Static Analysis for High Level Programs

Deliverable D2-2

ANR project VECOLIB

May 2017

Abstract

This deliverable presents the static analyses developed during the Vecolib project for the analysis of programs with containers. These analyses are based on abstract interpretation and specific abstract domains for containers implementing multiset and sequence mathematical structures.

1 Motivation and Related Work

Containers are general-purpose data structures for inserting, retrieving, removing, and iterating over elements. Examples of containers are array, list, vector, map, set, stack, queue, etc. They are widely used in client programs are therefore provided by common programming languages or standard libraries.

There are many different kinds of containers, varying in the convenience or efficiency of certain operations. However, from a point of view of the iteration operation, containers divide in two classes, as observed by Dilling et al [3]:

- position dependent containers where a position inside the container has a well-defined meaning and the iteration over its elements is done in a pre-defined order; such containers are array, vector, list, stack, and queue containers;
- value dependent containers where the iteration order may be undefined, for example set, multiset and map.

This classification captures the kind of properties could be specified on a container. Therefore, the position dependent properties like sorting for the first class, i.e., properties relating the values stored at different positions of the container:

$$\forall i_1, i_2 \in \texttt{positions}(c).\ i_1 \leq i_2 \Rightarrow \texttt{get}(c, i_1) \leq \texttt{get}(c, i_2),$$

where i_1, i_2 are positions and c is the container for which accessing the value et some position is done using `get` function. For the value dependent containers, universal properties over values stored, e.g., all element greater than 5:

$$\forall i \in \texttt{positions}(c).\ \texttt{get}(c, i_1) \geq 5.$$

Another class of properties focus on relations between several containers, e.g., all values of the container c_1 are greater than the values stored by c_2:

$$\forall i_1 \in \texttt{positions}(c_1), i_2 \in \texttt{positions}(c_2).\ \texttt{get}(c_1, i_1) \geq \texttt{get}(c_2, i_2).$$

The verification of programs using containers, called client programs in the following, could require to deal with such constraints. Notice that, for position dependent containers, the class of properties as above involving positions related by complex constraints may not have a decidable satisfiability or entailment problems. A static analyser manipulating such constraints should either work on a decidable class of constraints or propose sound procedure for the decision problems required by the analysis.

In [3], Dillig et al proposes a static analysis based on a domain of indexed constraints to automatically infer properties on position or value dependent containers. Their domain has been implemented in the tool Compass and applied to the analysis of client programs using `C++` libraries. The experimental results obtained demonstrate the good precision of the analysis proposed: more than 50% of false positives have been eliminated.

Another approach to represent constraints over containers has been proposed in [6] and it considers higher-order functional programs. It uses the Liquid Types framework [7] to reduce the safety of a functional program to the safety of a first-order imperative program. In this way, the safety of the functional program can be checked using any program analysis for imperative programs introduced in the literature.

2 Principles of Static Analyses

In general, a static analysis is built on (i) an abstract representation of sets of program configurations, where the set of values of this abstraction is called an abstract domain, and (ii) an encoding of the program statements in terms of abstract values transformations, also called abstract transformers.

We fix the set of operators a client program may call on a container depending on its kind in Tables 1 and 2, where the intuitive semantics of operations is provided. Next sections of this report define, for each class of containers, an abstract domain and its abstract transformers for the fixed operators.

3 Abstract Domain for Sequences

Sequences are natural abstractions for unbounded, position dependent containers like vectors and lists. The abstract domain we propose is inspired from our previous work on the analysis of programs manipulating dynamic linked lists [1], where sequences where used as model for the heap content. The main difference with this previous work is the abstract transformers required on the sequence abstract domain. For programs using a low level manipulation of the heap, the operations on the sequence model where: split the sequence in a head value and the remaining of the sequence, add a value at the end of the sequence, concatenate two sequences. For the programs manipulation position dependent containers, we have to implement the operations in the table above including the ones concerning positions, which is a novelty w.r.t. the previous usage of this domain.

Table 1: Operations provided by position dependent containers

`init(c)`	create d the empty container
`is_empty(c)`	test for empty container
`add_first(c,v)`	push a value before the first position
`add_last(c,v)`	push a value after the last position
`remove_first(c)`	remove the value at the first position
`remove_last(c)`	remove the value at the last position
`first_pos(c)`	return the first position or -1
`next_pos(c,p)`	return the next position or -1
`last_pos(c)`	return the last position or -1
`get_at(c,p)`	return the value at a valid positon or nil
`isin(c,v)`	return the position of the value in the container or nil

Table 2: Operations provided by value dependent containers

`init(c)`	create d the empty container
`is_empty(c)`	test for empty container
`add(c,v)`	push a value
`remove(c,v)`	remove a value
`isin(c,v)`	return true iff the value is in the container

Abstract values: A value $a^\sharp$ of the sequence abstract domain $\mathcal{A}_\mathbb{S}$ is a constraint over the following sets of typed variables:

- a set $\mathcal{C}$ of position dependent containers,
- a set $\mathcal{V}$ of integer variables,
- a set $\mathcal{P}$ of positions variables (although in most of languages positions are usually integers, we keep them separate to identify easily valid positions for a container), and
- a set $\mathcal{L}_\mathcal{C}$ of integer variables, one for each container variable in $\mathcal{C}$, representing the size of the container.

The syntax of an abstract value $a^\sharp$ has the following form:

$$\big(A \wedge E \wedge \bigwedge_{G(\vec{y})\in\mathcal{G}} \forall\vec{y}.\ G(\vec{y}) \Rightarrow U(\vec{y})\big),\ \text{where}$$

- A is a value of an abstract domain for equality, used to track aliasing between containers. It is called the *aliasing part* of the abstract value.
- E is a value of a numerical abstract domain $\mathcal{A}_\mathbb{Z}$ (such as the Octagons abstract domain, the Polyhedra abstract domain [5], etc.) constraining the set of variables $\mathcal{V} \cup \mathcal{P} \cup \mathcal{L}_\mathcal{C}$. It is called the *existential part* of the abstract value.

- $\vec{y}$ is a set of fresh *position variables* interpreted as integers representing positions in the sequences,
- $\mathcal{G}$ is a set of guards $G(\vec{y})$, which belong to a specific set of patterns. From our experience with the analysis of client programs, we include in this set the following patterns (for every variable in $\mathcal{C}$):
  - $y \in \texttt{positions}(s)$ which corresponds to a universal quantification over all the values in the sequence s,
  - $y_1 \in \texttt{positions}(s_1) \wedge y_2 \in \texttt{positions}(s_2)$ which allows to compare the values of two containers,
  - $y_1 \leq y_2 \in \texttt{positions}(s)$ which allows to express sorting properties, and
  - $y_1 \in \texttt{positions}(s_1) \wedge y_2 \in \texttt{positions}(s_2) \wedge y_1 = y_2$ which allows to compare, position by position, the values of two sequences.
- $U(\vec{y})$ is a value in a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ constraining the set of variables $\texttt{get}(s, y)$, denoting the integer at position y in the sequence s, $\texttt{len}(s)$, and $\texttt{get}(s, 0)$. A term $\texttt{get}(s, y)$ appears in $U(\vec{y})$ only if the guard $G(\vec{y})$ contains a constraint $\vec{y} \in \texttt{positions}(s)$. This restriction is used to avoid undefined terms. For instance, if s denotes a sequence of length 2 then the term $\texttt{get}(s, y)$ with y interpreted as 3 is undefined.

For example, the following abstract value represent configurations of a position dependent container c which has length less than ℓ and it is sorted:

$$\texttt{len}(c) < \ell \wedge \forall y_1 \leq y_2 \in \texttt{positions}(c) \Rightarrow \texttt{get}(c, y_1) \leq \texttt{get}(c, y_2)$$

Abstract transformers: To define the abstract transformers implementing the container operations, we use the basic operations described in [4, 2] for the domain of sequences:

- lattice operations: built top $\top$ and bottom $\perp$ abstract values, test inclusion of abstract values $a_1^\sharp \sqsubseteq a_2^\sharp$; the last operation is a sound over-approximation of the entailment between the constraints presented in [2],
- widening of two abstract values $a_1^\sharp \nabla a_2^\sharp$,
- in an abstract value $a^\sharp$, split a sequence s after the position p to obtain new sequences s_h and s_t, $\texttt{split}(a^\sharp, s, p, s_h, s_t)$,
- in an abstract value $a^\sharp$, fold a sequence of sequence variables $[s_1, \ldots, s_n]$ into a sequence variable s (for existential quantifier elimination), $\texttt{fold}(a^\sharp, [s_1, \ldots, s_n], s)$.

We also define the following new basic operations:

- built a sequence s of one value v, $\texttt{newunit}(a^\sharp, s, v)$.

Using the above basic operations, the abstract transformers for operations on containers are defined as follows:

- `init(c)` introduces a new constraint for the container assigned in the existential part: $\texttt{len}(c) = 0$,

- `is_empty(c)` intersects the abstract value with the constraint $\texttt{len}(c) = 0$ and returns true iff the value obtained is not empty (i.e., the constraint is still satisfiable).
- `add_first(c,v)` applies first $\texttt{newunit}(a^\sharp, s, v)$ to obtain $a_1^\sharp$, then applies folding $\texttt{fold}(a_1^\sharp, [s, c], c)$.
- `add_last(c,v)` applies first $\texttt{newunit}(a^\sharp, s, v)$ to obtain $a_1^\sharp$, then applies folding $\texttt{fold}(a_1^\sharp, [c, s], c)$.
- `remove_first(c)` applies splitting at position 0, $\texttt{split}(a^\sharp, c, 0, c_h, c)$, and the value of c_h is bound to the result of the operation.
- `remove_last(c)` applies splitting at position $\texttt{len}(c) - 2$, $\texttt{split}(a^\sharp, c, \texttt{len}(c) - 2, c, c_l)$, and the value of c_l is bound to the result of the operation.
- `first_pos(c)` applies emptiness test and if it returns false then returns 0.
- `next_pos(c,p)` intersects with the constraint $\texttt{len}(c) > p+1$ and is the obtained abstract value is not empty, sets the results to $p + 1$; otherwise returns -1.
- `last_pos(c)` applies emptiness test and if it returns false then returns $\texttt{len}(c) - 1$.
- `get_at(c,p)` intersects with the constraint $\texttt{len}(c) > p >= 0$ and if it returns an non empty abstract value, then applies $\texttt{split}(a^\sharp, c, p-1, c_h, c_t)$, and constrains the the result to be equal to $\texttt{get}(c_t, 0)$.
- `isin(c,v)` generates a set of four possible abstract values depending on the position of the value v: at the first resp. last position, in the middle of c or never present in c. Notice that the abstract transformer for this operation is not very precise because we can not represent inside the abstract values negative assertions.

4 Abstract Domain for Multisets

Multisets are natural abstractions for unbounded, value dependent containers like bags and even sets. The abstract domain we propose is inspired from our previous work on the analysis of programs manipulating dynamic linked lists [1], where multisets where used as an abstraction of the heap content. We extend this previous work by adding ordering constraints between multisets of integers, due to the new procedure we proposed (see the first section) on multisets. The abstract transformers proposed in [1] are enough for the abstract transformers required in programs over containers.

Abstract values: A value $a^\sharp$ of the multiset abstract domain $\mathcal{A}_{\mathbb{M}}$ is a constraint over the following sets of typed variables:

- a set $\mathcal{C}$ of position dependent containers,
- a set $\mathcal{V}$ of integer variables,
- a set $\mathcal{P}$ of positions variables (although in most of languages positions are usually integers, we keep them separate to relate easily valid positions),

- a set $\mathcal{L}_{\mathcal{C}}$ of integer variables, one for each container variable in $\mathcal{C}$, that represent the size of the container, and
- a set $\mathcal{M}_{\mathcal{C}}$ of multiset variables, one for each container variable in $\mathcal{C}$, that represent the size of the container, and

The constraints stored have the following for:

$$A \wedge E \wedge \bigwedge_{c \in \mathcal{C}} \forall y \in \texttt{positions}(c).\ U(\vec{y}) \quad \wedge \quad \big(\bigwedge_{i} t_1^i \# t_2^i\big) \text{ where}$$

- A is a value of an abstract domain for equality, used to track aliasing between containers. It is called the *aliasing part* of the abstract value.
- E is a value of a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ (such as the Octagons abstract domain, the Polyhedra abstract domain [5], etc.) constraining the set of variables $\mathcal{V} \cup \mathcal{P} \cup \mathcal{L}_{\mathcal{C}}$. It is called the *existential part* of the abstract value.
- y is a set of fresh *position variable* interpreted as an integer representing position in the multiset,
- $U(\vec{y})$ is a value in a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ constraining the set of variables $\texttt{get}(c, y)$, denoting the integer at position y in the container c and $\texttt{len}(c)$. A term $\texttt{get}(c, y)$ appears in $U(\vec{y})$ only if $y \in \texttt{positions}(c)$. This restriction is used to avoid undefined terms.
- t_1^i, t_2^i are multiset terms of the form $u_1 \cup \cdots \cup u_n$ ($n \geq 1$ and $\cup$ is the union of multisets) where basic terms u_i are of the form (1) d representing the singleton containing the value of d or the empty multiset $\emptyset$, or (2) $\texttt{ms}_c$ representing the multiset containing all the integers of the container c. The operator $\#$ belongs to $\{=, \leq\}$. The comparison $\texttt{ms}_{c_1} \leq \texttt{ms}_{c_2}$ means that all elements of c_1 are less or equal to the elements of c_2.

Abstract transformers: To define the abstract transformers implementing the container operations, we use the basic operations described in [4, 2] for the domain of multisets:

- `init(c)` introduces the new constraints $\texttt{len}(c) = 0$ and $\texttt{ms}_c = \emptyset$.
- `is_empty(c)` intersects the abstract value with the constraints $\texttt{len}(c) = 0$ and $\texttt{ms}_c = \emptyset$ and returns true iff the value obtained is not empty (i.e., the constraint is still satisfiable)
- `add(c,v)` substitutes c by a fresh variable c' even in terms $\texttt{len}(c)$ and $\texttt{ms}_c$, then adds the constraints $\texttt{len}(c) = \texttt{len}(c') + 1$ and $\texttt{ms}_c = \texttt{ms}_{c'} + v$. The variable c' is then eliminated by the existential variable elimination.
- `remove(c,v)` first tests the emptiness and if false, it substitutes c by a fresh variable c' even in terms $\texttt{len}(c)$and $\texttt{ms}_c$, then adds the constraints $\texttt{len}(c) = \texttt{len}(c') - 1$ and $\texttt{ms}_c + v = \texttt{ms}_{c'}$. The variable c' is then eliminated by the existential variable elimination.
- `isin(c,v)` generates two cases: (1) for v not in c using the constraint $\forall y \in \texttt{positions}(c).\ v > \texttt{get}(c, y) \vee v < \texttt{get}(c, y)$ and (2) for v in c using the constraint $\texttt{ms}_c = v + \texttt{ms}_{c'}$ with c' a fresh variable, and $\texttt{len}(c) \geq 1$.

References

[1] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.

[2] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, volume 7148 of *LNCS*, pages 1–22. Springer, 2012.

[3] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. *SIGPLAN Not.*, 46(1):187–200, Jan. 2011.

[4] C. Dragoi. *Automated verification of heap-manipulating programs with infinite data*. PhD thesis, Paris Diderot – Paris 7, 2011.

[5] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV'2009*, volume 5643 of *LNCS*, pages 661–667, 2009. http://apron.cri.ensmp.fr/library/.

[6] R. Jhala, R. Majumdar, and A. Rybalchenko. *HMC: Verifying Functional Programs Using Abstract Interpreters*, pages 470–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.