$ if and only if $\varsigma_1 \neq \mathit{nil} \wedge h(a) = \begin{cases} \mathit{val}_\sigma(s, s', \varsigma_2) & \text{if } a = \mathit{val}_\sigma(s, s', \varsigma_1) \\ \bot & \text{otherwise} \end{cases}$

$(h, s) \overset{s'}{\vDash} \mathit{Ls}(\varsigma_1, \varsigma_2)$ if and only if $\varsigma_1 \neq \mathit{nil} \wedge \mathit{val}_\sigma(s, s', \varsigma_1) \neq \mathit{val}_\sigma(s, s', \varsigma_2) \wedge$
$((h, s) \overset{s'}{\vDash} \varsigma_1 \mapsto \varsigma_2 \vee (h, s) \overset{s''}{\vDash} (\varsigma_1 \mapsto z' * \mathit{Ls}(z', \varsigma_2)))$
for some fresh z' and $s'' = s'[z' \mapsto h(\mathit{val}_\sigma(s, s', \varsigma_1))]$

$(h, s) \overset{s'}{\vDash} F * G$ if and only if $h = h_1 \uplus h_2$ where $(h_1, s) \overset{s'}{\vDash} F \wedge (h_2, s) \overset{s'}{\vDash} G$

where, $h_1 \uplus h_2$ is disjoint union of h_1 and h_2 and is undefined if $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) \neq \varnothing$. ■

Given the modeling relation, we define the the set of concrete states represented by a set of symbolic states as follows.

Definition 5.2 (Representation Function)**.** The representation function $\gamma : 2^{\mathfrak{S}_\sigma} \to 2^{\mathfrak{S}_\kappa}$ is the function that maps a set of symbolic states to the set of concrete states represented by them.

$$\gamma(S) = \begin{cases} \{(h, s) \in \mathfrak{S}_\kappa \mid (\Sigma, \Pi) \in S \wedge (h, s) \vDash (\Sigma, \Pi)\} & \text{if } \top_\sigma \notin S \\ \mathfrak{S}_\kappa & \text{otherwise} \end{cases}$$

■

In the following, we show that a symbolic state (Σ, Π) is inconsistent, i.e., $(\Sigma, \Pi) \vdash \mathit{false}$ if and only if there are no concrete states modeling it. Furthermore, we show that whenever two symbolic states are equal, a concrete state either models both of them or none of them. This means that $(\Sigma_1, \Pi_1) = (\Sigma_2, \Pi_2)$ implies, $\gamma(\{(\Sigma_1, \Pi_1)\}) = \gamma(\{(\Sigma_2, \Pi_2)\})$.

Theorem 5.3. Let (Σ, Π) be a symbolic state, then $(\Sigma, \Pi) \vdash \mathit{false}$ if and only if there is no concrete state (h, s) such that $(h, s) \models (\Sigma, \Pi)$. ■

Proof. We prove each side separately as follows:

$\Rightarrow$ $(\Sigma, \Pi) \vdash \mathit{false}$, then $\forall (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \not\models (\Sigma, \Pi)$:
$(\Sigma, \Pi) \vdash \mathit{false}$ if at least one of the conditions I, II or III in 4.3 holds. We show that for each of these cases, (Σ, Π) has no model.

- I: There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ or $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ and we have $\varsigma_1 =_\Pi \mathit{nil}$. If $(h, s) \models (\Sigma, \Pi)$, there must exist an $s' : \mathit{Vars}' \to \mathit{Values}$ such that $\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \mathit{nil}) = \mathit{nil}$ and we should have $\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{nil} \in \mathit{dom}(h)$ which is impossible as $h : \mathbb{N} \rightharpoonup \mathit{Values}$.
- II: There are two chunks $P_1(\varsigma_1, \varsigma_2)$ and $P_2(\varsigma_3, \varsigma_4)$ such that $\varsigma_1 =_\Pi \varsigma_3$. This means, there must be s' and a subheap h' such that $\mathit{dom}(h') \subseteq \mathit{dom}(h)$ and $(h', s) \overset{s'}{\models} P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4)$. Thus, we should have $h' = h'_1 \uplus h'_2$ such that $(h'_1, s) \overset{s'}{\models} P_1(\varsigma_1, \varsigma_2)$ and $(h'_2, s) \overset{s'}{\models} P_2(\varsigma_3, \varsigma_4)$. This can not be as we have
$$\mathit{val}_\sigma(s, s', \varsigma_1) = \mathit{val}_\sigma(s, s', \varsigma_3) \in \mathit{dom}(h'_1) \cap \mathit{dom}(h'_2)$$
which contradicts the fact that $h' = h'_1 \uplus h'_2$, as it is undefined.
- III: There is a chunk $Ls(\varsigma_1, \varsigma_2)$ such that $\varsigma_1 =_\Pi \varsigma_2$. Then, there must be an s' and a subheap h' such that $\mathit{dom}(h') \subseteq \mathit{dom}(h)$ and $(h', s) \models Ls(\varsigma_1, \varsigma_2)$ which in turn means, we should have $\mathit{val}_\sigma(s, s', \varsigma_1) \neq \mathit{val}_\sigma(s, s', \varsigma_2)$ which is a contradiction (see Definition 5.1).

$\Leftarrow$: $\forall (h, s) \in \mathfrak{S}_\sigma.\ (h, s) \not\models (\Sigma, \Pi)$, then $(\Sigma, \Pi) \vdash \mathit{false}$:
Instead of a direct proof, we show that the contrapositive holds. Namely, we show $(\Sigma, \Pi) \not\vdash \mathit{false} \to \exists (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \models (\Sigma, \Pi)$:

As for any heap h, it holds that $h = h \uplus h_\perp$ where $\mathit{dom}(h_\perp) = \varnothing$, whenever $(h, s) \models (\Sigma, \Pi)$, also $(h, s) \models (\Sigma * \mathit{emp}, \Pi)$. Therefore, in the rest of the proof we assume that there are no emp chunks in the spatial part.

Let $\Pi = \{\pi_0, \ldots, \pi_n\}$ then, we define $s : \mathit{Vars} \to \mathit{Values}$ and $s' : \mathit{Vars}' \to \mathit{Values}$ as follows:

$$s(x) = \begin{cases} i & \text{if } x \in \pi_i \wedge \mathit{nil} \notin \pi_i \\ \mathit{nil} & \text{if } \{x, \mathit{nil}\} \subseteq \pi_i \\ \mathcal{U}(x) & \text{otherwise} \end{cases} \qquad s'(x') = \begin{cases} i & \text{if } x' \in \pi_i \wedge \mathit{nil} \notin \pi_i \\ \mathit{nil} & \text{if } \{x', \mathit{nil}\} \subseteq \pi_i \\ \mathcal{U}(x') & \text{otherwise} \end{cases}$$

where $\mathcal{U} : \mathit{Vars} \cup \mathit{Vars}' \to \{n+m+1, n+m+2, \ldots\}$ is an injective mapping that maps each program variable or primed variable to some unique natural number greater than $n + m$, where m is the number of Junk chunks in Σ. Furthermore, let $h : \mathbb{N} \rightharpoonup \mathit{Values}$ be defined as follows:

$$h(a) = \begin{cases} \mathit{val}_\sigma(s, s', \varsigma_2) & \text{if } a = \mathit{val}_\sigma(s, s', \varsigma_1) \wedge \Sigma = P(\varsigma_1, \varsigma_2) * \Sigma' \\ n + i & \text{if } a = n + i \wedge 1 \leq i < m \\ \perp & \text{otherwise} \end{cases}$$

It is easy to see that $(h,s) \overset{s'}{\vDash} (\Sigma,\Pi)$, thus, we omit the proof here. □

Theorem 5.4. Let (Σ_1,Π_1) and (Σ_2,Π_2) be two symbolic states such that $(\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2)$. Then a concrete state either models both or neither of them. In other words,

$$\text{if } (\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2) \text{ then } \left(\forall (h,s) \in \mathfrak{S}_\kappa.\ (h,s) \vDash (\Sigma_1,\Pi_1) \text{ if and only if } (h,s) \vDash (\Sigma_2,\Pi_2)\right)$$

■

Proof. To prove this theorem, we assume $(\Sigma_1,\Pi_1) = (\Sigma_2,\Pi_2)$ and show that for an arbitrary concrete state $(h,s) \in \mathfrak{S}_\kappa$, $(h,s) \vDash (\Sigma_1,\Pi_1)$ then $(h,s) \vDash (\Sigma_2,\Pi_2)$. The other side (i.e., $(h,s) \vDash (\Sigma_2,\Pi_2)$ then $(h,s) \vDash (\Sigma_1,\Pi_1)$) follows through a similar reasoning and is therefore omitted.

Let $r : \mathit{Vars}' \to \mathit{Vars}'$ be the equalizer renaming of (Σ_1,Π_1) and (Σ_2,Π_2) and $s' : \mathit{Vars}' \to \mathit{Values}$ be such that $(h,s) \overset{s'}{\vDash} (\Sigma_1,\Pi_1)$. Then, $(h,s) \overset{s' \circ r}{\vDash} (\Sigma_2,\Pi_2)$.

This simply follows from the fact that

$$\forall x' \in \mathit{Vars}'.\ \mathit{val}_\sigma(s,s',x') = \mathit{val}_\sigma(s,s' \circ r, r(x'))$$

Which can be easily seen from the definition of $\mathit{val}_\sigma(s,s',\varsigma)$, for a symbol ς, in Definition 5.1. □

In preparation for the main result here, i.e., symbolic semantics being a sound overapproximation of concrete semantics, we show that the set of concrete states represented by a set of symbolic states is preserved under replacement of a symbol with a fresh primed variable. Furthermore, we show that the rearrangement to reveal a program variable is an overapproximation, i.e., if $(h,s) \vDash (\Sigma,\Pi)$ then $(h,s) \in \gamma(\mathit{Rearr}((\Sigma,\Pi),x))$ for any $x \in \mathit{Vars}$.

Lemma 5.5. Let (Σ,Π) be a symbolic state, $\varsigma \in \mathsf{Sym}$ be a symbol and $x' \in \mathit{Vars}'$ be a primed variable that does not appear in (Σ,Π) . Then,

$$\gamma(\{(\Sigma,\Pi)\}) \subseteq \gamma(\{(\Sigma,\Pi)[x'/\varsigma]\})$$

■

Proof. Let $s'_1 : \mathit{Vars}' \to \mathit{Values}$ and $s'_2 : \mathit{Vars}' \to \mathit{Values}$ be two valuation functions for primed variables such that:

$$s'_2(y') = \begin{cases} \mathit{val}_\sigma(s,s'_1,\varsigma) & \text{if } y' = x' \\ s'_1(y') & \text{otherwise} \end{cases}$$

We show that for any $(h,s) \in \mathfrak{S}_\kappa$,

$$(h,s) \overset{s'_1}{\vDash} (\Sigma,\Pi) \to (h,s) \overset{s'_2}{\vDash} (\Sigma,\Pi)[x'/\varsigma]$$

To show this, we only need to show that for any symbol $\varsigma_1 \in \mathsf{Sym}$,

$$\mathit{val}_\sigma(s,s'_1,\varsigma_1) = \mathit{val}_\sigma(s,s'_2,\varsigma_1[x'/\varsigma])$$

This, on the other hand, is obvious from the definition of s'_2 above. □

Lemma 5.6. Let $st \in \mathfrak{S}_\sigma$ be a symbolic state and $x \in \mathit{Vars}$ be a program variable. Then,

$$\gamma(\{st\}) \subseteq \gamma(\mathit{Rearr}(st,x))$$

■

Proof. If $st = \top_\sigma$, $\gamma(Rearr(\top_\sigma, x)) = \gamma(\{\top_\sigma\}) = \mathfrak{S}_\kappa$. If $st = (\Sigma, \Pi)$, we can have three cases:

- There is a Σ' such that $\Sigma = \varsigma_1 \mapsto \varsigma_2 * \Sigma'$ for some ς_2 and $\varsigma_1 =_\Pi x$. In this case, $Rearr((\Sigma, \Pi), x) = \{(x \mapsto \varsigma_2 * \Sigma', \Pi')\}$. In addition, if $(h, s) \vDash (\Sigma, \Pi)$, there is an s', h_1, h_2 such that $h = h_1 \uplus h_2$ and $(h_1, s) \overset{s'}{\vDash} (\varsigma_1 \mapsto \varsigma_2, \Pi)$ and $(h_2, s) \overset{s'}{\vDash} (\Sigma', \Pi)$. On the other hand, $\varsigma_1 =_\Pi x$ and hence $val_\sigma(s, s', x) = val_\sigma(s, s', \varsigma_1)$. Thus, $(h_1, s) \overset{s'}{\vDash} (x \mapsto \varsigma_2, \Pi)$ and as a result
$$(h, s) \vDash (x \mapsto \varsigma_2 * \Sigma', \Pi')$$
- There is a Σ' such that $\Sigma = Ls(\varsigma_1, \varsigma_2) * \Sigma'$ for some ς_2 and $\varsigma_1 =_\Pi x$. In this case,
$$Rearr((\Sigma, \Pi), x) = \{\underbrace{(x \mapsto \varsigma_2 * \Sigma', \Pi)}_{st_1}, \underbrace{(x \mapsto x' * Ls(x', \varsigma_2) * \Sigma', \Pi)}_{st_2}\}$$
According to Definition 5.1 and a reasoning similar to that of the previous case, if $(h, s) \vDash (\Sigma, \Pi)$, then, either $(h, s) \vDash st_1$ or $(h, s) \vDash st_1$.
- None of the above hold. In this case, $\gamma(Rearr((\Sigma, \Pi), x)) = \gamma(\{\top_\sigma\}) = \mathfrak{S}_\kappa$.

□

Lemma 5.7 (Symbolic Semantics of Basic Statements Overapproximates Concrete Semantics)**.** Let c be a basic statement. Then,
$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\sigma(S))$$

■

Proof. Since semantics functions and representation function, γ, are both continuous, we simply need to show that
$$\forall st \in \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(\{st\})) \subseteq \gamma([\![c]\!]_\sigma(\{st\}))$$
If $st = \top_\sigma$,
$$\gamma([\![c]\!]_\sigma(\{\top_\sigma\})) = \gamma(\top_\sigma) = \mathfrak{S}_\kappa$$
If $st = (\Sigma, \Pi)$,

- $c = \textbf{new}(x)$;

  For $st \in [\![c]\!]_\sigma(\{(\Sigma, \Pi)\})$, we have $st = (x \mapsto y' * \Sigma[x'/x], \Pi[x'/x])$ where x' and y' are some fresh primed variables.

  On the other hand, if $(h, s) \vDash (\Sigma, \Pi)$, i.e., $(h, s) \in \gamma(\{(\Sigma, \Pi)\})$, then, for any $st' \in [\![c]\!]_\kappa(h, s)$, we have,
  $$st' = (h[m \mapsto n], s[x \mapsto m])$$
  for some $m \notin dom(h)$ and $n \in \mathbb{N}$. Furthermore, $h[m \mapsto n] = h \uplus h'$ where,
  $$h'(x) = \begin{cases} n & \text{if } x = m \\ \bot & \text{otherwise} \end{cases}$$
  According to Lemma 5.5, $(h, s) \vDash (\Sigma[x'/x], \Pi[x'/x])$. On the other hand, since x does not appear in $(\Sigma[x'/x], \Pi[x'/x])$, we have
  $$(h, s[x \mapsto m]) \vDash (\Sigma[x'/x], \Pi[x'/x])$$

In addition,

$$(h', s[x \mapsto m]) \vDash (x \mapsto y', \Pi[x'/x])$$

Thus,

$$(h[m \mapsto n], s[x \mapsto m]) \vDash (x \mapsto y' * \Sigma[x'/x], \Pi[x'/x])$$

Consequently,

$$(h[m \mapsto n], s[x \mapsto m]) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = \textbf{free}(x)$;

If $Rearr((\Sigma, \Pi), x) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for $(h, s) \vDash (\Sigma, \Pi)$, According to Lemma 5.6, $(h, s) \in \gamma(Rearr((\Sigma, \Pi), x))$. Hence, there is a $(\Sigma' * x \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), x)$ such that

$$(h, s) \vDash (x \mapsto \varsigma * \Sigma', \Pi)$$

Therefore, $h = h_1 \uplus h_2$ such that $(h_1, s) \vDash (x \mapsto \varsigma, \Pi)$ and $(h_2, s) \vDash (\Sigma' * \Pi)$. Thus, for $(h[s(x) \mapsto \bot], s) \in [\![c]\!]_\kappa(h, s)$,

$$(h[s(x) \mapsto \bot], s) = (h_2, s) \in \gamma(\Sigma' * \Pi)$$

Consequently,

$$(h[s(x) \mapsto \bot], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x := e$;

Let x' be a fresh primed variable and (h, s) be a concrete state such that $(h, s) \vDash (\Sigma, \Pi)$. According to Lemma 5.5, we have $(h, s) \vDash (\Sigma[x'/x], \Pi[x'/x])$. Here, we consider two cases, first if x and e are the same (an assignment of x to itself!) and second if x and e are two different simple expressions.

If x and e are the same variables, then we have $s[x \mapsto val_\kappa(s, e)]) = s$ and

$$(\Sigma[x'/x], (\Pi[x'/x])[x :=: (y[x'/x])]) = (\Sigma[x'/x], \Pi[x'/x])$$

If x and e are two different symbols, we have $(e[x'/x]) = e$ and thus, we have to show

$$(h, (s[x \mapsto val_\kappa(s, e)])) \vDash (\Sigma[x'/x], (\Pi[x'/x])[x :=: e])$$

To see this, observe that

$$(h, (s[x \mapsto val_\kappa(s, e)])) \vDash (\Sigma[x'/x], \Pi[x'/x])$$

Notice that x does not appear in $(\Sigma[x'/x], \Pi[x'/x])$ and hence the value of $s(x)$ is irrelevant. Thus, there is an s' such that

$$(h, (s[x \mapsto val_\kappa(s, e)])) \overset{s'}{\vDash} (\Sigma[x'/x], \Pi[x'/x])$$

On the other hand, since

$$\forall \varsigma_1, \varsigma_1 \in \mathsf{Sym}.\ \varsigma_1 =_\Pi \varsigma_2 \to val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

we have

$$(h, (s[x \mapsto val_\kappa(s, e)])) \overset{s'}{\vDash} (\Sigma[x'/x], (\Pi[x'/x])[x :=: e])$$

Consequently,

$$(h, (s[x \mapsto val_\kappa(s, e)])) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x.\textbf{next} := e$;

If $Rearr((\Sigma, \Pi), x) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, there is a $(\Sigma' * x \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), x)$, such that

$$(h, s) \vDash (\Sigma' * x \mapsto \varsigma, \Pi)$$

On the other hand, since $(h, s) \vDash (\Sigma' * x \mapsto \varsigma, \Pi)$, we have $h = h_1 \uplus h_2$, such that $(h_1, s) \vDash (\Sigma', \Pi)$ and $(h_2, s) \vDash (x \mapsto \varsigma, \Pi)$. Therefore, since $(h_2[x \mapsto val_\kappa(e)], s) \vDash (x \mapsto e, \Pi)$, we have

$$(h[x \mapsto val_\kappa(s, e)], s) \vDash (\Sigma' * x \mapsto e, \Pi) \in [\![c]\!]_\sigma(\{(\Sigma, \Pi)\})$$

Consequently,

$$(h[x \mapsto val_\kappa(s, e)], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x := y.\textbf{next}$;

If $Rearr((\Sigma, \Pi), y) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, we have

$$(h, s) \in \gamma(Rearr((\Sigma, \Pi), y))$$

Hence, there is a $(\Sigma' * y \mapsto \varsigma, \Pi) \in Rearr((\Sigma, \Pi), y)$ such that

$$(h, s) \vDash (\Sigma' * y \mapsto \varsigma, \Pi)$$

Now, let x' be a fresh primed variable. Then, according to Lemma 5.5, $(h, s) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$.

Here, we should consider two cases, first if ς and x are the same (y's next cell is x) and second if ς and x are two different symbols.

If x and ς are the same variables, then, since $h(s(y)) = s(x)$, we have

$$s[x \mapsto h(s(y))] = s$$

and

$$((\Sigma'' * y \mapsto \varsigma)[x'/x], ((\Pi[x'/x])[x :=: (\varsigma[x'/x])])) = ((\Sigma'' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$$

Thus,

$$(h, s[x \mapsto h(s(y))]) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

On the other hand, if x and ς are two different symbols, since $(h, s) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$, we have

$$(h, s[x \mapsto h(s(y))]) \vDash ((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$$

Notice that x does not appear in $((\Sigma' * y \mapsto \varsigma)[x'/x], \Pi[x'/x])$ and thus the value of $s(x)$ is irrelevant. Thus, there is a s' such that

$$(h, (s[x \mapsto h(s(y))])) \overset{s'}{\vDash} (\Sigma[x'/x], \Pi[x'/x])$$

On the other hand, since

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_\Pi \varsigma_2 \rightarrow val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

we have,

$$\forall \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \varsigma_1 =_{(\Pi[x'/x])[x:=:\varsigma]} \varsigma_2 \rightarrow val_\sigma((s[x \mapsto s(y)]), s', \varsigma_1) = val_\sigma((s[x \mapsto s(y)]), s', \varsigma_2)$$

and as a result,

$$(h, (s[x \mapsto s(y)])) \overset{s'}{\vDash} (\Sigma[x'/x], (\Pi[x'/x])[x :=: \varsigma])$$

Consequently,

$$(h, (s[x \mapsto s(y)])) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

– $c = x.\textbf{next} := y.\textbf{next}$;

If $Rearr((\Sigma, \Pi), x, y) = \{\top_\sigma\}$, it trivially holds.

Otherwise, for any $(h, s) \vDash (\Sigma, \Pi)$, according to Lemma 5.6, there is

$$(\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi') \in Rearr((\Sigma, \Pi), x, y)$$

such that $(h, s) \vDash (\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi')$.

On the other hand, since $(h, s) \vDash (\Sigma' * x \mapsto \varsigma_1 * y \mapsto \varsigma_2, \Pi)$, we have $h = h_1 \uplus h_2 \uplus h_3$, such that

$$(h_1, s) \vDash (\Sigma', \Pi),\ \ (h_2, s) \vDash (x \mapsto \varsigma_1, \Pi)\ \text{ and }\ (h_3, s) \vDash (y \mapsto \varsigma_2, \Pi)$$

Therefore, we have

$$(h[x \mapsto h(s(y))], s) \vDash (\Sigma' * x \mapsto \varsigma_2 * y \mapsto \varsigma_2, \Pi)$$

Consequently,

$$(h[x \mapsto h(s(y)))], s) \in \gamma([\![c]\!]_\sigma(\{(\Sigma, \Pi)\}))$$

□

After proving overapproximation results for basic statements, we show overapproximation of the filter function before proving the final result, i.e., overapproximation for the whole of *SimpleLang*.

Lemma 5.8. Let $S \subseteq \mathfrak{S}_\sigma$ be a set of symbolic states and $cond$ be a loop condition of the form $e1$ == e_1 or e_1 != e_2 for some simple expressions e_1 and e_2. Then,

$$\mathit{filter}_\kappa(cond)(\gamma(S)) \subseteq \gamma(\mathit{filter}_\sigma(cond)(S))$$

■

Proof. Since the representation function, γ, and both filter functions are continuous, the results for singletons naturally extend to all sets. In other words, we only need to show that,

$$\forall st \in \mathfrak{S}_\sigma.\ \mathit{filter}_\kappa(cond)(\gamma(\{st\})) \subseteq \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

If $st = \top_\sigma$, $\gamma(\mathit{filter}_\sigma(cond)(\{st\})) = \mathfrak{S}_\kappa$. On the other hand, if $st = (\Sigma, \Pi)$ and $(\Sigma, \Pi) \vdash \mathit{false}$, according to Theorem 5.3 and Definition 5.2, $\gamma(\{(\Sigma, \Pi)\}) = \varnothing$ and thus,

$$\mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma, \Pi)\})) = \varnothing$$

Therefore, in the rest of the proof we consider $st = (\Sigma, \Pi)$ such that $(\Sigma, \Pi) \nvdash \mathit{false}$. We proceed by case analysis on $cond$.

Case: $cond$ = 'e_1 == e_2'

If $(h,s) \in \mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma,\Pi)\}))$, we have $(h,s) \vDash (\Sigma,\Pi)$ and $val_\kappa(s,e_1) = val_\kappa(s,e_2)$. As a result, $(h,s) \vDash (\Sigma,\Pi[e_1 :=: e_2])$. This, according to Theorem 5.3 and Definition 4.4, shows that $(\Sigma,\Pi) \nvdash e_1 \neq e_2$. Hence,

$$(\Sigma,\Pi[e_1 :=: e_2]) \in \mathit{filter}_\sigma(cond)(\Sigma,\Pi)$$

Consequently,

$$(h,s) \in \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

Case: $cond$ = 'e_1 != e_2'

If $(h,s) \in \mathit{filter}_\kappa(cond)(\gamma(\{(\Sigma,\Pi)\}))$, we have $(h,s) \vDash (\Sigma,\Pi)$ and $val_\kappa(s,e_1) \neq val_\kappa(s,e_2)$. According to Definition 4.4, we have $(\Sigma,\Pi) \nvdash e_1 = e_2$. Hence,

$$(\Sigma,\Pi) \in \mathit{filter}_\sigma(cond)(\Sigma,\Pi)$$

Consequently,

$$(h,s) \in \gamma(\mathit{filter}_\sigma(cond)(\{st\}))$$

□

Finally, we can state and prove the main theorem of this section, i.e., the correspondence of semantics.

Theorem 5.9 (Symbolic Semantics Overapproximates Concrete Semantics)**.** Let c be any statement or block of statements. Then,

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\sigma(S))$$

■

Proof. We prove this theorem by structural induction on the structure of statements:

Base Case: The statement c is a basic statement. This is already proven in Lemma 5.7.

Inductive Case1: The statement c is of the form $c_1; c_2$ and, as induction hypothesis, we have:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_1]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_1]\!]_\sigma(S)) \text{ and } \forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_2]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_2]\!]_\sigma(S))$$

From the definition of semantics of composition of statements, $[\![c_1;c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$, it follows that:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c_1;c_2]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c_1;c_2]\!]_\sigma(S))$$

Inductive Case2: The statement c is of the form **while**$(b)\{c'\}$ and, as induction hypothesis, we have:

$$\forall S \subseteq \mathfrak{S}_\sigma.\ [\![c']\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c']\!]_\sigma(S))$$

Assuming $S \subseteq \mathfrak{S}_\sigma$ is a set of symbolic states, we show that:

$$[\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![\textbf{while}(b)\{c'\}]\!]_\sigma(S))$$

Assuming that $f_{\kappa,\gamma(S)}$ and $f_{\sigma,S}$ are as defined for general semantics of the while loops, respectively for concrete and symbolic semantics, we have:

$$[\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S)) = \mathit{filter}_\kappa(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)})$$

and

$$[\![\textbf{while}(b)\{c'\}]\!]_\sigma(S) = \mathit{filter}_\sigma(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\sigma,S})$$

By a simple induction on n, using induction hypothesis and Lemma 5.8, we can see that:

$$\forall n \in . \ f^n_{\kappa,\gamma(S)} \subseteq f^n_{\sigma,S}$$

Thus,

$$\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)} \subseteq \bigcup_{n\in\mathbb{N}} f^n_{\sigma,S}$$

On the other hand, according to Lemma 5.8, we have:

$$\mathit{filter}_\sigma(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\sigma,S}) \subseteq \mathit{filter}_\kappa(\neg b)(\bigcup_{n\in\mathbb{N}} f^n_{\kappa,\gamma(S)})$$

Consequently,

$$[\![\textbf{while}(b)\{c'\}]\!]_\sigma(S) \subseteq [\![\textbf{while}(b)\{c'\}]\!]_\kappa(\gamma(S))$$

□

As a corollary, we get that whenever the concrete execution of the program, starting in the empty memory, indicates an error in the program, so does the symbolic execution, starting in the empty memory.

Corollary 5.10. For a statement (program) c,

$$\text{if } \ \top_\kappa \in [\![c]\!]_\kappa(h_\varepsilon, s_\varepsilon) \ \text{ then } \ \top_\sigma \in [\![c]\!]_\sigma(emp, \varnothing)$$

where,

$$h_\varepsilon(a) = \bot \ \text{ and } \ s_\varepsilon(x) = nil$$

■

6 Abstract Semantics

In this section we discuss an abstraction that maps symbolic states to abstract symbolic states. Abstract semantics then is simply symbolic semantics having abstraction applied to their results.

We define the abstraction in two phases; in the first phase, we remove all primed variables from the pure part and in the second phase, we contract the spatial part by collapsing consecutive chains of linked list segments (if possible) and gathering all memory leaks into a single Junk predicate.

Definition 6.1 (Removing Primed Variables from Pure Part)**.** Let (Σ, Π) be a symbolic state and $\pi \in \Pi$ be an equivalence class in the pure part. The removal of primed variables appearing in π, denoted as $rp_\pi(\Sigma, \Pi)$ is defined as follows:

$$rp_\pi(\Sigma, \Pi) = (\Sigma[\underbrace{rep(\pi), \ldots, rep(\pi)}_{n \text{ times}}/x'_1, \ldots, x'_n], (\Pi \setminus \{\pi\}) \cup \xi(\pi))$$

where

$$rep(\pi) = \begin{cases} x \in \pi \setminus \{x'_1, \ldots, x'_n\} & \text{if } \pi \setminus \{x'_1, \ldots, x'_n\} \neq \varnothing \\ x'_i & \text{for some } 1 \leq i \leq n \end{cases}$$

and

$$\xi(\pi) = \begin{cases} \{\pi \setminus \{x'_1, \ldots, x'_n\}\} & \text{if } |\pi \setminus \{x'_1, \ldots, x'_n\}| \geq 1 \\ \varnothing & \text{otherwise} \end{cases}$$

and $\{x'_1, \ldots, x'_n\} \subseteq \pi$ is the set of all primed variables in π and $[b_1, \ldots, b_n / a_1, \ldots, a_n]$ denotes simultaneous replacement of a_1 by $b1$, $\ldots$, a_n by b_n.

In the sequel we will use $rp_\Pi(\Sigma, \Pi)$ defined as

$$rp_\Pi(\Sigma, \Pi) = rp_{\pi_1}(\ldots(rp_{\pi_m}(\Sigma, \Pi))\ldots) \text{ where } \Pi = \{\pi_1, \ldots, \pi_m\}$$

to be the result of removal of all primed variables from the pure part of a symbolic heap. ■

Definition 6.2 (Spatial Abstraction Rules)**.** The following are the rules for the abstraction of the spatial part of a symbolic state.

$$\frac{x' \notin Vars'(\Sigma, \Pi) \qquad \varsigma \in \mathsf{Sym}}{(\Sigma * P(x', \varsigma), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma \cup Junk, \Pi)} \quad (Garbage1)$$

$$\frac{x', y' \notin Vars'(\Sigma, \Pi)}{(\Sigma * P_1(x', y') * P_2(y', x'), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma \cup Junk, \Pi)} \quad (Garbage2)$$

$$\frac{y' \notin Vars'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2\} \qquad \varsigma_1, \varsigma_2 \in \mathsf{Sym} \qquad \varsigma_2 =_\Pi nil}{(\Sigma * P_1(\varsigma_1, y') * P_2(y', \varsigma_2), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma * Ls(\varsigma_1, nil), \Pi)} \quad (Abs1)$$

$$\frac{y' \notin Vars'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\} \quad \varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4 \in \mathsf{Sym} \quad \varsigma_2 =_\Pi \varsigma_3}{(\Sigma * P_1(\varsigma_1, y') * P_2(y', \varsigma_2) * P_3(\varsigma_3, \varsigma_4), \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma * Ls(\varsigma_1, \varsigma_2) * P_3(\varsigma_3, \varsigma_4), \Pi)} \quad (Abs2)$$

Where,

$$\Sigma \cup Junk = \begin{cases} \Sigma & \text{if } \Sigma \text{ has a } Junk \text{ chunk} \\ \Sigma * Junk & \text{otherwise} \end{cases}$$

and $Vars'(\Sigma, \Pi)$ is the set of all primed variables that appear in Σ or Π. ■

The rule $Garbage1$ simply collects all memory chunks that are not pointed to by any program variables directly or indirectly – as x' does not appear in Σ or Π. The rule $Garbage2$ handles the cases where there is a cycle that is not reachable directly or indirectly from program variables (cyclic memory leak).

The rule $Abs1$ and $Abs2$ both merge linked list segments (or direct links) that form a continuous linked list. The fact that we require the end of the contracting linked list to be nil or already allocated in a separate portion of the heap is to make sure that the segments being merged don't form a cyclic linked list. Indeed if the end of the linked list segments is allocated in another separate portion of the memory or is nil, it can't be pointing to the middle or beginning of that linked list segment and it is thus safe to merge them to a single *acyclic* linked list segment.

Before we formally define the abstraction operation, we show that the set of rules for abstraction of the spatial part are strongly normalizing, i.e., there are no infinite chains of reduction.

Lemma 6.3 (Strong Normalization of Spatial Abstraction)**.** Let (Σ_0, Π_0) be a symbolic state, then, there does not exist an infinite sequence $(\Sigma_0, \Pi_0), (\Sigma_1, \Pi_1), \ldots$ such that $\forall i \in \mathbb{N}.\ (\Sigma_i, \Pi_i) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma_{i+1}, \Pi_{i+1})$. ■

Proof. It suffices to note that any of these rules decreases the collective number of Ls and $\mapsto$ chunks in the spatial part of the given symbolic states. □

In the sequel, we use $\overset{\mathcal{A}}{\rightsquigarrow}{}^*$ to denote the reflexive transitive closure of $\overset{\mathcal{A}}{\rightsquigarrow}$ and we use $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightsquigarrow}$ to indicate that (Σ, Π) is a normal form, i.e., none of the rules of spatial abstraction are applicable to (Σ, Π).

Before, defining the abstraction operation, we define the set of abstract states. That is, the set of all consistent symbolic states that have no primed variables in their pure part and none of the abstraction rules are applicable to them, together with the symbolic error state, $\top_\sigma$.

Definition 6.4 (Abstract States)**.** The set of abstract states $\mathfrak{S}_\alpha$ is defined as follows:

$$\mathfrak{S}_\alpha = \{(\Sigma, \Pi) \in \mathfrak{S}_\sigma \mid (\Sigma, \Pi) \not\models \mathit{false} \wedge (rp_\Pi(\Sigma, \Pi)) = (\Sigma, \Pi) \wedge (\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightsquigarrow}\} \cup \{\top_\sigma\}$$

■

The rewriting rules of Definition 6.2 form a rewriting relation, $\overset{\mathcal{A}}{\rightsquigarrow}$. Since this rewrite system is terminating, we can extract a function out of this relation that maps each symbolic state to one of its normal forms. This can simply be done by fixing an order on applications of the rules. In our implementation, we simply go through abstraction rules from top to bottom and for each of them keep reapplying them until they are no longer applicable. This process is repeated until none of the rules are applicable. In the sequel, we assume $\mathfrak{red}_\mathcal{A} : \mathfrak{S}_\sigma \to \mathfrak{S}_\sigma$ is such a function.

Definition 6.5 (Abstraction)**.** The abstraction function, denoted by $\mathfrak{Abs} : 2^{\mathfrak{S}_\sigma} \to 2^{\mathfrak{S}_\alpha}$ is defined as follows:

$$\mathfrak{Abs}(S) = \{(\Sigma', \Pi') \mid (\Sigma, \Pi) \in S \wedge \mathfrak{red}_\mathcal{A}(rp_\Pi(\Sigma, \Pi)) = (\Sigma', \Pi') \wedge (\Sigma', \Pi') \not\models \mathit{false}\} \cup \{\top_\sigma \mid \top_\sigma \in S\}$$

■

Abstract semantics is simply symbolic semantics where abstraction is applied to its result.

Definition 6.6 (Abstract Semantics)**.** Let c be a basic statement, then the abstract semantics of c denoted by $[\![c]\!]_\alpha : 2^{\mathfrak{S}_\alpha} \to 2^{\mathfrak{S}_\alpha}$ is defined as follows:

$$[\![c]\!]_\alpha(S) = \mathfrak{Abs}([\![c]\!]_\sigma S)$$

■

Figure 8 shows the abstract computation of the least fixpoint of the semantics of the while loop of the dispose program, depicted in Figure 2, starting in a set of states consisting of a single abstract state where x points to a linked list ending in nil.

Figure 9 shows the abstract computation of the least fixpoint of the semantics of the while loop for the infinite allocation program, depicted in Figure 3, starting in a set of states consisting of a single abstract state where the heap is empty and there are no pure assertions. Contrary to the case of symbolic semantics and concrete semantics, the least fixpoint of the while loop in this program, in abstract semantics, is obtainable by iterative computation of $f^n_{S_0}(\varnothing)$.

$$S_0 = \{(Ls(x, nil), \varnothing)\} \quad [\![dispose]\!]_\alpha(S_0) = \{(emp, \{y, nil, x\})\}$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(Ls(x, nil), \varnothing)\}$
2	$\{(Ls(x, nil), \varnothing), (emp, \{y, nil, x\}), (Ls(y, nil), \{y, x\})\}$

Figure 8: Computation of the least fixpoint of the while loop semantics for the dispose program (Figure 2) in the abstract semantics. In this case, $f^2_{S_0}$ is the least fixpoint. In addition, $[\![dispose]\!]_\alpha$ is the abstract semantics of the whole dispose program (while loop).

$$S_0 = \{(emp, \varnothing)\} \quad [\![infalloc]\!]_\alpha(S_0) = \varnothing$$

n	$f^n_{S_0}(\varnothing)$
0	$\varnothing$
1	$\{(emp, \varnothing)\}$
2	$\{(emp * x \mapsto x'_1, \varnothing)\}$
3	$\{(emp * x \mapsto x'_3 * Junk, \varnothing)\}$

Figure 9: Computation of the least fixpoint of the while loop semantics for the infinite allocation program (Figure 3) in the abstract semantics. In this case, $f^3_{S_0}$ is the least fixpoint. In addition, $[\![infalloc]\!]_\alpha$ is the abstract semantics of the whole dispose program (while loop).

7 Overapproximation of Concrete Semantics by Abstract Semantics

To show that abstract semantics is a sound overapproximation of concrete semantics, we first show that abstraction is an overapproximation.

Lemma 7.1 (Overapproximation of Removal of Primed Variables)**.** Let (Σ, Π) be a symbolic state. Then,

$$\forall (h, s) \in \mathfrak{S}_\sigma.\ (h, s) \vDash (\Sigma, \Pi) \to (h, s) \vDash pr_\Pi(\Sigma, \Pi)$$

■

Proof. Assume that $pr_\pi(\Sigma, \Pi) = (\Sigma', \Pi')$ for some $\pi \in \Pi$. We show that if $(h, s) \vDash (\Sigma, \Pi)$, then $(h, s) \vDash (\Sigma', \Pi')$. Assume that $\{x'_1, \ldots, x'_n\} \subseteq \pi$ is the set of all primed variables in π and $rep(\pi)$ is as in Definition 6.1. In addition, let $ren : \mathsf{Sym} \to \mathsf{Sym}$ be as follows:

$$ren(\varsigma) = \begin{cases} rep(\pi) & \text{if } \varsigma = x'_i \text{ for some } 1 \le i \le n \\ \varsigma & \text{otherwise} \end{cases}$$

We show that, for any $s' : Vars' \to Values$,

$$(h, s) \overset{s'}{\vDash} (\Sigma, \Pi) \to (h, s) \overset{s'}{\vDash} pr_\pi(\Sigma, \Pi)$$

First, note that for all $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$,

$$\varsigma_1 =_{\Pi'} \varsigma_2 \to \varsigma_1 =_\Pi \varsigma_2 \to val_\sigma(s, s', \varsigma_1) = val_\sigma(s, s', \varsigma_2)$$

Second, for any chunk A of Σ, $A' = A[\underbrace{rep(\pi), \ldots, rep(\pi)}_{n \text{ times}}/x'_1, \ldots, x'_n]$ and subheap h' of h, we have,

$$(h, s) \overset{s'}{\models} (A, \Pi) \rightarrow (h, s) \overset{s'}{\models} (A', \Pi')$$

There are four possibilities for A; $Junk$, emp, $\varsigma_1 \mapsto \varsigma_2$ and $Ls(\varsigma_1, \varsigma_2)$. In the first two cases, $A' = A$. According to Definition 5.1, the the case for Ls, follows form the case for $\mapsto$. Therefore, we consider the case where $A = \varsigma_1 \mapsto \varsigma_2$. Due to Definition 5.1, $(h, s) \overset{s'}{\models} (\varsigma_1 \mapsto \varsigma_2, \Pi)$, then,

$$h(a) = \begin{cases} val_\sigma(s, s', \varsigma_2) & \text{if } a = val_\sigma(s, s', \varsigma_1) \\ \bot & \text{otherwise} \end{cases}$$

On the one hand, $A' = ren(\varsigma_1) \mapsto ren(\varsigma_2)$. Thus, since $\forall \varsigma \in \mathsf{Sym}.\ \varsigma =_\pi ren(\varsigma)$, we have

$$\forall \varsigma \in \mathsf{Sym}.\ val_\sigma(s, s', \varsigma) = val_\sigma(s, s', ren(\varsigma))$$

Therefore, $(h, s) \overset{s'}{\models} (A', \Pi')$. □

Lemma 7.2 (Overapproximation of Spatial Abstraction)**.** Let (Σ, Π) and (Σ', Π') be two symbolic states such that $(\Sigma, \Pi) \overset{\mathcal{A}}{\rightsquigarrow} (\Sigma', \Pi')$, then,

$$\forall (h, s) \in \mathfrak{S}_\kappa.\ (h, s) \models (\Sigma, \Pi) \rightarrow (h, s) \models (\Sigma', \Pi')$$

■

Proof. We consider all the four cases. In what follows, we assume $(h, s) \overset{s'}{\models} (\Sigma, \Pi)$.

*Garbage*1. This can be if $\Sigma = \Sigma_1 * P(x', \varsigma)$ for x' not appearing in Σ_1. In this case, there must be two concrete heaps h' and h'' such that $h = h' \uplus h''$ and $(h', s) \overset{s'}{\models} (\Sigma_1, \Pi)$ and $(h'', s) \overset{s'}{\models} (P(x', \varsigma), \Pi)$ which means $dom(h'') \neq \varnothing$.

Here, there are two cases to consider $\Sigma_1 = \Sigma_2 * Junk$ or Σ_1 has no $Junk$ chunk. let $h' = h'_1 \uplus h'_2$ such that in the first case $(h'_1, s) \models (\Sigma_2, \Pi)$ and $dom(h'_2) \neq \varnothing$ and in the second case, $h'_1 = h'$ and $dom(h'_2) = \varnothing$.

In both cases, $dom(h'_2 \uplus h'') \neq \varnothing$ and thus, we have $(h, s) \models (\Sigma_1 \cup Junk, \Pi)$. Note that since $h = (h'_1 \uplus h'_2) \uplus h''$, $dom(h'_2) \cap dom(h'') = \varnothing$ and thus $h'_2 \uplus h''$ is not undefined.

*Garbage*2. Proof is very similar to the previous case and is thus omitted.

*Abs*1. This can be if $\Sigma = \Sigma_1 * P_1(\varsigma_1, y') * P_2(y', \varsigma_2)$ for $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$ and y' neither appear in Σ_1 nor in Π nor is it equal to ς_1 or ς_2 and we have $\varsigma_2 =_\Pi nil$. We prove that if $(h, s) \models (\Sigma, \Pi)$ then $(h, s) \models (\Sigma_1 * Ls(\varsigma_1, nil), \Pi)$.

Since ς_1 is allocated in (Σ, Π) and $val_\sigma(s, s', \varsigma_2) = nil$, $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$. On the other hand, $h = h_1 \uplus h_2 \uplus h_3$ such that $(h_1, s) \overset{s'}{\models} (\Sigma_1, \Pi)$, $(h_2, s) \overset{s'}{\models} (P_1(\varsigma_1, y'), \Pi)$ and $(h_3, s) \overset{s'}{\models} (P_2(y', \varsigma_2), \Pi)$. Therefore, $(h_2 \uplus h_3, s) \overset{s'}{\models} (Ls(\varsigma_1, \varsigma_2), \Pi)$ and hence $(h, s) \overset{s'}{\models} (\Sigma_1 * Ls(\varsigma_1, nil), \Pi)$.

*Abs*2. This case follows from a reasoning very similar to the previous case. The only thing to note is that the reason for having $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$ is different. Namely, it is due to the fact that $\varsigma_2 =_\Pi \varsigma_3$ and therefore $val_\sigma(s, s', \varsigma_2) = val_\sigma(s, s', \varsigma_3)$ and since ς_1 and ς_3 are allocated in disjoint parts of the heap, we know they can not be corresponding to the same address and thus $val_\sigma(s, s', \varsigma_2) \neq val_\sigma(s, s', \varsigma_1)$.

□

Theorem 7.3 (Abstraction Overapproximates)**.** Let $S \subseteq \mathfrak{S}_\sigma$ be a set of symbolic states, then:

$$\gamma(S) \subseteq \gamma(\mathfrak{Abs}(S))$$

■

Proof. Since $\mathfrak{Abs}$ is a continuous function, we need only to show:

$$\forall st \in \mathfrak{S}.\ \gamma(\{st\}) \subseteq \gamma(\mathfrak{Abs}(\{st\}))$$

If $st = \top_\sigma$, $\gamma(\mathfrak{Abs}(\{st\})) = \mathfrak{S}_\kappa$. Otherwise, if $st = (\Sigma, \Pi)$, let's assume

$$\mathfrak{red}_{\mathcal{A}}(rp_\Pi(\Sigma, \Pi)) = (\Sigma', \Pi')$$

In this case, according to Lemma 7.1 and Lemma 7.2,

$$\gamma(\{(\Sigma, \Pi)\}) \subseteq \gamma(\{(\Sigma', \Pi')\})$$

Therefore, according to Theorem 5.3, $(\Sigma', \Pi') \vdash \mathit{false}$, only if $(\Sigma, \Pi) \vdash \mathit{false}$. Hence,

$$\gamma(\{st\}) \subseteq \gamma(\mathfrak{Abs}(\{st\}))$$

□

Finally, the main result of this section, the overapproximation of concrete semantics by symbolic semantics, is given below.

Theorem 7.4 (Abstract Semantics Overapproximates Concrete Semantics)**.** Let c be any statement or block of statements. Then,

$$\forall S \subseteq \mathfrak{S}_\alpha.\ [\![c]\!]_\kappa(\gamma(S)) \subseteq \gamma([\![c]\!]_\alpha(S))$$

■

Proof. This follows using a similar reasoning as in the proof of Theorem 5.9. The only difference being the proof of the base case, i.e., the case of basic statements, follows from Lemma 5.7 together with Theorem 7.3. □

As an important corollary, we get that whenever the concrete execution of the program, starting in the empty memory, indicates an error in the program, so does the abstract execution, starting in the empty memory.

Corollary 7.5. For a statement (program) c,

$$\text{if } \top_\kappa \in [\![c]\!]_\kappa(h_\varepsilon, s_\varepsilon) \text{ then } \top_\sigma \in [\![c]\!]_\alpha(emp, \varnothing)$$

where,

$$h_\varepsilon(a) = \bot \text{ and } s_\varepsilon(x) = nil$$

■

8 Termination of Analysis with Abstract Semantics

In order to show termination of fixpoint computation – as fixpoints are used for the computation of semantics of while loops –, we show that the set of abstract states is finite. We define a notion of reduced-ness and show that a symbolic state is reduced if and only if it is not changed under abstraction. Then, we show that the set of reduced symbolic states is finite.

In order to formalize the concept of reduced-ness, we first define paths in symbolic states and define some properties of symbols in symbolic states.

Definition 8.1 (Paths, Cycles, Length and Reachability)**.** Let (Σ, Π) be a symbolic state, then, a path in (Σ, Π) is a sequence $\varsigma_0, \varsigma_1, \ldots, \varsigma_{n-1}, \varsigma_n$ such that

$$\forall 0 \leq i \leq n.\ \varsigma_i \in \mathsf{Sym}$$

$$\forall 1 \leq i \leq n.\ \exists \varsigma', \varsigma'' \in \mathsf{Sym}.\ \varsigma_{i-1} =_\Pi \varsigma' \wedge \varsigma_i =_\Pi \varsigma'' \wedge \Sigma\varsigma', \varsigma'')$$

A cycle is a path $\varsigma_0, \ldots, \varsigma_n$ if $\varsigma_0 =_\Pi \varsigma_n$.

In addition, the length of a path (or cycle) $\varsigma_0, \ldots, \varsigma_n$, n here, is the syntactical length of a path (or cycle), i.e., we don't distinguish $\mapsto$ and Ls.

For $\varsigma_0, \varsigma_n \in \mathsf{Sym}$, we say ς_n is reachable from ς_0 in (Σ, Π), if there is a path $\varsigma_0, \varsigma_1, \ldots, \varsigma_{n-1}, \varsigma_n$ in (Σ, Π). ■

Definition 8.2 (Shared, Internal, Possibly Dangling and Pointing to Possibly Dangling Symbols)**.** Let (Σ, Π) be a symbolic state, then,

> ***Shared Symbols:*** A symbol ς is a shared symbol, if $\Sigma = \Sigma' * P_1(\varsigma_1, \varsigma_2) * P_2(\varsigma_3, \varsigma_4)$ such that $\varsigma =_\Pi \varsigma_2 =_\Pi \varsigma_4$.
>
> ***Internal Symbols:*** A symbol ς is an internal node of a cycle in (Σ, Π) if and only if it is not shared.
>
> ***Possibly Dangling Symbols:*** A symbol ς is a possibly dangling symbol if $\varsigma \neq_\Pi nil$ and
>
> $$\exists \varsigma_1, \varsigma_2 \in \mathsf{Sym}.\ \Sigma = \Sigma' * P(\varsigma_1, \varsigma_2) \wedge \varsigma_2 =_\Pi \varsigma$$
>
> and
>
> $$\nexists\ \varsigma_3, \varsigma_4 \in \mathsf{Sym}.\ \Sigma' = \Sigma'' * P'(\varsigma_3, \varsigma_4) \wedge \varsigma_3 =_\Pi \varsigma$$
>
> ***Pointing to Possibly Dangling Symbols:*** A symbol ς points to a possibly dangling symbol if $\Sigma = \Sigma' * P(\varsigma_1, \varsigma_2)$ for some $\varsigma_1, \varsigma_2 \in \mathsf{Sym}$ such that $\varsigma =_\Pi \varsigma_1$ and ς_2 is a possibly dangling symbol.
>
> ■

Definition 8.3 (Reduced Symbolic States)**.** Let (Σ, Π) be a symbolic state, then, (Σ, Π) is reduced if and only if

1. There are no primed variables appearing in Π.
2. Every primed variable appearing in Σ is reachable from some program variable.
3. If a primed variable x' appears in Σ, then, at least one of the followings hold
   (a) x' is shared

(b) x' is the internal node of a cycle of length exactly 2

(c) x' is pointing to a possibly dangling variable

(d) x' is possibly dangling

■

Lemma 8.4. Let (Σ, Π) be a symbolic state such that $(\Sigma, \Pi) \not\models \mathit{false}$. Then, (Σ, Π) is reduced if and only if $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightarrow}$. ■

Proof. We prove 'if' and 'only if' parts as follows:

$\Rightarrow$. If (Σ, Π) is a reduced, we show that $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\rightarrow}$. As (Σ, Π) is reduced, there are no primed variables in Π and thus, $rp_{\Pi}(\Sigma, \Pi) = (\Sigma, \Pi)$. Furthermore, we show that if any of the spatial abstraction rules is applicable in (Σ, Π), (Σ, Π) is not reduced which contradicts our assumption.

$\mathit{Garbage}1$: If $\mathit{Garbage}1$ is applicable in (Σ, Π), we should have $\Sigma = \Sigma' * P(x', \varsigma)$ for some $\varsigma \in \mathsf{Sym}$ such that $x' \notin \mathit{Vars}'(\Sigma', \Pi)$. In such a case, x' is a primed variable not reachable from any program variable which contradicts condition 2 of Definition 8.3.

$\mathit{Garbage}2$: If $\mathit{Garbage}2$ is applicable in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(x', y') * P_2(y', x')$ for some $x', y' \in \mathit{Vars}'$ such that $x', y' \notin \mathit{Vars}'(\Sigma', \Pi)$. In such a case, x' and y' are primed variables not reachable from any program variable which contradicts condition 2 of Definition 8.3.

$\mathit{Abs}1$: If $\mathit{Abs}1$ applies in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(\varsigma_1, y') * P_2(y', \varsigma_2)$ for $\varsigma_1, y', \varsigma_2 \in \mathsf{Sym}$ such that $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2\}$ and $\varsigma_2 =_{\Pi} \mathit{nil}$.
Since $y' \notin \mathit{Vars}'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2\}$, y' is not shared. In addition, y' can not be on a cycle since there should be path from ς_2 to ς_1 which is impossible as $\varsigma_2 =_{\Pi} \mathit{nil}$ and thus we would have $(\Sigma, \Pi) \vdash \mathit{false}$ which contradicts our assumption.
On the other hand, since $\varsigma_2 =_{\Pi} \mathit{nil}$, ς_2 can not be a dangling symbol and thus y' can not be pointing to a dangling symbol. Moreover, y' can not itself be possibly dangling as it is appearing on the left hand side of $P_2(y', \varsigma_2)$.

$\mathit{Abs}2$: If $\mathit{Abs}2$ applies in (Σ, Π), we should have $\Sigma = \Sigma' * P_1(\varsigma_1, y') * P_2(y', \varsigma_2) * P_3(\varsigma_3, \varsigma_4)$ for $\varsigma_1, y', \varsigma_2, \varsigma_3, \varsigma_4 \in \mathsf{Sym}$ such that $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$ and $\varsigma_2 =_{\Pi} \varsigma_3$.
Since $y' \notin \mathit{Vars}'(\Sigma, \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$ holds, y' is not shared. Moreover, since $\varsigma_2 =_{\Pi} \varsigma_3$, ς_2 can not be a dangling symbol and thus y' can not be pointing to a dangling symbol. On the other hand, y' can not itself be possibly dangling as it is appearing on the left hand side of $P_2(y', \varsigma_2)$.
In addition, y' can not be on a cycle of length exactly two. There are three ways that can result in y' to be part of a cycle of length two and we show that all three of these cases are impossible.

1. We have $\varsigma_2 =_{\Pi} \varsigma_1$ which is impossible given $(\Sigma, \Pi) \not\models \mathit{false}$ as $\varsigma_2 =_{\Pi} \varsigma_3$ and ς_3 is allocated in a portion of the heap that is disjoined from the portion where ς_1 is allocated.
2. We have $\Sigma' = \Sigma'' * P_0(\varsigma_5, \varsigma_6)$ for $\varsigma_6 =_{\Pi} y'$ and $\varsigma_5 =_{\Pi} \varsigma_2$. This is impossible as $\varsigma_5 =_{\Pi} \varsigma_5$ would result in $(\Sigma, \Pi) \vdash \mathit{false}$ which contradicts our assumption.
3. We have $y' =_{\Pi} \varsigma_4$. This is impossible as $\varsigma_4 =_{\Pi} y'$ would require y' appearing in Π which contradicts condition 1 of Definition 8.3 or have $\varsigma_4 = y'$ which contradicts $y' \notin \mathit{Vars}'(\Sigma', \Pi) \cup \{\varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4\}$.

$\Leftarrow$. We assume that $(\Sigma, \Pi) \overset{\mathcal{A}}{\not\to}$ and show that (Σ, Π) is reduced. First, we observe that since no spatial abstraction rule changes Π, we have $rp_\Pi(\Sigma, \Pi) = (\Sigma, \Pi)$ which means there are no primed variables in Π. This justifies the first condition of Definition 8.3.

Let x' be a primed variable in Σ not reachable from any program variable. We show that at least one of the abstraction rules will apply which contradicts our assumption. We consider the following cases. In the following, we use left occurrence and right occurrence for referring to a symbol appearing in the first or second argument of a P chunk, respectively.

- The primed variable x' can not be only appearing in the left hand side of a P chunk. If there is only one such chunk, $Garbage1$ would be applicable. If there are more than one such chunks, (Σ, Π) would be inconsistent. Therefore, there always exists a chunk $P(\varsigma, x')$ in Σ for some $\varsigma \in \mathsf{Sym}$.
- We have x' is not part of any cycles. Let's assume that $\rho = x'_1, \ldots, x'_n, x'$ is a maximal path ending in x', i.e., there does not exist y' such that there is a path from y' to x'_1. Then, rule $Garbage1$ would be applicable as x'_1 can not be appearing in any chunk other that the one being part of ρ, otherwise, either consistency of (Σ, Π) or maximality of ρ is violated.
- We have x' is part of a cycle ρ. First note that ρ can not have any program variables or nil as it would contradict x' not being reachable by any program variables or consistency of (Σ, Π).

  Second, let us note that no symbol can be part of two different cycles in a consistent symbolic state. Therefore, if any of the primed variables y' of ρ is shared, it can only be pointed to by some primed variable (as otherwise, it would violate x' not being reachable by any program variables or consistency of (Σ, Π)). In such a case, a reasoning like that of the previous case can show that $Garbage1$ would be applicable to the beginning of such a path ending in y'.

  In addition, note that the length of the cycle must be at least 2. If the cycle is of length 1, i.e., we have $P(x', x')$, if x' is shared, we know that the beginning of the path pointing to it has $Garbage1$ applicable to it. If x' is not shared, since (Σ, Π) is consistent, rule $Garbage1$ would be applicable to $P(x', x')$.

  Hence, we assume that all primed variables of ρ are not shared and the length of the cycle is at least 2. As a result, no primed variable y' in ρ can be appearing in any chunk except for the chunks $P_1(x'_1, y')$ and $P_2(y', x'_2)$ such that x'_1, y', x'_2 is part of cycle ρ. Consequently, rule $Abs2$ would be applicable to $P_1(x'_1, y') * P_2(y', x'_2)$ as x'_2 is part of the cycle ρ and is hence equal to some allocated symbol under Π.

Now we consider the cases where $\Sigma = \Sigma' * P(x', \varsigma)$ for some $\varsigma \in \mathsf{Sym}$ or there is no such chunk $P(x', y)$ in Σ. We consider these two cases respectively as follows:

Case 1: $\Sigma = \Sigma' * P(x', \varsigma)$. If x' is shared, the condition 3.a of Definition 8.3 is satisfied. So in the rest of this case, we assume that x' is not shared.

If x' is part of a cycle, as mentioned earlier, the length of such a cycle must be at least 2. If x' is on a cycle of length 2, then the the condition 3.b of Definition 8.3 holds. If x' is on a cycle of length greater than 2, then $Abs2$ would be applicable which contradicts our assumption. Thus, in the rest of this case, we assume that x' is not part of any cycles.

Since x' is not part of any cycles, and x' is reachable from some program variable, we know that there must be a path $x, \varsigma_1, \ldots, \varsigma_n, x', \varsigma$, for some program variable x.

If the length of this path is greater than 2, ($n \geq 1$), as x' is assumed not be shared, the rule $Abs2$ would be applicable which is contradictory. Therefore, we assume that $\Sigma = \Sigma' * P_1(x, x') * P_2(x', \varsigma)$. In such a case, if $\varsigma =_\Pi nil$, $Abs1$ would be applicable which is contradictory. If $\varsigma =_\Pi \varsigma_2$ for some symbol ς_2 and ς_2 occurs on the left hand side of a P chunk, $Abs2$ would be applicable which is again contradictory. Thus, ς must be a dangling symbol, which implies that x' is pointing to a dangling symbol which is the condition 3.c of Definition 8.3.

Case 2: There is no Σ' such that $\Sigma = \Sigma' * P(x', \varsigma)$. In this case, x' only appears as the second argument of some P chunk. So let's assume $\Sigma = \Sigma' * P(\varsigma_1, x')$ for some $\varsigma_1 \in \mathsf{Sym}$. As x' does not appear at the left hand side of any P chunk, and there are no primed variables appearing in Π, nothing equal to it based on Π can be appearing as the left hand side of a P chunk. Consequently, x' must be a dangling symbol which is condition 3.d of Definition 8.3.

□

Now, in order to show that the set of consistent reduced symbolic states (which we just proved is the same as the set of abstract symbolic states) is finite, we show that if a consistent symbolic state is reduced, then, there is an upper bound on the number of primed variables that can be appearing in it.

Lemma 8.5 (Partitioning of Primed Variables)**.** Let (Σ, Π) be a consistent reduced symbolic state. Then, the set $\{X_s, (X_c \setminus X_s), (X_p \setminus X_s), (X_d \setminus X_s)\}$ where X_s, X_c, X_p and X_d are defined below, is a partitioning of primed variables appearing in Σ.

- $x' \in X_s$ if x' is shared
- $x' \in X_c$ if s' is the internal node of a cycle of length exactly 2
- $x' \in X_p$ if x' is pointing to a possibly dangling variable
- $x' \in X_d$ if x' is possibly dangling

■

Proof. Since (Σ, Π) is reduced, we obviously have $Vars'(\Sigma) = X_s \cup (X_c \setminus X_s) \cup (X_p \setminus X_s) \cup (X_d \setminus X_s)$. The only thing that remains to be shown is that these sets are pairwise disjoint.

Obviously,

$$X_s \cap (X_c \setminus X_s) = X_s \cap (X_p \setminus X_s) = X_s \cap (X_d \setminus X_s) = \varnothing$$

If any primed variable is part of a cycle, it can neither be dangling nor can it be pointing to a dangling symbol. Hence,

$$(X_c \setminus X_s) \cap (X_p \setminus X_s) = (X_c \setminus X_s) \cap (X_d \setminus X_s) = \varnothing$$

On the other hand, if a symbol is dangling, it can not be pointing to any (dangling) symbol. As a result,

$$(X_c \setminus X_s) \cap (X_d \setminus X_s) = \varnothing$$

□

Lemma 8.6 (Bound on Primed Variables of Consistent Reduced Symbolic States)**.** Let (Σ, Π) be a reduced symbolic state such that $(\Sigma, \Pi) \not\vdash \mathit{false}$, then the number of primed variables appearing in Σ is bound by $3n + 2$, where n is the number of program variables. ■

Proof. Let X_s, X_c, X_p and X_d be as defined in Lemma 8.5. Furthermore, let m_s, m_c, m_p and m_d be respectively the cardinalities of sets X_s, $(X_c \setminus X_s)$, $(X_p \setminus X_s)$ and $(X_d \setminus X_s)$.

Since, symbols appearing on the left hand sides of Ls and $\mapsto$ predicates must be unique (as $(\Sigma, \Pi) \not\models \mathit{false}$) and considering that members of m_d can not appear as the left hand side of any predicate, we have that

$$|\Sigma| \leq n + m_s + m_c + m_p + 1$$

where $|\Sigma|$ is the number of conjuncts of Σ. The 1 at the end is added for the sake of $Junk$ predicate. Furthermore, we do not consider emp chunks.

On the other hand, since all primed variables are reachable from some program variable, they must all be appearing on the right hand side of some Ls or $\mapsto$. Particularly members of m_s must appear at least twice (as they are shared). Hence,

$$2m_s + m_c + m_p + m_d \leq |\Sigma|$$

Altogether,

$$2m_s + m_c + m_p + m_d \leq n + m_s + m_c + m_p + 1$$

From which we can draw the conclusion that

$$m_s + m_d \leq n + 1 \tag{3}$$

For the upper bound of $m_c + m_p$, we can see that if $x' \in m_c \cup m_p$, there must be a chunk $P(\varsigma, x')$ in Σ for $\varsigma \in \mathsf{Sym}$. In such a case, ς can not be in m_c, m_p and m_d. If $\varsigma \in m_c$, then, ς is a primed variable and together with x' form a cycle of length 2 and are both not shared which means rule $Garbage2$ should be applicable which, according to Lemma 8.4, is a contradiction to the fact that (Σ, Π) is reduced. If $\varsigma \in m_p$, then ς should be pointing to a possibly dangling symbol but x' is already allocated (as it is part of a cycle or is pointing to a possibly dangling symbol) and can not be a possibly dangling symbol. If $\varsigma \in m_d$, then ς must be a possibly dangling symbol which means it can not appear on the left hand side of any Ls or $\mapsto$ predicate which contradicts our initial assumption that $P(\varsigma, x')$ is a chunk of Σ. Consequently, ς can only be a shared primed variable or a program variable. Hence,

$$m_c + m_p \leq n + m_s$$

From Equation 3, we get that $m_s \leq n + 1$ which means:

$$m_c + m_p \leq 2n + 1 \tag{4}$$

From Equation 3 and Equation 4, we have that

$$m_s + m_c + m_p + m_d \leq 3n + 2 \tag{5}$$

Which means that a consistent reduced symbolic state can have at most $3n + 2$ primed variables appearing in it. □

Theorem 8.7 (Finiteness of Abstract States)**.** Let n be the number of program variables, then, the number of reduced states is bounded by $2^{(2^{n+1})(16n^2+20n+7)}$.

Proof. Since there are no primed variables in the pure part of a reduced symbolic state, there can be at most $n + 1$ elements in each equivalence class. This means that we can get a very coarse bound of $2^{2^{n+1}}$ equivalence relations (pure parts).

On the other hand, from Lemma 8.6, we have that there can be at most $4n + 2$ symbols (primed and program variables) appearing on the left hand side of a Ls or $\mapsto$ predicate and at most $4n + 3$ symbols (primed and program variables together with nil) appearing on the right hand side. Therefore, there can be at most $16n^2 + 20n + 6 + 1$ chunks (one for $Junk$ predicate) that can possibly appear in spatial part. Therefore, there can be at most $2^{16n^2+20n+7}$ different spatial parts.

This means that in total there can be at most $2^{(2^{n+1})(16n^2+20n+7)}$ different reduced symbolic states. Hence, $\mathfrak{S}_\alpha$ is finite. □

Acknowledgements

This work was partially funded by EU FP7 FET-Open project ADVENT under grant number 308830.

References

[BCO05] Josh Berdine, C Calcagno, and P W O'Hearn. Symbolic Execution with Separation Logic. *Programming Languages and . . .* , 3780(Chapter 5):52–68, 2005.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. *Relational inductive shape analysis*, volume 43. ACM, New York, New York, USA, January 2008.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C Necula. Shape Analysis with Structural Invariant Checkers. *Static Analysis*, 4634(Chapter 24):384–401, 2007.

[DOY06] Dino Distefano, Peter W O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. pages 287–302, 2006.

[LRS04] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement for 3-valued-logic analysis. *Submitted for publication*, 2004.

[OCDY06] P W O'Hearn, C Calcagno, Dino Distefano, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. pages 182–203, 2006.

[Rey] J C Reynolds. Separation logic: a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Comput. Soc.