Programmation 1

TD n°14

5 janvier 2021

1 Rule of Sign

Introduction

Let us revisit the rule of sign example from the lecture. We will define an abstract interpretation of a language of integer expressions including only literals, addition, and multiplication. The goal of the abstract interpretation will be to determine whether each (sub)-expression is negative, zero, or positive.
Syntax. Expressions involve literals, addition, and multiplication.

$$\begin{aligned} \mathsf{exp} &\to n && \text{(literals)} \\ &\to \mathsf{exp} + \mathsf{exp} && \text{(addition)} \\ &\to \mathsf{exp} * \mathsf{exp} && \text{(multiplication)} \end{aligned}$$

Standard Interpretation. We define the standard interpretation of expressions using denotational semantics ; i.e., we add the definitions of the valuation functions below.

$$\mathbf{E} : \text{Exp} \to \text{Int}$$

$$\begin{aligned} \mathbf{E}[\![n]\!] &= n \\ \mathbf{E}[\![E1 + E2]\!] &= \mathbf{E}[\![E1]\!] + \mathbf{E}[\![E2]\!] \\ \mathbf{E}[\![E1 * E2]\!] &= \mathbf{E}[\![E1]\!] * \mathbf{E}[\![E2]\!] \end{aligned}$$

Abstract Domain

We define the abstract domain, called Sign, as follows. We want the abstract interpretation to tell us whether an expression is negative, zero, or positive. Note that we cannot always know the sign (for example, a negative plus a positive), hence, we define a "don't-know" value called *num*.

$$\text{Sign} = \{ \text{ zero} , \text{ pos} , \text{ neg} , \text{ num} \}$$

We define the valuation functions for the abstract interpretation in terms of the two tables below, which define abstract addition and multiplication operations.
Here are the abstract valuation functions ; note that the abstract meaning of an integer expression is a Sign.

$$\mathbf{E_{abs}} : \text{Exp} \to \text{Sign}$$

$$\begin{aligned} \mathbf{E_{abs}}[\![n]\!] &= \text{ if } n < 0 \text{ then neg else if } n = 0 \text{ then zero else pos} \\ \mathbf{E_{abs}}[\![E1 + E2]\!] &= \mathbf{E_a bs}[\![E1]\!] \oplus \mathbf{E_{abs}}[\![E2]\!] \\ \mathbf{E_{abs}}[\![E1 * E2]\!] &= \mathbf{E_{abs}}[\![E1]\!] \otimes \mathbf{E_{abs}}[\![E2]\!] \end{aligned}$$

$\oplus$	**neg**	**zero**	**pos**	**num**
neg	neg	neg	num	num
zero	neg	zero	pos	num
pos	num	pos	pos	num
num	num	num	num	num

$\otimes$	**neg**	**zero**	**pos**	**num**
neg	pos	zero	neg	num
zero	zero	zero	zero	zero
pos	neg	zero	pos	num
num	num	zero	num	num

Galois connection

A Galois connection is a pair of functions, α and γ between two partially ordered sets $(C, \subseteq)$ and $(A, \leq)$, such that both of the following hold.

— $\forall a \in A, c \in C : \alpha(c) \leq a$ iff $c \subseteq \gamma(a)$

— $\forall a \in A : \alpha(\gamma(a)) \leq a$

Exercise 1 :

1. Apply the abstract interpretation to the expression $\mathbf{E_{abs}}[\![-22 * (14 + 7)]\!]$.

Solution:

1.

$$\begin{aligned}
\mathbf{E_{abs}}[\![-22 * (14 + 7) &= \mathbf{E_{abs}}[\![-22]\!] \otimes \mathbf{E_{abs}}[\![14 + 7]\!] \\
&= \text{neg} \otimes (\mathbf{E_{abs}}[\![14]\!] \oplus \mathbf{E_{abs}}[\![7]\!] \\
&= \text{neg} \otimes (\text{ pos } \oplus \text{ pos }) \\
&= \text{neg} \otimes \text{ pos} \\
&= \text{neg}
\end{aligned}$$

Relationship between standard and abstract interpretations

1. Define two partially-ordered sets (posets) C and A. The elements of C are all non-empty sets of values from the concrete domain. The elements of A are the values from the abstract domain.
2. Define an **abstraction function** α that maps non-empty sets of concrete values to abstract values (i.e., α is of type $C \to A$).
3. Define a **concretization function** γ that maps abstract values to non-empty sets of concrete values (i.e., γ is of type $A \to C$).
4. Show that α and γ form a Galois connection.
5. For each possible form of expression exp, show that

$$\{\mathbf{E}[\![\text{exp}]\!]\} \subseteq \gamma(\mathbf{E_{abs}}[\![\text{exp}]\!])$$

where $\subseteq$ is the ordering of poset C, i.e., the subset ordering.

Exercise 2 :

For the above example, we now show that the abstract semantics is consistent with the standard semantics. For this, we follow the steps enumerated above.

1. Define the abstraction and the concretization functions.
2. Consider the entire set A and the following elements of the set C : $\{1,2\}$, $\{0\}$, $\{-1,-2\}$, $\{0,1\}$, $\{-1\}$, $\{1\}$, $\{0\}$, set of all negative integers, set of all positive integers, and the set of all integers. Draw the α and γ mappings between these elements.
3. Show that α and γ form a Galois connection.
4. Finally, prove that for every exp,

$$\{\mathbf{E}[\![\text{exp}]\!]\} \subseteq \gamma(\mathbf{E}_{\mathbf{abs}}[\![\text{exp}]\!])$$

5. In what way does proving the above inclusion show that the rule-of-sign abstract interpretation is consistent with the standard semantics ?

Solution:

1. **Abstraction function** α. The abstraction function is defined as follows :

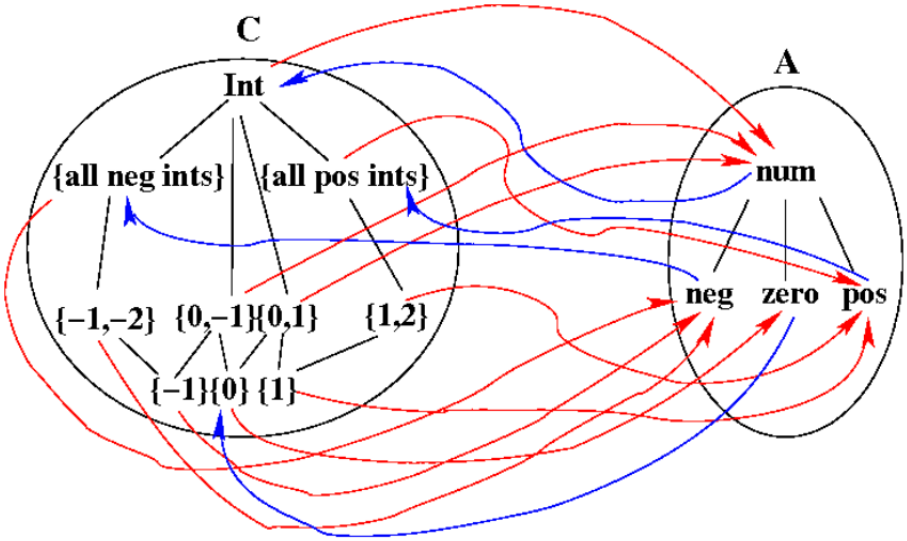
$$\alpha(\{0\}) = \text{ zero}$$

$$\begin{aligned}\alpha(S) = &\text{ if all values in S are greater than 0 then pos}\\ &\text{else if all values in S are less than 0 then neg}\\ &\text{else num}\end{aligned}$$

Concretization function γ is defined as follows :

$$\begin{aligned}\gamma(\text{zero}) &= \{0\}\\ \gamma(\text{pos}) &= \{\text{all positive ints}\}\\ \gamma(\text{neg}) &= \{\text{all negative ints}\}\\ \gamma(\text{num}) &= \text{Int (i.e., all ints)}\end{aligned}$$

2. The figure is below (α is indicated by the red arrows, and γ by the blue).



3. First, we will show that Relationship 1 holds : given an arbitrary concrete element c (a set of int values), for every abstract element a such that a $> \alpha$(c), c $\subseteq \gamma$(a). We will do this using cases on concrete element c. Case 1 : c contains only positive ints is shown below (the rest are straightforward extensions)

   α(c) = pos // def of alpha
   the elements of A that are $\geq$ pos are {pos, num} // def of the ordering of A
   γ(pos) = { all positive ints } // def of gamma
   c $\subseteq$ { all positive ints } // def of the (subset) ordering of C
   γ(num) = { all ints } // def of gamma
   $c \subseteq$ { all ints } // def of the (subset) ordering of C

   We now show that Relationship 2 holds : given an arbitrary abstract element a, $\alpha(\gamma(\text{a})) \leq$ a. We will do this using cases on abstract element a. Here is the first case : Case 1 : a is num. (The rest follow).
   In this case, γ(a) = { set of all ints } (by definition of gamma), and $\alpha(\gamma(\text{a}))$ = num (by definition of alpha), which is the same as a.

4. This can be done using structural induction. Base case : exp is literal k. This case has three parts (based on the definition of $\mathbf{E_{abs}}$). We show one part :
   $k < 0$: In this case,

   $\mathbf{E}[\![k]\!] = k$ // def of $\mathbf{E}$
   $\mathbf{E_{abs}}[\![k]\!] =$ neg // def of $\mathbf{E_{abs}}$
   γ(neg) = { all negative ints } // def of γ
   and $\{k\}$ is a subset of {all negative ints} (case proved).

   Inductive Step
   The inductive step is quite tedious. There are two cases (one for addition and one for multiplication), and each has 16 sub-cases (for all possible combinations of the signs of the two sub-expressions). Here is one example to show the flavour of the proof.
   Inductive case 1 : *exp* is $e1 + e2$.

   RHS $\gamma(\mathbf{E_{abs}}[\![e1 + e2]\!])$
   $= \gamma(\mathbf{E_{abs}}[\![e1]\!] \oplus \mathbf{E_{abs}}[\![e2]\!])$ // def of $\mathbf{E_{abs}}$

   sub-case 1 : both $\mathbf{E_{abs}}[\![e1]\!]$ and $\mathbf{E_{abs}}[\![e2]\!]$ are neg.
   $= \gamma(\text{neg} \oplus \text{neg})$
   $= \gamma(\text{neg})$ // def of $\oplus$
   = { all negative ints } // def of γ

   LHS $\mathbf{E}[\![e1 + e2]\!]$
   $= \mathbf{E}[\![e1]\!] + \mathbf{E}[\![e2]\!]$ // def of $\mathbf{E}$

   By the induction hypothesis, $\mathbf{E}[\![e1]\!]$ is a subset of $\gamma(\mathbf{E_{abs}}[\![e1]\!])$, which is γ(neg), which is the set of all negative ints. The same applies to $\mathbf{E}[\![e2]\!]$. Thus, the LHS is the sum of two negative ints, which is a negative int, which is certainly a subset of all negative ints (the final value for the RHS).

5. The rule-of-sign abstract interpretation of an expression exp is the result of applying the abstract valuation function to exp : $\mathbf{E_{abs}}[\![exp]\!]$ .This gives us some abstract value a. Applying gamma to a gives us some concrete value, S, a set of (integer) values.

   The actual meaning of the expression is the result of applying the standard evaluation function to exp : $\mathbf{E}[\![exp]\!]$. This produces a single (integer) value c – the value of the expression.

   What might be the relationship between c and S, and which of these possible relationships would we consider to be OK, i.e., in which cases would we say that our abstract interpretation is sound, though possibly imprecise ?

   — S = { c } In this case, the abstract interpretation is sound and precise.

   — c ∈ S In this case, the abstract interpretation is sound, but not precise.

   — not (c ∈ S) In this case, the abstract interpretation is not sound.

   $\{\mathbf{E}[\![exp]\!]\} \subseteq \gamma(\mathbf{E_{abs}}[\![exp]\!])$ only for cases 1 and 2. Therefore, if we can show that this relationship holds, we have shown that our abstract interpretation is consistent with the standard semantics.

2 Standard and Collecting Semantics for CFGs

For more realistic static-analysis problems, however, the standard denotational semantics is usually not a good place to start. This is because we usually want the results of static analysis to tell us what holds at each point in the program, and program points are usually defined to be the nodes of the program's control-flow graph (CFG).

For example, for constant propagation (process of substituting the values of known constants in expressions at compile time) we want to know, for each CFG node, which variables are guaranteed to have constant values when execution reaches that node. Therefore, it is better to start with a (standard) semantics defined in terms of a CFG.

Standard Semantics

The most straightforward is to define an operational semantics ; think of it as an interpreter whose input is the entry node of a CFG plus an initial state (a mapping from variables to values), and whose output is the program's final state. We'll define the standard sementics in terms of transfer functions, one for each CFG node. These are (semantic) functions whose inputs are states and whose outputs are pairs that include both an output state and the CFG node that is the appropriate successor. A node's transfer function captures the execution semantics of that node and specifies the next node to be executed.

Consider the CFG given in the diagram.

For this example, the transfer function for node 2 would be defined as follows :

$$\lambda s.(s[a \leftarrow 1], 3)$$

where $s[a \leftarrow 1]$ means "a new state that is the same as s except that it maps variable a to 1."

Exercise 3 :

1. What is the transfer function for the node 4 ?

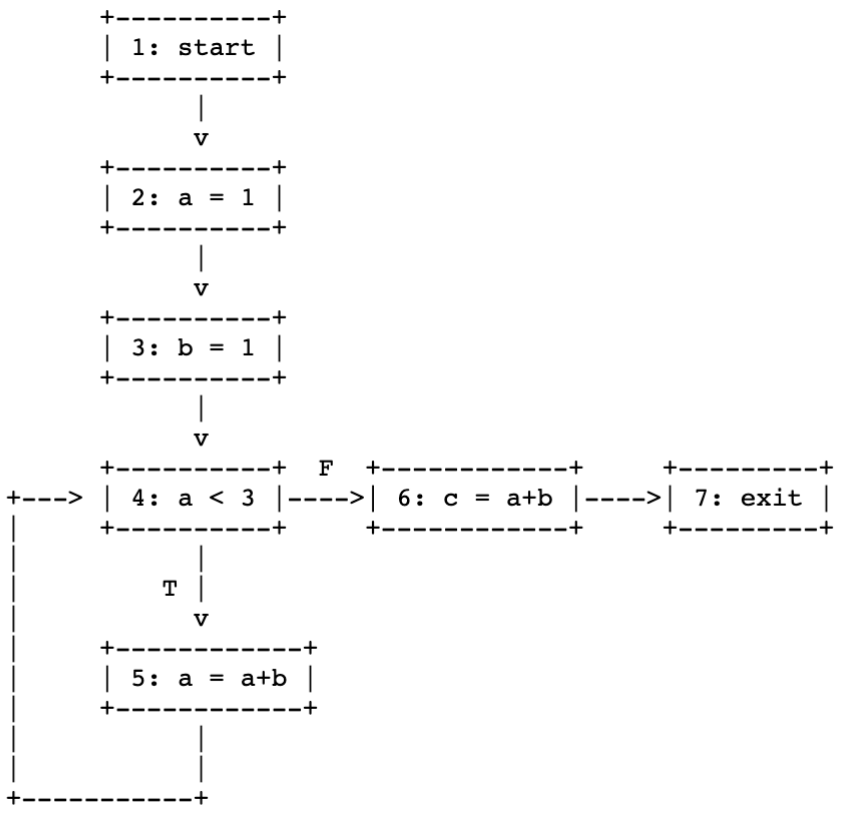


FIGURE 1 – Example CFG

Solution:

1. For node 4, the transfer function would be

$$\lambda s.(\text{if lookup}(s, a) < 3 \text{ then } (s, 5) \text{ else } (s, 6))$$

Operational semantics

Here is a (recursive) definition of the interpreter (the operational semantics). We use f_n to mean the transfer function defined for CFG node n.

$$\begin{aligned}\text{interp} = \lambda s.\lambda n.\ &\text{if isExitNode}(n) \text{ then } s \\ &\text{else let } (s', n') = f_n(s) \text{ in interp } s'n'\end{aligned}$$

Because this definition is recursive, we need to use the usual trick of abstracting on the function and defining the operational semantics as the least fixed point of that abstraction :

$$\begin{aligned}\text{semantics } = \text{fix}(\lambda F.\lambda s.\lambda n.\ &\text{if isExitNode}(n) \text{ then } s \\ &\text{else let } (s', n') = f_n(s) \text{ in F } s'n')\end{aligned}$$

Collecting Semantics

While the operational semantics discussed above is defined in terms of the program's CFG, it has two properties that are undesirable as the basis for an abstract interpretation :

1. It is still just a function from a program's input state to its final state ; the result of applying the operational semantics tells us nothing about the intermediate states that arise at each CFG node.
2. It maps a particular initial state to the corresponding final state. We want a semantics that tells us what can happen for every possible initial state.

To obtain a guarantee about the relationship between the program's semantics and the analysis results, we need a semantics that includes information about the set of states that can arise at each CFG node given any possible initial state. That kind of semantics is called a collecting semantics.
We will define a collecting semantics that maps CFG nodes to sets of states ; i.e., for each CFG node n, the collecting semantics tells us what states can arise just before n is executed. The "approximate semantics" that we define using abstract interpretation will compute, for each CFG node, (a finite representation of) a superset of the set of states computed for that node by the collecting semantics. By showing that our abstract interpretation really does compute a superset of the possible states that can arise at each CFG node, we show that it is consistent with the program's actual semantics.
Because the collecting semantics involves sets of states, we need to define transfer functions whose inputs and outputs are sets of states. We'll define one function $f_n \to m$ for each CFG edge $n \to m$. That transfer function will be defined in terms of the (original) transfer function f_n defined for the CFG node n :

$$f_{n\to m} = \lambda S.\{s' \mid s \in S \text{ and } f_n(s) = (s', m)\}$$

For example, the transfer function for edge$2 \to 3$ of the example CFG given above would be defined as follows :

$$\lambda S.\{s[a \leftarrow 1] \mid s \in S\}$$

Exercise 4 :

1. What is the transfer function (for the collecting semantics) for edge $4 \to 5$ of the example CFG ?
2. The collecting semantics will be of type CFG-node $\to$ set-of-states. It defines the set of states that holds just before node n to be the union of the sets of states produced by applying the transfer functions of all of n's in-edges to the sets of states that hold just before the sources of those in-edges execute. Write the non-recursive definition for it, if the recursive definition is given below.

$$\begin{aligned}\text{recColl } = \lambda n.\ &\text{if isEnterNode}(n) \text{ then } \{ \text{ all states } \} \\ &\text{else let } P = \text{ preds}(n) \text{ in } \cup p \in P f_{p\to n}(\text{ recColl}(p))\end{aligned}$$

Solution:

1. $\lambda S.\{s \mid s \in S \text{ and lookup}(s, a) < 3\}$.
2. $$\begin{aligned}\text{coll} = \text{fix}(\lambda F.\lambda n.\ &\text{if isEnterNode}(n) \text{ then } \{ \text{ all states } \} \\ &\text{else let } P = \text{preds}(n) \text{ in } \cup_{p\in P} f_{p\to n}(F(p))\end{aligned}$$

Abstract Interpretation

To define an abstract interpretation we need to do the following :

1. Define the abstract domain A, the abstraction function α, and the concretization function γ.
2. Show that α and γ form a Galois connection.
3. For each CFG edge $n \to m$, define an abstract transfer function $f\#_{n\to m}$.
4. Show that the abstract transfer functions are consistent with the concrete ones ; i.e., for each abstract $f\#$ and corresponding concrete f :
   (a) start with an arbitrary concrete-domain item c
   (b) let $c' = f(c)$
   (c) let $a = \alpha(c)$
   (d) let $a' = f\#(a)$
   (e) let $c'' = \gamma(a')$
   (f) show that $c' \subseteq c''$

Given an abstract interpretation, we can define the abstract semantics recursively or non-recursively, as we did for the collecting semantics. The definitions given below define the abstract semantics as a mapping CFG-node → abstract state. The abstract state that holds at CFG node n (a safe approximation to the set of concrete states that hold just before n executes) is the join of the abstract states produced by applying the abstract transfer functions of all of node n's incoming CFG edges to the abstract states that hold before those edges' source nodes.

$$\begin{aligned}\text{recAbs} = \lambda n.\ &\text{if isEnterNode}(n) \text{ then } \alpha(\{\text{all states}\}) \\ &\text{else let } P = \text{preds}(n) \text{ in} \\ &\cup_{p\in P}\, f\#_{p\to n}(\,\text{recAbs}(p))\end{aligned}$$

And here's the non-recursive definition :

$$\begin{aligned}\text{abs} = \text{fix}(\lambda F.\lambda n.\ &\text{if isEnterNode}(n) \text{ then } \alpha(\{\text{ all states }\}) \\ &\text{else let } P = \text{preds}(n) \text{ in} \\ &\cup_{p\in P}\, f\#_{p\to n}(F(p)))\end{aligned}$$

Exercise 5 :

We now adapt it to the example of constant propagation. The elements of the abstract domain A are abstract states that map variables to values, including the special value ? (which means that the corresponding set of concrete states includes states that map the variable to different values). The abstract domain also includes a special bottom element $\bot$. The ordering of the abstract domain is based on the underlying flat ordering of individual values in which ? is the top element, and all other values are incomparable. Given two abstract states, a_1 and a_2, $a_1 \leq a_2$ iff

— a_1 is $\bot$, or

— every variable x mapped to a non-? value in a_2 is mapped to the same value in a_1.

The concrete domain is the one defined earlier, whose elements are sets of states (each with a value for every variable), and whose ordering is subset (i.e., $S_1 \subseteq S_2$ iff S_1 is a subset of S_2).

1. What are the abstraction and concretization functions ?
2. Show that they form a Galois connection.
3. Define the abstract transfer functions.

Solution:

1. The abstraction function maps the empty set to $\bot$; it maps every non-empty set S of concrete states to a single abstract state : For each variable x, if x has the same value v in every concrete state in S, then it is mapped to v in the abstract state. Otherwise, it is mapped to the special value ?.

   The concretization function is the obvious dual of the abstraction function : It maps $\bot$ to the empty set. Given an abstract state a, if no variable is mapped to ? in a, then the concrete set of states S contains just one state a. Otherwise, S is an infinite set. For every variable x that is mapped to a non-? value v in a, every state s in S maps x to v ; all other variables are mapped to all possible combinations of values.

2. Clear from the definition.

3. The transfer function $f\#_{n\to m}$ is defined as follows :
   — If node n doesn't modify any variables, then $f_{n\to m}$ is the identity function.
   — If node n represents x = y + z, then $f_{n\to m}$ is defined as follows :

   $$\lambda s.s[x \leftarrow \text{ lookup}(s, y) \oplus \text{ lookup}(s, z)]$$

   where s is an abstract state, and $\oplus$ returns ? if either of its arguments is ? (and otherwise is the same as regular +).