Shape Analysis Using Separation Logic

Author

Tomáš Brablec

Supervisors

Tomáš Vojnar, Tomáš Dacík

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Introduction

- The goal is to develop a prototype of shape analysis
- Find bugs in code that handles dynamic memory allocation
- Develop as a plugin for the Frama-C framework
- Using separation logic to describe program's memory at each point

- Modular framework for analysis of C code
- Developed in the OCaml programming language
- C code represented as C Intermediate Language (CIL)
- CIL is based on a modified AST with extra information
- Analyses are developed using a plugin system

Software Analyzers

Dataflow analysis

- Runs on the Control Flow Graph (CFG) of the input code
- Each node of the CFG is assigned a state
- States of all nodes are updated until a fixpoint is found for all of them
- User defines a transfer function that computes new states for nodes based on the states of predecessor nodes
- Updates are propagated along the edges of the CFG

- Attempts to solve the problem of globality in analyses
- An SL formula describes the state of the program's memory in an abstract way
- Points-to atoms, list segments, equalities
- Separating conjunction
- Dedicated solver for SL – Astral
- Symbolic heap is the fragment of SL used for this analysis

$$x = y * y \mapsto z * \mathrm{ls}(z, w)$$

Astral

- Derives the shapes of data structures in the heap
- List segments can abstract linked lists of arbitrary length
- Previous work defines shape analysis using SL for a minimal language, without the use of a dedicated solver
- My task is to adapt their method to work on (a subset of) C, and to use Astral for solving SL formulae
- by using a dedicated solver for SL, we can always extend the analysis further (predicates for doubly linked lists, nested lists, even non-pointer data)

Implementation

- Currently limited to programs without loops, function calls, and complex assignments
- A list of symbolic heaps as the state for each node
- The implementation consists of the following:
  - Transfer function
  - Join operation
  - Filtering states for branches of `if` statement
  - Filtering out unsatisfiable formulae, and simplification

Transfer function

- Generates new data for a statement based on the previous statement's data
- Only variable initializations, assignments, and dereferences are analyzed
- Non-pointer variables are ignored
- Initializations by a call to `malloc` split state formula to two, representing a successful and failed allocation
- Assignments are done by substitution of variable's name in the formula

Transfer function (cont.)

- An assignment with a dereference on the right-hand side – the target is found by the transitive closure of equality for the left-hand side variable
- An assignment with a dereference of the left-hand side – similar to the above
- These basic operations will be enough to analyze more complex assignments composed from them

```
a = b;
a = *b;
*a = b;
a = malloc(...);
```

statements, for which the analysis is implemented

Join operation

- When a new state is computed for a node, this function joins it together with the old stored state, and stores the result
- Iterates through all formulae of the new state, and checks its entailment to the whole old state
- Only formulae that do not satisfy the entailment are added
- Otherwise, we would be adding redundant information

$$\varphi_{\text{new}} = \bigvee \varphi_{\text{new}_i}$$

$$\varphi_{\text{new}_i} \models \varphi_{\text{old}}$$

Conditionals

- Generating two states for the branches of an `if` statement based on the condition
- Only equalities and inequalities of two variables are analyzed, other conditions are treated as nondeterministic
- Only formulae that are satisfiable with the given condition added to them are sent to the respective branch

Example of analysis results

`int *nullptr = NULL;`	$\mathrm{nullptr} = \mathrm{nil}$
`void *x = malloc(sizeof(void *));`	$\mathrm{nullptr} = \mathrm{nil} * x \mapsto f_0$ $\mathrm{nullptr} = \mathrm{nil} * x = \mathrm{nil}$
`if (x == nullptr) {`	
`    ;`	$\mathrm{nullptr} = \mathrm{nil} * x = \mathrm{nil}$
`} else {`	
`*(void**)x = malloc(sizeof(void *));`	$\mathrm{nullptr} = \mathrm{nil} * x \mapsto f_0 * f_0 \mapsto f_1$ $\mathrm{nullptr} = \mathrm{nil} * x \mapsto f_0 * f_0 = \mathrm{nil}$
`}`	
`; // join of branches`	$\mathrm{nullptr} = \mathrm{nil} * x = \mathrm{nil}$ $\mathrm{nullptr} = \mathrm{nil} * x \mapsto f_0 * f_0 \mapsto f_1$ $\mathrm{nullptr} = \mathrm{nil} * x \mapsto f_0 * f_0 = \mathrm{nil}$
`x = nullptr;`	$\mathrm{nullptr} = \mathrm{nil} * f_2 = \mathrm{nil} * x = \mathrm{nullptr}$ $\mathrm{nullptr} = \mathrm{nil} * f_3 \mapsto f_0 * f_0 \mapsto f_1 * x = \mathrm{nullptr}$ $\mathrm{nullptr} = \mathrm{nil} * f_4 \mapsto f_0 * f_0 = \mathrm{nil} * x = \mathrm{nullptr}$

Future work

- Abstraction to list segments (needed for termination of loops)
- Analysis across function calls
- Preprocessing passes over CIL to simplify code
- Pull variable declarations to the top scope of each function
- Split more complex assignments into a series of simpler ones
- Find structs that represent linked lists, and rewrite struct member accesses to simple dereferences
- Inline functions to avoid analyzing function calls

Examples of preprocessing

```
*a = **b;
```

↓

```
tmp_1 = *b;
tmp_2 = *tmp_1;
*a = tmp_2;
```

```
list->next = malloc(...);
tmp = list->next;
tmp2 = list2->data;
```

↓

```
*list = malloc(...);
tmp = *list;
// this line would be omitted, only list would be checked, whether it is allocated
```

Summary

- Studied topics: Frama-C framework, dataflow analysis, separation logic, Astral solver, shape analysis
- Implemented a first prototype of shape analysis using separation logic, in the form of a Frama-C plugin