Allocation-Site Aware Shape Analysis and Applications in Hard Real-Time Systems

Jörg Herter
Saarland University, Saarbrücken, Germany
jherter@cs.uni-saarland.de

Abstract

Shape analysis aims at determining invariants of heap-allocated structures that arise during the execution of a program. Current shape analysis techniques are stateless, i.e. they only model the structures arising on the heap and completely ignore their memory locations and where they were allocated. This paper proposes an extended, allocation-site aware shape analysis and briefly sketches fields of applications for such an analysis in the area of (hard) real-time systems.

1 Introduction

Shape analysis denotes a static program analysis that determines the *shape* of—or invariants that hold for—the heap at the different program points. Most recent approaches to shape analysis rely either on separation logic [8] to express inferred properties of structures arising on the heap [2], or they model the heap by 3-valued logical structures [9]. The commonality of all approaches is that they are *stateless*. I.e. they only model *what* heap structures may arise, not *where*, i.e. at what memory address, they reside on the heap nor *where* and *when*, i.e. at what allocation site and which invocation thereof, they were allocated.

For the current field of applications for shape analyses like checking data structure invariants [9, 2] and memory safety or even verifying partial program correctness [7], information about where and when heap objects were allocated is not required and it may hence be safely abstracted from in order to increase performance. However, in a real-time setting, applications for allocation-site aware shape analyses arise. Consider for example dynamic memory allocation in hard real-time applications. To enable tight bounds on the worst-case execution-times (WCET) of such programs, a WCET analysis must be able to correctly classify most accesses to heap objects as cache hits or cache misses. The *cache mapping* of objects depends on their addresses in memory, i.e. *where* they reside on the heap.

More concretely, we are currently aware of three approaches to enable the determination of tight WCET bounds for programs using dynamic memory allocation. Schoeberl proposes to use predictable hardware caches to separate dynamically and statically allocated objects [10]. To support this approach, a static analysis would have to associate (heap) objects with allocation sites (or just allocation technique: static or dynamic) to decide in which hardware cache an object may reside. Herter et al. propose to use a predictable memory allocator that takes as an additional argument the cache set to which the returned address shall be mapped [5]. As a result, the cache set mapping becomes explicit and statically known. However, for a WCET analysis to benefit from this, heap objects need to be associated with invocations of the dynamic allocator, i.e. *where* and *when* they were allocated. The third approach is only applicable to a subset of hard real-time applications with statically derivable regularities in allocation behavior and aims at removing dynamic allocation completely by replacing it by precomputed memory addresses [4, 3]. This approach heavily relies on a precise static analysis of the program to enable the computation of good memory addresses. It also requires that heap objects can be associated with allocation sites. While a data structure analysis [6] can be used to connect heap structures with allocation sites, an allocation-site aware shape analysis as proposed in this paper can yield more precise information resulting in a more efficient set of precomputed memory addresses.

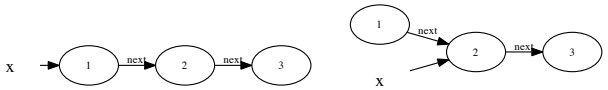
The remainder of this paper is organized as follows. Section 2 briefly introduces the framework for stateless shape analysis via 3-valued logic as proposed in [9]. In Section 3, we sketch how allocation-site awareness can be incorporated into this framework. Section 4 discusses static program analyses that would be enabled by or at least profit from an allocation-site aware shape analysis.

2 Shape Analysis using 3-valued Logic

This section briefly summarizes the framework for shape analysis via three-valued logic. For a detailed discussion, we refer to [9].

Two-valued logical structures can be used to describe concrete heap states. Each heap allocated object is represented by a logical individual, each pointer variable by a unary predicate that evaluates to true iff its argument

is the individual representing the heap object to which the variable points. Field pointers referencing one heap object from another are analogously modeled by binary predicates. Additional so-called *instrumentation predicates* can be defined to increase precision or performance of the analysis. Properties of the heap can be formulated as logical formulæ and checked by evaluating their defining formulæ on the logical structure describing the current heap state. Effects of program statements on the heap state are captured by predicate-update formulæ that state how predicates are updated to yield a structure describing the heap state after execution of a program statement. Figure 1(a) shows a graphical representation of a logical structure describing three objects organized in a singly linked list. Logical individuals are depicted as circles, predicates evaluating to *true* as arrows. Predicates evaluating to *false* are not drawn.



(a) (b)

Figure 1. Two shape graphs each depicting 3 objects organized in a singly linked list.

Applying the effects of the program statement $x = x{\rightarrow}next$; modeled by predicate-update formulæ

$$\begin{aligned} x(v) &\leftarrow \exists u.x(u) \wedge next(u,v) \\ next(u,v) &\leftarrow next(u,v) \end{aligned}$$

yields a structure as depicted in Figure 1(b).

A shape analysis of a given program/method can then be implemented as a fixed point computation collecting for each program state the set of logical structures describing all heap states that may arise there, starting with some initial heap description for the starting point of the program/method. However, an unbounded number of concrete heap description may arise at program points. We therefore introduce abstract heap descriptions using three-valued logical structures that can themselves represent a possibly infinite number of concrete two-valued logical structures. A concrete logical structure is abstracted by partitioning the individuals into equivalence classes such that all individuals within one class yield the same truth values for a predefined set $\mathcal{A}$ of *abstraction predicates*. The individuals of the abstracted structures correspond to these equivalence classes. Abstract individuals that may represent more than one concrete individual are called *summary nodes*. Predicates not in $\mathcal{A}$ need to be reevaluated and may evaluate to the indefinite truth value $1/2$ iff not all concrete individuals summarized by the abstract individual evaluate to the same definite truth value. Abstracting the structure from Figure 1(a) under $\mathcal{A} = \{x\}$ results in the 3-valued logical structure depicted in Figure 2, where dotted arrows represent predicates evaluating to $1/2$ and summary nodes are drawn doubly circled.

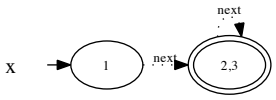


Figure 2. Abstract shape graph embedding in particular the structure from Figure 1(a).

To model the effects of program statements on abstract heap descriptions the same update formulæ as in the concrete setting are used and simply evaluated using 3-valued logic. However, to increase precision, before applying update formulæ the *relevant* parts of the structure are concretized (*focus* or partial concretization). As focusing may generate contradicting or less precise structures, after application of the update formula, the resulting structures are *coerced* into more precise structures and contradicting structures are completely removed.

3 An Allocation-Site Aware Shape Analysis

In order to make the previously described framework allocation-site aware, we associate with each heap object *where* and *when* it was allocated. The number of allocation sites is statically known and for most programs very small. Hence, to model *where* an object was allocated, we introduce additional unary predicates $alloc_m \in \mathcal{A}$ such that $alloc_m(u) = 1$ iff u was allocated at program location m. Furthermore, to model *when* the object was allocated, we construct a function $t^\natural : \mathcal{U} \mapsto \mathbb{N}$ that maps individuals of a concrete structure to invocations of an allocation site. In an abstract structure, we map to intervals of possible invocations: $t : \mathcal{U} \mapsto \mathbb{I}$, where the set of intervals is defined as $\mathbb{I} = \{[l,u] \,|\, l \in \mathbb{N} \wedge u \in \mathbb{N} \wedge l \leq u\}$. Analogously, we can add functions $s^\natural$ and s to associate heap objects with their (requested) sizes. Summarization of two individuals v_1 and v_2 is adapted as follows. Let the new summary node be v_{sm}, then $t(v_{sm}) = t(v_1) \sqcup t(v_2)$ and $s(v_{sm}) = s(v_1) \sqcup s(v_2)$ where $[l_1,u_1] \sqcup [l_2,u_2] = [\min\{l_1,l_2\}, \max\{u_1,u_2\}]$. The logical predicates are reevaluated as in the stateless framework.

Consider the C program given in Listing 1. Figure 3 shows an abstract allocation-site aware shape graph describing the possible heap states occuring after executing line 4. Being more precise than existing data structure analysis, we can identify two data structures and associate their objects precisely with the same occurrence of malloc in program line 12.

In a real-time setting, shape analysis can be performed arbitrarily precise. As in the general setting, we can add instrumentation predicates to increase precision, but we can also deactivate abstraction, i.e. summarization of individuals, completely as no unbounded structures may arise due to known loop and recursion bounds. However, abstract heap structures are still desirable as they may lead to significantly shorter analysis time. The following set of (instrumentation) predicates and additional

Listing 1. C program working on linked lists

```
1  int main() {
2      list * p = buildList(16, ...);
3      list * data = buildList(256, ...);
4      list * x = data;
5      ...
6  }
7  list * buildList(int size, ...) {
8      list * result;
9      ...
10     while(...) {
11         ...
12         ... = malloc(sizeof(list));
13         ...
14     }
15     return result;
16 }
17 struct dll_el * copy( struct sll_el * src ) {
18     struct dll_el * result;
19     ...
20     while ( src != NULL ) { /* loop bound exactly 256 */
21         ... = malloc(...);
22         ...
23         free(...);
24         ...
25     }
26     ...
27     return result;
28 }
```

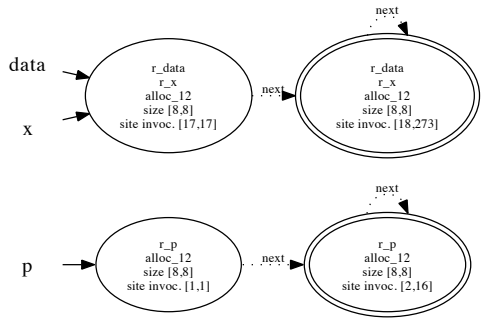


Figure 3. Analysis result after execution of line 4. `site invoc` **corresponds to our** t **function.**

precision increasing techniques have shown good trade-offs between precision and complexity of allocation-site aware shape analyses. To separate different data structures, a predicate $r_x(v)$ modeling reachability from program variables is used: $r_x(v) := \exists u.x(u) \wedge fr(u,v)^*$, where x and fr are predicates corresponding to pointer variables and field references, respectively. Deallocated objects are not removed but marked as freed by a unary predicate $deallocated(v)$. We further increase precision of partial concretization w.r.t. numeric intervals by allowing the analysis to mark predicates modeling field references with superscripts $^<$ and $^>$, indicating that, iff the predicate evaluates to true, both arguments to the predicate are allocated directly after or before each other at the same allocation site. We also introduce an additional abstraction technique that substitutes in intervals of length 1, as in $[5,5]$, the numerical value by newly introduced variables, yielding in the example the interval $[i,i]$. This enables embedding of structures that differ only in numerical values used as interval bounds. The shape graph depicted in Figure 4 is one of the 3 shape graphs arising after execution of line 25, when embedding is extended as described.

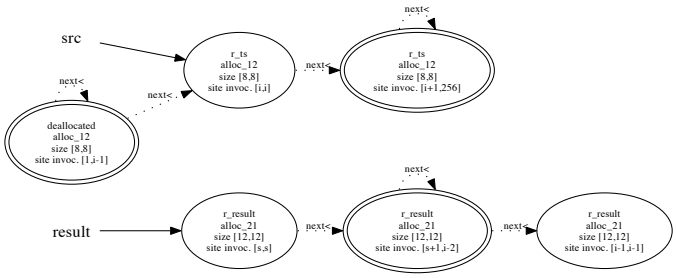


Figure 4. A possible heap state at line 25.

4 Applications

Applying an analysis as discussed in the previous section yields sets of allocation-site aware shape graphs for an analyzed program. Information extracted from these graphs can be used to enable program transformations that increase timing predictability and can even constitute analysis results for other, subsumed analyses. This section gives an overview on how information from shape graphs can be of benefit and sketches the applications for allocation-site aware shape analysis we identified so far.

4.1 Allocation Behavior Analysis

Our main motivation was to enable a more precise timing analysis for hard real-time applications. For programs with statically derivable regularities in object lifetimes, replacing dynamic memory allocation by a precomputed static allocation scheme yields many advantages. The memory addresses of heap allocated objects become known to the timing analysis and unpredictability introduced by the memory allocator is removed together with the allocator itself.

The precomputation of suitable memory addresses for heap objects as proposed in [3] relies on a formal description of a program's allocation behavior. This formal description is given by a six-tuple, $(M, U, L, A, \mathcal{C}, \mathcal{B})$. M is the set containing all allocation sites and U contains upper bounds on how often each allocation site may be reached, i.e. how often this function call may be invoked. Additional knowledge about the relations between elements of U, such as $u_1 < u_2$, is collected in the set L. For each allocation site m, a function f_m is constructed such that $f_m(i)$ evaluates to an interval describing the size of the memory block requested the i-th time allocation site m is reached. A is the set of all such functions. The set R where $R = \left\{(m,i) \mid m \in M \wedge i \in \mathbb{N}^{\leq u_m \in U}\right\}$ contains all allocation requests that may occur during program execution. C is a conflict function $\mathcal{C} : 2^R \mapsto \{0,1\}$ that

evaluates to 1 iff its argument requests at least two memory blocks with overlapping lifetimes. To exploit simple cache placement heuristics, a bias function $\mathcal{B}$ is given as $\mathcal{B} : (R \times R) \mapsto \{0, 1\}$ where $\mathcal{B}(r_1, r_2)$ evaluates to 1 iff the block requested in r_1 is likely to be accessed prior to the one requested in r_2. While M can be directly extracted from the program code, U and L are often provided by the user. A is constructed from M, L, and the requested sizes. These sizes, the conflict function $\mathcal{C}$, and the bias function $\mathcal{B}$ have to be derived by a static program analysis. The size of a heap object—or requested memory block—is explicitly stored in the shape graphs. Functions $\mathcal{B}$ and $\mathcal{C}$ can be extracted from an allocation-site aware shape analysis as follows. $\mathcal{B}(r_1, r_2)$ evaluates to 1 iff there exists a field-pointer predicate evaluating to true for the individuals representing r_1 to r_2, $\mathcal{C}$ evaluates to 1 iff representatives of at least two elements of its argument set are present in the same shape graph and none is marked as deallocated.

4.2 Cache Analysis for Heap Allocated Objects

For programs from which we cannot statically remove dynamic memory allocations, a cache-aware predictable memory allocator may be used. Such an allocator as proposed in [5] can be guided with respect to the cache set mapping of returned addresses via an additional cache set argument. Knowing to what cache sets the memory locations of heap objects are mapped, cache and subsequent WCET analyses may be able to predict cache hits or cache misses for accesses to dynamically allocated objects. Failing to be able to correctly classify a significant number of memory accesses as cache hits or misses would result in high overestimations of a program's WCET. A shape analysis as proposed in this paper can be used to automatically find suitable cache set arguments for allocated objects by extracting program logical structures from the resulting shape graphs (see Section 4.3) and applying a suitable cache set mapping strategy to the respective structures. Also, combining a cache analysis [1] with an allocation-site aware shape analysis is current on-going work.

4.3 Combined Data Structure and Escape Analysis

Data Structure Analysis attempts to identify disjoint instances of program logical data structures and their internal and external connectivity properties [6]. An escape analysis categorizes objects into escaping and not escaping their allocating function [11]. An object is said to escape the function it was allocated in if it may still be accessible after returning from this function. The aim of such an analysis is typically to identify objects that can be allocated on the stack instead of the heap to increase program performance.

Both analysis results can be extracted from allocation-site aware shape graphs. A program logical data structure can always be defined as a connected component of the shape graph. With more knowledge about the data structures used in a program, we can even introduce new predicates to more precisely associate heap objects with program logical data structures. To identify escaping objects, we check within the shape graphs occurring at the exit point of functions whether objects allocated within the function are reachable from returned, static or class objects or also arguments to the analyzed function. Objects passed as arguments to functions called within the analyzed method also escape. Reconsider Figure 3 describing all possible heap states after execution of line 4 of our example code, where an allocation-site aware shape analysis was able to separate all heap allocated objects into two disjoint data structures. Existing data structure analyses like [6] yield a much less precise overapproximation associating all heap objects with one structure.

5 Conclusions

The current stateless shape analysis framework via 3-valued logic can be extended to track information about where and when heap objects were allocated as well as their respective sizes. For applications of shape analysis considered within the community so far, this additional information is not necessary and only tends to increase analysis times. However, for hard real-time programs, applications have emerged that depend on this additional information. Additionally, performance of static analyses is less critical in this setting as the analyzed programs are normally less complex and higher analysis times are justifiable.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96*, London, UK, 1996. Springer-Verlag.

[2] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL '08*, New York, NY, USA, 2008. ACM.

[3] J. Herter and S. Altmeyer. Precomputing memory locations for parametric allocations. In *WCET'10*, 2010.

[4] J. Herter and J. Reineke. Making dynamic memory allocation static to support WCET analyses. In *WCET'09*, 2009.

[5] J. Herter, J. Reineke, and R. Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[6] C. Lattner and V. Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Technical report, University of Illinois at Urbana-Champaign, 2003.

[7] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, 2000.

[8] J. Reynolds. Separation logic: A logic for shared mutable data structures. IEEE Computer Society, 2002.

[9] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

[10] M. Schoeberl. Time-predictable cache organization. In *STFSSD '09*, Washington, DC, USA, 2009.

[11] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. *SIGPLAN Not.*, 34(10), 1999.