Software Analysis and Verification 2007 Miniproject

Yuanjian Wang Zufferey Simon Blanchoud

June 30, 2007

Abstract

In this report, we will introduce the results of our project on the variable range analysis. We will first explain the necessity people have to analyze the range of variable and some works that have been done before. Then we will repeat some theories that we have used as support for our project. Later we explain how we implemented our variable range analysis program and some results we have got along with some limits we have met. We will talk about the improvement in the future in the last section.

1 Introduction

Almost all the papers that talk about the system dependence will mention the failure of the Ariane 5 launcher in 1996 because of an overflow bug. Every programmer always keep in mind not to perform a division by zero but we can never be sure it will not happen. In the most cited example for bank account, people have to check that the operations will not violate the invariants. To avoid infinite loop in program, we have to guarantee that the stop condition will be reached. Program compilers always have to check the correctness of the index value or of the right pointers.

All the examples above show us that if we could check the range of variables before the execution, time, money and even lives could be spared as we could find the problem earlier

There exist several variable range analysis works that have been done before :

- The most well-known is the fixpoint approximation method by Cousot and Cousot [1] that evaluates the variable interval by increasing and descending approximation methods
- Karr´s domain that discovers affine equalities between variables.
- Cousot and Halbwachs polyhedron domain for affine inequalities
- Grangers congruence domain,
- And many more

In practice, for some programming languages, we can find some part of variable range analysis functions that are implemented directly in the compiler. For example, Java has “Range Check Elimination” optimization since, at least, Java

1.3.1 in order to be able to check the array bounds. In C++, the implicit check (by declaration) and the explicit check (by writing explicit code to verify) is strongly recommended in the practice for the index of array. Overflow problem in both cases is not easy to trace. We can check the external inputs by using limited bounds but we have great difficulties to find out the overflow of the internal input.

In our project, we followed the fixpoints approximation methods and implemented the methods to calculate the range of multiple variables in sequent, conditional and loop situation. We used the increasing approximation process (widening) and descending approximation process (narrowing) methods to analyze the range of multiple variables.

2 Variable Range Analysis

2.1 Principle Method

We strictly followed the theory written in Cousot and Cousots paper [1]. We will introduce this paper and show the principle that we have used to guarantee the correctness of our project. In Cousot and Cousots paper [1], they introduced the lattice of abstract interpretations. Firstly, we need to show what the abstract interpretations and the corresponding lattice are. We take the formal definition of the paper [1]: An abstract interpretation I of a program P is a tuple :

$I =< A - Cont, \circ, \leq, \top, \bot >$

$A - Cont$ **:** Set of abstract contexts (Context: the set of values that a variable can take)

$\circ, \leq$ **:** a complete $\circ$-semi lattice with ordering $\leq$, for example $x \leq y \Leftrightarrow x \circ y = y$

$\top$ **:** the top of A-Cont

$\bot$ **:** the bottom of A-Cont.

The set of context vectors is defined by: $\underline{A - \widetilde{Cont}}$, and the function $\underline{Int}$ defines the interpretation of basic instructions: $\underline{Int} : Arcs^0 \times A - \widetilde{Cont} \rightarrow A - Cont$. Here $Arcs$ is the set of edges of $program$ that connect two nodes of $program$. Each node of $program$ represents either entry, one exit, one assignment, one test or one conjunction.

The local interpretation of elementary program constructs which is defined by $\underline{Int}$ is used to associate a system of equations with the program. It is defined in the paper as $\underline{\widetilde{Int}} : \underline{A - \widetilde{Cont}} \rightarrow \underline{A - \widetilde{Cont}}$. It is order-preserving and it has fixpoints.

We will not repeat the theory here but cite the examples that they have shown in Cousot and Cousots paper [1] as they are useful to understand the abstraction process.

Let P be a program with a single variable. The value of the variable is an integer. The set of values at some program points(i.e. its context), is denoted as S. S may be abstracted by an abstracting process α that returns a more abstracted closed interval: $\alpha(S) = [min(S), max(S)]$. When S is infinite, the bounds will be $-\infty$ or ∞.Conversely, the concretization process γ is defined as:

$\gamma([a,b]) = \{x | a \leq x \leq b\}$, will return a more concrete interval. The abstract process is shown in Fig.1 from (a) to (d) and the concretization process is the reverted process, that is from (d) to (a), on the same figure:

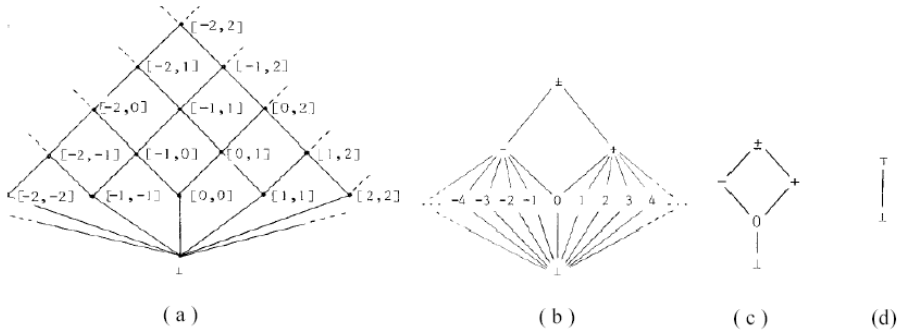

Figure 1: The abstraction and the concretization processes

Secondly, we will show the two fixpoint approximation methods they mentioned in the paper [1]: one is the finite iterative and increasing approximation of the least fixpoint starting from a lower bound and the other is a decreasing approximation sequence. To show how the two methods work, we will use the same example that was given in paper [1]. It calculates the range of variable x, the transformed program graph is shown in Fig.2. The analyzed code is the following :

```
x:=1;
while(x<=100)
{
  x:=x+1;
}
```

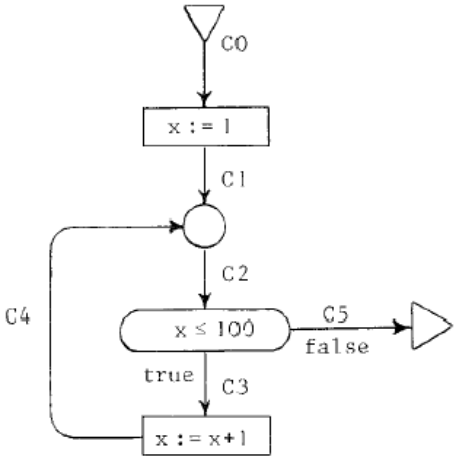

Figure 2: The graph corresponding to the example program

The resulting system of equations is the following :

$$C_0 = [,] \tag{1}$$
$$C_1 = [1,1] \tag{2}$$
$$C_2 = C_1 \cup C_4 \tag{3}$$
$$C_3 = C_2 \cap [-\infty, 100] \tag{4}$$
$$C_4 = C_3 + [1,1] \tag{5}$$
$$C_5 = C_2 \cap [101, +\infty] \tag{6}$$

In the increasing approximation process, they have used the widening method ∇ :

- $[,]$ is the null element of ∇
- $[i,j]\nabla[k,l] = [\underline{if}\ k < i\ \underline{then}\ -\infty\ \underline{else}\ i\ \underline{fi},\ \underline{if}\ l > j\ \underline{then}\ +\infty\ \underline{else}\ j\ \underline{fi}]$

The increasing approximation sequence needs to be placed on one of the loop arcs. In the paper´s example, they modified the system of equations as follow :

$$C_0 = [,] \tag{7}$$
$$C_1 = [1,1] \tag{8}$$
$$C_2 = C_2 \nabla (C_1 \cup C_4) \tag{9}$$
$$C_3 = C_2 \cap [-\infty, 100] \tag{10}$$
$$C_4 = C_3 + [1,1] \tag{11}$$
$$C_5 = C_2 \cap [101, +\infty] \tag{12}$$

The process starts from the lower bound of x, i.e. its bottom $[,]$, and uses the widening methods in order to find the fixpoint. The full convergence process can be found in the paper [1].

In the descending approximation process, the narrowing method $\triangle$ is defined:

- $[,]$ is the null element of $\triangle$
- $[i,j]\triangle[k,l] = [\underline{if}\ i = -\infty\ \underline{then}\ k\ \underline{else}\ min(i,k)\ \underline{fi},$
  $\underline{if}\ j = +\infty\ \underline{then}\ l\ \underline{else}\ max(j,l)\ \underline{fi}]$

In order to precise the result, they replace the ∇ operator by the $\triangle$ one and start converging from the fixpoint found by the previous method.

2.2 Implementation

The main idea of this project was to create something that could be used for real analysis. That is why we decided to try to implement this functionality in the Jahob[1]. As this program is implemented in O´Caml, we decided to use the same language for our implementation. Moreover, functional languages have great advantages for this kind of work. Unfortunately we did not achieved to interface our work with Jahob, nevertheless this could be done quite easily.

We decided to develop our program in an incremental way in order to ease its debugging and to ensure its correctness. The more important steps were :

[1] http://lara.epfl.ch/dokuwiki/doku.php?id=jahob_system

1. Develop a grammar that suits our needs
2. Produce the correct system of equations
3. Use the ∇ and $\triangle$ operators in order to find the fixpoints
4. Generalize for any number of variables

For the steps 2 and 3, we based ourselves on the paper´s example we showed in the previous section. All this work is based on a previous O´Caml code we wrote for the Homework 4 of this course.

The general organization of our program can be seen on Fig.3.

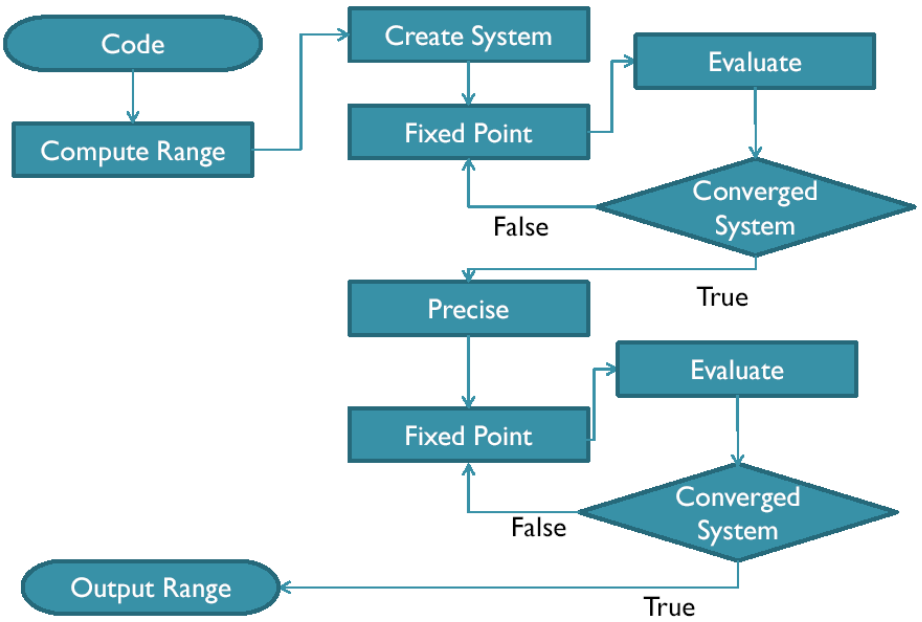

Figure 3: The general organization of our program

2.2.1 The Grammar

This is where we were able to reuse some code from our previous Homework : the structure of a grammar defining some programming language. We used a very simple grammar that allows :

- The `+`, `-`, `*` and `/` operators
- `<`, `>` and `=` tests for conditions
- A `while` loop and an `if-then-else` statement
- The use of arrays
- The call of procedures
- The creation of new objects

In order to be as general as possible, we used `float` variables for all of them. The restriction to `int` is obvious.

In order to be able to handle this code, we defined the grammar corresponding to the system of equations we will create. Designing correctly the basic entity of the system of equation was crucial as all our analysis works with this structure. A line of a system of equations is defined as the following tuple :

(c1, r1, v1, i1, eq1)

c1 A binary flag that help us reminding if this line has already converged

r1 The last computed range for this line, we compare it with the new one to know if we have converged

v1 The main variable of this line i.e. the variable for which we are computing the range

i1 The index of this line, each variable have independent indexes

eq1 The range equation of this line

On its side, a range can be :

- A real range, defined by `Is(t1, b1, b2, t2)` where `b1, b2` are the values of the bounds and `t1, t2` are boolean flags used to indicate if the bounds are included in the range or not
- A reference to another line, we can only reference to a line that has the same main variable as ours
- A reference to the range of another variable, occasionally we can indicate a particular line where to take the value from. This is useful when we have conditional branching in the code

For the equations, we needed to have the following operations available :

- $\cap$ and $\cup$ between ranges
- The `+`, `-`, `*` and `/` operators, in order to be able to reflect the actions of the code
- The ∇ and the $\triangle$ operators
- `<` and `>` because some times the main variable depends on the value of other variables we still do not know. So we keep the constraint like this until we know their value

2.2.2 The System of Equations

The creation of the system is defined recursively by the function `create_system` that returns a `system` of equations :

create_system (c : code)(s : system)(l : (string*int) list) : system

c The code we want to translate

s The system resulting from the translation of the preceding code

l The list of indexes where to point to. This list gives for each variable the value of the index where we should take its value. When the index is -1 this means that we should take the value of the closest line

The s variable is initialized by the function `create_init` which adds a line with index 0 for all the variables in the program. By adding these lines, we ensure to have at least one line where to reference to. The l is initialized such that it contains all the variables of the code with -1 for all of them. A `system` is simply a list of lines.

The basic idea of this function is simple, we create a new line that has not converged ($c = 0$), with an empty range, the variable we have just read, a new index (we wrote a function that looks in s to find this value) and its corresponding equation.

For most of the statements of our programming language, this translation is quite straight-forward so we will not discuss them. We will instead focus on the more interesting one :

- new objects, arrays and procedure calls are ignored directly at this step. Adding these functionalities in our project would simply mean finding a way to translate these instructions into the structure we have just discussed
- `if-then-else`
  1. Transform the condition into a suitable formula
  2. Compute its negation
  3. Get the next index value
  4. Translate the formula into a range equation
  5. Add this line to s
  6. Translate the code of the first part
  7. Get the next index value
  8. Add the negation equation to our new s
  9. Translate the code of the second part
  10. Unify both paths in one equation and add it to s
- `while` the idea is the same as for `if-then-else`. For more precise information, read directly our implementation code.

As we can see, the two main issues in creating the system are :

1. Transforming the formula so that we can use it
2. Creating the links correctly

The second part was quite easy to do as soon as we had implemented all the helping functions we needed (like functions for parsing lists, retreiving indexes, equations and so on, we have more than 30 of them so we will not present them here). The first point was a bit more complex, even if for only one variable it is fairly easy. Our idea was simply to isolate the main variable on one side

of the comparison operator so that we can simply remove it and translate the other side into an equation. In fact that is what we did and it works fine. The only problem we found, and that we did not succeed to solve, is how to handle > and < when we have to perform a multiplication or a division. The problem comes from the fact that when we multiply/divide one of these operators with a negative number we must invert them, but as we are using range, it is possible that we have both negative and positive numbers. For now on we have decided not to modify them.

2.2.3 The Convergence

This part is maybe one of the easiest one in our program, once again as soon as we have implemented the helping functions (around 25). We had to rewrite all the operations over both the ranges and the bound as both are not only `float` values.

The general algorithm of this part, performed by the function `comput_range`, is the following :

1. Evaluate each line of the system
2. If all the lines have converged, goto 3. otherwise goto 1.
3. Replace ∇ by $\triangle$
4. Evaluate each line of the system
5. If all the lines have converged, end otherwise goto 4.

A line has converged if and only if the value of its equation is the same as the range of the line, this means that we have computed two times the same results in two different iterations of the fixpoint mathod, and if all its predecessors have also converged. Both conditions are logical as one means that we are stable and the other one that our basis is stable.

At each run we reevaluate all the lines as some may have entered a state of pseudo-stability in which they are stable, according to the previous definition, but they may come back to an unstable state because of the change of another variable.

In case of loops in the program, we will also have a loop in the references between the different lines. This can create a deadlock in the stability check which can lead the program into an infinite loop. In order to avoid this, the stability of a reference which is located after us in program execution order is not taken into account.

2.2.4 Multi-Variables

At first sight, modifying our program so that it could handle more than one variable looked trivial. After spending various hours on the problem, we realized that it was not.

Our main inspiration for this part comes from [3]. What we did, as they suggested, was to create one equation for each variable that takes part in the condition. This means that we isolate each variable of the condition on one side one at the time. In order not to cycle forever and produce more than once each

equation, we have decided to pass trough a transition state. IN this state, we isolate all the variables of the equation on the left side of the comparison. After that it is quite easy to take out alternatively all the variables, this process is performed by the `produce_all` function.

One problem that arise here is how to handle conjuntions and disjunctions of formulas. With only one variable we can include it in the equation as we know that both main variables are the same, but know we have no idea. Of course we can think of performing tests on the main values in order to know if we can keep it, but then you can think of a more complicated condition for which you test would probably not work. As we are using intervals, we cannot keep easily the relation between the variables and here is where we lose a lot of information for the final result.

We decided, in order to be as simple as possible, to split the formulas whenever they avec either an `And` or an `Or`. As we lost some information, we must stay more general than the real formula in order to be sure to include at least its case. That is why we regroup them afterwards using `Or` only, as it is weaker than the `And`. We say that we regroup them because, as we want only one line per variable and because the splitting process may have produced more than one formula with the same main variable, we regroup the equations that have an identical one.

Another problem that may arise if we implement only this part comes from wrong references. In case you have variables inside a block that is not part of the condition, and that is often the case in real code, then our index searching procedure will get out from the block and search into the wrong piece of code, think of the second block of a `if-then-else` statement. In order to avoid that, we use the l variable to specify the right line to which these "free" variables must refer to. Using this technique, we are sure that they will always point to the good value without adding extra equations.

Using all these modifications, our program can successfully analyze the range of various variables in the same program.

2.3 Discussion

The most surprising result of this project was its size. At first sight this project looked a lot easier than it is. We were expecting to be able to implement more functionalities to this project but the time we spent on the basic problems prevented us from doing so. The biggest deception due to this is that we were not able to interface it with Jahob. Because of that, our project will probably stay purely theoretical.

Using linear ranges for the variables was chosen as it is the simplest method to compute this kind of information. Nevertheless, we can easily see that with this simple technique we are loosing a lot of information. This results is, in most complex cases, a very vague approximation of the final values. This is clearly the main limitation of this work as we want to be as precise as possible in order to avoid as many false positive as possible. Improving this well cannot be done by improving any part of our program and this implies that we would need to modify our approach. Other techniques, such as the octagon abstract domain [5], or the polyhedral domain, would allow us to do that but this implies a lot heavier methodology.

Separating the clauses of the formulas makes us loose a lot of information too (sometimes from a perfect answer to an infinity-bounded approximation). Improving this part would also allow us to improve our precision in a lot easier way than changing our abstract domain.

Having only one loop and one conditional branching statement is not a limitation as all the other kind of conditional structures can easily be translated into these two ones.

Except these limitations, our implementation is well scalable. It is not limited neither by the size of the code nor by the number of variables and even not by the number of formulas. There is no explosion in the number of equations, it is equal to the number of assigned variables in the code plus the number of variables used in the formulas. Knowing the results people often gets when analyzing software, this is a very good results.

3 Conclusion

In this project, we used the Cousot and Cousot[1] increasing and descending approximation methods in order to realize the analysis of multiple variables range. Actually, in this paper we did not get the precise explanation of how to analyze more than one variable at the same time.

Our project could be improved mainly in one aspect : its precision. Using this range technique on a real-scale program would probably lead to too approximated results. Using another abstraction domain is probably the solution. In [3] another approach is explained in order to improve our results without modifying our abstract domain : discovering the dependencies between the variables in order to deduce from it an approximation order. We did not explore this possibility but this could be an easier solution than the octagon one

Our project may also be extended in several directions :

- Implementing the missing parts about the procedure call, the arrays, the objects and even function calls would certainly be very useful in order to be able to analyze real code
- By adding the context of a program, the analysis of global variables and local variables would become possible
- Analyzing not only the numerical variables but other types of variables could be interesting even if they may need it less
- Interfacing it with Jahob would allow us to analyze real code without having to implement a file parser ourselves in addition to the benefits both applications would obtain

This project was very interesting as we had to build it totally from zero, using only the literature we found. Even if we may be a bit disappointed by the final results it is clearly a good basis from which we could start in order to create a more usable tool.

Our big hope about our program is that all these improvements and extensions will once be implemented and that one day we people could use it to avoid some nasty bugs.

References

[1] Patrick Cousot and Radhia Cousot: *Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints*, ACM Press, New York, 1977

[2] Patrick Cousot and Radhia Cousot: *Static Verification of Dynamic Type Properties of Variables*, Research Report, University of Grenoble, 1975

[3] Johnnie Birch, Robert van Engelen and Kyle Gallivan: *Value Range Analysis of Conditionally Updated Variables and Pointers*, CPC, 2004

[4] Yury Markovskiy : *Range Analysis with Abstract Interpretation*, Semester Project, CS 263, 2002

[5] Antoin Miné : *The octagon abstract domain*, Kluwer Academic Publishers, Hingham, USA, 2006