From Low-Level to High-Level Programs

Deliverable D3-3

ANR project VECOLIB

September 2017

Abstract

The code used in Ada or C implementations of standard libraries of containers includes complex data structures where the program heap is explicitly managed using pointers and dynamic allocation. Those containers are then used by client code, in an abstract way. In order to properly analyze such programs, it is important to be able to bridge the gap between the low- and high-level view of the containers' implementations.

This deliverable reports on two approaches developed in the VECOLIB project to tackle this challenge. The first approach is implemented in FRAMA-C. Customizable memory abstractions are provided, that hide the notion of pointers from abstrat domains. Aggregates (structs and arrays) are also flattened – and abstracted for big arrays.

1 Introduction

The static analyser of FRAMA-C, VALUE [2] as well as its new version EVA [1] are able to deal with such view due to a specific memory model, where the contents of locations in memory are seen as sequences of bytes and pointers are abstracted in both integers and addresses. During the VECOLIB project, a customizable *memory abstraction* has been added, which is used to hide from abstract domains the complexities brought by pointers. Instead, they only need to understand *cells*, as explained in Section 2.1.

1.1 Related works

2 High-level abstractions of memory

2.1 Customizable memory abstractions

Pointers are ubiquitous in C. Yet, they are a challenge to usual relational domains (e.g. octagons or polyhedra), which can only relate variables. Aggregates (structs and unions) pose similar problems.

Memory abstractions [4, 3] are generally used to bridge this gap. They abstract memory locations by a set of abstract variables (usually). However, such memory abstractions are generally fixed, and part of the abstract interpretation engine. In EVA, each abstract domain is instead free to choose its own abstraction, resulting in an increased expressivity. For convenience, EVA however supplies customizable memory abstractions, to ease the writing of new analysis domains.

```
module type Value = sig
  type t

  val top : t
  val join : t -> t -> t
  val widen : t -> t -> t
  val is_included : t -> t -> bool

  (** This function must return [true] if the given variable should be
      tracked by the domain. All untracked variables are implicitely
      mapped to [V.top]. *)
  val track_variable: Cil_types.varinfo -> bool
end

module Make_Memory (Value: Value) : sig
  type t
  type value = Value.t

  val top: t
  (** The top abstraction, which maps all variables to {!V.top}. *)

  val join: t -> t -> t
  val widen: t -> t -> t
  val is_included: t -> t -> bool

  (** [add loc typ v state] binds [loc] to [v] in state. If [typ] does
      not match the effective type of the location pointed, [V.top] is
      bound instead. This function automatically handles the case where
      [loc] abstracts multiple locations, or when some locations are not
      tracked by the domain. *)
  val add: Precise_locs.precise_location -> Cil_types.typ -> value -> t -> t

  (** [find loc typ state] returns the join of the abstract values stored
      in the locations abstracted to by [loc] in [state], assuming the
      result has type [typ]. When [loc] includes untracked locations, or when
      [typ] does not match the type of the locations in [loc], the
      result is approximated. *)
  val find: Precise_locs.precise_location -> Cil_types.typ -> t -> value

  (** [remove loc state] drops all information on the locations pointed to
      by [loc] from [state]. *)
  val remove: Precise_locs.precise_location -> t -> t

  (* [...] *)
end
```

Figure 1: OCaml signature for Eva customizable memory abstractions

Figure 1 shows the signature `Value` that must be supplied in order to obtain a memory abstraction. It consists of two parts:

1. An OCaml type `t` that abstracts values (integers, floating-point values, pointers), equipped with a join-lattice structure, plus a widening operation.

2. A function `track_variable` that decides whether the domain is interested in a certain variable. Untracked variables will behave as if mapped to the imprecise value `top`, but for efficiency reasons they will not appear in the OCaml datastructure at all.

The OCaml functor `Make_Memory` returns a memory abstraction that maps all tracked variables to an abstract value. The result of this functor can then be used to implement most of the memory-related operations required by an Eva domain. For example, the function `add` can be used to implement the abstract operation for assignment. The argument of type `Precise_locs.precise_location` (which is the type of abstract locations within Eva) is automatically converted into a set of abstract variables.

This memory abstraction will be part of Frama-C Sulfur.

2.1.1 Extension to aggregates

Currently, the memory abstraction presented above is limited to scalar variables. We are experimenting with an extension to aggregates types, namely structs and arrays. The idea would be to represent each individual field by an abstract variable, which we call a *cell*. The function `track_variable` would be generalized into a function `track_cell`. There are however two difficulties in doing so:

1. By using pointer arithmetics to access a memory region with an improper type, memory accesses can refer to only *parts* of a cell, or worse overlap multiple cells. Although some of the code patterns are forbidden by so-called C strict aliasing rules, many programs use GCC option `-fnostrict-aliasing` and nevertheless perform them. We give examples below.

   ```
   int x = 1;
   // Accesses part of x. Valid because the type used is char
   char *y = *(((char*) &x)+1);

   // Accesses part of x. Invalid
   short *z = *(((short*) &x)+1);

   int t[3] = {1, 2, 4};
   // Accesses parts of t[0] and t[1]. Invalid pointer arithmetics
   int w = *((int *)(((char *)&t)+2));
   ```

   Our cell-based memory abstraction automatically handles such examples, by warning when an access to a cell is partial. In those cases, the memory abstraction performs an imprecise read or update operation.

2. Big arrays can lead to the generation of an unwieldy number of cells, which will in turn degrade the performance of e.g. relational numeric domains. For example, octagons have a cubic complexity, while polyhedra are exponential in the number of variables. It

is thus of particular importance to *smash* those arrays, into a single summary cell. Such summary cells are special, since the information on their contents can only grow. It is indeed impossible to learn a newer, more precise information. This will be automatically handled by the memory abstraction.

We expect this extension of our memory abstraction to also be part of Frama-C Sulfur.

References

[1] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions: EVA an Evolved Value Analysis for Frama-C.* PhD thesis, University of Rennes, 2017.

[2] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[3] V. Laporte. *Vrification danalyses statiques pour langages de bas niveau.* PhD thesis, Universit de Rennes 1, 2012.

[4] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.