Abstract Interpretation

Arvind W. Kiwlelekar

November 16, 2004

Abstract

Abstract Interpretation is a theory of semantics approximation applied to programming languages. Standard or concrete semantics allows us to interpret the properties of a program in some concrete or *standard domain*. Semantics approximation allows us to interpret the properties of program in an *abstract domain*. Abstract domain is an approximate version of standard domain. The main objective of the paper is to illustrate how an abstract interpretation is derived for a given program.

1 Introduction

Abstract Interpretation is one of the widely used technique for program analysis. Purpose of such program analysis is to obtain as much information as possible about the run time behavior of the program. In a *standard execution* of the program run time behavior is obtained by actually running the program over a machine. Sometimes it is also known as *program profiling*. *Non-standard execution* of the program derives information about run time behavior without running the program and not using standard or concrete values. In non-standard execution of the program, *value descriptions* (properties describing the values of variables) or *abstract values* are used in place of the actual computed values. The term *abstract interpretation* is used to denote semantics based version of non-standard execution [6]. Nonstandard execution can be roughly described as follows-

- Execute commands in a program using values from abstract domain.
- Deduce information about the programs computation on actual input data from the resulting abstract descriptions of values.

The main objective of the paper is to illustrate the methodology of deriving an abstract interpretation of a program. To achieve our object, in Section 2, we are first deriving an abstract interpretation for a small language that supports only arithmetic expressions. Section 3 will extend that small language to support usual control structure found in any programming language and will explain the methodology of deriving an abstract interpretation for a full fledged imperative programming language. Lattices are the underlying mathematical foundation for deriving abstract interpretation of a programming language [5]. Paper assumes familiarity with lattices and Tarski fixpoint theorem. In Section 4, will give the rationale behind selecting a particular mathematical structure i.e. lattice and imposing restrictions (monotonicity) over the functions (i.e. abstract function, concretization function) that are applied to these mathematical structures.

2 Abstract Interpretation of a Language having only Arithmetic Expressions

In order to get the better insight of the methodology involved in deriving an abstract interpretation, we are first restricting ourselves to a small language that supports only arithmetic expressions over integer type of data. The example that we are discussing here is adopted from [2]. The main objective of giving this simple example is to identify *elements* of abstract interpretation. *Mathematical rationale* behind selecting these elements will be understood in the Section 4. In this example, we are abstracting an integer number with its sign. Arithmetic operations supports multiplication, unary substraction, addition and division operations. The syntax of the arithmetic expression is given by-

$$exp = n \ \mid exp_1 * exp_2 \ \mid \ -exp_1 \ \mid \ exp_1 + exp_2 \ \mid \ exp_1/exp_2$$

The semantics is given by

$$\begin{aligned}[exp] = n \ &\mid [exp_1] * [exp_2] \ \mid \ -[exp_1] \\ &\mid \ [exp_1] + [exp_2] \ \mid \ [exp_1]/[exp_2]\end{aligned}$$

The term [exp] denotes the value of an expression. When an expression is in the simple form i.e. exp =n, the value of that expression is nothing but the integer number, n.

In a standard program execution, we are using a concrete values like 10, 5, etc.... Table 1 shows a the execution of these arithmetic expressions for two numbers i.e. 10, and 5. Calculating the behavior of this small language for all possible values of n is next to impossible. So, we require to take the help of non-standard execution.

Expression	Value
$e = 5$	$e = 5$
$e = -5$	$e = -5$
$e = 10 * 5$	$e = 50$
$e = 10 + 5$	$e = 15$
$e = 10 - 5$	$e = 5$
$e = 10/5$	$e = 2$

Table 1: Standard Execution

In the non-standard program execution, actual values are replaced by abstract values or properties of concrete values. Here, we are partitioning all integer values in three partitions called positive (*pos*), negative (*neg*) and zero (0). *pos* denotes $\{n|n > 0\}$. *neg* denotes $\{n|n < 0\}$. Hence, our abstract domain is a set $Abs = \{pos, neg, 0\}$

Now, we will require to define the semantics for multiplication, substraction, addition and division operations in the abstract domain. Abstract semantics for multiplication operation is straightforward and dose not pose any problems. The multiplication operation in abstract domain is denoted by- $\otimes$. The semantics for $\otimes$ is given in the Table 2

$\otimes$	*pos*	*neg*	0
pos	*pos*	*neg*	0
neg	*neg*	*pos*	0
0	0	0	0

Table 2: Abstract Semantics for Multiplication

Abstract semantics for unary minus operation is also straightforward i.e. is given in Table 3

Providing abstract semantics for addition operation poses some problem. When we perform $pos \oplus neg =??$ or $neg \oplus pos =??$, the result of the operation is not exactly predictable in terms of pos, neg, and 0. The result belongs to set of integers Z, this complete set of integers is denoted by the partial element $\top$. So, we require to extend our abstract domain by including $\top$ -

	pos	neg	0
$\ominus$	neg	pos	0

Table 3: Abstract Semantics for Unary Minus

$A = \{pos, neg, 0, \top\}$. So, the abstract semantics for addition is given in the Table 4.

$\oplus$	pos	neg	0	$\top$
pos	pos	$\top$	pos	$\top$
neg	$\top$	neg	neg	$\top$
0	pos	neg	0	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Table 4: Abstract Semantics for Addition

Similarly, abstract semantics of division operation poses the problem of no-termination when we divide a certain number by zero. This non-termination is captured by yet another partial element called bottom $\perp$. Again, we are extending our abstract domain by including $\perp$- $A = \{pos, neg, 0, \top, \perp\}$. The abstract semantics for the operation division $\oslash$ is given in the Table 5

$\oslash$	pos	neg	0	$\top$	$\perp$
pos	pos	neg	0	$\top$	$\perp$
neg	neg	pos	0	$\top$	$\perp$
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 5: Abstract Semantics for Division

After adding partial elements $\top$ and $\perp$ our abstract domain i.e. $Abs = pos, neg, 0, \top, \perp$ becomes a lattice, $Abs_L =< Abs, \top, \sqsubseteq \perp, \top >$, where $\sqsubseteq$ is an approximation ordering. With $\sqsubseteq$ ordering, pos, int, and 0 becomes comparable elements of the lattice, $\perp$ becomes a minimal element and $\top$ becomes maximal element. Throughout this paper we refereing this lattice as an abstract domain.

After imposing a lattice structure over the abstract domain, we are in a position to define functions called abstract functions and concretization function.

- **Abstraction function (α and β)**: Abstraction function (α) maps a value from a concrete domain to an abstract domain. Here, both abstract domain and concrete domain forms a lattice. We have seen that abstract domain forms a lattice. Similarly, our concrete domain i.e. set of integers forms (Z) forms a lattice if Z is extended by adding $\top$ and $\perp$. Let us define Abs denotes abstract domain and CT denotes concrete domain. Abstraction function (α) is defined as-

$$\begin{aligned} \alpha : CT \rightarrow Abs \\ \alpha(n) \mapsto 0 \quad if\ n = 0\ \ and\ n \in CT \\ \alpha(n) \mapsto pos\ \ if\ n > 0\ \ and\ n \in CT \\ \alpha(n) \mapsto neg\ \ if\ n < 0\ \ and\ n \in CT \end{aligned}$$

  The other form of **abstraction function** β maps a set of values S from a concrete domain to an abstract domain.

$$\begin{aligned} \beta : P(CT) \rightarrow Abs \\ \beta(S) \mapsto lub\{\ \alpha(x) \mid x \in S\}\ and\ S \subseteq P(CT) \end{aligned}$$

- **Concretization Function** (γ): Concretization function maps a value from an abstract domain to the set of values that it represents from the concrete domain.

$$\begin{aligned} \gamma : Abs \rightarrow P(CT) \\ \gamma(a) \mapsto 0\ if\ a = 0 \\ \gamma(a) \mapsto \{\ n \mid n > 0\}\ if\ n = pos \\ \gamma(a) \mapsto \{\ n \mid n < 0\}\ if\ n = neg \end{aligned}$$

Proving the correctness of the abstraction function is not a trivial task. For example abstract interpretation of an expression like $3+4 = 7$ yields the correct result i.e. $alpha(3+4 = 7) = \alpha(3) + alpha(4) = alpha(7) = pos$ But at the same if expression is like (1+2) -3 = 0 is giving $\top = 0$. Which is an imprecise answer. This is due to loss of information. Abstract interpretation gives us *safe* answers rather than correct answers i.e. giving an answer 'pos' when the correct answer '0' is an incorrect answer. But saying that an answer is 'integer' is a *safe* answer!!

To summarize this example, we identified following elements of the abstract interpretation-

- Concrete and Abstract Domain.
- Abstraction and Concretization function.
- Correctness criteria (i.e. safe answers).

In addition to above elements, we are putting following restrictions.

- Abstract and concrete domains are always *lattices*.
- Abstraction and concretization functions are always *monotonic functions*.
- Following Correctness condition must be satisfied.
  - $\beta(\gamma(a)) = a$ for all $a \in Abs$
  - $\gamma(beta(S)) \supseteq S$ for all $S \in P(CT)$

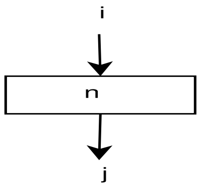
3 Abstract Interpretation of a Flow-Chart Based Programming Languages

In his seminal paper [5], P. Cousot develops a rigorous framework for abstract interpretation of an imperative programming languages. His main idea of deriving an abstract interpretation is based on the concept of *collecting semantics*. Collecting semantics associates with each program point the set of all memory stores that can ever occur when program control reaches that point, as the program is run on data from a given initial data space. It was shown that a wide variety of flow analysis may be realized by finding finitely computable approximations to the collecting semantics.[6]

In the P. Cousot's framework, program is represented by a graph; the nodes corresponds to program operations and the arcs corresponds to control flow. Each node is can be classified into one of the three types i.e. *Assignment*, *Test*, and *Junction*. Each node has a set of successor nodes and a set of predecessor nodes associated with it. These sets are given by the following functions.

$$n\_succ, n\_pred : Nodes \rightarrow 2^{Nodes}$$

. For each type of nodes there are constraints on the cardinalities of the successor and predecessor sets. These constraints determines the syntax of such a language. The syntax for assignment, test, and join node is given in the Figure **??**.



- **Syntax**: $|n\_pred(n)| = 1 \wedge |n\_succ(n)| = 1$
- **Collecting Semantics**:

$$C_j = \{\sigma[x = n] | \sigma \in C_i \wedge [e]\sigma = n\}$$

- **Abstract Semantics**:

$$a_j = \beta(\{\sigma[x = n] | \sigma \in \gamma(a_i) \wedge [e]\sigma = n\})$$

Figure 1: Syntax and Semantics for Assignment Node

In order to define the semantics of such a language we are identifying a set of program points and associating with it all the states that are possible while executing the program. Program points are annotated with *labels*. Let us define state (σ) the program point is variable and value pair that is given by the function

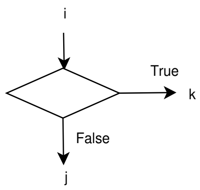
$$\sigma \in \Sigma = Var \rightarrow Z$$

Let us define one more function called *Context* that will map a given *label* to the set of states associated with that label. Hence, a context at a particular program point is nothing but the set of all possible states at that point. Which is also known as collecting semantics.

$$C \in Contexts = Labels \rightarrow P(\Sigma)$$

Let us define C_k denote the collecting semantics at program point k. The collecting semantics, C_k, at a particular point k is derived from the collecting semantics of it's predecessor point i The collecting semantics for assignment, test, and join node is given in the table **??**.

The set $P(\Sigma)$ of all sets of stores forms a lattice with set inclusion $\subseteq$ as its partial order, so any two context sets A and B have at least a upper



- **Syntax**: $|n\_pred(n)| = 1 \wedge |n\_succ(n)| = 2$
- **Collecting Semantics**:

$$C_j = \{\sigma | \sigma \in C_i \wedge [b]\sigma = False\}$$

and

$$C_k = \{\sigma | \sigma \in C_i \wedge [b]\sigma = True\}$$

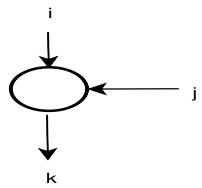
- **Abstract Semantics**:

$$a_j = \alpha(\{\sigma | \sigma \in \gamma(a_i) \wedge [b]\sigma = False\})$$

and

$$a_k = \alpha(\{\sigma | \sigma \in \gamma(a_i) \wedge [b]\sigma = True\})$$

Figure 2: Syntax and Semantics for Test Node



- **Syntax**: $|n\_pred(n)| = 2 \wedge |n\_succ(n)| = 1$
- **Collecting Semantics**:

$$C_k = C_i \cup C_j$$

- **Abstract Semantics**:

$$a_k = \alpha(\gamma(a_i) \cup \gamma(a_i)) = lub(a_i, a_j)$$

Figure 3: Syntax and Semantics for Join Node

bound $A \cup B$ and the greatest lower bound $A \cap B$. The lattice $P(\Sigma)$ is a complete, meaning that any collection of sets of stores has a lub in $P(\Sigma)$, namely its union.

As the state representing program variables forms a lattice, we are in a position to simulate the program behavior by applying Tarski's least fixed point theorem. In the following discussion, C_k^i denotes collecting semantics at program point k during i^{th} iteration. Application of lfp is done as follwos

- We start with $C_1 0 = \lambda L.\phi$ i.e. program that takes label as an input starting collecting semantics as an empty set.
- We apply the relations between C_j^i and C_k^i to construct C_k^i from C_j^i.
- We stop when $C_j^i = C_j^{i-1}$.

The problem with this approach is that state space of the program being an infinite the program executes forever. Technique of abstract interpretation overcomes this situation by approximating this infinite state space.

Various approximations can be expressed by simpler lattices, connected to $P(\Sigma)$ by an abstraction function (α and β). Where $\alpha : 2^{P(\Sigma)} \rightarrow Abs$

and $\beta : P(\Sigma) \rightarrow Abs$. Abs is the abstract domain representing the approximation. An abstract function is most often used together with a dual concretization function $\gamma : Abs \rightarrow P(\Sigma)$. The pair of functions β and γ are required to satisfy the conditions mentioned in the last section.

As the abstract domain is represented by the lattice, the same machinery employed for deriving collecting semantics is also used for deriving abstract semantics. The difference between deriving collecting semantics and abstract semantics is that solution for the program behavior in an abstract domain converges earlier than in a concrete domain.

3.1 Example Program illustrating Collecting Semantics in Concrete Domain

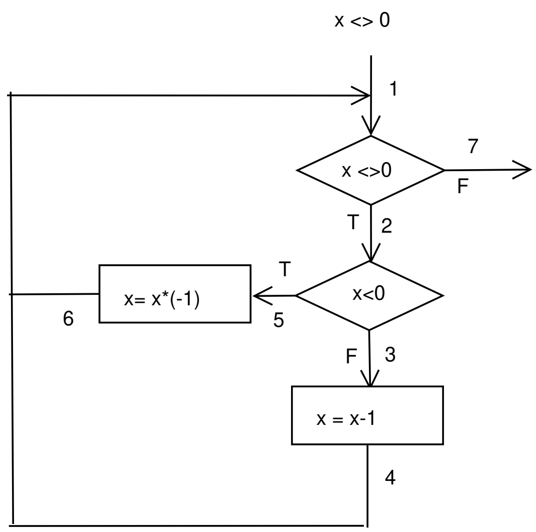


Figure 4: Flowchart for the Program 1

Program 1

while ($x <> 0$) do
  begin
    if($x <> 0$) then
      $x = x * (-1)$
    else
      $x = x - 1$

end

This example gives the collecting semantics at each program point and shows the values taken by the collecting semantics when the initial value of x is -2.

At program point 1, the collecting semantics is given by-

$$C_1^{i+1} = C_1^i \cup C_4^i \cup C_6^i$$

$C_1^0 = \phi$
$C_1^1 = C_1^0 \cup \{v|v \in s(x)\} = \{-2\}$

At program point 2, the collecting semantics is given by-

$$C_2^{i+1} = C_2^i \cup \{v|v \in C_1^{i+1} \wedge s(x <> 0) = True\}$$

and

$$C_2^1 = \{-2\}$$

At program point 3, the collecting semantics is given by-

$$C_3^{i+1} = C_3^i \cup \{v|v \in C_2^{i+1} \wedge s(x < 0) = False\}$$

and

$$C_3^1 = \phi = \perp$$

At program point 4, the collecting semantics is given by-

$$C_4^{i+1} = C_4^i \cup \{v|v \in C_3^{i+1} \wedge s(x) = s(x) - 1\}$$

and

$$C_4^1 = \phi = \perp$$

At program point 5, the collecting semantics is given by-

$$C_5^{i+1} = C_5^i \cup \{v|v \in C_2^{i+1} \wedge s(x < 0) = True\}$$

and

$$C_5^1 = \{-2\}$$

At program point 6, the collecting semantics is given by-

$$C_6^{i+1} = C_6^i \cup \{v|v \in C_5^{i+1} \wedge s(x) = s(x) * (-1)\}$$

and

$$C_6^1 = \{2\}$$

At program point 7, the collecting semantics is given by-

$$C_7^{i+1} = C_7^i \cup \{v|v \in C_7^{i+1} \wedge s(x <> 0) = False\}$$

and

$$C_2^1 = \phi = \perp$$

Similarly, during **Second** iteration the collecting semantics at all program points is given below-
$C_1^2 = \{-2, 2\}$
$C_2^2 = \{-2, 2\}$
$C_3^2 = \{\phi, 2\}$
$C_4^2 = \{\phi, 1\}$
$C_5^2 = \{-2, \phi\}$
$C_6^2 = \{2, \phi\}$
$C_7^2 = \{\phi, \phi\}$

Note that i^{th} element in the set C_k^i indicates the value taken by x during i^{th} iteration at program point k.

Similarly, during **Third** iteration the collecting semantics at all program points is given below-
$C_1^3 = \{-2, 2, 1\}$
$C_2^3 = \{-2, 2, 1\}$
$C_3^3 = \{\phi, 2, 1\}$
$C_4^3 = \{\phi, 1, 0, \}$
$C_5^3 = \{-2, \phi, \phi\}$
$C_6^3 = \{2, \phi, \phi\}$
$C_7^3 = \{\phi, \phi, \phi\}$

Similarly, during **Fourth** iteration the collecting semantics at all program points is given below-
$C_1^4 = \{-2, 2, 1, 0\}$
$C_2^4 = \{-2, 2, 1, \phi\}$
$C_3^4 = \{\phi, 2, 1, \phi\}$
$C_4^4 = \{\phi, 1, 0, \phi\}$
$C_5^4 = \{-2, \phi, \phi, \phi\}$
$C_6^4 = \{2, \phi, \phi, \phi\}$
$C_7^4 = \{\phi, \phi, \phi, 0\}$

3.2 Collecting Semantics in Abstract Domain

Collecting semantics in an abstract domain is derived in the same way like deriving collecting semantics in concrete domain. Abstract semantics at each program point is given in the Table 7

The collecting semantics in an abstract domain that we defined in the section 2 using the concrete value of x = -2 is given in the Table 6

a_k^i	1	2	3	4
1	{neg }	{neg, pos }	{neg, pos, pos }	{neg, pos, pos, 0 }
2	{neg }	{neg, pos }	{neg, pos, pos }	{neg, pos, pos, ϕ }
3	{ϕ }	{ϕ, pos }	{ϕ pos, pos }	{ϕ, pos, pos, ϕ }
4	{ϕ }	{ϕ, pos }	{ϕ pos, 0 }	{ϕ pos, 0, ϕ }
5	{neg }	{ neg, ϕ }	{neg, ϕ, ϕ }	{neg, ϕ, ϕ, ϕ }
6	{pos }	{ pos, ϕ }	{pos, ϕ, ϕ }	{pos, ϕ, ϕ, ϕ }
7	{ϕ }	{ ϕ, ϕ }	{ϕ, ϕ, ϕ }	{ϕ, ϕ, ϕ, 0 }

Table 6: Collecting Semantics in Abstract Domain

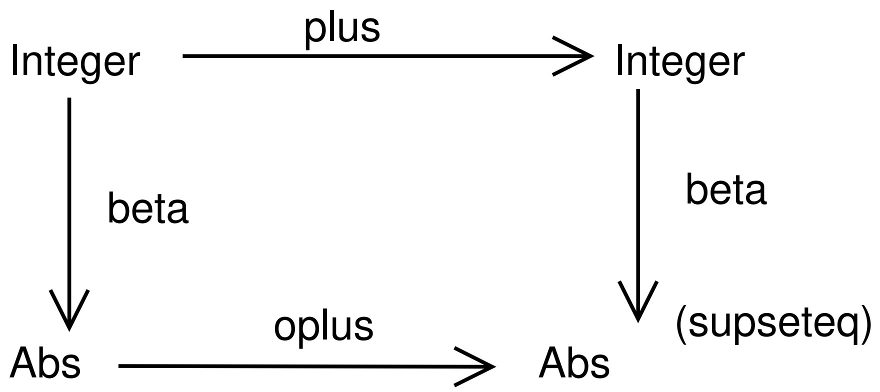
4 Understanding Mathematical Rationale behind Abstract Interpretation

The examples of abstract interpretation discussed in the Section 2 and 3 are based on the mathematical framework proposed by P. Cousot in [5]. As mentioned earlier main elements of this framework are-

1. Abstract and Concrete domains must be always *lattices*.
2. Abstraction and Concretization functions are always *monotonic functions*.
3. Following Correctness condition must be satisfied.
   (a) $\beta(\gamma(a)) = a$ for all $a \in Abs$
   (b) $\gamma(\beta(S)) \supseteq S$ for all $S \in P(CT)$

   Such a function of pair i.e. β and γ is also called Galois Insertion.

Condition 1 imposes a lattice structure over the concrete and abstract domains. As mentioned earlier, abstract interpretation is all about no-standard execution of programs. The main reason of using lattices for modelling state space in abstract and concrete domain is that- lattices equipped



with Tarski Fixed point theorem gives us the needed machinery to simulate the execution of the program. Tarski fixed point theorem allows us to simulate the execution of a program that uses iterations or recursions. Normally, the lattice used is a complete lattice having finite height. That allows us to capture termination of the program. At the same time, abstract and concrete domain are extended by including $\top$ and $\perp$ element in the sate space. Inclusion of $\perp$ captures the meaning of either no progress in program execution, or non-termination of the program, or unreachability of a particular program point. Inclusion of $\top$, models the whole sate space in the abstract or concrete domain.

Condition 2 Condition 2 puts the restriction of monotonocity over the α and β. To understand the rationale behind this restriction, let us recall the semantics provided for addition operation in the earlier section $\oplus$ (oplus). This can be viewed as a two different algebraic structures. First algebraic structure modelling addition operation in an integer domain i.e. $(Integer, +)$. The second algebraic structure models addition operation in an abstract domain $(Abs, \oplus)$. The β relates these two different algebraic structures as shown in the Figure 4. In usual algebraic structure mapping, the function β (beta) will be a homomorphic function. The homomorphic functions expects us to give precise and unique results. When we perform $pos \oplus neg$ operation in an abstract domain we are unsusre about the result. So, we are relaxing the condition of being homomorphic to monotonic. This allows us to give approximate answers. $x \sqsubseteq$ y means x is more precise than y.

Condition 3 The condition 3.a is natural one which intends to faithfully represent concrete to abstract domain mapping and vice-versa. The condition 3.b allows us to err always on safe side. For example, when we perform *pos* $\oplus$ *neg* operation, the result is an integer number ($\top$) and not either +, 0 or -.

These three conditions guarantees safety and correctness.

5 Conclusions

The main objective of the paper is to illustrate how an abstract interpretation is derived for the imperative programs. The paper demonstrates the same by giving two examples. The methodology used in deriving abstract interpretation is based on the mathematical framework proposed by P. Cousot, paper tries to understand the rationale behind it.

References

[1] Abstract interpretation course slides. http://www.cs.bris.ac.uk/Teaching/Resources/COMS30122/lectures/abstract.p.pdf, 2003.

[2] Abstract interpretation course slides. http://www.cs.ucdavis.edu/ su/teaching/ecs289c/, 2004.

[3] Samson Abramsky. An introduction to abstract interpretation. *Journal of Logic and Computation*, 1(1):5–41, 1990.

[4] Prof Aiken. Abstract interpretation course slides. http://www.cs.berkeley.edu/ necula/294softqual/lectures/lecture02.pdf.

[5] Patrick Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th POPL*, pages 238–252, 1977.

[6] Neil D. Jones. Abstract interpretation : a semantics based tool for program analysis. *Handbook of Logic In Computer Science*.

[7] Arnaud Venet. Abstract interpretation: A methodology for the rapid development of provably correct static analyzers (slides). arnaud@email.arc.nass.gov.

Program Point	Abstract Semantics
1	$$a_1^{i+1} = \beta(a_1^i \cup a_4^i \cup a_6^i)$$
2	$$a_2^{i+1} = \beta(a_2^i \cup \{v \mid v \in a_1^{i+1} \wedge s(x <> 0)True\})$$
3	$$a_3^{i+1} = \beta(a_3^i \cup \{v \mid v \in a_2^{i+1} \wedge s(x < 0) = False\})$$
4	$$a_4^{i+1} = \beta(a_4^i \cup \{v \mid v \in a_3^{i+1} \wedge s(x) = s(x) - 1\})$$
5	$$a_5^{i+1} = \beta(a_5^i \cup \{v \mid v \in a_2^{i+1} \wedge s(x < 0) = True\})$$
6	$$a_6^{i+1} = \beta(a_6^i \cup \{v \mid v \in a_5^{i+1} \wedge s(x) = s(x) * (-1)\})$$
7	$$a_7^{i+1} = \beta(a_7^i \cup \{v \mid v \in a_7^{i+1} \wedge s(x <> 0) = False\})$$

Table 7: **Abstract Semantics of Program 1**