Semi-Automatic Verification for Heap-Allocated Data Structures, Focusing on the Perl Interpreter

11th of July, 2009

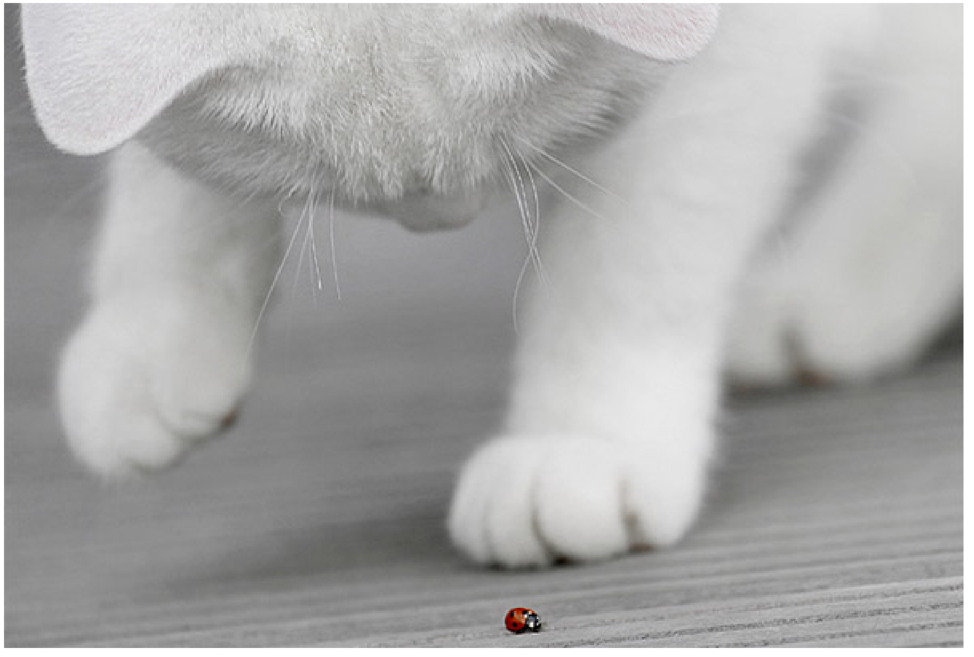
Will Klieber, Jeehoon Kang

Overview

- Motivation
- Description of our Approach
- Example of our Approach
- Comparison to TVLA
- Conclusion

General Motivation

- Many programs use heap-allocated data structures.
- It is often important to verify properties involving such data structures (e.g., memory safety).
- However, existing shape analysis tools (e.g, TVLA) don't seem to scale well to large, messy, real-world programs.



Specific Motivation

We aim to develop a domain-specific analyzer, specialized for the Perl Interpreter, that can verify:

- "Shape" properties of the Perl Interpreter's data structures
- In particular, memory safety
  - Null pointer dereference.
  - Dereference of a dangling pointer.
  - Calling `free` on an address that wasn't allocated via `malloc` or that has already been freed.
  - Accessing memory past the bounds of a struct or an array.

Overview of Our Approach

- We use the Abstract Interpretation framework.
- The memory is represented in terms of predicates.
  - E.g., M = {“ListSeg(p1, p2)” : true, “IsValidPtr(p3)” : true, ...}
- Summarization and Focusing:
  - We summarize the memory state at the bottom of loops and recursive functions.
  - We “bring into focus” (create a concrete representation of) a summarized memory cell whenever we read from it or write to it.

Abstract Representation of Memory

- We represent memory states in terms of predicates.
  - Concretely-represented portions of memory.
    These preds are built into the analyzer.
  - Abstractly-represented (summarized) portions of memory.
    These preds are defined by the user.
- *Simple memory state*: Each predicate mapped to a logical value.
- *Complex memory state*: Conceptually, a collection of simple memory states, as in *collecting semantics*.
- For function summarization, the memory state is parameterized by the input memory state at entry to the function.

Summarization and Focusing

- At the bottom of loops and recursive functions, we verify that the user-supplied predicates hold true of the data structures.
  - These predicates summarize the data structure.
- When we need to create a concrete representation of a summarized memory cell, we do so by using the Focusing operation supplied by the user.
- The Focus and Verify operations are defined directly in terms of the abstract memory representation.

Example: Singly-Linked List

In a simple memory state M, the predicate

`ListSeg(`pA`, `pB`)`

signifies that there is a list segment from pA to pB (or $pA == pB$ in the base case), with no aliasing except as entailed by other predicates true in the simple mem state.

Let us write "p1@L1" to denote the value of `p1` at program point L1.

Example: Singly-Linked List

```
struct node {
  node *pNext;
};

void main() {
    node *p1 = 0;
    node *p2 = malloc(4);
    while (non_det()) {
L1:     node *pTmp = malloc(4);
        pTmp->pNext = p1;
        p1 = pTmp;
L2:     verify(ListSeg(p1, 0));
    }
    p2->pNext = 0xDEADBEEF;
X:  verify(ListSeg(p1, 0));
}
```

In a simple memory state *M*, the predicate

`ListSeg(`pA`, `pB`)`

signifies that there is a list segment from *pA* to *pB* (or *pA*==*pB* in the base case), with no aliasing except as entailed by other predicates true in the simple mem state.

Let us write "p1@L1" to denote the value of `p1` at program point L1.

Example: Singly-Linked List

```
struct node {
  node *pNext;
};

void main() {
    node *p1 = 0;
    node *p2 = malloc(4);
    while (non_det()) {
L1:     node *pTmp = malloc(4);
        pTmp->pNext = p1;
        p1 = pTmp;
L2:     verify(ListSeg(p1, 0));
    }
    p2->pNext = 0xDEADBEEF;
X:  verify(ListSeg(p1, 0));
}
```

To focus on *pA* for

`ListSeg(`*pA*`, `*pB*`)`

we split the memory into two simple mem states:
(1) replace the original predicate with pA==pB,
(2) replace it with pA->pNext == pX and ListSeg(pX, pB),
where pX is a pointer to a new representation of a concrete cell.

In the 2nd case, pX is only aliased where entailed by other predicates.

Example: Singly-Linked List

```
struct node {
  node *pNext;
};

void main() {
    node *p1 = 0;
    node *p2 = malloc(4);
    while (non_det()) {
L1:     node *pTmp = malloc(4);
        pTmp->pNext = p1;
        p1 = pTmp;
L2:     verify(ListSeg(p1, 0));
    }
    p2->pNext = 0xDEADBEEF;
X:  verify(ListSeg(p1, 0));
}
```

To verify

ListSeg(pA, pB)

we check that one of the following holds true:

(1) pA == pB, or

(2) pA == pX and ListSeg(pX, pB) for some pX.

In the second case, we also check that pX is not aliased except where entailed by other predicates.

Related Work: TVLA
(Three-Valued Logic Analyzer)

- TVLA is a state-of-the-art shape analysis engine.
- TVLA's motivation:
  - Parametric framework for developing new shape analysis techniques. "A yacc for shape analysis".
  - Tries to *discover* which data structures have a given shape.
- Tom Reps and Mooly Sagiv

Our Approach vs TVLA (1)

- Unlike TVLA, our approach aims only to *verify* the shape properties of data structures, not to *discover* them.
- We rely on user annotations and heuristics to determine which shape properties should hold true of which data structures.
- This should greatly reduce the computational costs and allow us to scale up to messy real-world programs like the Perl interpreter.

Our Approach vs TVLA (2)

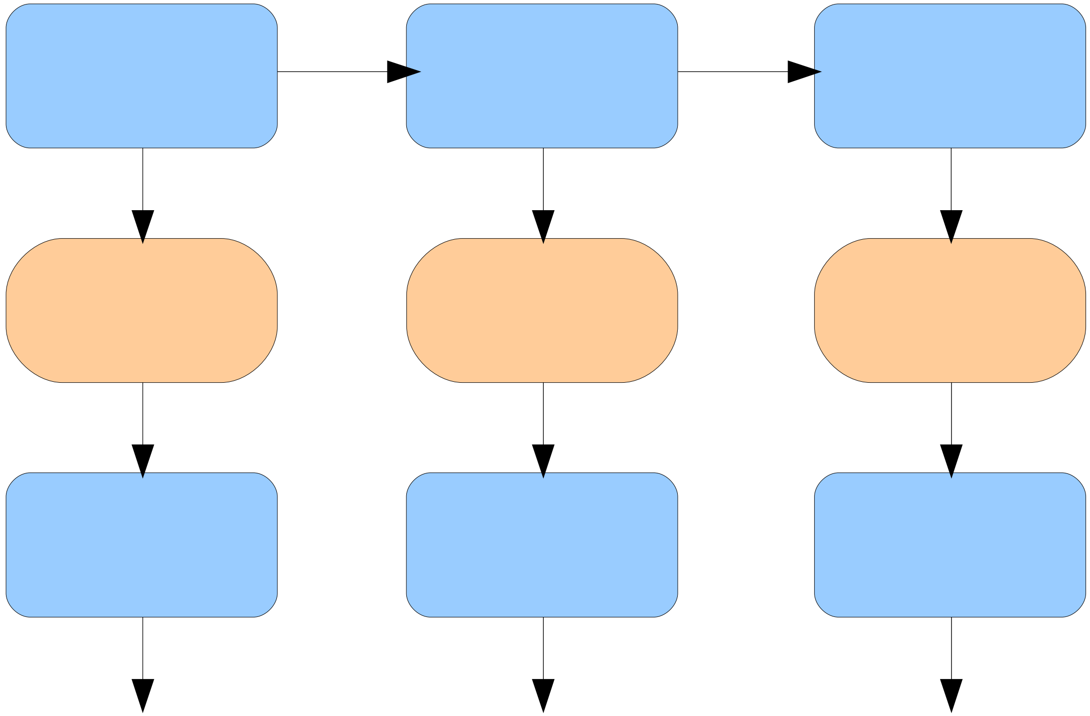
- In TVLA, the user defines a predicate by a formula in First-Order Logic with Transitive Closure (FO+TC).
- Example for singly-linked list:
  - Suppose the predicate *next* describes the forward ptr of a node.
  - Specifically, *next*(*v1*, *v2*) means `v1->next` == `v2`.
  - The transitive closure of *next*, written "$next^{+}$", signifies reachability via the *next* field.
  - Specifically, "$next^{+}(arg1, arg2)$" signifies that *arg2* is reachable by one or more pointer hops from *arg1* via the *next* field.
- User must also supply an *update relation* (transfer function) for each predicate.
  - Indicates effect of a single stmt on the value of the predicate.

Our Approach vs TVLA (3)

- TVLA preds: First-Order Logic with Transitive Closure (FO+TC).
- TVLA's restriction to FO+TC makes it difficult to cleanly express properties of mutually co-recursive data structures.
  - E.g.: In Perl, a CMD may have a pointer to a STAB (“Symbol Table”), and a STAB may have a pointer to another CMD.

```
struct cmd_t {
    cmd_t  *pNext;
    stab_t *pStab;
    ...
};

struct stab_t {
    cmd_t  *pCmd;
    ...
};
```



What about Aliasing/Sharing Within a Data Structure?

- For aliasing within a data structure (i.e., the type of aliasing that occurs in a DAG but not in a tree), the user must explicitly specify the nature of the aliasing in the definition of the predicates and the focusing operations.

Conclusion

- We believe our handling of predicates is more flexible than TVLA's and better suited to messy real-world programs.
- For computational scalability, we require the user to annotate the program to specify which properties hold true of which data structures.
  - We can use heuristics to propagate or guess this information to minimize the burden on the user.
- We hope to verify memory safety and shape properties of the Perl interpreter using this method.

http://www.hacksomnia.com/wp-content/uploads/2009/03/computer-bug.jpg

THE END!

Extra Example: Singly-Linked List

```
void main() {
  node *p1 = 0;
  node *p2 = malloc(4);
  node *pMid = 0;
  while (non_det()) {
L1: node *pTmp = malloc(4);
    if (rand()) {pMid = p1;}
    pTmp->pNext = p1;
    p1 = pTmp;
L2: verify(ListSeg(p1, pMid));
    verify(ListSeg(p1, 0));
  }
  p2->pNext = 0xDEADBEEF;
X:verify(ListSeg(p1, pMid));
  verify(ListSeg(p1, 0));
}
```

At beginning of iteration:

ListSeg(p1@L1, 0)

p2 != p1, pTmp is uninitialized

At end of iteration:

pTmp->pNext@L2 == p1@L1

ListSeg(pTmp->pNext@L2, 0)

p1->pNext@L2 == pTmp@L2

ListSeg(p1@L2, 0)