Attacking Large Industrial Code with Bi-Abductive Inference

Dino Distefano

Queen Mary, University of London

Abstract. In joint work with Cristiano Calcagno, Peter O'Hearn, and Hongseok Yang, we have introduced *bi-abductive inference* and its use in reasoning about heap manipulating programs [5]. This extended abstract briefly surveys the key concepts and describes our experience in the application of bi-abduction to real-world applications and systems programs of over one million lines of code.

1 Introduction

Automatic software verification has seen an upsurge of interest in recent years. This is exemplified by tools such as SLAM [1] and ASTRÉE [4], which have been used to verify properties of special classes of real-world software, e.g., device drivers and avionics code. Crucial in this reinvigoration of software verification has been the employment of methods from static program analysis which have the advantage to lessen annotation burden (e.g., by automatically inferring loop invariants and procedure summaries).

While these advances are impressive, a persistent trouble area stands in the way of verification-oriented program analysis for a wider range of real software: *the heap.* The heap is one of the hardest open problems in automatic verification and prominent tools such as ASTRÉE and SLAM either eschew dynamic allocation altogether or use coarse models that assume pointer safety.

Shallow pointer analyses, which infer dereferencing information of bounded length, often do not give enough information for verification purposes. For example, for automatically proving that a device driver manipulating a collection of nested cyclic linked lists, does not dereference `null` or a dangling pointer, the analysis technique needs to be able to look unboundedly deep into the heap. This is done by shape analyses [13]. Shape analyses are program analyses which aim to be accurate in the presence of deep-heap update—They go beyond aliasing or points-to relationships to infer properties such as whether a variable points to a cyclic or acyclic linked list.

Until very recently shape analyses could only be applied to tiny toy programs written to test an analysis. SpaceInvader [8,2,10] is an automatic tool aiming at bringing such analyses into the real world. The driving force behind Space Invader is the idea of local reasoning, which is enabled by the Frame Rule of separation logic [11]:

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}$$

M. Alpuente, B. Cook, and C. Joubert (Eds.): FMICS 2009, LNCS 5825, pp. 1–8, 2009.

In this rule R is the *frame*, i.e., the part of the heap which is not touched by the execution of the command C. The connective $*$ is called *separating conjunction* and it states that its operands hold for disjoint parts of memory. The Frame Rule allows pre and postconditions to concentrate on the *footprint*: the cells touched by command C. In by-hand proofs this enables specifications to be much more succinct than they might otherwise be. SpaceInvader takes as its aim to port the concept of footprint into automatic verification in order to enjoy similar benefits and keep the proof process manageable.

2 Bi-Abduction

In moving from by-hand to automatic verification the ability to deduce the frame becomes a central task. Computation of the frame is done by *frame inference*, which can be formally defined as:

Definition 1 (Frame inference). *Given (separation logic) formulae H and H' compute a formula $\mathcal{F}$ such that $H \vdash H' * \mathcal{F}$ holds.*

An algorithm for inferring frames was introduced in [3]. Interestingly, crucial tasks necessary to perform automatic heap analysis — such as rearrangement (materialization) and abstraction — can be reduced to solving frame inference questions [9].

In our attempts to deal with incomplete code and increase automation in Space Invader, we discovered that the idea of *abductive inference* (or abduction) — introduced by Charles Peirce in the early 1900s in his writings on the scientific process [12] — is highly valuable. When reasoning about the heap, abductive inference, often known as inference of explanatory hypotheses, is a natural dual to the notion of frame inference, and can be defined as follows:

Definition 2 (Abductive Inference). *Given (separation logic) formulae H and H' compute a formula $\mathcal{A}$ such that $H * \mathcal{A} \vdash H'$ holds.*

In this definition we call $\mathcal{A}$ the "anti-frame".

Bi-abductive inference (or bi-abduction) is the combination of frame inference and abduction. It consists of deriving at the same time frames and anti-frames.

Definition 3 (Bi-Abductive inference). *Given (separation logic) formulae H and H' compute a frame $\mathcal{F}$ and an anti-frame $\mathcal{A}$ such that $H * \mathcal{A} \vdash H' * \mathcal{F}$ holds.*

Many solutions are possible for $\mathcal{A}$ and $\mathcal{F}$. A criterion to judge the quality of solutions as well as a bi-abductive prover were defined in [5].

Example 1. Let $H \triangleq z \mapsto \mathsf{nil} * x \mapsto \mathsf{nil}$ and $H' \triangleq \mathsf{list}(x) * \mathsf{list}(y)$. Informally H represents a heap with two disjoint cells allocated at addresses x and z which contain the value nil[1]. H' stands for a heap with two disjoint allocated lists starting at x and y, respectively. Consider now the bi-abduction question:

[1] The semantics of the predicate $a \mapsto b$ is a heap with precisely one allocated cell at address a with content b.

$$z \mapsto \mathsf{nil} * x \mapsto \mathsf{nil} * \mathcal{A} \vdash \mathsf{list}(x) * \mathsf{list}(y) * \mathcal{F}$$

There are many solutions for the pair $\mathcal{A}$ and $\mathcal{F}$, some of which are

$$\begin{array}{ll}
\mathcal{A} \triangleq \mathsf{list}(y) & \mathcal{F} \triangleq z \mapsto \mathsf{nil} \\
\mathcal{A} \triangleq y \mapsto \mathsf{nil} & \mathcal{F} \triangleq \exists v. z \mapsto v \\
\mathcal{A} \triangleq y \mapsto \mathsf{nil} & \mathcal{F} \triangleq \mathsf{list}(z) \\
\mathcal{A} \triangleq y \mapsto \mathsf{nil} * w \mapsto 0 & \mathcal{F} \triangleq \mathsf{list}(z) * \exists v. w \mapsto v
\end{array}$$

Notice how in synthesizing $\mathcal{A}$ we are discovering the part of the heap which is missing in H w.r.t. H'. Dually, $\mathcal{F}$ represents the part of the heap H which is superfluous w.r.t. H'. Given that there are many solutions, an automatic prover will essentially make pragmatic choices in order to synthesize only one. In our experience aiming for the "best" solution is hard.

3 Compositional Shape Analysis

Bi-abduction allows us to automatically compute (approximations of) *footprints* of commands and preconditions of procedures. In particular, bi-abduction is the main ingredient which allows for an analysis method where pre/post specs of procedures are inferred independently of their context. This has opened up a way to design compositional shape analyses for sequential [5], and recently concurrent programs [6]. Such analyses can be seen as the attempt to build proofs for Hoare triples of a program. More precisely, given a program composed by procedures $p_1(\boldsymbol{x_1}), \ldots, p_n(\boldsymbol{x_n})$ a compositional analysis automatically synthesizes preconditions $P_1, \ldots, P_n$ and postconditions $Q_1, \ldots, Q_n$ such that the following are valid Hoare triples:

$$\{P_1\}\, p_1(\boldsymbol{x_1})\, \{Q_1\}, \ldots, \{P_n\}\, p_n(\boldsymbol{x_n})\, \{Q_n\}$$

The triples are constructed by symbolically executing the program and by *composing* existing triples. The composition (and therefore the construction of the proof) is done in a bottom-up fashion starting from the leaves of the call-graph and then using their triples to build other proofs for procedures which are on a higher-level in the call-graph. To achieve that we use a special rule for sequential composition which embeds directly the concept of bi-abduction:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \qquad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * \mathcal{A}\}\, C_1; C_2\, \{Q_2 * \mathcal{F}\}} \qquad Q_1 * \mathcal{A} \vdash P_2 * \mathcal{F}$$

A compositional analysis has a great ability to scale since procedures are analyzed in isolation and, moreover, the analysis results of procedures can be easily reused. When dealing with large programs, the ability to analyze parts of the program independently of others, allows us to load only small parts of the source program into memory avoiding to overspill the RAM and cause the analysis to thrash. Finally, compositional analysis is *incremental*: that is, if the program changes after being analyzed, only the modified part need to be re-analyzed.

The results of the previous analysis are still valid for those parts of the program which did not change. All these features provide a strong boost to accurate heap analysis and make it scale up to millions of lines of code. Previous shape analyses were whole-program, non-compositional and therefore did not scale.[2]

We have implemented a compositional shape analysis which uses abduction in a new version of SpaceInvader called SpaceInvader/Abductor (or Abductor for short).

4 Application to Real Code

In this section we discuss our experience of running SpaceInvader/Abductor on large open source codebases (e.g. a complete Linux Kernel distribution with over 2.5 million lines of code). Figure 1 reports the results we obtained from these experiments. The case studies were run on a machine with two 2.66GHz Quad-Core Intel Xeon processors with 4GB memory. The number of lines of C code was measured by instrumenting `gcc` so that only code actually compiled was counted. The analysis was run using only one core in all examples except Linux for which, instead, we used 8 cores. The experiments were run using a timeout of one second.

The green bars indicate the percentage of procedures with at least one consistent non-trivial specification found from the analyzer. The precondition of a discovered specification denotes a set of states on which it is safe to run the procedure: that is, states for which one will not get pointer errors such as a double-free, dereference of null/dangling pointers, or memory leaks. Thus, for example, if a procedure disposes an acyclic list the precondition will not describe cyclic lists, because otherwise the procedure would commit a null-pointer violation.

The red bars instead show the percentage of procedures for which the analyzer was not able to synthesize any specification. The best results were obtained for the IMAP experiment for which SpaceInvader/Abductor synthesized specifications for 68.3% of the total number of procedures. The worst was OpenSSH for which 45.3% of consistent specs were found. For Linux, specs for 58.4% of procedures were discovered.

Focussing on the IMap example. Currently, Space Invader/Abductor provides little support for interpreting the data resulting from the analysis. Given this current user support and the huge quantity of results we decided to look closely only at the data related to IMAP.[3] Here we briefly summarize the outcomes. As indicated above consistent specifications were found for 68.3% of the procedures. Among the discovered specifications, we observed that 18 procedures (i.e., 1% of the total and 1.5% of the successfully analyzed procedures) reported preconditions involving complex data structures (e.g., different kinds of nested and

[2] The largest example of whole-program shape analysis in the literature is around 10K lines of code [10].

[3] We used `cyrus-impad-2.3.13` downloaded from `http://cyrusimap.web.cmu.edu`

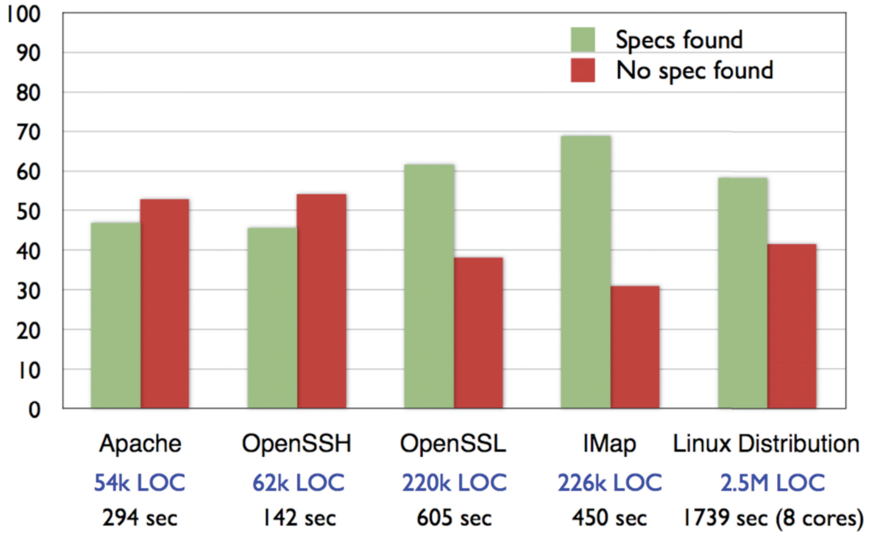

Fig. 1. Results of SpaceInvader/Abductor's analysis on large open source projects

non-nested lists). This indicates that a minority of procedures actually traverse data structures.

Figure 2 reports (in a pictorial form) one of the three (heap) specifications discovered for the procedure `freeentryatts`. The precondition is given by the box on the top labelled by "PRE1". On the bottom there are two post-conditions labelled by "POST1" and "POST2", respectively. The intuitive meaning is that when running the procedure `freeentryatts` starting from a state satisfying PRE1, the procedure does not commit any pointer errors, and if it terminates it will reach a state satisfying either POST1 or POST2. A pre (or a post) displays a heap structure. A small white rectangle with a label denotes an allocated cell, a red rectangle stands for a possibly dangling pointer and a green rectangle denotes nil. A long grey rectangle represents a list. A dashed blue box shows the internal structure of the elements of a list. Hence we can observe that the footprint of `freeentryatts` consists of a nested non-circular singly linked-list.

Figure 3 shows one specification of the function `freeattvalues`. It deallocates the fields in the list pointed to by its formal parameter `l`. The procedure `freeentryatts` calls `freeattvalues(l->attvalues)` asking to free the elements of the inner list. Notice how the bottom-up analysis composes these specifications. In `freeentryatts` the elements of the inner list pointed to by `attvalues` are deallocated by using (composing) the specification found for `freeattvalues` which acts on a smaller footprint. The field `entry` is instead deallocated directly inside `freeentryatts`.

This relation between `freeentryatts` and `freeentryattvalues` illustrates, in microcosm, the modularizing effect of bi-abductive inference. The specification of `freeentryattvalues` does not need to mention the enclosing list from `freeentryatts`, because of the principle of local reasoning. In a similar way, if

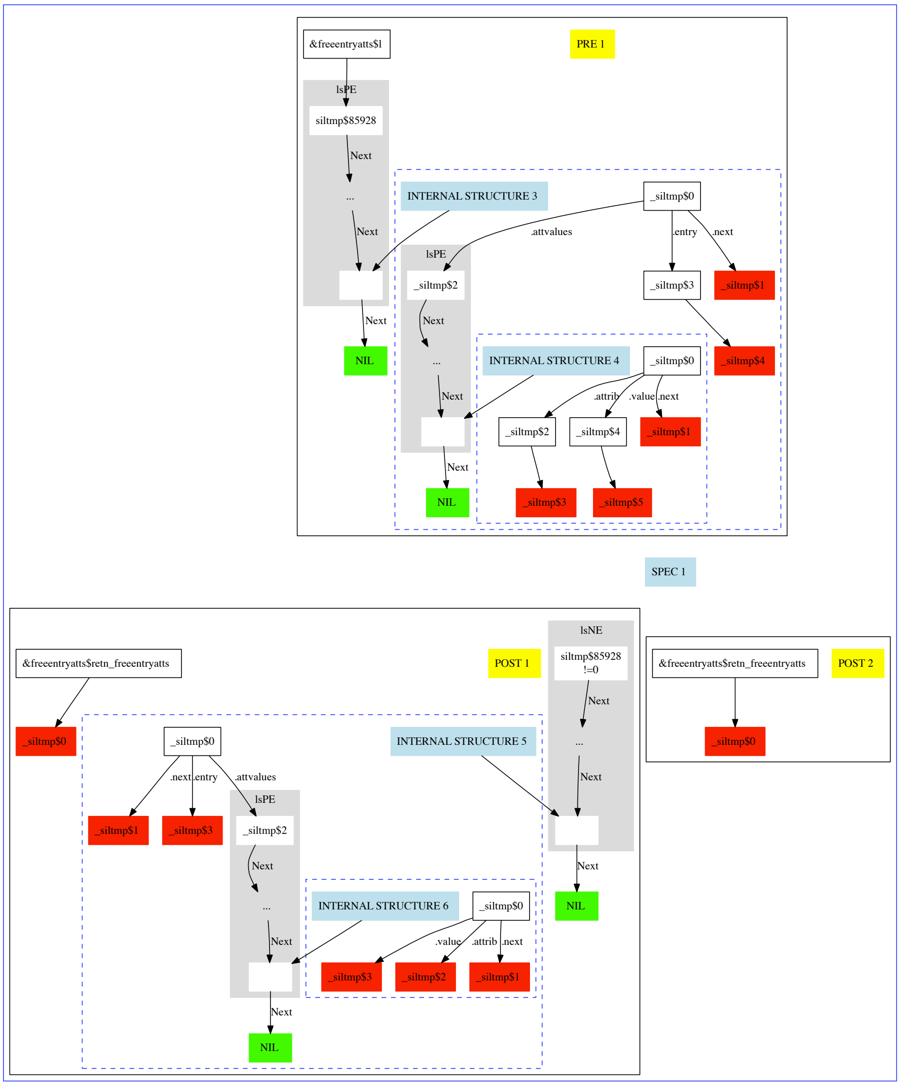

Fig. 2. A specification automatically synthesized by SpaceInvader/Abductor for the procedure `freeentryatts` of the IMap example

a procedure touches only two or three cells, there will be no need to add any predicates describing entire linked structures through its verification. In general, analysis of a procedure does not need to be concerned with tracking an explicit description of the entire global state of a system, which would be prohibitively expensive.

Only 4 procedures timed out (that is 0.4% of the total). Among the procedures for which the analysis was unable to synthesize specifications, 84 potential

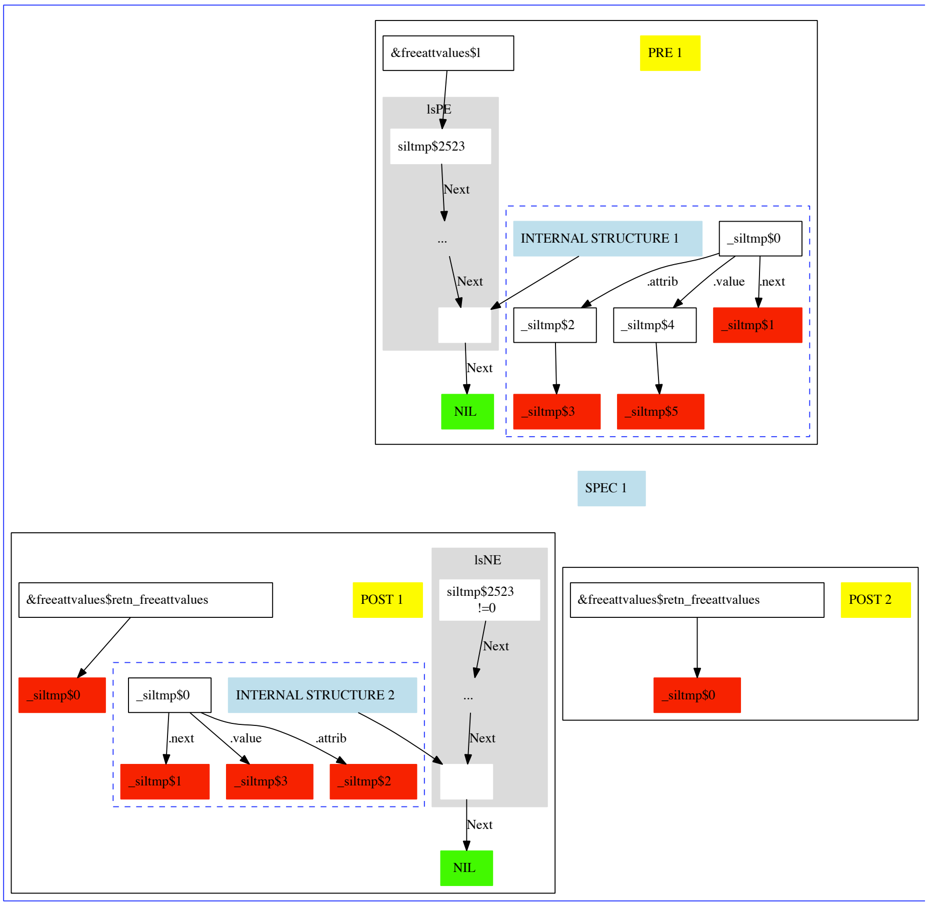

Fig. 3. A specification for the procedure `freeattvalues` called by `freeentryatts`

memory leaks were reported by SpaceInvader/Abductor. A quick inspection of these possible errors revealed that 19 cases (22.6%) were clearly real leaks, whereas 26 cases (30.9%) were false bugs. For the remaining 39 cases (46.4%), it was not easy to establish whether or not they were genuine bugs. This would require a good knowledge of the source code and/or better user support in reporting possible errors, a feature that is currently lacking in Abductor.[4] Nevertheless, given that SpaceInvader/Abductor was not designed as a bug catcher, but rather as a proof tool, we found the unveiling of several real bugs a pleasant surprising feature of our technology. In this context, we add a final consideration. We emphasize that SpaceInvader/Abductor computes a genuine over-approximation (with respect to an idealized model) in the sense of abstract interpretation [7]. Thus, in contrast to several unsound bug-catching tools that can detect some

[4] This feature has high priority in our to-do list of future work.

heap errors, when Abductor finds a specification it has constructed a proof which shows that no pointer errors can occur. For instance, from Figure 2 we can infer that `freeentryatts` *does not* leak memory, *does not* dereference a null/dangling pointer, and *does not* double-free memory.

Acknowledgments. I would like to thank Peter O'Hearn for many invaluable suggestions and helpful ideas on preliminary versions of this paper. This work was supported by a Royal Academy of Engineering research fellowship.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: PLDI, pp. 203–213. ACM, New York (2001)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis of composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003, pp. 196–207. ACM, New York (2003)
5. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300. ACM, New York (2009)
6. Calcagno, C., Distefano, D., Vafeiadis, V.: Compositional resource invariant synthesis (submitted, 2009)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM, New York (1977)
8. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
9. Distefano, D., Parkinson, M.: jStar: Towards Practical Verification for Java. In: OOPSLA, pp. 213–226. ACM, New York (2008)
10. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
11. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
12. Peirce, C.: Collected papers of Charles Sanders Peirce. Harvard University Press, Cambridge (1958)
13. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM TOPLAS 20(1), 1–50 (1998)