Static Analyzer Project

Matthieu Boyer Antoine Groudiev

Last updated June 8, 2024

Introduction

This is the static analyzer project of Antoine Groudiev and Matthieu Boyer, for the class *Semantics and application to verfication*. It implements the base domains, that is the constant, sign, interval, congruence domains, the reduced product between the interval and the congruence domains ; furthermore, support for the Karr's linear equality analysis and Backward analysis have been added as extensions.

The provided analyzer should pass every single test included with the project. Specialized tests for each domain and feature have been added.

1 Iterator

As recommended, the iterator (in the file `iterator.ml`) implements a classic worklist algorithm. The Hashtbl `environment` maintains a map from nodes to abstract values. The main logic is contained inside `iterate_function`, which handles the worklist, and `iter_arc`, which applies the appropriate rule of the domain to the visited arc.

Widening is handled by applying initially a DFS to the CFG and marking nodes which might need widening. When this node is encoutered during the main worklist procedure, widening is applied if necessary. Note that the iterator never calls the `narrow` function.

2 Domains

2.1 Value Domain to Domain

A `VALUE_DOMAIN` is transformed into a `DOMAIN` using the `Domain` functor in `domain.ml`. It abstracts a set of mappings from variables to integers, abstracting sets of integers with the given `VALUE_DOMAIN`.

2.2 Constant domain

The constant domain abstracts a set of integers as:

```
type t = | Top | Bot | Int of Z.t
```

In most cases, operations such as `widen`, `meet` and `join` do not convey much information. Similarly, any non-trivial `rand` is abstracted as `Top`.

The functions `binary` and `bwd_binary` are slightly more complicated. For `binary`, multiplication with `Int Z.zero` is guaranteed to result in `Int Z.zero`. Values $\bot$ and $\top$ cases are absorbant. Finally, we can compute the operation if the value of both operands is known; in case of division by zero, the result is $\bot$, which will notably cause future assertion to pass, as seen in the example `constant/constant_div_zero.c`. For `bwd_binary`, the cases of $+$, $-$, $\times$ are easy to handle

since these operators are invertible. The case of division and modulo is harder, since the special case of zero needs to be extracted.

2.3 Sign domain

We chose the following representation for the sign domain elements:

```
type t = | Bot | Top | Pos | Neg | Null
```

A more complicated sign latice could have been chosen, separating for instance positive values into stricly positive, and positive or null.

Pretty much every function is straighforward. Division can zero can still be detected and abstracted as $\bot$, but there is no added difficulty compared to the constant domain.

2.4 Interval domain

To represent an interval, a bound element has been introduced:

```
type bound = | MinusInf | Finite of Z.t | PlusInf
```

which can later be used in the definition of an abstract type:

```
type t = | Bot | Top | Interval of bound * bound
```

Firstly, I later realized that I could have abstracted `Top` as `Interval (MinusInf, PlusInf)`, which would have reduced the length of the code. Nevertheless, $\top$ is in general fairly easy to handle, so it introduced no fundamental complexity.

Operations on `bound` have been introduced, to handle infinite cases. Note that in practice, for an analyzer, cases such as $+\infty - (+\infty)$ never happen, because substractions are only called with one left and one right bound. A function `bottomize_if_necessary` is used to transform `Interval` into `Bot` when the bounds are inverted.

For `binary`, only multiplication and division are tricky to handle. The main idea is to enumerate all possible cases and to take the most general one. Similarly, I found `bwd_binary` extremly difficult to handle in the case of multiplication. Cases where the interval contains zero need to be taken care of, we have to be careful to round intervals – otherwise multiple tests do not pass... It was not fun to code, and even less to debug.

2.5 Congruence domain

For this domain, a set is either of the form $\varnothing = \bot$ or $a\mathbb{Z} + b$. This is classically represented by:

```
type t = | Bot | Modulo of Z.t * Z.t
```

Note that $\top = \mathbb{Z} = 1 \times \mathbb{Z} + 0$, and that $\{b\} = 0 \times \mathbb{Z} + b$.

Most operations are easy to handle since little information can be deduced. This can be seen, for example, in `compare`, which results most of the time in $(\bot, \bot)$. Only the `meet` and `join` is interesting since one can sometimes use rafinments such as GCD to give a precise abstraction.

2.6 Reduced product

The reduced product domain maintains in parallel two abstract representations for an element:

```
type t = IntervalDomain.t * CongruencesDomain.t
```

Most operations simply apply the respective functions of both domains to each component of the pair. The only interesting cases are the ones of `widen` and `narrow`, where a `reduction` function use information from both domains to obtain a precise abstraction. This is done by implementing in `IntervalDomain` functions to convert intervals into congruences, and congruences into intervals.

2.7 Karr's linear equality

In this domain, we only check affine assignations or equalities, else we resort to non-deterministic assignment. This domain is not based on a value domain, since we only consider relational constraints between multiple variables (multiple can be only one, this might be seen as fixing the value of a variable). We represent the domain using the following type:

```
type t = | Bot | Constraints of (Q.t * Q.t array) array
```

where both arrays should be of length $n = |\texttt{Vars}|$. Indeed, if we have more than n constraints, then, either we can simplify the matrix and multiple constraints are equivalent, or the matrix represents the empty set since we consider an n-dimensional space for the values of the variables. `Constraints d` can be seen as a matrix of the coefficients in each affine constraint and a vector of each constant in the affine constraint. For each row i, we then have

$$\sum_{j=0}^{n-1}(\texttt{snd } d.(i)).(j)V_j = \texttt{fst } d.(i)$$

where V_j is the actual value of the j-th variable. In that case, the null matrix and the null vector are used to represent the whole space, and will be called `Top`.
To actually compute operations, we use the Row-Echelon Normal Form of the matrix. We can compute this form for any matrix of any size, thus, intersection can be done by considering the concatenation of the two arrays, simplifying by gaussian elimination, and taking all the non-empty rows, or returning `Bot`. This however, poses problems in the implementation as, for still unknown reasons, this function is called multiple times on different inputs, and thus returns `Bot` or `Top` depending on the situation. We implement equality by checking coefficients in the row-echelon normal form, and then we can compute the inclusion by considering that $A \subseteq B \Leftrightarrow A \cap B = A$.

For the assignation of variables, we simplify the right side of the assignement to

$$\nu_k < -\sum_{i=0}^{n-1} \alpha_i \nu_i + \beta$$

where ν_i is the i-th variable. Then, depending on α_k we can substitute in our domain the new value of ν_k. The same thing goes for guarding, where we simplify the condition, and then compute the minimal domain in which it is valid and find if our domain is included in it.

The implementation proposed here is not working, for reasons that are behind my comprehension. I have spent hours coding and debugging this, and I still have no clue about what goes wrong... I would also like to mention that this project was mostly carried by Antoine and that my failure is mine alone.

3 Backward analysis

3.1 Presentation

When the backward analysis mode is enabled, the analyzer will process failing assertions in more detail. When it encounters a failing assertion, it will try to go back to find values for the variables which will trigger the assertion. If no environment do trigger the assertion, we can conclude that the assertion always passes, and continue the analysis. The idea is to apply the exact same iteration procedure but in reverse (in practice, changing for example `node_in` in `node_out`) starting from the assertion node using the environment which triggers the false assertion.

A `bwd_assign` function is implemented for every value-domain based domain, that is every domain except for Karr's one. It does exactly what is hinted by the subject, and its implementation is quite short and uses the `Domain` functor to write the code once for all value domains.

3.2 Example

We use the file `backward_abs.c` as an example:

```
void main(){
    int x = rand(-10, 10);
    int z = 0;
    int y = 0;
    if (x==0) {
        z = 0;
    } else {
        y = x;
        if (y < 0) { y = -y; }
        z = x / y;
    }
    assert(x > 0); //@OK
}
```

Figure 1: Program `backward_abs.c`

The forward analysis fails since in the `else` branch, `x` takes values into $[-10, 10] \setminus \{0\}$, which is abstracted by the interval domain as $[-10, 10]$. Therefore, the limitations of our domain do not allow us to detect that $x \neq 0$.

The backward analysis will go up, starting from the assertion, and will identify that the assertions fails when $y = 0$. It will then deduce that it means that $x = 0$, which is impossible in this branch. Therefore, the backward analysis could not find any trace in which the assertion is false: therefore, the assertions passes.