Chapter 9

From a Concrete Interpreter to an Abstract Interpreter

A mathematical view of a concrete interpreter.

Let us look at the interpreter from the previous chapter. It proceeds in small steps, as follows (cf., Fig. 9.1):

- It takes a state of a program and computes the following state
- every state consists of a program counter, as well as an environment binding variable names to concrete values (in this case integers). The environment also contains a stack of concrete values.
- In order to compute the following state the interpreter applies some concrete operations such as addition, multiplication, equality checks.
- The overall interpreter proceeds computing these small steps until the program terminates. For every state there is a unique successor state.

We have a set of possible program states St, made up of values from a concreted domain $\mathcal{C}$, a set of possible initial states $S_0 \subseteq St$, we have for a program P a semantic function $succ_P : St \mapsto St$, which given a state computes the immediate successor state. Let us say we are interested in the set R of program states reachable from S_0, e.g., to validate or infer a certain property about our program P. We can formally define R as follows:

- $R = F_P \uparrow^{\infty} (\emptyset)$, where
- $F_P(S) = S_0 \cup S \cup succ^*_P(S)$ and where
- f^* is the lifted version of a function f, defined by : $f^*(S) = \{f(s) \mid s \in S\}$.

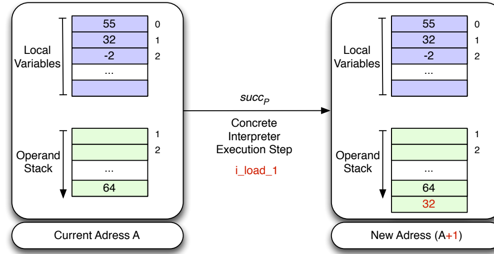


Figure 9.1: Concrete Interpreter Step (Java bytecode example)

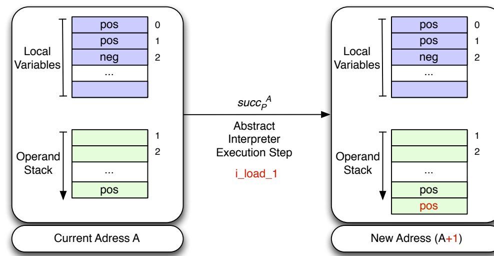


Figure 9.2: Abstract Interpreter Step

- $f \uparrow^0 (S) = S$ and $f \uparrow^{n+1} (S) = f(f \uparrow^n (S))$.

For our Java Bytecode interpreter, a program state could be a triple (PC, S, V), consisting of a program counter PC, the sequence of values S representing the stack, and the set of local variable bindings V. The initial state would be $S_0 = \{(0, \langle\rangle, \emptyset)\}$. After executing `iconst_2` we would have the state $succ^*_P(S_0) = S_1 = \{(1, \langle 2 \rangle, \emptyset)\}$, and after executing the second instruction `istore_1` we would obtain the state $succ^*_P(S_1) = \{(2, \langle\rangle, \{1 \mapsto 2\})\}$. Note that $F_P(S_0) = S_0 \cup S_1$.

Note that for a method with parameters, we would have a larger initial set of states, e.g., $S_0 = \{(0, \langle\rangle, \{1 \mapsto x\}) \mid x \in -2^{63}..2^{63}-1\}$ for a method with one integer parameter.

Observe that F_P is monotonic: $S \subseteq S' \Rightarrow F_P(S) \subseteq F_P(S')$. Hence, F_P has by the Knaster& Tarski Fixpoint theorem a least fixpoint $lfp(F_P)$. Observe that F_P is also finitary (and hence continuous), in the sense that for every infinite sequence $I_0 \subseteq I_1 \subseteq \ldots$ we have $F_P(\bigcup_{n=0}^{\infty} I_n) \subseteq \bigcup_{n=0}^{\infty} F_P(I_n)$. Hence, $lfp(F_P) = R$.

The problem is that the above approach may of course fail to stabilise in finite time (i.e., for no number n do we have that $F_P \uparrow^n (\emptyset) = F_P \uparrow^{n+1} (\emptyset) = F_P \uparrow^{\infty} (\emptyset)$. Furthermore, even if it does stabilise can be very expensive (basically running the program on all possible inputs), and depending on the concrete domain, the result R maybe large.

The idea of abstract interpretation is to replace the set of concrete values by

a set of abstract values, in order to ensure that the above process always terminates and provides finite representations of all possible program behaviours. In particular, the state of an abstract interpreter would contain abstract values rather than concrete values. Also, instead of applying concrete operations, the abstract interpreter applies abstract counterparts of these operations.

Let us take an example: the interpreter from the previous chapter operated on the concrete set $\mathcal{C}$ of integer values. A possible abstract set of values is $\mathcal{A} = \{\mathbf{0}, Z^+, Z^-, \top\}$.

To give these abstract values a meaning, abstract interpretation introduces two functions:

- The concretisation function $\gamma : \mathcal{A} \mapsto 2^{\mathcal{C}}$
- The abstraction function $\alpha : 2^{\mathcal{C}} \mapsto \mathcal{A}$

In our case we define $\gamma(\mathbf{0}) = \{0\}$, $\gamma(Z^-) = MININT.. - 1$, $\gamma(Z^+) = 1..MAXINT$, and $\gamma(\top) = \mathcal{C}$. The function γ also induces a partial order on $\mathcal{A}$:

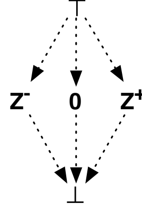
$$a_1 \sqsubseteq a_2 \textit{ iff } \gamma(a_1) \subseteq \gamma(a_2)$$

We also write

$$a_1 \sqsubset a_2 \textit{ iff } a_1 \sqsubseteq a_2 \wedge a_2 \not\sqsubseteq a_1$$

Hence, in our case, $\mathbf{0} \sqsubset \top$, $Z^+ \sqsubset \top$, $Z^- \sqsubset \top$.

Quite often, one also works with a bottom element $\bot$ such that $\gamma(\bot) = \emptyset$. We thus have, $\bot \sqsubset \mathbf{0}$, $\bot \sqsubset Z^+$, $\bot \sqsubset Z^-$ and $\bot \sqsubset \top$, illustrated graphically as follows (the arrow from $\top$ to $\bot$ implied by transitivity is not shown):



For correctness it is required that for any $S \subseteq \mathcal{C}$ we have $\gamma(\alpha(S)) \supseteq S$. Quite often one defines $\alpha(S)$ from γ by requiring that $\alpha(S)$ is the most precise element of $\mathcal{A}$ such that $\gamma(\alpha(S)) \supseteq S$. In our case, $\alpha(\{1, 3, 5, 7\})$ could be either Z^+ or $\top$, as $\gamma(Z^+) \supseteq \{1, 3, 5, 7\}$ and $\gamma(\top) \supseteq \{1, 3, 5, 7\}$. However, the most precise abstraction is Z^+, as $Z^+ \sqsubset \top$.

In order to merge two abstract values, we introduce the following operation $\sqcup$ (also called *least upper bound*, lub), which is the counterpart of the set union at the concrete level.

Definition 9.0.2 First, b is called an upper bound of a_1 and a_2 iff $b \sqsupseteq a_1$ and $b \sqsupseteq a_2$. Then $a_1 \sqcup a_2$ denotes the least upper bound of a_1 and a_2, i.e., it is smaller ($\sqsubseteq$) than all upper bounds of a_1 and a_2.

The least upper bound is unique if it exists. From now on, we will suppose it exists (which is the case if our abstract domain is a semi-lattice). By $\sqcup_S$ we denote the least upper bound of a set of abstract elements.

Definition 9.0.3 Let f be an n-ary function on the concrete domain, i.e., $f : \mathcal{C} \times \ldots \times \mathcal{C} \mapsto \mathcal{C}$. Let f_a be an n-ary function on the abstract domain, i.e., $f_a : \mathcal{A} \times \ldots \times \mathcal{A} \mapsto \mathcal{A}$. We say that f_a is a *safe approximation* of f iff $\forall a_1, \ldots, a_n$ we have $\gamma(f_a(a_1, \ldots, a_n)) \supseteq \{f(c_1, \ldots, c_n) \mid c_1 \in \gamma(a_1) \wedge \ldots c_n \in \gamma(a_n)\}$.

In our case this concept can be applied both to the small step function $succ_P$, to the semantic function F_P, or to more elementary operations such as multiplication employed by $succ_P$. Indeed, to obtain a safe approximation of F_P, we need a safe approximation of $succ_P$, for which we need in turn a safe approximation of the elementary operations (such as addition, multiplication,...).

Let F_P^A be a safe approximation of F_P. Abstract interpretation consists in computing $R^A = F_P^A \uparrow^\infty (\bot)$, where $\bot$ is such that $\gamma(\bot) = \emptyset$. Under certain conditions we have that $\mathit{lfp}(F_P) \subseteq \gamma(\mathit{lfp}(F_P^A))$. In other words, the least fix point obtained by abstract interpretation is a safe approximation of all possible reachable states (the concrete semantics). In still other words, any property that is true for all elements of the abstract semantics ($\mathit{lfp}(F_P^A)$) will be true for *every* possible concrete execution at runtime. For example, if abstract interpretation determines that a certain variable is positive at a certain program point, then it is *guaranteed* that for every possible concrete execution at runtime the variable is indeed positive at runtime at that program point.

Usually, the concrete semantics is infinite (i.e., the set $\mathit{lfp}(F_P)$ is infinite), or very large. Luckily, for well chosen abstract domains, the construction $F_P^A \uparrow^\infty (\bot)$ will stabilise after some finite number of steps and produce a finite approximation of the infinite concrete semantics. This is indeed the reason of existence of abstract interpretation.

Exercise 9.0.4 Write abstract versions of addition, subtraction, multiplication, division, modulo for our abstract domain. Verify that you have developed a safe approximation.

Exercise 9.0.5 Use the abstract domain to approximate the possible values of a and b after executing the following piece of pseudo-code. Suppose that a and b are strictly positive upon entry.

```
a := a+b;
b := b+1;
a := a*b;
```

What about the following piece of code:

```
a := a+b;
b := b-1;
a := a*b;
```

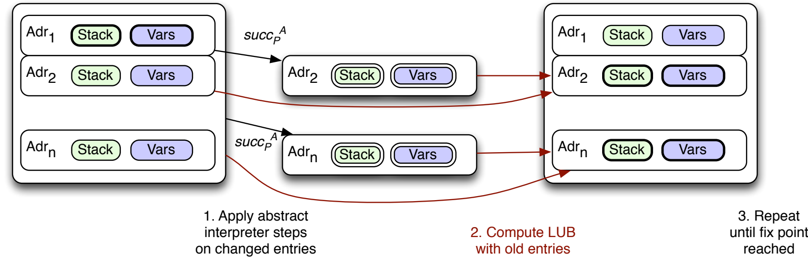


Figure 9.3: Abstract Interpretation Scheme for Java bytecode interpreter

How could one improve the situation, i.e., obtain a more accurate result?

Exercise 9.0.6 Use the abstract domain to approximate the possible values of a and b during and after executing the following piece of pseudo-code. Suppose that a and b are strictly positive upon entry.

```
while b>=0 do
  a := a+b;
  b := b-1
od
a := a*b
```

What has changed wrt the previous exercise?

9.1 A schema to compute $lfp(F_P^A)$

We need to develop an abstract version of the semantic function F_P^A. In our case, having an interpreter which computes *succ*, we can start as follows: $F_P^A(S) = succ_P^A(S) \sqcup \alpha(S_0)$ and where $succ_P^A$ is a safe approximation of *succ*.

Exercise 9.1.1 (We could add $\sqcup S$ to $F_P^A(S)$. Does it change anything ?)

In order to define $succ_P^A$ we need to look at the concrete state (in our case `PC * seq(INT) * (Var +-> INT)`) and decide which parts of it will be abstracted, and how. A typical approach would be to not abstract the program counter (after all there are only finitely many program points), and not to abstract the "structure" of the stack/environment but only abstract the concrete datavalues. We would also provide a special abstract value for program points which have not been reached (bottom). I.e, an abstract state would be a set of elements from either `PC *` $\perp$ or `PC * (seq(A) * (Var +-> A))`. Hence, $succ^A$ is basically the interpreter as before, but:

- using elements of the concrete data domain $\mathcal{A}$ rather than $\mathcal{C}$

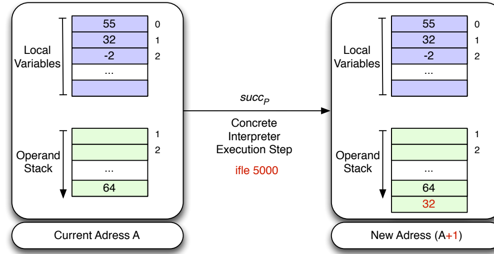


Figure 9.4: Concrete Interpreter Step (Java bytecode example)

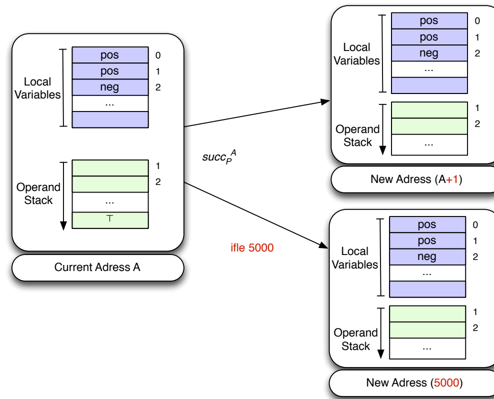


Figure 9.5: Abstract Interpreter Step

- using abstract operations on $\mathcal{A}$, which are safe approximations of the concrete counterparts for $\mathcal{C}$ in *succ*

This will guarantee that F_P^A is a safe approximation of F_P, and hence that the result of abstract interpretation is sound.

[We actually use a scheme where we have a set of abstract values, one for each program point. See Figure 9.3. An alternative view, is that a single abstract value is a mapping from program points to abstract stacks and abstract local variable environments.]

Note that, while *succ* is deterministic, our Prolog version of $succ^A$ sometimes has to be non-deterministic (given that we do not abstract the program counter).[1] E.g., $if\ 2 < 3\ goto\ 10$ is deterministic, $if\ pos\ <\ pos\ goto\ 10$ needs to explore both alternatives. See also Figure 9.5 for another example on the Java Bytecode interpreter.

[1]This means we deviate from the pure abstract interpretation framework as described earlier, and allow sets of abstract values.

In order to compute *lfp*(F_P^A) we can simply proceed as follows:

- keep a table which for every program point tracks the abstract information about the possible environments at that program point. This can be most easily done with a dynamic fact database. The fact database would be initialised with the empty relation, signifying that initially no program point is reachable (this corresponds to setting the abstract environment to $\perp$ for all program points).
- when reaching a program point:
  - if this program point was not reached before: simply store the current abstract environment in the table and proceed normally
  - if this program point was reached before: we need to merge the previous and new abstract environment. This is the LUB (Least Upper Bound) operation. We then need to check if the LUB is different from the currently store abstract environment. If it is: we need to update the table and proceed with the abstract interpretation. If not, we can simply return.

Note: because of the non-determinism, we need an outer loop to drive the interpreter until no more updates to the table occur. After that the table contains the representation of *lfp*(F_P^A).

First, implementing the least upper bound for abstract values:

```
/* Abstract Domain:
   0, pos, neg, top
lub(X,X,R) :- !,R=X.
lub(X,Y,top) :- X\=Y.
```

Now, implementing abstract operations for arithmetic operators:

```
aex_op(*,0,_,0).
aex_op(*,pos,X,X).
aex_op(*,neg,0,0).
aex_op(*,neg,pos,neg).
aex_op(*,neg,neg,pos).
aex_op(*,neg,top,top).
aex_op(*,top,0,0).
aex_op(*,top,X,top) :- X\=0.

aex_op(+,0,X,X).
aex_op(+,pos,0,pos).
aex_op(+,pos,pos,pos).
aex_op(+,pos,neg,top).
aex_op(+,pos,top,top).
aex_op(+,neg,0,neg).
aex_op(+,neg,pos,top).
```

```
aex_op(+,neg,neg,neg).
aex_op(+,neg,top,top).
aex_op(+,top,_,top).
% - missing
% TO DO for homework
```

Now, implementing boolean test predicates:

```
% Redefinition of predicate from javabc_interpreter.pl:
test_op(<=,X,X).
test_op(<=,top,X) :- X \= top.
test_op(<=,neg,X) :- X \= neg.
test_op(<=,0,pos).
test_op(<=,0,top).
test_op(<=,pos,top).

test_op(<,top,_).
test_op(<,neg,_).
test_op(<,0,pos).
test_op(<,0,top).
test_op(<,pos,pos).
test_op(<,pos,top).

test_op(>=,X,Y) :- test_op(<=,Y,X).
test_op(>,X,Y) :- test_op(<,Y,X).


% Redefinition of predicate from javabc_interpreter.pl:
false_op(<,A1,A2) :- test_op(>=,A1,A2).
false_op(>,A1,A2) :- test_op(<=,A1,A2).
false_op(<=,A1,A2) :- test_op(>,A1,A2).
false_op(>=,A1,A2) :- test_op(<,A1,A2).
```

Note: here we become non-deterministic, for some combinations of abstract values both `test_op` and `false_op` succeed. Success means: this test could be true (resp. false) for some concrete values.

Now, for the rest of the code:

```
aint :- print('Starting ABSTRACT Interpretation'),nl,
        execute(_X),fail.
aint :- print('FIXPOINT reached'),nl,
        print('ABSTRACT INFORMATION AT PROGRAM POINTS: '),nl,
        memo(PC,AEnv),
        print(PC), print(' : '), print(AEnv),nl,
        fail.
aint :- nl,flush_output(user).

:- dynamic memo/2, sol/2.
```

```
% Redefinition of predicate from javabc_interpreter.pl:
init_env(env([],[])) :- retractall(memo(_,_)).

% Redefinition of predicate from javabc_interpreter.pl:
interpreter_loop(PC,In,_Out) :- print(PC), print(': '),
   lub_program_point(PC,In,AIn,Change),
   (Change=true ->
      instr(PC,Opcode,Size),
      NextPC is PC+Size,
      print(Opcode),print(' '), print(AIn),nl,
      ex_opcode(Opcode,NextPC,AIn,_)
    ; print(fix(AIn)),nl
   ).


lub_program_point(PC,env(S,Vars),Res,Change) :-
  (retract(memo(PC,env(SM,VarsM)))
     -> lub_stack(S,SM,SR),
        lub_local_vars(Vars,VarsM,VarsR),
        Res = env(SR,VarsR), assert(memo(PC,Res)),
        (Res=env(SM,VarsM) -> Change=false
           ; print(' <REANALYZE> '),Change = true)
     ;  assert(memo(PC,env(S,Vars))),
        Res = env(S,Vars),
        Change=true
  ).

lub_stack([],[],[]).
lub_stack([],[_|_],_) :- print('*** Illegal bytecode: Varying stack pattern at PC'),nl,fail.
lub_stack([_|_],[],_) :- print('*** Illegal bytecode: Varying stack pattern at PC'),nl,fail.
lub_stack([H1|T1],[H2|T2], [H3|T3]) :-
   lub(H1,H2,H3),
   lub_stack(T1,T2,T3).

lub_local_vars([],X,X).
lub_local_vars([H|T],[],[H|T]).
lub_local_vars([Key/Val1|T1],[Key/Val2|T2],[Key/Val3|T3]) :- !,
   lub(Val1,Val2,Val3),
   lub_local_vars(T1,T2,T3).
lub_local_vars([Key1/Val1|T1],[Key2/Val2|T2],[H|T3]) :- Key1 \= Key2,
  (Key1 @<Key2
    -> (H=Key1/Val1, lub_local_vars(T1,[Key2/Val2|T2],T3))
    ;  (H=Key2/Val2, lub_local_vars([Key1/Val1|T1],T2,T3))
  ).


push(env(S,Vars),Value,env([AV|S],Vars)) :-
   abstract_value(Value,AV).
```

```
store(env(Stack,Vars),Key,Value,env(Stack,NVars)) :-
    abstract_value(Value,AV),
    update(Vars,Key,AV,NVars).

abstract_value(X,AV) :- number(X),!,
   (X=0 -> AV=0
     ; (X>0 -> AV = pos ; AV = neg)
   ).
abstract_value(X,X).

% Redefinition of predicate from javabc_interpreter.pl:
ex_op(OP,A1,A2,R) :- abstract_value(A1,AV1),
   abstract_value(A2,AV2),
   aex_op(OP,AV1,AV2,R).
```

Exercise 9.1.2 Adapt an interpreter from Chapter 7 to perform abstract interpretation.

Exercise 9.1.3 Adapt the above interpreter to work on a more precise domain, adding the abstract values Z_0^+ and Z_0^-.

For abstract domains with infinite ascending chains, it is necessary to include a widening step in the abstract interpretation procedure. E.g., in Figure 9.3 one would after step 2. optionally abstract the entries further, to avoid an infinite series of abstract states without ever reaching a fix point.

Below is a sample output of our abstract interpreter. For every program point, we obtain information about the operand stack and the local variables, where `top` denotes the abstract value which represents every possible value, `pos` only denotes strictly positive values.

```
Starting ABSTRACT Interpretation
FIXPOINT reached
ABSTRACT INFORMATION AT PROGRAM POINTS:
0 :  iconst(2) :   env([],       [])
1 :  istore(1) :   env([pos],    [])
2 :  iconst(5) :   env([],       [1/pos])
3 :  istore(2) :   env([pos],    [1/pos])
4 :  iload(2) :    env([],       [1/pos,2/pos])
5 :  istore(3) :   env([pos],    [1/pos,2/pos])
6 :  iconst(1) :   env([],       [1/pos,2/pos,3/pos])
7 :  istore(4) :   env([pos],    [1/pos,2/pos,3/pos])
9 :  iload(3) :    env([],       [1/pos,2/pos,3/top,4/pos])
10 : if1(<=,0,25) :env([top],    [1/pos,2/pos,3/top,4/pos])
13 : iinc(3,-1) :  env([],       [1/pos,2/pos,3/top,4/pos])
16 : iload(4) :    env([],       [1/pos,2/pos,3/top,4/pos])
18 : iload(1) :    env([pos],    [1/pos,2/pos,3/top,4/pos])
19 : iop(*) :      env([pos,pos],[1/pos,2/pos,3/top,4/pos])
20 : istore(4) :   env([pos],    [1/pos,2/pos,3/top,4/pos])
22 : goto(9) :     env([],       [1/pos,2/pos,3/top,4/pos])
25 : println(4) :  env([],       [1/pos,2/pos,3/top,4/pos])
27 : return :      env([],       [1/pos,2/pos,3/top,4/pos])
```

As can be seen, we have inferred for every program point the layout of the stack (the first part of the `env` term). For example for program point 19, performing the `imul` instruction, we have as stack layout `[pos,pos]`, i.e., there are exactly two values on the stack (which are also guaranteed to be positive). For compilation this means that when generating the compiled code for program point 19 we know exactly how the stack looks like and exactly from which memory location we need to take the two operands for the multiplication.

- when performing abstract interpretation, non-determinism arises naturally: because of the abstraction one can often not decide whether a test will fail or succeed. As such, Prolog is a good language to encode abstract interpreters in, even if we analyse deterministic, imperative languages.

Exercise 9.1.4 Add support for equality and disequality in `test_op` and `false_op`.

Exercise 9.1.5 Add a back-end to the abstract interpreter, to generate three-address code from the Java Bytecode zero-address code, by making use of the inferred stack layout.

9.2 Improving Precision

After a test is taken, we can often make an abstract value more precise. E.g., if we have $x > 0$ and x has as abstract value $\top$, then the test can succeed at runtime. If it does, we can narrow down x to be of type Z^+.

Exercise 9.2.1 Improve the abstract interpreter to perform this optimisation.

Hint: `test_op` will now take two extra arguments, the new abstract value of the operands. For example, for the $<$ operator, we have:

```
test_op(<,top,0,neg,0).
test_op(<,top,neg,neg,neg).
test_op(<,top,pos,top,pos).
test_op(<,top,top,top,top).
test_op(<,neg,X,neg,X).
test_op(<,0,top,0,pos).
test_op(<,0,pos,0,pos).
test_op(<,pos,pos,pos,pos).
test_op(<,pos,top,pos,pos).
```

You will then have to adapt `if_then_else` to store the changes.

9.3 Other Domains

Constant Propagation.

Domain: constants, + nac (not a constant; equivalent to $\top$).

With respect to the previous abstract interpreter, we only need to update the definitions of

1. the least upper bound `lub/3`.
2. the binary boolean comparators `test_op/3`, and `false_op/3`
3. the arithmetic operators `ex_op/4`.
4. the abstraction of concrete values `abstract_value/2`.

```
lub(X,X,R) :- !,R=X.
lub(_X,_Y,nac).

% Redefinition of predicate from javabc_interpreter.pl:
test_op(OP,X,Y) :- number(X),number(Y),test_op_concrete(OP,X,Y).
test_op(_OP,X,Y) :- \+ number(X) ; \+ number(Y).


% Redefinition of predicate from javabc_interpreter.pl:
false_op(OP,X,Y) :- number(X),number(Y),false_op_concrete(OP,X,Y).
false_op(_OP,X,Y) :- \+ number(X) ; \+ number(Y).

abstract_value(X,X).

ex_op(OP,X1,X2,R) :- %print(ex_op(OP,X1,X2,R)),nl,
       number(X1), number(X2), ex_op_concrete(OP,X1,X2,R).
ex_op(_,_,_,nac) :- \+ number(X) ; \+ number(Y).
```

9.3.1 Related Work

The CLIP group from the Technical University of Madrid has developed a class file loader Prolog library and analyse Java bytecode using their CiaoPP abstract interpretation engine; see, for example, [4, 3, 2, 35]. Another related work is [30] as well as part of [44]. Also, since Java SE 6 (version 50.0 of the class file format), class files now also contain information about the stack layout (see, e.g., Section 4.8.4 of [17]). Note that Section 4.11 of [17] contains Prolog code as a specification of the type checking verfication procedure for class files.