Octagon Abstract Interpreter

Jérôme Boillot
EPFL

Orégane Desrentes
EPFL

Siang-Yun Lee
EPFL

Dewmini Sudara Marakkalage
EPFL

Abstract

We present an implementation of a static analyser for C language using abstract interpretation with octagons [5] as the abstract domain. For simplicity, we only consider C functions with integer variables and maintain the program state at each execution point as a set of octagonal constrains on program variables. An octagonal constrains is a constraint of the form $\pm x \pm y \leq c$ where x and y represent a pair of variables and c is an arbitrary constant. To transform the abstract state at one execution point to that at the next execution point, we apply abstract transfer functions. These include functions for handling assignments, conditional statements, control-flow merge, and loops. Using abstract interpretation with octagon abstract domain, we show that we can verify simple invariants (both relational and non-relational) that are otherwise not possible with simpler abstract domains such as intervals.

In this report, we first review the background knowledge about abstract interpretation and the octagon domain and describe some implementation details in our system. Then, the analysis flow is demonstrated with some examples, and the complexity of several closure algorithms is examined with experimental results. Finally, future work that we have not done is discussed before we conclude the report.

2012 ACM Subject Classification Software and its engineering → Software verification and validation

Keywords and phrases formal verification, abstract interpretation, octagon domain

1 Introduction

1.1 Software Verification

Software is computer programs usually written by human engineers to perform some specific tasks. Because humans often make mistakes, it is important to *verify* that human-written programs really do what they are supposed to do. The *specifications* of software are the expected behaviors of a program defined by the users, which can be written in a formal way by defining some input-output relationship to be formally verified. As the scale of modern software grows, it becomes inevitable to rely on computers in software verification.

To verify software specifications, it is often easier to analyse *program states*, which are the values of the *program variables* hold at certain positions of execution (e.g., certain lines of the code). However, the number of possible program states may be too large to be enumerated and analysed one by one. Hence, they have to be *abstracted* into a less detailed domain so that several states can be represented together in a more compact way to ease the effort needed in verification.

1.2 Abstract Interpretation

Abstract interpretation is a technique to transform concrete program states into an abstraction, analysing them in the *abstract domain*, and interpret the conclusions back in the concrete domain. Abstraction is, in its nature, approximation, which means errors are inevitable. An important requirement for an abstract domain is that it should never produce false positive results, whereas some false negatives are acceptable. In other words, if the given program contains a bug, it must be caught; but when an abstract analyser reports a buggy behavior of the program, it may turn out to never happen in real executions.

Researchers have proposed several different abstract domains of different levels of precision. Like every computer science problem, there is a trade-off between precision of abstract domains and efficiency of their operations. For example, one of the simplest abstract domain is the *interval domain* [4], where a program variable is abstracted from taking a certain value into a range of possible values. Abstract operations in this domain is more efficient than in more complex domains, but due to its simplicity, an analyser in this domain may often fail to conclude anything useful. The interval domain is not *relational*, i.e., it does not consider relationships between program variables. Nevertheless, relationships between program variables are often the focus or even the target of software verification. Hence, a relational abstract domain is in need. To maintain efficiency, it should also not be too complicated.

1.3 The Octagon Domain

The Octagon domain [5] is an abstract domain describing program states with *octagonal constraints*, which are constraints on two program variables V_i, V_j of the form $b_i V_i + b_j V_j \leq c$, where $b_i, b_j \in \{0, +1, -1\}$ and c is a constant. The conjunction of a set of octagonal constraints forms an *octagon*, which is a set of concrete program states satisfying the constraints.

To deal with the positive and negative signs more easily, each program variable V_i is split into a positive form V'_{i+} and a negative form V'_{i-} during analysis. This way, all possible octagonal constraints can be written as a *potential constraint* of the form $V'_i - V'_j \leq c$.

An intuitive way to represent a set of potential constraints is the *potential graph*. In this graph, a node represents an extended variable (a program variable in its positive or negative form) and an edge pointing from node i to node j with a weight c represents a potential constraint $V'_j - V'_i \leq c$. In practice, to manipulate potential graphs easily, they are stored as adjacency matrices, called *difference bound matrices* (DBMs). An entry in a DBM $\mathbf{m}$ at row i, column j, $\mathbf{m}_{ij} = c$, encodes a potential constraint $V'_j - V'_i \leq c$.

A DBM $\mathbf{m}$ is said to be *coherent* if the constraints it encodes agree with the fact that V'_{i+} and V'_{i-} relate to the same program variable, or more precisely, if

$$\forall i, j,\ \mathbf{m}_{i+,j+} = \mathbf{m}_{j-,i-} \text{ and } \mathbf{m}_{i+,j-} = \mathbf{m}_{j+,i-} \text{ and}$$
$$\mathbf{m}_{i-,j+} = \mathbf{m}_{j-,i+} \text{ and } \mathbf{m}_{i-,j-} = \mathbf{m}_{j+,i+}$$

In the following of this report, all DBMs are assumed to be coherent unless explicitly specified. The prime symbol distinguishing an extended variable from a program variable is also neglected in the following.

1.4 Normalization of DBMs

As described earlier, the abstract representation we choose to represent octagons is DBM, and there can be different DBMs representing the same octagon. Such situations can make

the abstract interpreter assume constraints on variables that are not as tight as the best possible constraints and fail to verify certain conditions that are in fact true.

For example, consider the two DBMs (where $V'_{2i-1} = V'_{i+}$ and $V'_{2i} = V'_{i-}$):

$$\mathbf{m_1} = \begin{array}{c|cccc} & V'_1 & V'_2 & V'_3 & V'_4 \\ V'_1 & 0 & \infty & \infty & \infty \\ V'_2 & 4 & 0 & 3 & \infty \\ V'_3 & 1 & \infty & 0 & \infty \\ V'_4 & \infty & \infty & \infty & 0 \end{array} \quad \text{and} \quad \mathbf{m_2} = \begin{array}{c|cccc} & V'_1 & V'_2 & V'_3 & V'_4 \\ V'_1 & 0 & \infty & \infty & \infty \\ V'_2 & 10 & 0 & 3 & \infty \\ V'_3 & 1 & \infty & 0 & \infty \\ V'_4 & \infty & \infty & \infty & 0 \end{array}.$$

They represent the same octagon, but the explicitly stated constraint $V'_1 - V'_2 = 2V_1 \leq 10$ in $\mathbf{m_2}$ is not as tight as the respective constraint in $\mathbf{m_1}$.Nevertheless, there are two other constraints, namely $V'_1 - V'_3 \leq 1$ and $V'_3 - V'_2 \leq 3$, that imply $V'_1 - V'_2 = 2V_1 \leq 4$. If the abstract interpreter is asked to verify that there are no division by zero error when performing a division such as $1/(3 - V_1)$, it might not be able to directly do it without any additional processing to figure out the implicit constraints provided that it had the $\mathbf{m_2}$ as the abstract representation instead of $\mathbf{m_1}$.

Thus it is beneficial to always make implicit constraints explicit, and this process is called normalization. Intuitively, given a DBM $\mathbf{m}$, normalization is the process of finding the "smallest" DBM that represents the same octagon as $\mathbf{m}$. Depending on which implicit constraints are considered, we can obtain different normalization procedures.

1.4.1 Shortest-Path Closure

The most straight-forward normalization operation is called the shortest-path closure where we make all pair-wise constraints explicit. We call a matrix $\mathbf{m}$ is shortest-path closed if we have $\mathbf{m}_{ij} \leq \mathbf{m}_{ik} + \mathbf{m}_{kj}$ for all i, j, k and $\mathbf{m}_{ii} = 0$ for all i. Computing shortest-path closure is equivalent to replacing each entry $\mathbf{m}_{ij}$ in a DBM $\mathbf{m}$ with the shortest distance from i to j in the potential graph corresponding to $\mathbf{m}$, and it can be easily achieved using the well-known Floyd-Warshall algorithm. It runs in $O(n^3)$ time where n is the number of variables in the environment.

1.4.2 Strong Closure

Although the shortest-path closure is simple to implement, it disregards the structural constraints that V'_{2i} and V'_{2i-1} are in fact the same variable with opposite signs, and hence does not achieve the smallest possible DBM for a given octagon. By making such constraints explicit in addition to the constraints considered in the shortest-path closure, we obtain the strong closure. Formally, we call a DBM $\mathbf{m}$ strongly closed if it is shortest-path closed and $\mathbf{m}_{ij} \leq (\mathbf{m}_{i\bar{i}} + \mathbf{m}_{\bar{j}j})/2$ for all i, j where $\bar{i}$ is the index of the negated version of V'_i. In the work of Miné [5], it was shown that we can compute the strong closure in $O(n^3)$ time by modifying the Floyd-Warshall algorithm.

1.4.3 Tight Closure for Integral DBMs

While the strong-closure gives the tightest possible set of explicit constraints for a real-valued DBMs, we can achieve an even tighter normalization for integer-valued DBMs by incorporating integrality constraints as well. For example, suppose we had the constraint $V'_1 - V'_2 \leq 3$. Since $V'_1 - V'_2 = 2V_1$ is an even number if V_1 is an integral variable, we can

tighten this constraint to derive $V_1' - V_2' \leq 2$. Formally, a DBM $\mathbf{m}$ is tightly closed if it is strongly closed and $\mathbf{m}_{i\bar{i}}$ is even for all i. The work of Miné [5] presents and algorithm to compute the tight-closure of a DBM in $O(n^4)$ time, but it was later shown by Bagnara et al. [3] that a simple modification to the strong closure algorithm can compute the tight closure in $O(n^3)$ time if the coherence property (i.e., $\mathbf{m}_{ij} = \mathbf{m}_{\bar{j},\bar{i}}$ for all i, j) is maintained during updates.

1.5 Program Analysis with The Octagon Abstraction

The goal of an abstract interpreter is to prove (or disprove) some desired *invariants*, which are some properties that always hold at some point of the program. When loops are involved in the program, some program variables may take different values after the execution of the same line of code in a loop when it is executed several times, but some properties may remain valid whatever times the loop is executed or whatever user inputs the program receives.

An octagon abstract interpreter starts the analysis of a program with an octagon abstracting all possible program states. This octagon is an invariant at the very beginning of the program. Then, it tries to obtain an octagon at each point of the program (e.g., after each line of code) which is also an invariant at the point. This is done by translating each program statement into an *abstract transfer function* operating in the abstract domain. Finally, these octagons can be used to prove or signal possible invalidity of user-specified invariants at any point of the program.

The most common components in imperative programming languages include *assignment*, *(conditional) branching* and *looping* statements.

- **Abstract assignment**: The expression on the right-hand side of an assignment statement is analysed based on the constraints we have before the assignment. Then, the octagon is updated with the new constraints derived from the expression. If the expression involves more than two program variables, it has to be split into smaller sub-expressions, analysed separately and merged together again because octagonal constraints involve at most two variables.
- **Branching and merging (if-then-else)**: At the beginning of each branched code block, new constraints are introduced by the branching condition(s). They are analysed and decomposed into octagonal constraints and added into the octagon, this is called Backward Boolean Expression Analysis. After the whole branching block ends, all branches merge together, corresponding to computing the *union* of the octagons in the end of each branch. The union of two octagons is the *join* of their closed (normalized) DBMs.
- **Widening and narrowing for loops**: To analyse loops, we may need to loosen the constraints to get to a fixed point and conclude loop invariants in a finite time. The standard *widening* technique removes the constraints that are unstable after executing one iteration of the loop block. However, as widening is an over-approximation, after reaching to a stable point, we can try to refine it again to get back to a more precise abstraction with the *narrowing* technique.

Contributions

The main contributions of this project are as follows:

- OCaml implementation of octagon abstract domain for arbitrary precision integral variables with associated transfer functions.
- OCaml implementation of a static analyser for C programming language that supports a restricted set of C statements.

2 Implementation Details

We implemented an abstract interpreter in the Octagon domain to analyse C programs. The implementation in OCaml language can be found on GitHub[1].

Due to the limited time and scale of this project, we constrained the type of program variables to be always integers. Also, the structure and syntax of the input C program are also limited to be of some simple forms.

2.1 Data Structure and Basic Operations

Octagons are represented with DBMs, which are arrays of arrays. The `zarith` package [2] is used to deal with overflow problems as integers in C programs have a maximum representable value (`int_max` $= 2^{32} - 1$). We assume that the number of program variables is known in the beginning of analysis so that the size of all DBMs throughout the analysis is the same.

Basic operations on DBMs are implemented as follows:

- `top`: The greatest element $\top$ in the lattice of octagons is the DBM encoding no constraint and containing all possible program states. It has infinity at every entry.
- `bottom`: The smallest element $\bot$ in the lattice of octagons is a result of contradictory constraints and contains an empty set of program states. It is defined and handled separately without a real matrix.
- `add_constraint`: Add a new (octagonal) constraint to a DBM by finding the corresponding entry and replacing it with the constant in the new constraint if the original value in the matrix is larger.
- `is_in`: Check if a concrete program state is contained in the octagon defined by a DBM by checking if all constraints are satisfied.
- `is_inside`: Check if the octagon defined by a DBM is included in the other octagon defined by another DBM by checking if all entries in the first matrix are smaller than or equal to the corresponding ones in the second matrix. This defines the inclusion relation $\sqsubseteq$ among DBMs.
- `meet`: The meet operation $\sqcap$ over two DBMs takes the larger value among the two matrices for each entry separately.
- `join`: The join operation $\sqcup$ over two DBMs takes the smaller value among the two matrices for each entry separately.
- `widening`: The widening operation $\triangledown$ takes a DBM representing the state before the loop and a DBM representing the state after executing one iteration. The standard widening is implemented, which keeps the constraints as in the initial state if it is tighter after one iteration and deletes all other constraints.
- `narrowing`: The narrowing operation $\triangle$ takes a DBM representing the state before the loop and a DBM representing the state after executing one iteration. The standard narrowing is implemented, which refines the constraints with the final state if it is not constrained in the initial state and keeps all other constraints as in the initial state.

2.2 Closure Operations

We have implemented all three closure operations, shortest-path closure, strong closure, and tight closure, for integer-valued DBMs. For tight closure, we have in fact implemented two

[1] `https://github.com/lee30sonia/OctagonAI`

algorithms, the one with $O(n^4)$ running time introduced by Miné [5] and the one with $O(n^3)$ improved running time by Bagnara et al. [3].

```
1   (* Make 'm' coherent *)
2   for i = 0 to (2*n - 1) do
3     for j = 0 to (2*n - 1) do
4       m.(i).(j) <- Z.min m.(i).(j) m.(bar j).(bar i);
5     done;
6   done;
7
8   (* Compute shortest-path closure *)
9   for k = 0 to 2*n - 1 do
10    for i = 0 to 2*n - 1 do
11      for j = 0 to 2*n - 1 do
12        m.(i).(j) <- Z.min m.(i).(j) (m.(i).(k) #+ m.(k).(j));
13        (* Maintain coherence *)
14        m.(bar j).(bar i) <- m.(i).(j);
15      done;
16    done;
17  done;
18
19  (* Compute strong and tight closure *)
20  for i = 0 to 2*n - 1 do
21    for j = 0 to 2*n - 1 do
22      m.(i).(j) <- Z.min m.(i).(j)
23          ((m.(i).(bar i) #/ two) #+ (m.(bar j).(j) #/ two));
24      (* Maintain coherence *)
25      m.(bar j).(bar i) <- m.(i).(j);
26    done;
27  done;
```

Figure 1 Implementation of the optimized tight closure algorithm proposed by Bagnara et al.[3]

All four implementations use in-place updates as it is more memory and time efficient and is shown to be correct. For the optimized version of the tight closure, we explicitly make sure the coherence property is maintained both before and during the execution of the algorithm as it is required for the correctness of the algorithm.

For reference, we include the tight closure algorithm in Figure 1. We use Zarith package's arbitrary precision integers as DBM entries, and the operator '`#+`' is defined to be the addition operator. Note that the symbol '`two`' represents constant 2 and the division operator '`#/`' in Line 23 represents the integer division so that there is an implicit '`floor`' operation. Lines 2-6 make sure that the coherence property is maintained prior to executing the rest of the algorithm, and Lines 14 and 25 ensure the coherence property is maintained after each update. We remark that the given algorithm uses zero-based indexing.

Additionally, it is not necessary to compute the closures from the beginning every time we update a DBM. In fact, if we only update one (or a constant number of) constraint in the DBM, we can recompute the tight closure in $O(n^2)$ using the incremental tight closure algorithm presented by Miné [5], and we have also implemented this version.

2.3 Abstract Assignment

In this project, we restrict ourselves to assignments to integer variables where the assigned quantity is an integer linear combination of the variables. Our implementation considers two cases:

1. *The assigned quantity is a general integer linear combination of variables.*
   In this case, we simply consider the variable-wise maximum and minimum values read from the current DBM to derive the upper and lower bounds of the assigned quantity. Then we update all constraints involving the assigned variable using these bounds. However, this implementation may not get the best possible bounds in certain cases. A better way to handle such assignments is to solve a linear program to figure-out the best bound for each potential constraint, but this is too complicated and time consuming.
2. *The assigned quantity is of the form $ax+by+c$ where x, y are two variables, $a, b \in -1, 0, 1$, and c is an arbitrary constant.*
   For this kind of assignments, we can do better, and consider relational information when establishing new upper and lower bounds for octagonal constraints.

2.4 Parsing C Programs

We use the `FrontC` parser [1] to parse the input C program to be analysed.

We only use a very simple subset of the Frontc parser. We chose to go through the syntax tree and call the functions analysing the program on the fly.

We can analyse a program file corresponding to different function declarations (*analyseFunctions* and *analyseFunction* in the code). Those functions are made of a header containing information on the variables (this is not supported yet as this part of the parser is very intricate and complicated), and a body where the function is described. At the beginning of the function's body, we have the declarations of all the variables, building the context of the function (collected *analyseDefinitions*). The second part of the function's body is the statement (*analyseStatement*), which describe parts of code like a sequence, a loop or a computation. All of those statement contain expressions (*analyseExpression*), mostly computations, which triggers most of the functions doing the analysis of the code.

2.5 Program Analysis

We first parse the C program to obtain the syntax tree, and then go over the syntax tree of a function, computing the DBM at every point.

- If the statement is an assignment, we use abstract assignment function to get the next DBM.
- For conditional *if* statements, we need to compute the DBMs at the beginning of *then* block and *else* blocks. For this, we have to calculate the backward boolean analysis so the condition is taken into account. This backward boolean analysis is represented as a new DBM in which the conditions are added as constraints. Then, the analysis of all the branches are executed on their own initial abstract memory states and the results are joined so all the possibles ways are taken into account for the resulting state.
- For *while* statements, we have to calculate all the possible input (abstract memory) states. A first idea would be to execute on the input state the backward boolean analysis so the condition of the loop is met and then the analysis of the inner body of the loop is executed. We now have all the new possibles input states so we can add them to the input state and we iterate so we reach a fixpoint. The problem happens if at every step,

one abstract variable only increase by one so the recursion would be infinite. A solution is to add the new states calculated at the end of the body of the loop to the input states not by using a join but by using a widening operator. Its usage ensures the termination of the analysis while the result is still sound. However, the resulting state can be refined a bit by adding the usage of the narrowing operator instead of the widening after a fixpoint is reached. This operator will keep the termination an the soundness while restricting the output state.

At the end of every function the invariants found during its analysis are printed to the user.

3 Analysis Examples

In this section, we demonstrate how our octagon abstract interpreter analyses code with some example C programs. The analysis of these examples can be run in our system following the commands described in the README file.

3.1 Example 1

Our first example C program to analyse, shown in Figure 2, contains only assignments and branching. There are three program variables, x, y and z, where x, y, assigned with user inputs, are unconstrained. This simple program computes the absolute difference between x and y, so the result $z = |x - y|$ is expected to be non-negative. Though the invariant to verify (in line 8) is not relational, an interpreter in the interval domain cannot prove it because the branching condition is relational.

```
1  int x, y, z;
2  scanf("user input x: %d\n", &x);
3  scanf("user input y: %d\n", &y);
4  if (x >= y)
5      z = x - y;
6  else
7      z = y - x;
8  assert(z >= 0);
```

Figure 2 Example program 1.

From the beginning of analysis until line 3, the octagon is the top element with no constraint. Then, when entering the `if` block (line 4), we have an additional constraint $x \geq y$, or in the octagonal form $y - x \leq 0$. After executing the assignment in line 5, we have $z \leq x - y$ and $z \geq x - y \Rightarrow -z \leq y - x \leq 0 \Rightarrow z \geq 0$. On the other hand, in the `else` block, we have $x < y$ or $x - y \leq -1$. Similarly, after executing line 7, we get $z \leq y - x$ and $z \geq y - x \Rightarrow -z \leq x - y \leq -1 \Rightarrow z \geq 1$. In the end, taking the union of $z \geq 0$ and $z \geq 1$, the analyser concludes that $z \geq 0$ is indeed true.

3.2 Example 2

The second example, shown in Figure 3, demonstrates how the abstract domain in combination with the widening and narrowing operators are used to prove loop invariants. Again, an interval interpreter will not be able to prove the assertions because the important loop invariant $x + y = 10$ is relational.

```
1  int x, y;
2  x = 10;
3  y = 0;
4  while (x > y)
5  {
6      x = x - 1;
7      y = y + 1;
8  }
9  assert(x + y == 10);
10 assert(x <= 5);
11 assert(y >= 5);
```

Figure 3 Example program 2.

4 Experimental Results

In this section, we present some experimental results.

4.1 Performance of Closure Operations

The running times of different closure algorithms are measured and presented in Table 1. The experiments were run for different DBMs where the number of environment variables (n) are 25, 50, 100, and 200. The running times are measured on a 3.1GHz Dual-Core Intel Core i5 CPU with 16GB 2133MHz DDR3 RAM memory. The first row presents the timing results for the standard Floyd-Warshall shortest-path closure algorithm. The second and third rows respectively give the results for the strong-closure algorithm and the tight-closure algorithm proposed in [5]. The last row shows the results for the optimized tight-closure algorithm presented in [3].

Table 1 Running time of different closure algorithms

	n = 25	n = 50	n = 100	n = 200
Shortest Path Closure	0.002 s	0.014 s	0.097 s	0.799 s
Strong Closure	0.005 s	0.040 s	0.296 s	2.520 s
Tight Closure	0.538 s	7.942 s	132.238 s	> 1000 s
Tight Closure Optimized	0.003 s	0.024 s	0.186 s	1.344 s

Note that the optimized version of tight closure is in-fact more efficient than the strong closure algorithm proposed in [5]. This is because the strong closure algorithm we implemented is the one presented in [5], and it does strengthening passes (i.e., updates of the form $\mathbf{m}_{ij} = \min(\mathbf{m}_{ij}, (\mathbf{m}_{i\bar{i}} + \mathbf{m}_{\bar{j},j})/2)$ for all i, j) inside the outer for-loop of the traditional Floyd-Warshall algorithm. In contrast, Bagnara et al. [3] showed that the strengthening passes can be moved out of the outer for-loop.

4.2 Program Analysis

Figure 4 shows an example execution of the static analyser for the example C programs seen above and the invariants it is able to derive.

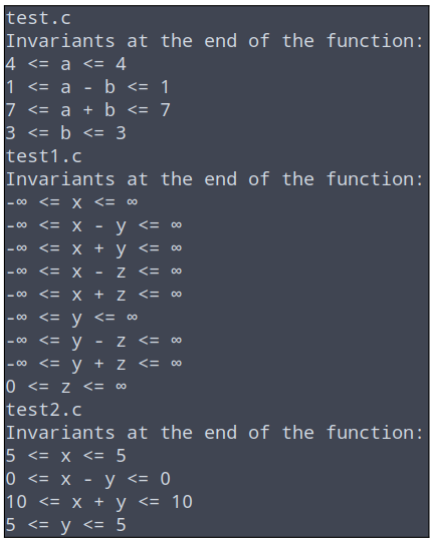

Figure 4 A screenshot of program analysis.

5 Future Work

In this section, we propose some potential improvements we can make to our implementation.

1. Supporting a bigger part of the C language.

   For example, we don't support at this moment calling functions. Moreover, the arguments of the functions are not taken into account. We could also add the analysis of the returned value by adding a new phantom variable that represents this result. At this point, the analysis of pointers is not supported since it needs two different abstract domains. Finally, some statements such as *do ... while* are not supported but they only are rewrites of supported statements so it would be easy (but still long) to implement their analysis.

2. Allow different types of integer variables with overflow analysis.

   Currently, we allow all integral variables but we do not handle overflow scenarios. For example, suppose we had a `unsigned int` variable x with constraint $0 \leq x \leq 2^{31} - 1$, and we have the assignment $x \leftarrow x+1$. Then, in the current implementation, we update the bound to be $1 \leq x \leq 2^{31}$, whereas the correct bound should be $-2^{31} \leq x \leq 2^{31} - 1$, and this is an important future improvement. Since different types of integral variables (i.e., `char, short, int, long` etc.) have different maximum values, we need a fine grained overflow analysis to support those types. One solution would be to expand the support of every variable to the infinite when one of it's bound is to the infinite at one step and raising a warning to the user.

3. Better handling of assignments.

   The present implementation does not handle the assignments of general linear combinations in the most precise manner as it only uses variable-wise upper/lower bounds. This could be improved to have better upper/lower bounds using relational information as well. One solution would be to decompose the expressions into multiple supported ones using phantom variables. However, it requires a first analysis of the program to know how many phantom variables are required.

6 Conclusions

In conclusion, we have experimented how the octagon domain works on a restriction of the C language, some algorithms that are involved and we have seen its interest compared to non-relational abstract domains such as the intervals. This analyser is still simple but remains a first step toward a more exhaustive one.

References

1 *FrontC*. URL: `https://opam.ocaml.org/packages/FrontC/`.
2 *Zarith*. URL: `https://opam.ocaml.org/packages/zarith/`.
3 Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 8–21, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
4 Patrick Cousot et al. Static determination of dynamic properties of programs. 1977.
5 Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.