RnD report

Abstract Interpretation and Abstract Domains

Guide: Prof. Supratik Chakraborty

Eashan Gupta - 160050045

May 2019

1 Introduction

The aim of this project was to learn about various abstract domains used for abstract interpretation and to implement them to be integrated with the CAnalyzer tool. Some basic introduction and implementation summary regarding the abstract domains implemented is recorded in this report.

1.1 Background

Abstract interpretation is used to observe important properties of a program. Different domains can be used to observe different such properties. These domains store the state of a program as we parse the program and we can summarize the program execution by managing the domain.

For implementation, for each domain we require to store an *abstractvalue* and a *stackvalue*. The *abstractvalue* stores the overall state of the program till now and *stackvalue* is a value stored in between operations to facilitate computation. It may be seen as an output of an operation on a single variable.

2 Congruence Domain

The congruence domain is a non-relational domain. This domain stores the properties of all the variables independently and follows the property that "variable x always satisfies n modulo m". So we store n and m for all the variables and are able to make comments depending on these.

2.1 Implementation Summary

Structures used:

1. *stackvalue* : `std::tuple<int,int,int>`

2. *abstractvalue* : `std::map<std::string,std::tuple<int,int,int>>`

The structure used stores three values b, c and a for each variable x. The utility for this was the earlier aim of implementing the domain which was to maintain:

$$x \,\texttt{mod}\, a \in [b, c]$$

$$0 < b < a, b \in \mathbf{Z}$$

$$0 < c < a, c \in \mathbf{Z}$$

But due to difficulty of expressing in this form it was reduced to two values such that c and b are the same in the implementation.

$$x = a\mathbf{Z} + b$$

2.2 Post Operations

The various post operations done follow from [2]. The approximations are done as follows:

- $\texttt{var}\, x = \texttt{int}\, num$
  In this case, $a = 0$ and $b = num$ where num is an integer.

- $\texttt{var}\, x = d \pm d'$
  If
  $$d = a\mathbf{Z} + b \,\texttt{and}\, d' = a'\mathbf{Z} + b'$$
  So,
  $$x = gcd(a, a')\mathbf{Z} \pm min(b, b')$$

- $\texttt{var}\, x = d \times d'$
  $$x = gcd(aa', ab', a'b)\mathbf{Z} + bb'$$

- $\texttt{var}\, x = d/d'$
  Best approximation in general is
  $$x = \mathbf{Z}$$

2.3 Lattice Operations

The lattice operations also follow from [2].

- $d \sqcup d'$

$$a_{res} = gcd(a, a', \|b - b'\|)$$
$$b_{res} = min(b, b')$$

- $d \sqcap d'$
  If $b \equiv b' \bmod gcd(a, a')$

$$a_{res} = lcm(a, a')$$
$$b_{res} = b$$

  else it is $\bot$

All these are followed from [2] and implemented accordingly. The values are stored for a and b in the `tuple` and retrieved from the `map` whenever required. For each operation done in the program, the values are updated or stored according to the operation rules described above.

2.4 Concerns

- The implemenetation uses a 3-tuple and this can be used if required to change to original implementation aim which required a, b and c.
- The widen operator maybe changed as currently join and widen do the same thing. This is because join already over-approximates the values.

3 Array Domain

The array domain is required to simulate arrays efficiently. Ideally, we could refer to an array by representing it with n number of separate variables, n being the size of the array. But [3] models it more efficiently by segmenting an array based on index and then representing the values stored in another separate abstract domain. This helps in efficient computation, in case we have large arrays and the changes done are linear on index, as we have fewer variables (segments) to keep track of.

This is a relational domain as the variables and arrays may be interrelated.

3.1 Implementation Summary

The Arrays are represented using separate structures. Each segment's values are stored in a separate abstract domain, interval domain in this implementation. The segment bounds are stored in the form of expressions which are easy to use and the structure directly used from the Z3 API [1]. The interval domain is directly used form the previous implementation which uses `apron` API. The structures are as follows:

1. *stackvalue* : for interval domain from API
2. *abstractvalue* : for interval domain from API
3. Array

```
struct Array{
    std::vector<std::set<expression>> expressions_v;
    std::vector<std::string> val_v;
};
```

Each array is represented by a vector of segment bounds, each represented by a set of expressions. Each segment is represented by a hidden variable whose name is store in the vector `val_v` and can be retrieved from the interval domain. An array is represented as $A : B_0P_0B_1P_1 \dots P_{n-1}B_n$ where B_i is the i^{th} boundary set of expressions and P_j is the j^{th} value predicate for the array A. eg:

$$A : \{0\} \top \{i\} \bot \{n\}$$

$$B : \{0, k\} [0, 3] \{(i-1), (j+2)\} \top \{n\}$$

4. *stackvalue* for aray domain:

```
struct StackVal_p{
    expression *e;
    stack_value_interval_domain* s;
};
```

For the implementation, since we do not have the exact expressions directly from the parser, we rebuild them according to the operations required and these are stored in e in the *stackvalue*.

5. *abstractvalue* for array domain:

```
struct Abstract{
    std::set<std::string> list_of_variables;
    std::map<std::string,Array*> list_of_arrays;
    abstract_value_interval_domain* inter;
};
```

The abstract value stores all the variables in the `list_of_variables` and their values are stored in the `abstract_value_interval_domain inter`. The arrays are stored in the `map` with keys as the name of array.

3.2 Basic Idea

The basic idea behind the implementation is that an array is divided into segments. The segment boundaries are stored in the form of expressions in terms of variables as assigned during runtime. The information regarding the variables used in these expressions is also stored. Any operation on these variables affects all the expressions. Any operation on the array requires context-free comparison of expressions. The context-free comparison of expression implies comparing without following the context. eg:
For abstract domain $A1$:

$$e1 = i + 1, i \in [1, 3]$$

For abstract domain $A2$:

$$e2 = i - 1, i \in [6, 7]$$

Now compare $e1$ and $e2$. If we follow a comparison with context, we have $e1 < e2$ but if we compare without context, as

$$i + 1 > i - 1$$

So $e1 > e2$ follows. This is useful in the cases where say i is a segment boundary and we wish to access array element at position $i - 1$ later, so we require context-free comparison.

For a detailed example on the unreeling of expressions, refer to Section 4.3 of [3].

3.3 Post Operations

1. **Create new array:**
   Create a new array $A[10]$. So it is initialised as:

   $$A : \{0\} \top \{10\}$$

2. **Assigning values to array elements:**

   $$A[b] = x$$

   $$A : B_0 P_0 B_1 P_1 \ldots P_{n-1} B_n$$

   Iterate over all expressions in B_i over all i to check where does the expression b fit and add a new segment between $\{b\}$ and $\{(b + 1)\}$ and assign this segment the value form $x \to s$ (note: form structure of *stackvalue_p*). eg:

   $$A : \{0\} \top \{i\} \perp \{n\}$$

   $$A[i - 2] = 4$$

   So

   $$A : \{0\} \top \{(i - 2)\} [4, 4] \{(i - 1)\} \top \{i\} \perp \{n\}$$

   and similar examples. There are some particular cases such as b or $(b + 1)$ coincides with bounds which can be taken care of.

3. **Get value of array element:**
   Get value of $A[b]$. So iterate over all expressions in B_i over all i to check in which segment b lies and take the predicate value from there.

4. **Assigning values to variables:**

$$x = f(y_0, y_1 \dots y_n)$$

Take inverse of the function w.r.t. y_i for all i,ie

$$y_i = f^{-1}(x, y_0, y_1 \dots y_{i-1}, y_{i+1} \dots y_n)$$

and replace it in all the expressions which contain y_i and add these new expressions to corresponding bounds. Then ignore previous expressions of x as it has a new value and those expressions are now useless. eg:

$$A: \{0\} \top \{i\} \perp \{n\}$$

$$B: \{0, k\}\,[0, 3]\,\{(i-1), (j+2)\} \top \{n\}$$

now,

$$j = i + 33$$

Take inverse so that

$$i = j - 33$$

replace and ignore previous expressions of j

$$A: \{0\} \top \{i, (j-33)\} \perp \{n\}$$

$$B: \{0, k\}\,[0, 3]\,\{(i-1), ((j-33)-1)\} \top \{n\}$$

3.4 Lattice Operations:

3.4.1 Segmentation Unification Algorithm

To join/meet/widen two instances of an array, this algorithm is used. Its purpose is to segment the whole array based on the two instances such that the segments are made as small as possible. Then we do the join/meet/widen operation on these segments respectively.

$$Abstractdomain1 : A : B_0 P_0 B_1 P_1 \dots P_{n-1} B_n$$

$$Abstractdomain2 : A : B'_0 P'_0 B'_1 P'_1 \dots P'_{n'-1} B'_{n'}$$

So now we tend to create a new segmentation such that it follows from the above two. For the detailed algorithm, refer to Section 11.4 of [3].

The main idea behind the algorithm is to follow recursively ignoring the predicates, we only compare the bounds.

1. If $B_i < B'_j$, then we place a bound B_i for new array A and move on to comparing B_{i+1} and B'_j, and vice-versa.

2. if $B_i \cap B'_j \neq \emptyset$ then put a bound with $B_i \cap B'_j$ as the new bound. Rest of the expressions are compared next to get the order. Note that the predicate value between this intersection for both the arrays depends on the lattice operation(for join/widen: $(\bot)$, meet: $(\top)$).

3. If B_i and B_j cannot be compared, we ignore both of them, and club the segments from the previous to the next ie $B_{i-1}P_{i-1} \sqcup P_iB_{i+1}$ and $B'_{j-1}P'_{j-1} \sqcup P'_jB'_{j+1}$.

So this algorithm gives a segmented array. We can get values of all the segments of the two arrays and unify them based on the operation. eg:

$$0 \ldots 10 \ldots 20$$

$$0 \ldots 5 \ldots 15 \ldots 20$$

Unifying:

$$0 \ldots 5 \ldots 10 \ldots 15 \ldots 20$$

Next we get values of all segments from the two domains and correspondingly unify them. Another example:

$$\{0, i\}\top\{n\} \texttt{ and } \{0, (i-1)\}0\{1, i\}\top\{n\}$$

join in steps:

$$\{0\}\bot\{i\}\top\{n\} \texttt{ and } \{0\}0\{(i-1)\}0\{1, i\}\top\{n\} \texttt{ so bound } \{0\}$$

$$\{i\}\top\{n\} \texttt{ and } \{(i-1)\}0\{1, i\}\top\{n\} \texttt{ so bound } \{(i-1)\}$$

$$\{i\}\top\{n\} \texttt{ and } \{1, i\}\top\{n\} \texttt{ so bound } \{i\}$$

$$\{n\} \texttt{ and } \{n\}$$

So

$$\{0\}\bot\{i-1\}\bot\{i\}\top\{n\} \texttt{ and } \{0\}0\{(i-1)\}0\{i\}\top\{n\}$$

We can also merge the segments which have same values. So removing unnecessary bounds

$$\{0\}\bot\{i\}\top\{n\} \texttt{ and } \{0\}0\{i\}\top\{n\}$$

Meet on corresponding bounds.

So, for calculating the operations join, meet and widen, segmentation unification has bee implemented which gives the two arrays with corresponding segments. Then taking unification gives the result. Also for rest of the variables, normal operations work and have been implemented so.

3.5 Concerns and Key Points regarding Implementation

1. The abstract domain used for segments in this case is interval domain. If required, the abstract domain can be separated as a template by generalizing all functions required for *abstractvalue* and *stackvalue*. Then the code can be used to simulate array domains with any underlying domain.

2. This implementation does not handle cases in which a variable value is obtained depending on an array element value and later that variable is used to bound the array. Such cases may give unexpected results. eg:

   ```
   i=A[3]+1;
   A[i]=5;
   ```

3. Merging of segments has not been implemented. It could be implemented to reduce the number of segments in case they increase drastically. For merging, first collapse the consecutive bounds whose segment values are same. Then while collapsing others, join the segment values of the previous two segments to get the segment value of the new segment. ie:

   $$B_{i-1}P_{i-1}B_iP_iB_{i+1}$$

   Collapse B_i to get

   $$B_{i-1}P_{i-1} \sqcup P_iB_{i+1}$$

4. This implementation does not take care of floating point numbers.

5. The paper [3] also refers to cases where we can further subdivide every segment over indices (eg. odd or even). This can also be added to the current implementation.

6. Some general helper functions have been defined in the class to facilitate the coding such as:

   - `test`: It takes as input two expressions and in case both are expressions of variables, it compares them without context or if there is a numeral, then it takes into account the context as well.
   - `contains`: To check if an expression is contained in another.
   - `check_equal`: To compare equality of two expressions.
   - `sequential_join`: It takes two arrays and returns the two arrays after proper segmentation so that they can be then reduced after joining (or meet/widen).

References

[1] Z3 theorem prover:
`https://github.com/Z3Prover/z3`.

[2] S. Bygde. *Abstract Interpretation and Abstract Domains with special attention to the congruence domain.* Master's thesis, Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden, 2006.

[3] R. C. . F. L. . P. Cousot. A parametric segmentation functor for fully automatic and scalable array content analysis. 2011.