Reduced Product between Extended Sign and Parity

Software Correctness, Security, and Reliability
Ca' Foscari University of Venice

Hernest Serani **Mirco Venerba**

{877028, 872653}@stud.unive.it

May 11, 2022

Abstract

In this paper we focus on Abstract Interpretation [1] which is a method to examine and analyze the behaviour of a program by approximating the values that the expressions can take during the execution of the program, and for this reason this manner of examination permits to check if some properties into the program are satisfied or not.

A simple abstract domain looks some specific properties into a specific domain but in our study case, we need to implement a more complex domain that is given by the union of extended sign domain and parity domain and to do this, we can use Cartesian Product Operator and the Reduced Product to elaborate better the informations by both domains. We implemented it using LiSA, a static analyzer, that takes in input *.imp* files and it analyzes them returning *.dot* resulting files.

1 Introduction

LiSA [2] is a JAVA library that implements a static analyzer for imperative programs. The analysis is done on a pseudo language called IMP. This library supports abstract interpretation theory and uses Control Flow Graphs (CFGs) to represent the program to be analysed. Internally it uses a fix point algorithm in order to implement all the analysis types that are possible such as available expressions, reaching definitions and many other things. Having such a powerful tool, allows us to have a safe and sound anylysis of a program based on some domains. The ouput of the result is a digraph which represents the control flow graph of the program and the analysis. A use case of this tool is the detection of bugs in a program.

2 The Cartesian Product among Abstract Domains

In this section, we are going to take in consideration the Cartesian Product of two abstract domains for semplicity, but the idea can be applied to any number of abstract domains. Since LiSA is a static analyzer written in Java, we can express abstract domains as classes and therefore implement abstract domains. The ability to express a new domain from a concrete one, is called abstract interpretation and allows the mapping from the concrete to the abstract one. The concrete domain, consists of values which basically have the same type as expressed in the programming language, such as integers, booleans, strings, etc. We can abstractly interpret these values. Examples can be Parity (Even or Odd), Sign (Positive or Negative), and many other things.

If we consider these as a set, they can be infinite in case of integers (without taking in consideration the maximum expressable number in a computer system), or finite in case of booleans. Taking the Cartesian Product of two sets, gives a set aswell, which is finite in case both sets are finite and infinite otherwise. The same idea can be applied for abstract domains if we consider them as sets, they produce another pseudo-domain, which is given as all the possible combinations of elements of the two abstract domains. This way we have obtained a new domain which inherits the properties that describe both abstract domains.

Among the elements of the abstract domain, we can perform different binary, urnary or ternary operations, and somehow the results of these, need to be handled, in order to still finish into the same abstract domain.

The Cartesian Product is given as: $C = A \times B$, where A and B are two sets (abstract domains in our case).

The partial order is defined as the conjunction of the partial orders of the two domains (eq. 1). Similarly, the *lub* (least upper bound), and *glb* (greatest lower bound) are defined as the component-wise application of the operators of the two domains (eq. 2). Since we have these operations, we can conclude that the Cartesian Product forms a lattice (eq. 3). The last important observation is that even the semantic operator $\mathbb{S}_C : C \to C$, is defined as the component-wise application of the abstract semantics of the two domains:

$$(a_1, b_1) \leq_C (a_2, b_2) \Leftrightarrow a_1 \leq_C a_2 \ \wedge \ b_1 \leq_B b_2 \tag{1}$$

$$\begin{aligned} (a_1, b_1) \sqcup_C (a_2, b_2) &= (a_1 \sqcup_A a_2, b_1 \sqcup_B b_2) \\ (a_1, b_1) \sqcap_C (a_2, b_2) &= (a_1 \sqcap_A a_2, b_1 \sqcap_B b_2) \end{aligned} \tag{2}$$

$$\langle C, \leq_C, \sqcup_C, \sqcap_C \rangle \tag{3}$$

$$\mathbb{S}_C[(a, b)] = (\mathbb{S}_A[a], \mathbb{S}_B[b]) \tag{4}$$

This way the semantics of the Cartesian Product is a sound over-approximation of the concrete semantics. One of the main issues of the Cartesian Product result is that it may contain several abstract values that represent the same thing. For instance:

$$([2, 4], Odd), ([2, 3], Odd), ([3, 4], Odd), and([3, 3], Odd) \tag{5}$$

All these values concretize to the 3 singleton.

3 Reduced Product among Abstract Domains

Using only the Cartesian Product between the two domains, we have a valid, cheap and effective implementation of the combined domain. However, we can improve the information precision by applying reductions. We saw how the result can have different precisions in the example shown above. Applying the reduction, improves the precision of the abstract representation, without affecting the concrete meaning.

Let's define the Cartesian Product as:

$$C = A \times B \tag{6}$$

The Reduced Product of two domains A and B can formally be defined as:

$$RP(A, B) = \{\rho(a, b) \mid (a, b) \in C\} \tag{7}$$

where

$$\rho(a, b) = \sqcap_C\{(a', b') \mid \gamma_C(a, b) \leq_C \gamma_C(a', b')\} \tag{8}$$

The function in (eq. 8) is the reduction operator, which uses the greatest lower bound, in order to reduce the value to more precise one.

There are two properties which a reduction operator needs to satisfy:

$$\rho(a, b) \leq_C (a, b) \tag{9}$$

$$\gamma(\rho(a, b)) = (a, b) \tag{10}$$

Equation 9 expresses that the result of the application of a reduction operator is a more precise abstract element.
Equation 10 expresses that an abstract element and its reduction represent the same property.

If we take again the example we saw above:

$$([2, 4], Odd) \tag{11}$$

We can easily see how:

$$\rho([2, 4], Odd) = \rho([3, 4], Odd) = \rho([3, 3], Odd) \tag{12}$$

4 Granger's Product

Using Granger's Product [3], we can have a more elegant way to compute the reduced product, and have an overapproximation of the reduction operator.

It is based on two operators:

$$\begin{aligned} \rho_1 &: C \to A \\ \rho_2 &: C \to B \end{aligned} \tag{13}$$

Each of these operators, refines the domain involved in the product. In order to have a sound reduction operator ρ_1 and ρ_2 they have to satisfy the following properties:

$$\begin{aligned} \rho_1(a, b) \leq_A a \wedge \gamma_C(\rho_1(a, b), b) = \rho_C(a, b) \\ \rho_2(a, b) \leq_B b \wedge \gamma_C(\rho_2(a, b), b) = \rho_C(a, b) \end{aligned} \tag{14}$$

5 Implementation

LiSA allows us to define a domain by using the Java classes. In order to inherit some of the general and common functionalities of a domain representation, we need to extend **NonRelationalValueDomain** class, and implement the abstract methods, update some of the parent methods based on our needs, and introduce new methods. This way we can achieve code reuse among various domains. To implement our domain, we created the class **ExtSignParityDomain**, which combines two other abstract domains (**ExtSignDomain** and **ParityDomain**). Our class internally has two state methods which represent the values of the two single domains. In order to construct an object of the combined domain, we can either pass the parity and the extended sign, or create a top element, which represents an unknown value. Internally we have defined all the possible combinations of the two domains using constant values. This way allows us to save some

memory by working with references. In order to make this work, we need to implement the hashcode and equals methods.

We didn't see the need to implement and apply the Granger's Product in order to reduce the value, instead we created a static method which returns the reduction. The result of the abstract domain combination, is always the most precise one (fixpoint reached in a single step) and an overapproximation is not needed, so we can conclude that the reduction is needed only once in order to get a sound, correct and precise value. Note that this is a particularity of our domain combination. If we were working with intervals i.e. we need to implement the Granger's Product in order to reduce the value since it's not possible to correctly reduce it with a single step (as in our case).

It is worth to mention the following cases of our reduce product:

$$\begin{aligned} \rho(0-, Odd) &= (-, Odd) \\ \rho(0+, Odd) &= (+, Odd) \\ \rho(0, Odd) &= (\bot, \bot) \end{aligned} \tag{15}$$

In order to make the code scalable and clean, all the methods which perform the evaluation (*evalNonNullConstant*, *evalUnaryExpression*, *evalBinaryExpression*), call directly the methods of the two domains, which are implemented in their respective classes. The result is the combination of the two domains after applying these methods. The same logic is applied for the methods which perform the satisfy of a unary of binary expression. We based the implementation of the **ParityDomain** class on the **Parity** class already present in LiSA, but we extended some functionalities such as support for modulo, satisfiability and the mothods which deal with the assumption. We also did the same thing for the **ExtSignParityDomain** class, by using the correct implementation given during previous tasks and performing the same improvements.

Some of the main improvements we made on Parity:

- Support for modulo operator during evaluation
- Satisfiability check for the numeric negation (-ODD = ODD and -$EVEN$ = $EVEN$)
- Satisfiability check for the equal and not equal comparison operators
- Assumption for the not equal comparison operator (if x != $ODD \rightarrow x = EVEN$, and vice versa)

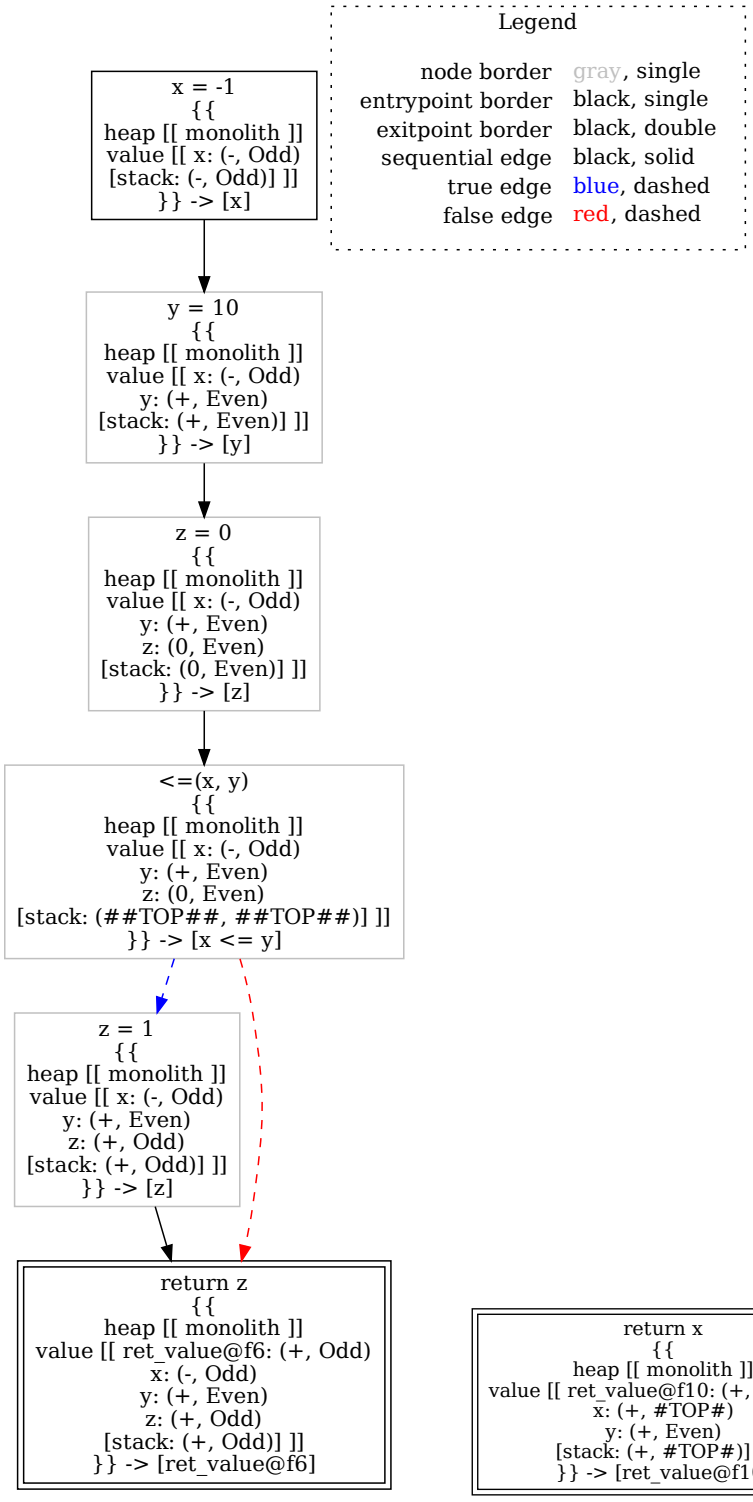
Some of the main improvements we made on Extended Sign:

- Support for modulo operator during evaluation
- Satisfiability check for all comparison operators
- Assumption for all comparison operators i.e.:
  - if $x! = 0+ \rightarrow x = \top$
  - if $x > 0+ \rightarrow x = +$

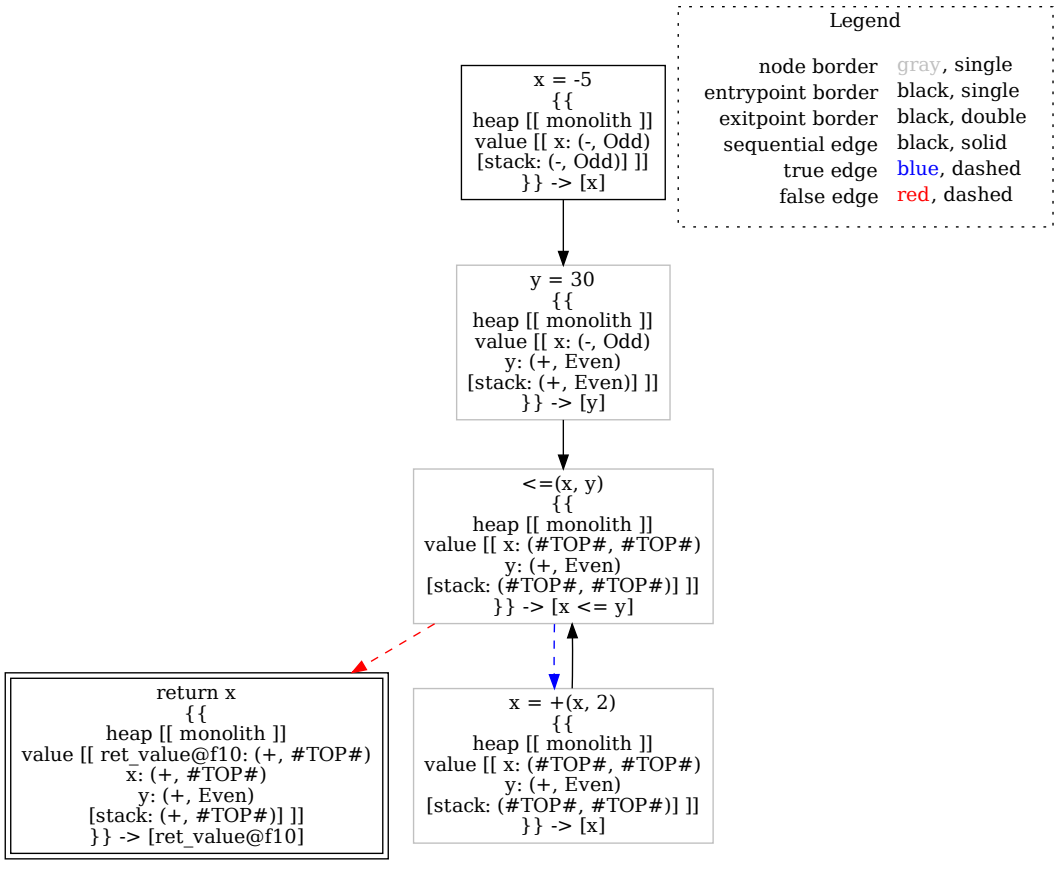
6 Test Cases

In this section we are going to demonstrate some tests that we have done using various functions which cover almost all the test cases that deal with our combined domain implementation.

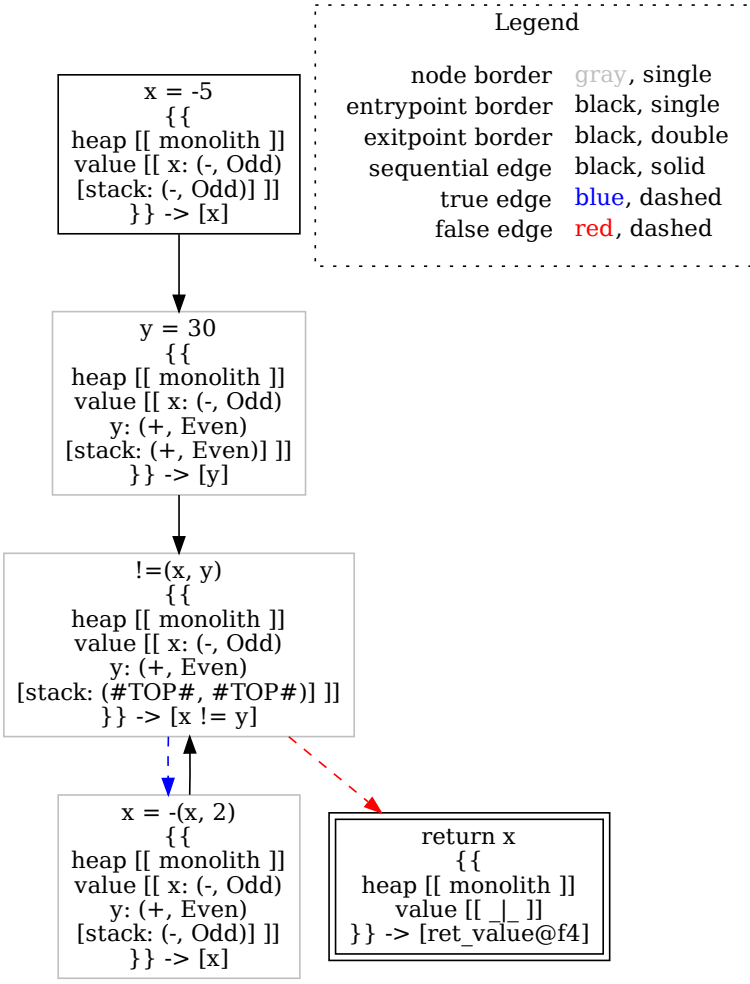
- **Less or Equal ($\leq$) Operator** This function tests the $\leq$ operator and we can also see that the expression in the if condition is true because the type of x is $(-, ODD)$ and the type of y is $(+, EVEN)$, and in fact we know that the negative odd numbers are less or equal than positive even numbers. At the end the function the type of z is $(+, ODD)$ because it changes the value into the if instruction.



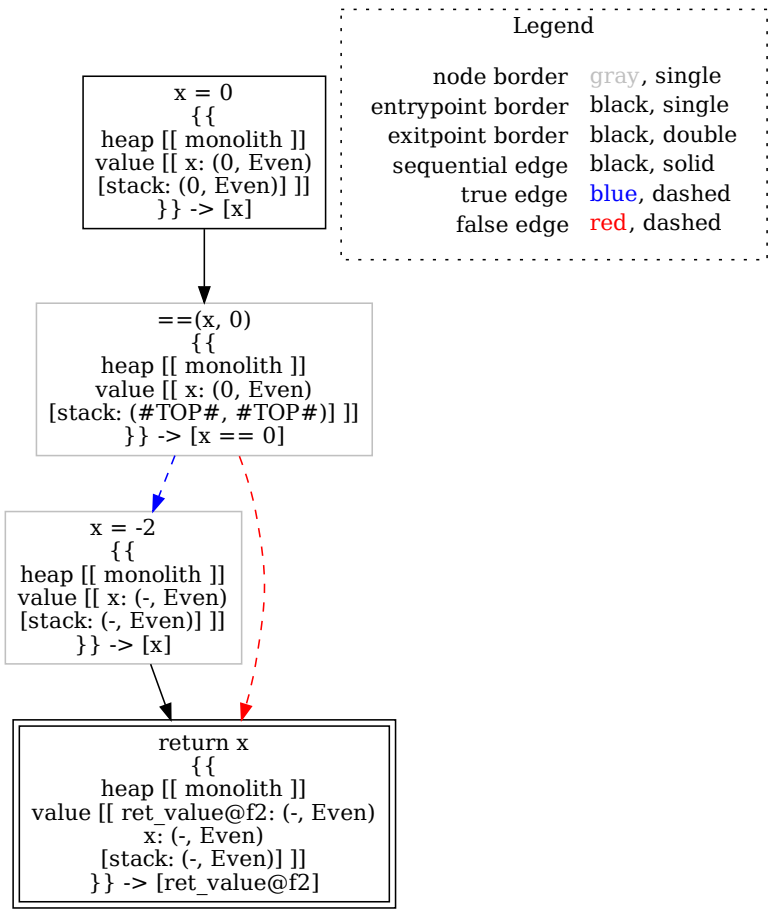
Less or equal



Finite while loop



Infinite while loop



If condition

- **Finite while loop** In this test, there is a while loop that goes on until x is less or equal to y and in fact we can see that when the loop finishes. The variable x has a value greater than y and actually we can see from the digraph that x is positive but TOP in the parity because it can assume both positive even numbers and positive odd numbers.
- **Infinite while loop** The value of x is always different from y because in the while loop there is a decrement of 2 of the x variable. Therefore we have an infinite while loop. In fact we can see from the image of digraph that the type of x remains always $(-, ODD)$.
- **If condition** This example tests if the variable $x = 0$ and this is. For this reason the final type of x is $(-, EVEN)$.

7 Conclusions

We can conclude saying that with this project, we have implemented an extension for the LiSA static analysis framework that permits the analysis of the reduced product between extended sign and parity domains, and for this reason, we proved also that is possible to exploit information and more complex structure nodes from two or more input abstract domains. Then we have also tested our implementation with some non-trivial and complex functions to prove that our implmentation is correct and it is useful to identify bugs in the code that concerns the sign and parity.

References

[1] A. Cortesi, G. Costantini, and P. Ferrara, "A survey on product operators in abstract interpretation," *EPTCS 129, 2013, pp. 325-336*, 2013.

[2] UniVE-SSV, "lisa," *https://github.com/UniVE-SSV/lisa*.

[3] P. Granger, "Improving the results of static analyses programs by local decreasing iteration," *FSTTCS*, 1992.