Stripes Domain

Software Correctness, Security and Reliability Project Report

Alessio De Biasi
Ca' Foscari University of Venice, Italy
870288@stud.unive.it

Jonathan Gobbo
Ca' Foscari University of Venice, Italy
870506@stud.unive.it

1 Introduction

The purpose of this project is to implement in LiSA [3], a library for static analysis developed by the group SSV @ Ca' Foscari, the Stripes abstract domain. This domain is originally proposed in the paper "Safer Unsafe Code for .NET" [1] by Ferrara, Logozzo, and Fähndrich.

The authors propose this domain in order to analyze unsafe C# code in which unsafe operations, such as manipulating pointers or performing arbitrary casts, are allowed.

In this case, the usual run-time bound checking is not performed, meaning that the programmer can involuntary cause a wide range of errors, such as buffer and array overflows.

The Stripes abstract domain is therefore used to statically verify memory accesses, in such a way that upper bounds of allocated memory are not exceeded.

2 Formal definition

The Stripes abstract domain tracks constraints in the form:

$$x - k_1 \cdot (y\ [+z]) > k_2 \tag{1}$$

where x, y and z (which is optional) are variables, while k_1 and k_2 are integer constants. Since in a program the only statements that can be expressed in this form are assignments and boolean conditions, only such statements are analyzed.

2.1 Domain representation

The elements in the Stripes abstract domain are represented by a map from variables to constraints. In particular, each constraint in the normalized form (see equation 1) is represented by a quadruple $(y,\ z,\ k_1,\ k_2)$.

If z is absent, the constraint is represented, instead, by the quadruple $(y,\ \bot,\ k_1,\ k_2)$.

Formally, let $\mathcal{V}$ be the set of all the variables in the analyzed program. A Stripes abstract domain element is the map:

$$\Big[\mathcal{V} \to \mathcal{P}\big((\mathcal{V} \times (\mathcal{V} \cup \bot) \times \mathbb{N} \times \mathbb{N})\big)\Big]$$

The top domain element $\top$ expresses the fact that the abstract state tracks no variables.

The bottom domain element $\bot$, instead, expresses the fact that the abstract state contains a contradiction (i.e., the statement, to which the $\bot$ domain element is associated to, is not reachable at run-time, whatever execution path the program takes).

2.2 Assignments

When an assignment in the form `x = <expr>` is encountered, all the constraints that mention variable x are dropped. Formally, we drop all the constraints:

$$(y,\ z,\ k_1,\ k_2) \mid y = x \vee z = x$$

Then, we can track the encountered assignment only if `<expr>`, once simplified, contains one or two variables and an integer constant, but does not mention variable `x`. Moreover, if the simplified expression mentions two variables, they must be multiplied by the same constant k_1.

Finally, to extract the constraint in the normalized form (see equation 1), we perform the following operations (between square brackets there are the optional parts):

$$\begin{aligned} x &= k_1 \cdot y\ [+k_1 \cdot z] + k_2 \\ x &\geq k_1 \cdot (y\ [+z]) + k_2 \\ x - k_1 \cdot (y\ [+z]) &\geq k_2 \\ x - k_1 \cdot (y\ [+z]) &> k_2 - 1 \end{aligned}$$

This will produce the constraint $(y,\ z,\ k_1,\ k_2 - 1)$, or $(y,\ \bot,\ k_1,\ k_2 - 1)$ if z is missing. This constraint is then associated to variable x in the map.

2.2.1 New constraints

If the assignment can be tracked according to the aforementioned conditions, then it is possible to infer other constraints based on the simplified expression.

2.2.1.1 Reverse assignment

If the simplified expression contains only one variable multiplied by the constant 1, i.e., it is in the form `y + k`, then we can infer the constraint (x, $\perp$, 1, $-k-1$) associated to variable y.

If, instead, the simplified expression contains only one variable but multiplied by -1, i.e., it is in the form `-y + k`, then we can infer the constraint (x, $\perp$, -1, $k-1$), again associated to variable y. Indeed:

$$\begin{aligned} x &= -y + k \\ -y &= x - k \\ y &= -x + k \\ y &\geq -x + k \\ y &> -x + k - 1 \\ y - (-x) &> k - 1 \end{aligned} \qquad \begin{aligned} x &= y + k \\ y &= x - k \\ y &\geq x - k \\ y &> x - k - 1 \\ y - x &> -k - 1 \end{aligned}$$

2.2.1.2 Equality/Inequality chain

In the case the simplified expression contains tracked variables, we can infer new constraints by applying inequality rules.
For example, if the abstract state contains the constraint $b-1\cdot c > 3$, the assignment `a = 2*b` will produce the new constraint (c, $\perp$, 2, 5) associated to a. Indeed:

$$\begin{aligned} a &= 2 \cdot b \\ a &> 2 \cdot b - 1 \\ a &> 2 \cdot (c + 3) - 1 \\ a &> 2 \cdot c + 5 \\ a - 2 \cdot c &> 5 \end{aligned}$$

2.2.1.3 Common expressions

Suppose, at some point, we track the assignment `a = <expr1>`. Then, suppose that later we track the assignment `b = <expr2>` such that `<expr2>` can be written as $k \cdot$ `<expr1>` $+$ `<expr3>`. If the assignment `a = <expr1>` is still tracked (i.e., no variable in `<expr1>` have been reassigned) and if this new expression can be written in the normalized form (see equation 1), then we can infer a new constraint.

For instance, the assignment `a = 2*(b+c) + 9` produces the constraint $a - 2 \cdot (b + c) > 8$. When the assignment `d = 4*(b+c) + 7` is encountered, this will produce the constraint (a, $\perp$, 2, -10) associated to d. Indeed:

$$\begin{aligned} d &= 4 \cdot (b + c) + 7 \\ d &> 4 \cdot (b + c) + 6 \\ d &> 2 \cdot a - 16 + 6 \\ d &> 2 \cdot a - 10 \\ d - 2 \cdot a &> -10 \end{aligned}$$

The same applies in the opposite way: if `<expr1>` can be written as $k \cdot$ `<expr2>` $+$ `<expr3>`, then we can infer a new constraint.

For example, the assignment `a = 2*b + 4` produces the constraint $a - 2 \cdot b > 3$. When the assignment `c = b + 8` is encountered, this will produce the constraint (c, $\perp$, 2, -13) associated to a. Indeed, since if $c = b+8$, then $b = c - 8$:

$$\begin{aligned} a &> 2 * b + 3 \\ a &> 2 \cdot (c - 8) + 3 \\ a &> 2 \cdot c - 16 + 3 \\ a &> 2 \cdot c - 13 \\ a - 2 \cdot c &> -13 \end{aligned}$$

2.2.1.4 Assignment propagation

If the simplified expression is in the form `u + v`, where `u` and `v` are variables, then we can substitute `x` with `u + v` in all the constraints that mention only `x`. This inferring must be performed before dropping the constraints that mention `x`.

2.2.2 Exception

If the simplified expression is in the form `u - v + k`, normally we cannot track the expression because `u` is multiplied by 1 while `v` is multiplied by -1, therefore, the expression cannot be written as $k_1 \cdot (u + v)$.
However, we can infer the constraint (x, v, 1, $-k-1$) associated to variable u by rewriting the whole assignment. Indeed:

$$\begin{aligned} x &= u - v + k \\ x + v - k &= u \\ u &= x + v - k \\ u &\geq x + v - k \\ u &> x + v - k - 1 \\ u - (x + v) &> -k - 1 \end{aligned}$$

2.3 Conditions

When a boolean condition is encountered, we can use the constraints tracked in the abstract state to establish

if the condition is either always true, always false or neither of the two.

If the condition contains at least one of the logical operators `&&` and `||`, then the condition on the left and the condition on the right of such operator are tested singularly.

Their results are then combined according to the following tables (`F` stands for "proved to be always false", `T` stands for "proved to be always true" while `U` stands for "neither of the two"):

E_1	E_2	E_1 && E_2
F	F	F
F	T	F
F	U	F
T	F	F
T	T	T
T	U	U
U	F	F
U	T	U
U	U	U

Table 1: AND operation

E_1	E_2	E_1 || E_2
F	F	F
F	T	T
F	U	U
T	F	T
T	T	T
T	U	T
U	F	U
U	T	T
U	U	U

Table 2: OR operation

In the case the condition to be tested is negated (i.e., it starts with the logical operator `!`), we test a new condition obtained by taking the condition that follows the operator `!` and negating all its operators according to the following table:

E	!E
E_1 > E_2	E_1 <= E_2
E_1 >= E_2	E_1 < E_2
E_1 < E_2	E_1 >= E_2
E_1 <= E_2	E_1 > E_2
E_1 == E_2	E_1 != E_2
E_1 != E_2	E_1 == E_2
E_1 && E_2	!E_1 || !E_2
E_1 || E_2	!E_1 && !E_2
!E_1	E_1

For example, to test if the condition `!(a > b)` is verified or not, we test if the condition `a <= b` is.

Eventually, we will reach a condition in the form:

$$E_1 \texttt{ <op> } E_2 \tag{2}$$

where E_1 and E_2 are non-constant and non-boolean expressions, while `<op>` is one of the comparison operators `==`, `!=`, `>`, `<`, `>=` and `<=`.
If it is not the case, then we are not able to tell if the condition is either always true or always false.

Conditions in such form (see equation 2) are then rewritten as:

$$E > F$$

If the comparison sign `<op>` is a *greater than* sign (`>`), then $E = E_1$ and $F = E_2$, otherwise we rewrite the condition according to the following table:

Condition	Rewritten Condition
E_1 == E_2	E_1 > E_2 - 1 && E_2 > E_1 - 1
E_1 != E_2	E_1 > E_2 - 1 || E_2 > E_1 - 1
E_1 >= E_2	E_1 > E_2 - 1
E_1 < E_2	E_2 > E_1
E_1 <= E_2	E_1 > $E_2 + 1$

Then, the condition $E > F$ is rewritten in the normalized form:

$$x - k_1 \cdot (y\ [+z]) > k_3 \tag{3}$$

If it is not possible to rewrite the condition in such form, we conclude that it is not possible to tell whether the condition is always satisfied or not.

A boolean condition in the normalized form (see equation 3) is satisfied if there exists a constraint (y, z, k_1, k_2) associated to x such that $k_2 \geq k_3$.

If we don't find such constraint, we test for the opposite condition (i.e., we apply the `!` operator): if it is satisfied, then we conclude that the original condition is always false.

If neither of these two cases applies, then we can't tell if the condition is satisfied or not.

2.3.1 New constraints

When a boolean condition is encountered and we are not able to tell if it is always true or always false, we can refine the abstract state by extracting new constraints based on the condition.

In this case, we can apply the same rules seen in 2.2.1.2, but, instead of applying them to a statement in the form:

$$\texttt{a} = \texttt{<expr>}$$

we apply them to a statement in the form:

$$\texttt{a} > \texttt{<expr>}$$

For example, if the abstract state contains the constraint $b - 1 \cdot c > 3$ and we encounter the condition `a > 2 * b + 7`, we can infer the constraint $(a, \perp, 2, 13)$ associated to a. Indeed:

$$\begin{aligned} a &> 2 \cdot b + 7 \\ a &> 2 \cdot (c + 3) + 7 \\ a &> 2 \cdot c + 13 \\ a - 2 \cdot c &> 13 \end{aligned}$$

3 Implementation

In this section we describe how we implemented in LiSA [3] the functionalities described earlier.

3.1 Representation

A constraint is represented by the `Constraint` class, which holds 4 fields that represents respectively the variables y and z (if it is $\perp$, then the field is `null`) and the constants k_1 and k_2.

The abstract domain is represented by the class `StripesDomain`, which holds the abstract state.

The abstract state is represented by a `Map`. This `Map` maps each tracked `Variable` into a `Set` containing all the `Constraint`s associated to the variable.

The bottom element ($\perp$) is represented by an instance of the `StripesDomain` class which has no domain elements. Comparisons with the bottom element are performed reference-wise (i.e., using the `==` operator).

The top element ($\top$) is again represented by an instance of the `StripesDomain` class which contains no tracked variables. Every element that has no tracked variable and that is not the bottom element, is considered top.

It is common in a program that variables go out of scope. When this happens, if the variable that goes out of scope is tracked, then we delete from the domain all the constraint associated to that variable and all the constraints that mention such variable.

3.2 Assignments

When an assignment in the form `x = <expr>` is encountered, we first delete from the map the entry (if present) associated to `x` and all the `Constraint`s that mention the variable `x`.

Then we simplify the assigned expression `<expr>` by collapsing multiple occurrences of the same variable and summing up all the integer constants.

The assignment `x = <expr>` can be tracked if and only if `<expr>`, once simplified, does not mention variable `x` and contains one or two variables. In particular, if there are exactly two, they must be multiplied by the same integer constant. Therefore, the simplified expression is in the form $k1 \cdot (y[+z]) + k_2$.

Consequently, we can insert in the map the constraint (y, z, k_1, k_2) associated to the variable x.

If the assignment falls in the exception case (see 2.2.2), we verify that the simplified expression does not mention variable `x` and contains one variable multiplied by 1 and another variable multiplied by -1.

Finally, if the assignments has been tracked,we try to infer new constraints as seen in 2.2.1.

3.3 Conditions

When a condition is encountered, we can establish if it is always true, always false or neither of the two.

If the condition contains logical operators `&&` and `||`, we recursively analyze the conditions on the left and on the right of such operators.

If, instead, the condition starts with the logical operator `!`, we test a new condition obtained by taking the condition the operator `!` is applied to and complementing all its operators.

When a condition in the normalized form (see equation 3) is reached, we can prove that it is always true if there exists a constraint (y, z, k_1, k_2) associated to any variable in the abstract state such that $k_2 \geq k_3$.

If we are not able to prove this, we test for the opposite condition (i.e., we complement the comparison operator): if this new condition is proved to be always true, then the previous one is proved to be always false.

If neither of the two cases applies, then we cannot tell whether the condition is always true or always false. In this case, we can refine the abstract state by computing new constraints according to what has been explained in 2.3.1.

4 Tests

In this section, we provide some example programs written in the IMP language [2] and we examine the results of the static analysis performed by LiSA.

4.1 Simple program

```
1  class Test1 {
2      main() {
3          def a = 10;
4          def b = a * 3 + 5/2 + a%3;
5          def c = a + 5*b - 7 - (6*a)/3 -
6                  3*(b - a - 2) + 6;
7          if (a > 3*b) {
8              def d = 5*b;
9              if (a > 3*b+7) {
10                 c = c+1;
11             }
12             b = 4*(a + d);
13             c = 2*a+2*b - 40;
14         }
15         return c;
16     }
17 }
```

Line 3 contains an assignment that assigns a constant value which cannot be turned in normalized form (see equation 1). Therefore, it is not tracked.

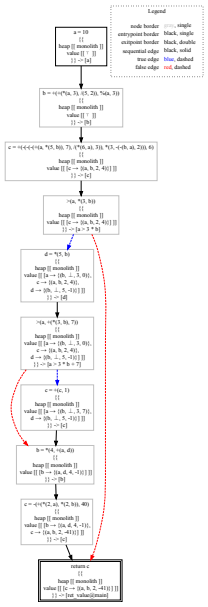

Figure 1: Output of the first test

Line 4 contains an assignment that assigns an expression that cannot be simplified due to the factor `a%3`, so it is not tracked.

Line 5, once simplified, becomes the assignment `c = 2*(a+b)+5`, so, we can extract the constraint $(a, b, 2, 4)$ associated to the variable `c`.

Line 7 contains a condition that is proved to be neither always true nor always false because variables `a` and `b` are not tracked. Therefore, we can refine the abstract state inside the `if` block by adding the constraint $(b, \perp, 3, 0)$ associated to variable a.

Line 8 contains an assignment that can be tracked, adding the constraint $(b, \perp, 5, -1)$ associated to variable `d`.

Line 9 contains another condition. Again, we cannot tell if the condition is either always true or always false, thus we refine the abstract state inside the `if` block. In this case, the new constraint $(b, \perp, 3, 7)$ is more specific than $(b, \perp, 3, 0)$ extracted on line 7, so the latter is substituted by the former, and this can be seen on line 10.

Line 10 contains an assignment but, since the assigned expression mentions the variable the expression is assigned to, that assignment is not tracked. When exiting the `if` block of line 11, the condition on line 9 is not true anymore, so the added constraint is deleted restoring the constraint extracted on line 7.

Line 12 and 13 contain assignments that can be tracked as $(a, d, 4, -41)$ and $(a, b, 2, -1)$ respectively.

Line 14 closes the `if` block. Several operations are performed:

- Variable `d` goes out of scope. Therefore every constraint associated to that variable and every constraint that mention that variable are deleted;
- The condition on line 7 is not true anymore. Therefore all the constraints extracted from that condition are deleted;
- The constraints tracked at the end of the `if` block are joined with the ones tracked before such block. This operation keeps the intersection of the constraints (i.e., the constraints that have the same values for x, y, z and k_1 on both states. The value for k_2 is the lowest k_2 value among the aforementioned constraints).

Line 15 contains a return statement, whose state reflects what was just said.

4.2 New constraints on assignment

```
1  class Test {
2      main() {
3          def a = 5;
4          def b = 3;
5          def c = 2*(a + b) + 1;
6          def d = (-c) + 1;
7          def e = 2*(-c) + 7;
8          def f = 4*(-c) + 14;
9          def g = a - b - 4;
10         def v1 = a;
11         a = b+c;
12     }
13 }
```

Lines 3 and 4 contain assignments that are not tracked, since they assign constant values.

Line 5 can be tracked as usual. The new constraint $(a, b, 2, 0)$ is added to the abstract state.

Line 6 can be turned into the normalized form (see equation 1). Therefore it creates the constraint $(c, \perp, -1, 0)$. Moreover, we can flip the assignment (as seen in 2.2.1.1) so that it becomes `c = -d + 1`, generating the new constraint $(d, \perp, -1, 0)$ associated to `c`.

Line 7 mentions again variable `c`. Since the assigned expression is a multiple (excluding the constant value) of the one assigned to `d`, we can rewrite the assignment as `e = 2d+5`, creating the constraint $(d, \perp, 2, 4)$ (as seen in 2.2.1.3), that is added to the usual $(c, \perp, -2, 6)$, both associated to variable `e`.

Line 8 is an assignment that assigns an expression to variable `f`. By substituting variable `e` with the expression assigned to it, we end up with the assignment `f = 4*d + 10`, which generates the constraint $(d, \perp, 4, 9)$ associated to variable `f`. Of course, as all the other assignments, we extract the constraint $(c, \perp, -4, 13)$ associated to `f`.

Line 9 is an example of the exception assignment (see 2.2.2). The extracted constraint will be (g, b, 1, 3) associated to variable a.

Line 10 is a simple assignment.

Line 11, instead, is an example of assignment propagation (see 2.2.1.4). Therefore, we generate the new constraint (b, c, 1, -1) associated to variable v1.

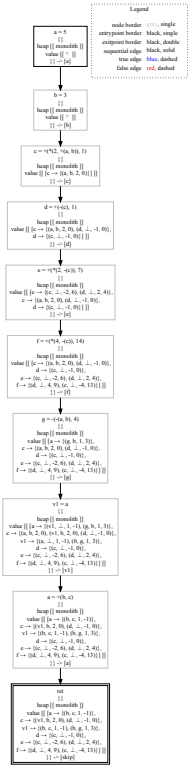

Figure 2: Output of the second test

4.3 Conditions always true/false

```
1  class Test {
2      main() {
3          def a = 10;
4          def b = 2*a + 7;
5          if (b > a+a+4) {
6              a = 2*b;
7          } else {
8              a = 7050*b;
9          }
10         def d = a+b;
11         while ((d < a+b-9 && a < 5*3) ||
                   (d <= a+b-5)) {
12             d = d*2;
13         }
14         return d;
15     }
16 }
```

Line 3 contains a non-trackable assignment while line 4 does.

Line 5 contains a condition that can be proved to be always true. Indeed the else block will be skipped and the abstract state of all the instructions in the else block is $\bot$.

Line 10 contains an assignment that can be tracked.

Line 11 contains a condition that evaluates always to false. Therefore, no instruction of the while block is ever executed. This fact is reflected on line 12 where the variable d is still tracked.

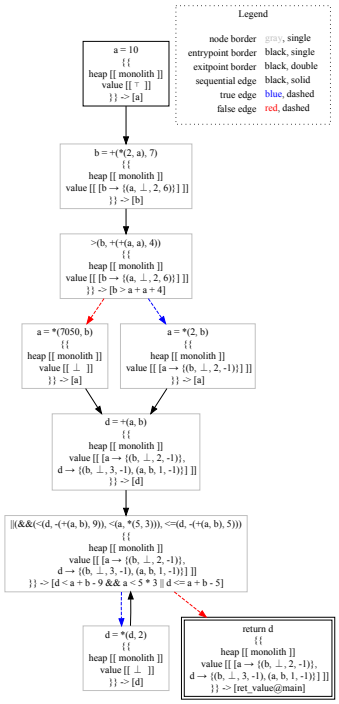

Figure 3: Output of the third test

References

[1] Pietro Ferrara, Francesco Logozzo, and Manuel Fähndrich. "Safer Unsafe Code for .NET". In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 329–346. DOI: 10.1145/1449955.1449791.

[2] SSV @ Ca' Foscari. *The IMP Language*. URL: https://unive-ssv.github.io/lisa/imp/.

[3] SSV @ Ca' Foscari. *LiSA (Library for Static Analysis)*. URL: https://unive-ssv.github.io/lisa/.