Certified Static Analysis by Abstract Interpretation

Frédéric Besson[1], David Cachera[2*], Thomas Jensen[3], and David Pichardie[1]

[1] INRIA Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France
[2] ENS Cachan (Bretagne), Campus de Ker Lann, 35170 Bruz, France
[3] CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. A certified static analysis is an analysis whose semantic validity has been formally proved correct with a proof assistant. We propose a tutorial on building a certified static analysis in Coq. We study a simple bytecode language for which we propose an interval analysis that allows to verify statically that no array-out-of-bounds accesses will occur.

1   Introduction

Static program analysis is a fully automatic technique for proving properties about the behaviour of a program without actually executing it. Static analysis is becoming an important part of modern security architectures, as it allows to screen code for potential security vulnerabilities or malicious behaviour before integrating it into a computing platform. A prime example of this is the Java byte code verifier that plays an important role in the Java Virtual Machine security architecture. The byte code verifier ensures by a static analysis that code is well typed and that objects are constructed properly—two properties that when properly verified improve the overall security of the virtual machine. At the same time, it is recognized that static analyzers are complex pieces of software whose integration into the trusted computing base (TCB) may pose problems. Indeed, static analyzers implement sophisticated logics for proving properties about programs and errors in the implementation are possible. Such errors in the implementation may compromise the security of the platform as was the case in some early implementations of the Java type checking mechanism [MF99]. In general, it is desirable to reduce the part of the static analyzer that forms part of the TCB.

The correctness of static analyses can be proved formally by following the theory of abstract interpretation [CC77] that provides a theory for relating two semantic interpretations of the same language. These strong semantic foundations constitute one of the arguments advanced in favor of static program analysis. The implementation of static analyses is usually based on well-understood constraint-solving techniques and iterative fixpoint algorithms. In spite of the

---

[*] Currently delegated as a full time researcher at INRIA, Centre Rennes - Bretagne Atlantique.

nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the machine. While this gap might be small for toy languages, it becomes important when it comes to real-life languages for which the implementation and maintenance of program analysis tools become a software engineering task. To eliminate this gap, is possible to merge both the analyser implementation and the soundness proof into the same logic of a proof assistant. This gives raise to the notion of *certified static analysis*, *i.e.* an analysis whose implementation has been formally proved correct using a proof assistant.

Related Work This tutorial paper follows [CJPR05], where a dataflow analysis for a byte code language has been formalized in Coq. In [CJPS05], the same kind of framework has been applied to a different kind of analysis, namely an analysis certifying the absence of infinite loops in a byte code interprocedural language.

Proving correctness of program analyses is one of the main applications of the theory of abstract interpretation [CC77]. However, most of the existing proofs are pencil-and-paper proofs of analyses (formal specifications) and not mechanised proofs of analysers (implementations of analyses). The only attempt of formalising the theory of abstract interpretation with a proof assistant is that of Monniaux [Mon98] who has built a Coq theory of Galois connections. The Coq proof assistant at this time did not benefit from the current extraction mechanism, which prevented the author from extracting analysers from the specifications. More recently, Bertot has proposed a Coq] formalization of abstract interpretation for a While language [Ber08], based on a weakest precondition calculus. His work however does not consider termination issues. Mechanical proofs about fixpoint iteration using widenings have not been considered by other authors, since widening operators require complex termination proofs. Other existing works only deal with ascending chain condition [KN02,BD04,CGD06,BGL06].

Other approaches use a proof assistant for certifying the correction of static analyses, without developing a framework for a general theory like abstract interpretation. Barthe and al. [BDHdS01] have shown how to formalize Java byte code verification, reasoning on an executable semantics. In [BDJ+01], they automatize the derivation of a certified verifier in a dedicated environment. In another approach, Klein and Nipkow have used Isabelle/HOL to formalize a bytecode verifier [KN02]. In [KN06], they propose a complete formalization of both source language and byte code language semantics, of a non-optimising compiler and of a bytecode verifier, for a Java-like language. The proofs however are not done at the analysis implementation level, and do not offer the same certification level as with a Coq extraction.

Finally, Lee *et al.* [LCH07] have certified the type analysis of a language close to Standard ML in LF and Leroy [Ler06] has certified some of the data flow analyses of a compiler back-end.

Overview In this work we rely on the Coq proof assistant [Coq09]. Coq is used here as programming language with a very rich type system where full functional correctness of functional programs can be formally established. The program extraction mechanism provides a tool for automatic translation of these programs into a functional language with a simpler type system, namely Ocaml. The extraction mechanism removes those parts of the lambda-terms that are only concerned with proving properties without actually contributing to the final result. In our context this mechanism allow us to obtain certified Ocaml implementation of static analysis that have been formally proved correct with Coq.

In order to mechanically prove correctness of static analyses we follow the abstract interpretation methodology that provide a rich framework for designing sound static analyses in a methodological way and comparing the precision of static analyses. We are mainly interested here in ensuring correctness of our analyses, rather that systematically design optimal ones. We will then only use a restricted part of the theory in our Coq formalization. Other parts will be presented and used in order to give a formal methodology for static analysis construction, but will not be explicitly defined inside Coq.

The methodology that we present here is generic but we have chosen to develop it in the concrete setting of a Java-like bytecode language for which we propose an interval analysis that allows to verify statically that no array-out-of-bounds accesses will occur.

Building a certified static analysis can be thought as assembling puzzle pieces, as shown in Figure 1, where each piece interacts with its neighbors. The basic components of a static analysis based on abstract interpretation are: (1) a (concrete) semantic domain and (2) semantic rules modelling the program behaviour, (3) an abstract domain allowing the expression of properties on the concrete domain, (4) logic links between abstract and concrete domains, describing how abstract objects represent concrete ones, (5) the specification of the analysis, under the form of constraints between abstract objects, (6) correctness proofs showing that any solution to the analysis specification indeed represents an approximation of the concrete semantics, (7) a way to generate analysis constraints from the program syntax, and (8) a way to compute a solution of the analysis specification.

We will know describe in turn all of these pieces, giving each time most of Coq code used in the development. The whole development is made available on-line at the following url:

```
http://www.irisa.fr/celtique/pichardie/fosad09/
```

2 Running language example

To illustrate the progress of our methodology, we will develop an interval analysis for a simple byte code language. The analysis is based on existing interval analyses for high-level structured languages [Cou99] but has been extended with
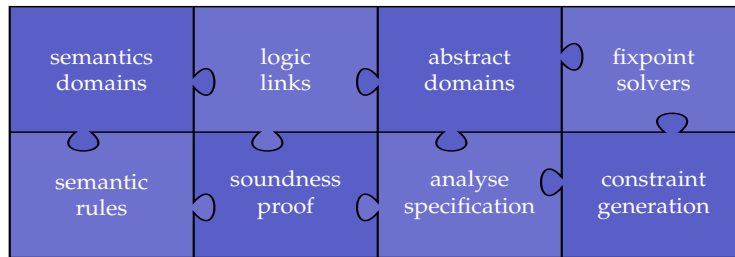
Fig. 1. The main blocks for building a certified static analysis

an abstract domain of *syntactic expressions* in order to obtain a similar precision at byte code level.

The language we treat is sufficiently general to illustrate the feasibility of our approach and perform experiments on code obtained from compilation of Java source code. We restrict ourselves to programs with only one method manipulating an operand stack, handling integers (with infinite arithmetic) and dynamically allocated arrays. Missing features like method calls, objects or multithreading could be added to this language, but would require more complex static analyses that are beyond the scope of this tutorial paper.

The syntax of our language is defined in Coq with inductive types.

```
Definition pc := word.
Definition var := word.
Inductive binop := Add | Sub | Mult.
Inductive cmp := Eq | Ne | Gt | Ge.
Inductive instruction :=
  | Nop | Push (n:integer)| Pop | Dup
  | Load (x:var) | Store (x:var) | Binop (op:binop)
  | Newarray | Arraylength | Arrayload | Arraystore
  | Input
  | If (c:cmp) (jump:pc) | Goto (jump:pc).
Definition program := list (pc * instruction).
```

The syntax we use here is similar to Caml's variant types. Type `word` is kept abstract here but will be made explicit in Section 10. The only requirement we have for the moment is an equality test on objects of this type.

```
Definition eq_word_dec : ∀ w1 w2 : word, {w1=w2}+{w1≠w2} := ...
```

We use here a dependent type: `eq_word_dec` is a function that takes two arguments (called `w1` and `w2` here) of type `word` and returns a *rich* boolean type which explicitly depends on the two arguments of the function. This inductive type contains two constructors. An element of type {w1=w2}+{w1≠w2} is either of the form (`Left h`) with `h` a proof of `w1=w2`, or (`Right h`) with `h` a proof of w1≠w2.

The type `word` is used to model local variable names and program counters. We handle addition, subtraction and multiplication over integers (type `binop`).

Integers can be tested with respect to four kinds of comparisons: equality, disequality, and order (strict or not). The language handles instructions for operand stack manipulation (`Pop`, `Dup`), local variable manipulation (`Load` x, `Store` x, respectively loading from or storing to memory, to of from the stack) and conditional (`If`) or unconditional (`Goto`) jumps. Arrays are dynamically allocated with `Newarray`, read with `Arrayload`, written with `Arraystore` and we can read their length with `Arraylength`. At last, the instruction `Input` pushes an arbitrary integer on the operand stack. A program is just an association list between program counters and instructions.

As an example, we give on Figure 2 (first column) the byte code of a program doing a bubble sort. The source version is given on Figure 3. Only program counters that are targets of a jump are displayed.

3 Semantic domains

The first piece of our puzzle relates to the semantic domain, that is, the runtime structures that are manipulated by the program during execution.

Semantic domains of the bytecode language Our byte code programs manipulate states of the form (pc, h, s, l) where pc is the control point to be executed next, h is a heap for storing allocated arrays, s is an operand stack, and l is an environment mapping local variables to values. An array is modeled by a dependent pair consisting of the size *length* of the array and a function that for a given index in the range $[0, length - 1]$, returns the value stored at that index. A special error state is used to model execution errors which arise here from indexing an array outside its bounds or allocating an array with a negative size. All this elements are defined with standard enumerate types. Arrays are defined with a record, where the field `array_values` is a function with a dependent type, taking two arguments: the first one (called `i`) is an integer and the second one is a proof that `i` is in the right range.

```
Inductive val : Set := | Num (i:integer)| Ref (l:location).

Inductive var_val : Set := Undef | Def (v:val).
Definition locvar : Set := var → var_val.

Definition opstack : Set := list val.

Record array : Set := {
  array_length : integer;
  array_values : ∀ i:integer, (0 <= i < array_length) → integer
}.

Inductive heap_val : Set :=
  No | Array (a:array).
Definition heap := location → heap_val.
```

`0:Ipush 10`		$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in\bot$	$t\in\bot$
`:Store 4`	`10`	$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in\bot$	$t\in\bot$
`:Load 4`		$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in[10,10]$	$t\in\bot$
`:Newarray`	`n`	$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in[10,10]$	$t\in\bot$
`:Store 5`	`n`	$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in[10,10]$	$t\in\bot$
`:Ipush 0`		$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in[10,10]$	$t\in[10,10]$
`:Store 1`	`0`	$i\in\bot$	$j\in\bot$	$tmp\in\bot$	$n\in[10,10]$	$t\in[10,10]$
`7:Load 1`		$i\in[0,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 4`	`i`	$i\in[0,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`i::n`	$i\in[0,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Sub`	`i::n::1`	$i\in[0,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:If Ge 58`	`i::(Sub n 1)`	$i\in[0,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 0`		$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Store 2`	`0`	$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`14:Load 2`		$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 4`	`j`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`j::n`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Sub`	`j::n::1`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 1`	`j::(Sub n 1)`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Sub`	`j::(Sub n 1)::i`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:If Ge 53`	`j::(Sub (Sub n 1) i)`	$i\in[0,8]$	$j\in[0,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`		$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Add`	`t::j::1`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arrayload`	`t::(Add j 1)`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`	`Top`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`Top::t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arrayload`	`Top::t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:If Ge 48`	`Top::Top`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`		$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arrayload`	`t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Store 3`	`Top`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`		$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`	`t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`t::j::t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`t::j::t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Add`	`t::j::t::j::1`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arrayload`	`t::j::t::(Add j 1)`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arraystore`	`t::j::Top`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 5`		$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 2`	`t`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`t::j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Add`	`t::j::1`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Load 3`	`t::(Add j 1)`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Arraystore`	`t::(Add j 1)::tmp`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`48:Load 2`		$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`j`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Add`	`j::1`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Store 2`	`(Add j 1)`	$i\in[0,8]$	$j\in[0,8]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Goto 14`		$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`53:Load 1`		$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Ipush 1`	`i`	$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Binop Add`	`i::1`	$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Store 1`	`(Add i 1)`	$i\in[0,8]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`:Goto 7`		$i\in[1,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$
`58:`		$i\in[9,9]$	$j\in[1,9]$	$tmp\in\top$	$n\in[10,10]$	$t\in[10,10]$

Fig. 2. Byte code for bubble sort.

```
class BubbleSort {
    public static void main(String argv ) {
        int i,j,tmp,n;
        n = 10;
        int[] t = new int[n];
        // We omit the initilisation of the array
        for (i=0; i<n-1; i++)
            for (j=0; j<n-1-i; j++)
                if (t[j+1] < t[j])
                    { tmp = t[j]; t[j] = t[j+1]; t[j+1] = tmp;}
    }
}
```

Fig. 3. Source code for buble sort.

```
Inductive state : Set :=
| St (i:pc) (s:opstack) (l:locvar) (h:heap)
| Error.
```

Complete lattice structure According to the base postulate of abstract interpretation, the semantic domain for expressing the concrete semantics can be given a complete lattice structure.

Definition 1 (Complete lattice). *A complete lattice is a triple* $(A, \sqsubseteq, \bigsqcup)$ *composed of a set A, equipped with a partial order $\sqsubseteq$ and a least upper bound (lub) $\bigsqcup$: for all subset S of A,*

- $\forall a \in S, \ a \sqsubseteq \bigsqcup S$
- $\forall b \in A, (\forall a \in S, \ a \sqsubseteq b) \ \Rightarrow \ \bigsqcup S \sqsubseteq b$

A complete lattice necessarily has a greatest lower bound (glb) operator $\bigsqcap$. It also necessarily has a greatest element $\top = \bigsqcap \emptyset = \bigsqcup A$, and a least element $\bot = \bigsqcup \emptyset = \bigsqcap A$.

Here, we can consider the set of subsets of `state` ordered by inclusion as concrete domain. Such a powerset domain $\mathcal{P}(\texttt{state})$ is given a complete lattice structure $(\mathcal{P}(\texttt{state}), \subseteq, \cup, \cap)$. Elements of $\mathcal{P}(\texttt{state})$ are seen as properties on objects of `state`. The partial order $\subseteq$ thus models the property accuracy: if $P_1 \subseteq P_2$, then P_1 gives more precision, since it describes a smaller set of behaviours.

Complete lattices can be formalised in Coq by means, for example, of a record construct. However, though it remains quite easy to prove general results on complete lattices in Coq, the effective instantiation of such a structure is much more intricate because of the constructive logic used by Coq. For example, if the base set is of an implementable type, the lub will be a function that for any predicate P on the lattice carrier[4] computes the least upper bound of all

---

[4] A subset in $\mathcal{P}(\texttt{A})$ is encoded as a predicate of type $\texttt{A} \rightarrow \textbf{Prop}$ in Coq.

elements satisfying P. But predicate P is of a logical type that is not necessarily implementable. Still, for the carrier $\mathcal{P}(\texttt{state})$ the lattice structure can be build in Coq but it is useless for the purpose of our case study. We thus end this section with a simple semantic domain $\mathcal{P}(\texttt{state})$ without a formal proof that it enjoys a canonical complete lattice structure.

4 Semantic rules

We still have to formally explain how a program manipulates the elements of the semantic domain during an execution. This is traditionally expressed by means of an operational semantics, under the form of a transition relation `-[p]->` between states, parameterized by a program `p`. The keyword **Inductive** is now used for a different purpose that during the syntax definition. We define here a relation with inductive rules. We rely on the Coq notation system to directly use the notation `-[p]->`. The definition handles one or two rules per instructions in the language. The function `instr_at` is a lookup search in the instruction list in order to find the instruction associated with a given program counter. Function `next` computes the successor of a program counter. The execution does not progress for ill-typed states, except for arrays out of bound access or allocation of an array with a negative size, that leads to an explicit error state.

```
Inductive step (p:program) : state → state → Prop :=
| step_nop : ∀ pc h s l,
    instr_at p pc = Some Nop →
    (St pc s l h) -[p]-> (St (next pc) s l h)
| step_push : ∀ pc h s l n,
    instr_at p pc = Some (Push n) →
    (St pc s l h) -[p]-> (St (next pc) (Num n :: s) l h)
| step_pop : ∀ pc h s l v,
    instr_at p pc = Some Pop →
    (St pc (v::s) l h) -[p]-> (St (next pc) s l h)
| step_dup : ∀ pc h s l v,
    instr_at p pc = Some Dup →
    (St pc (v::s) l h) -[p]-> (St (next pc) (v::v::s) l h)
| step_iload : ∀ pc h s l x v,
    instr_at p pc = Some (Load x) →
    l x = Def v →
    (St pc s l h) -[p]-> (St (next pc) (v::s) l h)
| step_istore : ∀ pc h s l x v l',
    instr_at p pc = Some (Store x) →
    subst l x v l' →
    (St pc (v :: s) l h) -[p]-> (St (next pc) s l' h)
| step_binop : ∀ pc h s l v1 v2 b,
    instr_at p pc = Some (Binop b) →
    (St pc (Num v2:: Num v1::s) l h)
        -[p]-> (St (next pc) (Num (binop_sem b v1 v2)::s) l h)
| step_newarray_1 : ∀ pc h h' s l n loc,
    instr_at p pc = Some Newarray →
```

```
      create_new_array h n h' loc →
      (St pc (Num n :: s) l h)
          -[p]-> (St (next pc) (Ref loc :: s) l h')
  | step_newarray_2 : ∀ pc h s l n,
      instr_at p pc = Some Newarray →
      n<0 →
      (St pc (Num n :: s) l h) -[p]-> Error
  | step_arraylength : ∀ pc h a s l loc,
      instr_at p pc = Some Arraylength →
      h loc = Array a →
      (St pc (Ref loc :: s) l h)
        -[p]-> (St (next pc) (Num a.(array_length) :: s) l h)
  | step_arrayload_1 : ∀ pc h a  i s l loc,
      instr_at p pc = Some Arrayload →
      h loc = Array a →
      ∀ hi:(0 <= i < a.(array_length)),
      (St pc (Num i :: Ref loc :: s) l h)
        -[p]-> (St (next pc) (Num (a.(array_values) i hi) :: s) l h)
  | step_arrayload_2 : ∀ pc h a  i s l loc,
      instr_at p pc = Some Arrayload →
      h loc = Array a →
      ¬ (0 <= i < a.(array_length)) →
      (St pc (Num i::Ref loc :: s) l h) -[p]-> Error
  | step_arraystore_1 : ∀ pc h i s l loc n h',
      instr_at p pc = Some Arraystore →
      heap_update_array h loc i n h' →
      (St pc (Num n :: Num i :: Ref loc :: s) l h)
          -[p]-> (St (next pc) s l h')
  | step_arraystore_2 : ∀ pc h i s l loc n a,
      instr_at p pc = Some Arraystore →
      h loc = Array a →
      ¬ (0 <= i < a.(array_length)) →
      (St pc (Num n :: Num i :: Ref loc :: s) l h) -[p]-> Error
  | step_goto : ∀ pc jump h s l,
      instr_at p pc = Some (Goto jump) →
      (St pc s l h) -[p]-> (St jump s l h)
  | step_if_1 : ∀ pc h s n1 n2 l cmp jump,
      instr_at p pc = Some (If cmp jump) →
      cmp_sem cmp n1 n2 →
      (St pc (Num n2 :: Num n1 :: s) l h) -[p]-> (St jump s l h)
  | step_if_2 : ∀ pc h s n1 n2 l cmp jump,
      instr_at p pc = Some (If cmp jump) →
      ¬ cmp_sem cmp n1 n2 →
      (St pc (Num n2 :: Num n1 :: s) l h)
          -[p]-> (St (next pc) s l h)
  | step_input : ∀ pc h s l n,
      instr_at p pc = Some Input →
      (St pc s l h) -[p]-> (St (next pc) (Num n :: s) l h)
where "s1 -[ p ]-> s2" := (step p s1 s2).
```

The predicate (`subst l s v l'`) expresses the fact that the set of local variables `l'` is the update of set `l` with the value `v` and for the variable `x`.

```
Inductive subst : locvar → var → val → locvar → Prop :=
  subst_def : ∀ (env env' : locvar) x n,
    env' x = Def n →
    (∀ y, x ≠ y → env y = env' y) →
    subst env x n env'.
```

The predicate (`create_new_array h n' loc`) expresses the fact that heap `h'` results from the allocation of a new array of size `n` in a heap `h` at location `loc`.

```
Inductive create_new_array :
                     heap → integer → heap → location → Prop :=
 create_new_array_def : ∀ h n h' loc a,
   h loc = No →
   (∀ loc', loc' ≠ loc → h loc' = h' loc') →
   h' loc = Array a →
   a.(array_length) = n →
   (∀ i (h:0<=i<a.(array_length)), a.(array_values) i h = 0) →
   create_new_array h n h' loc.
```

The predicate (`heap_update_array h loc i n h'`) expresses the fact that heap `h'` results from an update of an array at location `loc` in the heap `h`. The corresponding array is updated at index `i` with the value `n`.

```
Inductive array_subst :
                   array → integer → integer → array → Prop :=
  array_subst_def : ∀ a a' i n,
    ∀ h:(0 <= i < a'.(array_length)),
    a'.(array_length) = a.(array_length) →
    a'.(array_values) i h = n →
    (∀ j,
       i ≠ j →
       ∀ h:(0 <= j < a.(array_length)),
       ∀ h':(0 <= j < a'.(array_length)),
      a'.(array_values) j h' = a.(array_values) j h) →
    array_subst a i n a'.
Inductive heap_subst :
                   heap → location → heap_val → heap → Prop :=
  heap_subst_def : ∀ h h' loc hv,
    h' loc = hv →
    (∀ loc', loc ≠ loc' → h' loc' = h loc') →
    heap_subst h loc hv h'.
Inductive heap_update_array :
        heap → location → integer → integer → heap → Prop :=
  heap_update_array_def : ∀ h loc i n h' a a',
    h loc = Array a →
    array_subst a i n a' →
    heap_subst h loc (Array a') h' →
    heap_update_array h loc i n h'.
```

The relation `cmp_sem` gives a semantics to the comparison operators.

```
Definition cmp_sem (cmp:cmp) : integer → integer → Prop :=
  match cmp with
    | Eq ⇒ fun x y ⇒ x=y
    | Ne ⇒ fun x y ⇒ x≠y
    | Gt ⇒ fun x y ⇒ x>y
    | Ge ⇒ fun x y ⇒ x>=y
  end.
```

Semantics of binary operators is given by `binop_sem`.

```
Definition binop_sem (binop:binop) :
          integer → integer → integer :=
  match binop with
    | Add ⇒ Zplus
    | Sub ⇒ Zminus
    | Mult ⇒ Zmult
  end.
```

We can then define the set of reachable states during the execution of a program.

```
Inductive InitState (p:program) : state → Prop :=
  initState_def :
    InitState p  (St p nil (fun _ ⇒ Undef) (fun _ ⇒ No)).


Inductive ReachableStates (p:program) : state → Prop :=
    reach_init : ∀ st, InitState p st → ReachableStates p st
  | reach_next : ∀ st1 st2,
      ReachableStates p st1 → st1 -[p]-> st2 →
      ReachableStates p st2.
```

The predicate `ReachableStates p` (noted $[\![p]\!]$ in mathematical form) is generally given a least fixpoint characterisation

$$[\![p]\!] = lfp(\lambda X.S_0 \cup post_p(X))$$

where S_0 is the set of initial states (called `InitState` in Coq) and *post*, the operator formally defined by

$$post_p(S) = \{s_2 \mid \exists s_1 \in S, \ s_1\text{-}[p]\text{->}s_2\}$$

We do not prove here this characterisation in Coq. Instead, the inductive definition we use for `ReachableStates` gives us for free that:

1. $[\![p]\!]$ contains S_0 (rule `reach_init`),
2. $[\![p]\!]$ is stable by *post* (rule `reach_next`),
3. and this is the least property that satisfies this two conditions (by the intrinsic property of inductive definitions).

This least-postfixpoint characterisation will be sufficient for the rest of the work.

The global objective of the analysis is to prove that all reachable states of a program are safe, *i.e.* are not the error state. More formally, safe is defined by

```
Definition Safe (st:state) : Prop := st ≠ Error.
```

5 Abstract domains

The concrete semantics is unfortunately not computable. The key idea of abstract interpretation theory is to replace the previous semantic domains by a simpler one, where the computation of the program can be mimicked in a computable way. This new domain is called abstract domain. This simpler version can be seen as a restriction of the set of properties used to express the behaviour of a program. For example, if the concrete domain for a program manipulating integers has the form $\mathcal{P}(\mathbb{Z})$, an abstract domain could contain only the convex parts of $\mathcal{P}(\mathbb{Z})$, that is intervals of integers. Abstract domains are given a lattice structure where the least upper bound represents the disjunction of properties and greatest lower bound represents their conjunction.

Our bytecode analyser will rely on this abstract domain of intervals for abstracting numerical values. We first present this standard domain, before constructing the abstract domains for our byte code language analysis. The link between concrete and abstract domains will be formally explained in Section 7.

Interval lattice The set of intervals over integers is naturally equipped with a lattice structure, induced by the usual order on integers, extended to infinite positive and negative values in order to get completeness.

Definition 2 (Interval lattice). *The lattice $(Int, \sqsubseteq_{Int}, \sqcup_{Int}, \sqcap_{Int})$ of intervals is defined by*

- *a base set $Int = \{[a,b] \mid a, b \in \overline{\mathbb{Z}}, a \leq b\} \cup \bot$ where $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$,*
- *a partial order $\sqsubseteq_{Int}$ which is the least relation satisfying the following rules*

$$\frac{I \in Int}{\bot \sqsubseteq_{Int} I} \qquad \frac{c \leq a \quad b \leq d \quad a, b, c, d \in \overline{\mathbb{Z}}}{[a,b] \sqsubseteq_{Int} [c,d]}$$

- *meet and join binary operators, respectively defined by*

$$I \sqcup_{Int} \bot = = I, \ \forall I \in Int$$
$$\bot \sqcup_{Int} I = = I, \ \forall I \in Int$$
$$[a,b] \sqcup_{Int} [c,d] = [\min(a,c), \max(b,d)]$$

$$I \sqcap_{Int} \bot = \bot, \ \forall I \in Int$$
$$\bot \sqcap_{Int} I = \bot, \ \forall I \in Int$$
$$[a,b] \sqcap_{Int} [c,d] = \rho_{Int}([\max(a,c), \min(b,d)])$$

The bottom and top elements are $\bot_{Int} = \bot$ and $\top_{Int} = [-\infty, +\infty]$, respectively.

This lattice, of infinite width and height, can be depicted as on Figure 4.
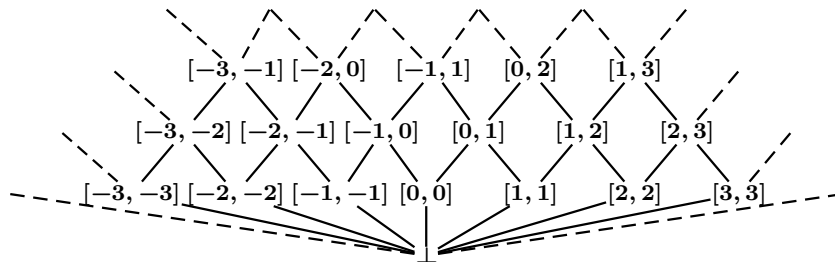
Fig. 4. The lattice of intervals

Abstract domain of the bytecode analysis The property we want to compute will be attached to each program point of a program. As a consequence, the abstract domain will be of the form

$$\mathtt{pc} \rightarrow \mathtt{mem}^\sharp$$

with $\mathtt{mem}^\sharp$ a domain that expresses properties on the heap, the local variables and the operand stack. Values are either numerics or references to an array. In the first case we will abstract the corresponding integer with an interval, in the second case we will abstract the size of the array with an interval too. Such an information will be computed for each local variable, and hence the abstract domain of local variables will have the form

$$\mathtt{var} \rightarrow \mathit{Int}$$

A naive choice would be to abstract operand stacks with a stack of intervals, in the same spirit as for local variables. Nevertheless, such a choice would have a bad impact on the precision of our analysis. We will discuss this point below. Instead, we will abstract operand stacks by stacks of symbolic expressions. Each symbolic expression represents a relation between the corresponding value of the operand stack and the local variables. The language of expressions is defined by the following inductive type.

```
Inductive expr :=
  Const (n:integer) | Var (x:var) | Bin (bin:binop) (e1 e2:expr).
```

We put a flat lattice structure on expressions, as depicted on Figure 5. For this purpose with the parameterized type flat given below.

```
Inductive flat (A:Set) : Set := Top | Base (x:A) | Bot.
```

Precision gain of the symbolic operand stack Representing abstract operands by symbolic expressions has a significant impact on the precision of the analysis, since it allows to preserve the information obtained through the evaluation of
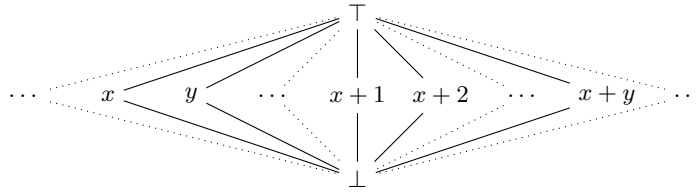
Fig. 5. The lattice of syntactic expressions

conditional expressions. At source level, a test such as `j+i>3` is a constraint over
the possible values of `i` and `j` that can be propagated into the branches of a
conditional statement. However, at byte code level, before such a conditional
the evaluation stack only contains a boolean (encoded by an integer). The ex-
plicit constraint between variables `i` and `j` is lost because their values have to be
pushed onto the stack before they can be compared. Using syntactic expressions
to abstract stack contents enables the analysis to reconstruct this information.

A lattice library Building complex lattices in Coq can be done in a modular
fashion. A library of Coq functors and modules has thus be developed [Pic08], in
order to easily construct complex lattices by composition from simpler ones. The
lattice properties are summarized in a Coq module signature named `Lattice`.

```
Module Type Lattice.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym :
    ∀ x y : t, order x y → order y x → eq x y.
  Parameter order_trans :
    ∀ x y z : t, order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter join  : t → t → t.
  Parameter join_bound1 : ∀ x y : t, order x (join x y).
  Parameter join_bound2 : ∀ x y : t, order y (join x y).
  Parameter join_least_upper_bound :
    ∀ x y z : t, order x z → order y z → order (join x y) z.
```

```
  Parameter meet   : t → t → t.
  Parameter meet_bound1 : ∀ x y : t, order (meet x y) x.
  Parameter meet_bound2 : ∀ x y : t, order (meet x y) y.
  Parameter meet_greatest_lower_bound :
    ∀ x y z : t, order z x → order z y → order z (meet x y).

  Parameter bottom : t.
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.

  Parameter widening : widening_operator eq order bottom.
  Parameter narrowing : narrowing_operator eq order meet.
End Lattice.
```

Widening and narrowing operators will be described in Section 10.

For our analysis, the base lattices are the interval lattice and the (flat) lattice of syntactic expressions. The functor `FlatLattice` allows to construct such a structure given a carrier type and an equivalence relation for equality. The other lattices corresponding to semantic subdomains such as local variables or stacks are simply constructed by application of the respective functors for lists, arrays and cartesian product.

```
 Module Expr
  Definition t := expr.
  Definition eq (x y : t) : Prop := x=y.

  Lemma eq_refl : ∀ x : t, eq x x.
  Proof. ... Qed.
  Lemma eq_sym : ∀ x y : t, eq x y → eq y x.
  Proof. ... Qed.
  Lemma eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Proof. ... Qed.
  Lemma eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.
  Proof. ... Qed.
End Expr.


Module ExprLat := FlatLattice Expr.
Module LocvarLat := ArrayBinLattice IntervalLat.
Module OpstackLat := ListLattice ExprLat.
Module MemLat := ProdLattice StackLat LocalVarLat.
Module GlobalLat := ArrayBinLattice MemLat.
```

The module `ExprLat` represents the lattice of symbolic expressions partially ordered according to the Hasse diagram of Figure 5. The module `LocvarLat` represents the module of functions from `word` to intervals. Functions are encoded with maps (binary trees). The corresponding partial order is the point-wise extension of the partial order on intervals. The module `OpstackLat` builds a lattice of lists whose elements live in `ExprLat.t`. Lists of the same length are ordered point-wise, while lists of different sizes are not comparable. The lattice is extended with two bottom and top extra values (using again the parameterized

type `flat`). `MemLat` is the product of the two previous lattices. `GlobalLat` is the final lattice. Again, it is built with the `ArrayBinLattice` functor of functions.

6 Analysis specification

The counterpart of the semantics rules is an operational description of the abstract execution of a program. Instead of manipulating concrete properties, this execution manipulates the abstract values that represent only a restricted set of properties. When the transformation of a property does not fit in the restricted set of the abstraction, a conservative alternative must be used instead. For example, the successor function on $\mathcal{P}(\mathbb{Z})$ transforms any property $P \subseteq \mathbb{Z}$ by $\{z + 1 \mid z \in P\}$ but, for the sign abstraction which only represents properties $\mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-, \{0\}$ and $\emptyset$, the image of $\{0\}$, *i.e.* set $\{1\}$, does not fit exactly a particular property and must be over-approximated by $\mathbb{Z}^+$. The set $\mathbb{Z}$ would have been an other sound approximation, but less precise.

For each instruction of the language we have to propose a sound transformation in the abstract domain that mimics the effect of a concrete execution of the instruction in the concrete domain. The word *sound* will be made more formal in the next section.

We specify the analysis under the form of a constraint-based specification. A constraint imposes on a function $X \in pc \to \text{mem}^\sharp$ an inequation of the form

$$f(X(i)) \sqsubseteq^\sharp X(j)$$

where i, j, f are the three parts of the constraint. Index $i \in pc$ is the source program point, $j \in pc$ the target program point and $f \in \text{mem}^\sharp \to \text{mem}^\sharp$ the transfer function that must be applied between these two points. The property expressed by a constraint is formalized with a predicate `cstr2prop`.

```
Record cstr : Set := C
  { source : pc; target : pc; transfer : Mem.t → Mem.t }.
```

```
Definition cstr2prop (s:GlobalLat.t) (c:cstr) : Prop :=
  MemLat.order
    (c.(constraint) (GlobalLat.get s c.(source)))
    (GlobalLat.get s c.(target)).
```

Each instruction gives rise to one or two constraints, generated by the function `gen_constraint`.

```
Definition gen_constraint (i:pc) (ins:instruction) : list cstr :=
  match ins with
    | Nop ⇒ C i (next i) (fun x ⇒ x) ::nil
    | Push n ⇒
        C i (next i)
          (lift0 (fun s l ⇒ (Base (Const n)::s,l))) ::nil
    | Pop ⇒
        C i (next i) (lift1 (fun _ s l ⇒ (s,l))) ::nil
```

```
    | Dup ⇒
        C i (next i) (lift1 (fun e s l ⇒ (e::e::s,l))) ::nil
    | Load x ⇒
        C i (next i)
          (lift0 (fun s l ⇒ (Base (Var x)::s,l))) ::nil
    | Store x ⇒
        C i (next i)
          (lift1
            (fun e s l ⇒
              (map (clear_var x) s,ab_store x e l))) ::nil
    | Binop op ⇒
        C i (next i)
          (lift2
            (fun e1 e2 s l ⇒ (ab_binop op e1 e2::s,l))) ::nil
    | Newarray ⇒
        C i (next i) (lift1 (fun e s l ⇒ (e::s,l))) ::nil
    | Arraylength ⇒
        C i (next i) (lift1 (fun e s l ⇒ (e::s,l))) ::nil
    | Arrayload ⇒
        C i (next i) (lift2 (fun _ _ s l ⇒ (Top::s,l))) ::nil
    | Iastore ⇒
        C i (next i) (lift3 (fun _ _ _ s l ⇒ (s,l))) ::nil
    | Input ⇒
        C i (next i) (lift0 (fun s l ⇒ (Top::s,l))) ::nil
    | If cmp jump ⇒
        C i jump
          (lift2 (fun e2 e1 s l ⇒ (s,ab_cmp cmp e1 e2 l))) ::
        C i (next i)
          (lift2
            (fun e2 e1 s l ⇒ (s,ab_cmp (neg_cmp cmp) e2 e1 l))) ::
        nil
    | Goto jump ⇒
        C i jump (fun x ⇒ x) ::nil
  end.
```

The lift0, lift1, lift2, lift3, functions are used here to simplify the presentation. Each of them is dedicated to symbolic stacks with at least 0, 1, 2 or 3 elements on top of them.

```
Definition lift1
  (f:ExprLat.t → list ExprLat.t → LocvarLat.t →
                        list ExprLat.t*LocvarLat.t)
  (m:MemLat.t) : MemLat.t :=
  let (s,l) := m in
    match s with
      | Top ⇒ m
      | Base (e::s) ⇒ let (s',l') := f e s l  in (Base s',l')
      | _ ⇒ (Bot,LocvarLat.bottom)
    end.
```

These functions allow to factorize many cases of the constraint generation. For example, without `lift1`, the branch of instruction `Dup` would be the following.

```
| Dup ⇒
    C i (next i) (fun (m:Mem.t) ⇒
                     let (s',l) := m in
                       match s' with
                       | Top ⇒ (Top,l)
                       | Base (e::s) ⇒ (Base (e::e::s),l)
                       | _ ⇒ (Bot,LocvarLat.bottom)
                       end)
```

We now informally describe each instruction. The `Nop` instruction does not modify the memory and we generate hence a constraint with an identity transfer function. The `Push n` instruction modifies only the abstract operand stack and pushes the symbol `Const n` on its top. The `Dup` instruction duplicates the expression on top of the abstract operand stack. The `Load x` pushes the symbol `Var x` on top of the abstract operand stack. The `Store x` modifies both the abstract operand stack and the abstract local variables. First, the abstract operand is cleaned up: every symbolic expression which contains x is replaced by `Top`, thanks to the operator (`clear_var x`). This is necessary, because after the execution of this instruction, such expressions will refer to the old value of x. The abstract local variables are updated with the operator `ab_store` defined below.

```
Definition ab_store (x:var) (e:flat expr) (l:LocvarLat.t) :
  LocvarLat.t :=
  match e with
    | Bot ⇒ LocvarLat.bottom
    | Base e ⇒ AbEnv.assign x e l
    | Top ⇒ AbEnv.forget x l
  end.
```

In the first case we propagate the bottom information. In the second case, we use an abstract assignment of the expression e in the abstract local variables. In the last case we need to forget the information currently known about the variable x. All operators that are specific to the abstraction of variables by intervals are gathered in a module `AbEnv`. We will not detail them in this document.

The `Binop` instruction modifies the abstract operand stack with the right symbolic expression.

```
Definition ab_binop (op:binop) (e1 e2:flat expr) : flat expr :=
  match e1, e2 with
    | Bot, _ | _, Bot ⇒ Bot
    | Top, _ | _, Top ⇒ Top
    | Base e1, Base e2 ⇒ Base (Bin op e1 e2)
  end.
```

`Newarray` and `Arraylength` do not really modify the abstract operand stack because a location has the same abstraction as the length of the corresponding array. For instruction `Arrayload`, the content of an array is always represented

by `Top` since our abstraction does not allow us to track more acutely the content of arrays. Updating an array does not modify its length, hence the `Arraystore` instruction keeps the queue of the operand stack and the local variables in their previous state. The exact value pushed on top of the operand stack by the instruction `Input` is unknown. We hence push `Top` on top of the abstract operand stack. The `If` instruction needs to generate two constraints, one for each potential branch. In each case the abstract local variables are updated according to the fact that the test has been satisfied or not. This operation relies on the operator `AbEnv.assert` on abstract local variables.

```
Definition ab_cmp (cmp:cmp) (e1 e2:flat expr)
                   (l:LocvarLat.t) : LocvarLat.t :=
  match e1, e2 with
    | Bot, _ | _, Bot ⇒ LocvarLat.bottom
    | Top, _ | _, Top ⇒ l
    | Base e1, Base e2 ⇒ AbEnv.assert cmp e1 e2 l
  end.
```

```
Definition neg_cmp (cmp:cmp) : sem.cmp :=
  match cmp with
    | Eq ⇒ Ne | Ne ⇒ Eq | Ge ⇒ Gt | Gt ⇒ Ge
  end.
```

The `Goto` instruction is straightforward.

At last, an analysis result is specified as an element of the global lattice that satisfies all the constraints imposed by all the instructions of the program (`w1` represents the first program counter of a program).

```
Definition Spec (p:program) (s:GlobalLat.t) : Prop :=
  MemLat.order (Base nil,LocvarLat.bottom) (GlobalLat.get s w1) ∧
  ∀ pc instr,
    instr_at p pc = Some instr →
    ∀ c, In c (gen_constraint pc instr) → cstr2prop s c.
```

Figure 2 gives an example of analysis result that fulfills all the constraints of the bubble sort program. In this Figure, we note `c` instead of (`Const c`), `x` instead of (`Var x`) and (`op e1 e2`) instead of (`Bin op e1 e2`). An empty operand stack is not represented. We note $\top$ the interval $[-\infty, +\infty]$.

7   Logical link between concrete and abstract domains

We have now to establish a formal link between the concrete and abstract domains. We first give a semantics to the expression language. Note that the abstraction of references by the length of the array they points-to is visible here since a same expression may evaluate both to a numerical value and a reference or the corresponding size.

```
Inductive sem_expr (h:heap) (l:locvar) : expr → val → Prop :=
| sem_expr_const : ∀ n, sem_expr h l (Const n) (Num n)
```

```
| sem_expr_var : ∀ x v, l x = Def v → sem_expr h l (Var x) v
| sem_expr_bin : ∀ op e1 e2 n1 n2,
  sem_expr h l e1 (Num n1) →
  sem_expr h l e2 (Num n2) →
  sem_expr h l (Bin op e1 e2) (Num (binop_sem op n1 n2))
| sem_expr_ref : ∀ e a loc,
  h loc = Array a →
  sem_expr h l e (Num a.(array_length)) →
  sem_expr h l e (Ref loc)
| sem_expr_length : ∀ e a loc,
  h loc = Array a →
  sem_expr h l e (Ref loc) →
  sem_expr h l e (Num a.(array_length)).
```

We then link each abstract domain $A^\sharp$ with its concrete counterpart A by means of a *concretization* function $\gamma \in A^\sharp \to \mathcal{P}(A)$ that maps each abstract element to a property in the concrete domain. Since subsets are encoded as predicates in the logic of Coq, γ functions can be defined with inductive relations. Note that it is sometimes necessary to parameterize some of the relations with the heap or the local variables.

We first give the concretization function for syntactic expressions, that is directly derived from their semantics.

```
Definition gamma_expr
       (e:ExprLat.t) (h:heap) (l:locvar) (v:val) : Prop :=
  match e with
    | Top ⇒ True
    | Base e ⇒ sem_expr h l e v
    | Bot ⇒ False
  end.
```

The concretization function for lists of expressions is an intermediate function used for defining concretization of stacks.

```
Inductive gamma_list (h:heap) (l:locvar) :
                      list ExprLat.t → list val → Prop :=
| gamma_list_nil : gamma_list h l nil nil
| gamma_list_cons : ∀ e L v q,
    gamma_expr e h l v → gamma_list h l L q →
    gamma_list h l (e::L) (v::q).


Definition gamma_opstack
    (S:OpstackLat.t) (h:heap) (l:locvar) (s:opstack) : Prop :=
  match S with
    | Top ⇒ True
    | Base ab_s ⇒ gamma_list h l ab_s s
    | Bot ⇒ False
  end.
```

Concretization of numerical values or references uses the interval concretization function, and is extended to environments. get denotes here the operator on maps that allow to retrieve the value associated with a key.

```
Inductive gamma_val (i:IntervalLat.t) (h:heap) : val → Prop :=
| gamma_val_num : ∀ n,
  IntervalAbstraction.gamma i n →
  gamma_val i h (Num n)
| gamma_val_ref : ∀ a loc,
  h loc = Array a →
  IntervalAbstraction.gamma i a.(array_length) →
  gamma_val i h (Ref loc).

Definition gamma_locvar
            (L:LocvarLatt.t) (h:heap) (l:locvar) : Prop :=
  ∀ x v, l x = Def v → gamma_val (LocvarLat.get L x) h v.

Definition gamma_mem :
                  MemLat.t → heap → locvar → opstack → Prop :=
  fun M h l s ⇒
    let (S,L) := M in
      match S with
        | Top ⇒ True
        | Base _ ⇒ gamma_opstack S h l s ∧ gamma_locvar L h l
        | Bot ⇒ False
      end.
```

We finally define the concretization function for the whole program as follows.

```
Definition gamma (p:program) (F:GlobalLat.t) : state → Prop :=
  fun st ⇒
    match st with
      | St i s l h ⇒ gamma_mem (GlobalLat.get F i) h l s
      | _ ⇒ True
    end.
```

A natural property required on concretization functions is monotony, *i.e*, if $\sqsubseteq^\sharp$ is the partial order relation on abstract domain $A^\sharp$, a concretization function $\gamma \in A^\sharp \to \mathcal{P}(A)$ must satisfy

$$\forall a_1^\sharp, a_2^\sharp \in A^\sharp, \ a_1^\sharp \sqsubseteq^\sharp a_2^\sharp \ \Rightarrow \ \gamma(a_1^\sharp) \subseteq \gamma(a_2^\sharp)$$

This property is indeed natural, since it expresses the fact that the abstract partial order is related to logical implication in the concrete domain. For our bytecode analysis, each concretization function for each semantic sub-domain is monotone, as expressed by the following lemmas (proofs are omitted here).

```
Lemma gamma_expr_monotone : ∀ e1 e2 h l v,
  gamma_expr e1 h l v → ExprLat.order e1 e2 →
  gamma_expr e2 h l v.
Lemma gamma_opstack_monotone : ∀ S1 S2 h l s,
  gamma_opstack S1 h l s → OpstackLat.order S1 S2 →
  gamma_opstack S2 h l s.
Lemma gamma_val_monotone : ∀ V1 V2 h v,
  gamma_val V1 h v → IntervalLat.order V1 V2 →
```

```
  gamma_val V2 h v.
Lemma gamma_locvar_monotone : ∀ L1 L2 h l,
  gamma_locvar L1 h l → LocvarLat.order L1 L2 →
  gamma_locvar L2 h l.
Lemma gamma_mem_monotone : ∀ M1 M2 h l s,
  gamma_mem M1 h l s → MemLat.order M1 M2 →
  gamma_mem M2 h l s.
Lemma gamma_global_monotone : ∀ p F1 F2 st,
  gamma p F1 st → GlobalLat.order F1 F2 →
  gamma p F2 st.
```

The standard abstract interpretation framework The presentation order
we have chosen from now is not exactly the *official way* since we have presented
the analysis specification before the logical interpretation of abstract domains.
The theory of abstract interpretation proposes a different way for presenting the
construction of static analyses. Abstracting the semantics of a program consists
in restricting the set of properties used to express its behaviour. The notion of
abstraction is naturally represented by a subset of all possible properties. For
instance, the sign abstraction is composed of the five properties $\mathbb{Z}, \mathbb{Z}^+$, $\mathbb{Z}^-$, $\{0\}$
and $\emptyset$ which all belong to $\mathcal{P}(\mathbb{Z})$.

The question that naturally arises is: what is a good abstraction?, *i.e.,* what
are the best choices for defining the set of abstract properties? The "reason-
able hypothesis" proposed by Patrick and Radhia Cousot in their seminal pa-
per [CC79] is that for any property P, the set of all correct approximations
of P must have a least element. In addition, abstract properties are defined as
elements of a distinct lattice, instead of being a subset of the set of concrete
properties. This gives birth to the notion of a Galois connection.

Definition 3. *Galois connection Let* $(L_1, \sqsubseteq_1, \bigsqcup_1, \prod_1)$ *and* $(L_2, \sqsubseteq_2, \bigsqcup_2, \prod_2)$ *be
two complete lattices. A pair of functions* $\alpha \in L_1 \to L_2$ *and* $\gamma \in L_2 \to L_1$ *is a*
Galois connection *if*

$$\forall x_1 \in L_1, \forall x_2 \in L_2, \ \alpha(x_1) \sqsubseteq_1 x_2 \iff x_1 \sqsubseteq_2 \gamma(x_2)$$

In that case, we use the following notation.

$$\left(L_1, \sqsubseteq_1, \bigsqcup_1, \prod_1\right) \xrightarrow[\alpha]{\gamma} \left(L_2, \sqsubseteq_2, \bigsqcup_2, \prod_2\right)$$

Given such a Galois connection, the following properties hold [CC92a].

- α and γ are monotonic functions
- α is a union morphism: $\forall S \subseteq L_1, \ \gamma\left(\bigsqcup_1 S\right) = \prod_2 \alpha\left(S\right)$
- γ is an intersection morphism: $\forall S \subseteq L_2, \ \gamma\left(\prod_2 S\right) = \prod_1 \gamma\left(S\right)$

Conversely, if $(L_1, \sqsubseteq_1, \bigsqcup_1, \prod_1)$ and $(L_2, \sqsubseteq_2, \bigsqcup_2, \prod_2)$ are two complete lattices,
and if $\gamma \in L_2 \to L_1$ is an intersection morphism, then there exists a unique

function $\alpha \in L_1 \to L_2$ such that (α, γ) is a Galois connection between L_1 et L_2. α is defined by $\alpha(x) = \bigsqcap_2 \{ y \mid x \sqsubseteq_2 \gamma(y) \}$.

From now on, we will assume that our concrete and abstract semantics are expressed in two lattices A and $A^\sharp$, respectively, in relation with a Galois connection:

$$\left(A, \sqsubseteq, \bigsqcup, \bigsqcap \right) \xrightleftharpoons[\alpha]{\gamma} \left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp \right)$$

Given a concrete semantics $[\![P]\!] \in A$ of a program P, the approximated behaviour of P will be computed by an abstract semantics $[\![P]\!] \in A^\sharp$. The abstraction will be correct iff $[\![P]\!] \sqsubseteq \gamma([\![P]\!]^\sharp)$, or equivalently $\alpha([\![P]\!]) \sqsubseteq^\sharp [\![P]\!]^\sharp$. $\alpha([\![P]\!])$ thus represents the best abstract semantics we can get with this choice of (α, γ). Fortunately, the theory of abstract interpretation gives more details on how we can construct $[\![P]\!]^\sharp$. The idea is to follow the structure of $[\![P]\!]$. If $[\![P]\!]$ is expressed as a composition of functions, $[\![P]\!]^\sharp$ will be a composition of abstract functions: this is the principle of abstract evaluation, that we briefly develop now.

Let us consider a function $f \in A \to A$. A concrete computation in A is represented by a couple $(a, f(a))$ with $a \in A$. We have to define a function $f^\sharp \in A^\sharp \to A^\sharp$ that correctly approximates all concrete computations. Formally, we can state the following definition.

Definition 4 (Correct function approximation). *Function $f^\sharp \in A^\sharp \to A^\sharp$ is a correct approximation of $f \in A \to A$ if*

$$\forall a \in A, a^\sharp \in A^\sharp, \alpha(a) \sqsubseteq^\sharp a^\sharp \Rightarrow \alpha(f(a)) \sqsubseteq^\sharp f^\sharp(a^\sharp)$$

If the concrete or the abstract function is monotone, we can give the following equivalent correctness criteria.

Theorem 1. *If $f \in A \to A$ or $f^\sharp \in A^\sharp \to A^\sharp$ is monotonic, the following assertions are equivalent:*

(i) $f^\sharp$ is a correct approximation of f,
(ii) $\alpha \circ f \mathrel{\dot{\sqsubseteq}^\sharp} f^\sharp \circ \alpha$
(ii) $\alpha \circ f \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} f^\sharp$
(iv) $f \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} \gamma \circ f^\sharp$

where $\dot{\sqsubseteq}$ is the point-wise order between functions.

Abstract interpretation provides a characterisation of correct abstractions of the concrete semantics, but also tackles an optimality issue. The case where $f^\sharp = \alpha \circ f \circ \gamma$ indeed gives an optimal approximation for $f^\sharp$. However, this is rather a specification than an implementation, since function α is rarely computable. Using an abstraction function α thus requires providing proofs that a given subset of concrete states respect a intensionally defined property. On the other hand, we are mainly interested here in ensuring correctness of our analyses, rather that systematically design optimal ones. We thus only retain concretization functions in our framework, requiring that these functions are monotone, which is one condition of the minimalist framework proposed by P. and R. Cousot [CC92b].

8   Soundness proof

The analysis specification must be proved sound with respect to the concrete operational semantics and the logical interpretation of the abstract domain previously defined. The main component of this proof is a *subject reduction* theorem. This theorem intuitively expresses the fact that progression of one step in the concrete operational semantics preserves the correctness w.r.t the concretization function, provided that the abstract state respects the locally generated constraints.

```
Lemma step_correct : ∀ p F i ins h l s s2,
  instr_at p i = Some ins →
  (∀ c, In c (gen_constraint i ins) → cstr2prop F c) →
  (St i s l h) -[p]-> s2 →
  wf_state (St i s l h) →
  gamma p F (St i s l h) →
  gamma p F s2.
```

Here `wf_state` represents a well-formedness property on reachable states. We have to prove that it is a semantic invariant.

```
Inductive wf_val (h:heap) : val → Prop :=
| wf_val_num : ∀ n, wf_val h (Num n)
| wf_val_ref : ∀ a loc, h loc = Array a → wf_val h (Ref loc).

Definition wf_state (st:state) : Prop :=
  match st with
    | St pc s l h ⇒ (∀ x v, l x = Def v → wf_val h v)
                                        ∧ (∀ v, In v s → wf_val h v)
    | Error ⇒ True
  end.

Lemma wf_state_invariant_step : ∀ p st1 st2,
  st1 -[p]-> st2 → wf_state st1 → wf_state st2.
```

In order to prove the subject reduction lemma, we proceed by a case study on `(St i s l h) -[p]->s2`. For each semantic step towards a state `st2` of the form `(St i2 s2 l2 h2)`, we have to prove a result of the form

```
        gamma_mem (GlobalLat.get F i2) h2 l2 s2.
```

To achieve this, we choose among the constraints associated with the current instruction the one which has `i2` as target. Using the monotony of `gamma_mem`, there remains to show

```
        gamma_mem (transf (GlobalLat.get F i)) h2 l2 s2
```

where `f` is the transfer function of the chosen constraint. Combined with the hypothesis `gamma_mem (GlobalLat.get F i) h l s`, we thus have recovered the standard criterion

$$f \circ \gamma \subseteq \gamma \circ f^{\sharp}$$

Thanks to the subject reduction lemma, we can establish that any solution of the constraint-based specification is a correct approximation of the set of reachable sets.

Lemma spec_correct : ∀ p F,
  Spec p F → ∀ st, ReachableStates p st → gamma p F st.

The proof is based on the induction principle associated with ReachableStates which generate two sufficient properties to be proved.

  ∀ p F, Spec p F, ∀ st, InitState st → gamma p F st   (1)

  ∀ p F, Spec p F, ∀ st1 st2,
     ReachableStates st1 → st1 -[p]-> st2 →   (2)
     gamma p F st1 → gamma p F st2

The first property is proved thanks to the constraint on (GlobalLat.get F w1). The second property is easily proved with the subject reduction lemma and the proof of invariance of wf_state.

9 Constraint generation

It remains to collect all the constraints of a program in order to generate an equation system. This part is generic: it does not depend on the constraints chosen for each instruction. It is then adaptable for many static analyses on our bytecode language.

Fixpoint collect_all_cstr (gen:pc → instruction → list cstr)
                      (l:list (pc*instruction)) : list cstr :=
  match l **with**
    | nil ⇒ nil
    | (i,ins)::q ⇒ (gen i ins)++(collect_all_cstr f q)
  end.

The collected list of constraints is then turned into a global operator on the global lattice.

Definition global_fun (l:list cstr) : GlobalLat.t → GlobalLat.t
  := **fun** s ⇒
  fold_left
    (**fun** s' c ⇒
      GlobalLat.modify s' c.(target)
        (MemLat.join (c.(constraint) (GlobalLat.get s c.(source))))
)
    l
    (GlobalLat.modify
      GlobalLat.bottom w1 (**fun** _ ⇒ (Base nil,LocvarLat.bottom))).

We rely here on the operator modify of the maps library which updates a map. More precisely, modify m x f return a new map equal to m, except for the key x which is associated to the value (f (get m x)).

We end this part by proving that any post-fixpoint of the operator

```
global_fun (collect_all_cstr gen_constraint p)
```

is a solution of the constraint-based specification.

```
Lemma global_fun_correct : ∀ p s,
  GlobalLat.order
    (global_fun (collect_all_cstr gen_constraint p) s) s →
  Spec p s.
```

10 Fixpoint computations

The semantics of a program can often be expressed as the least fixpoint of a monotone operator. The abstract semantics mimics this situation, and approximate the concrete fixpoint by an abstract one. The classical setting of complete lattices provides fundamental results, that we briefly recall and illustrate with the example of the interval lattice, before addressing the particularities of this issue in constructive logic.

Classical results The Knaster-Tarski theorem states that in any complete lattice, there exists a smallest fixpoint for any monotone function. This theorem ensures that this smallest fixpoint coincides with the smallest post-fixpoint (see Figure 6(a)), however it does not say anything on the method that could be followed to compute such a fixpoint. If the function is continuous, the Kleene theorem provides a more effective characterisation by providing a computation method: iterating the function from the smallest element of the lattice indeed reaches the smallest fixpoint, *if the iteration terminates* (see Figure 6(b)). The most simple criterion ensuring termination is the *ascending chain condition* (ACC), that forbids existence of infinite strictly increasing chains. Unfortunately, this condition is quite strong, and does not admit constructive proofs in general, even for lattices as simple as $\{\bot, \top\}$ [Pic05].

Instead of computing a least (post-)fixpoint of a monotone function, we can accomodate ourselves with an over-approximation, keeping correction in mind, but loosing optimality. This decision is generally taken when the lattice does not respect the ACC property, or when the iteration chain would be too long to allow an acceptable computation time.

The solution proposed by P. and R. Cousot consists in accelerating the ascending iteration, thus reaching a post-fixpoint, but not necessarily the least one. It is done by using a binary *widening* operator, that extrapolates both of its arguments. Intuitively, at the n-th iteration, the n-th iterate of the function is compared to the preceeding value, in order to detect the beginning of a possible infinite or very long chain. If this is the case, we skip to a point farther up in the iteration. If the binary operator fulfills the requirements of a widening operator (see [CC92a] for details), then this modified iteration is guaranteed to converge to a post-fixpoint in finite time (see Figure 6(c)).

When a post-fixpoint is reached, a new descending iteration starting from this point allows to improve the approximation. The termination issue is similar to that of the ascending iteration, and a *narrowing* operator can be used to accelerate and ensure convergence.
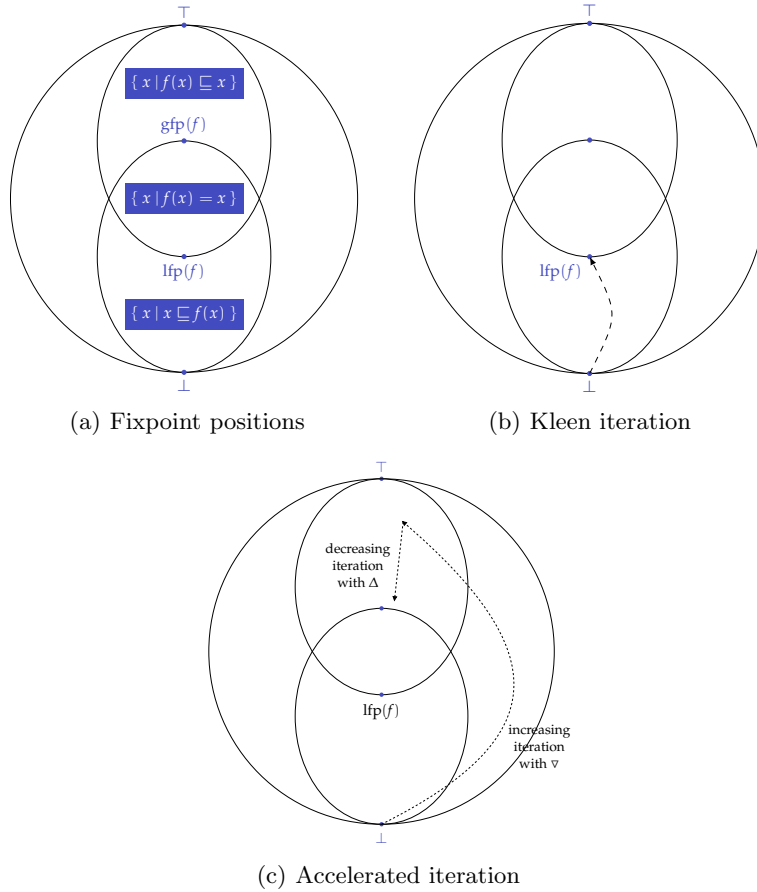

(a) Fixpoint positions

(b) Kleen iteration

(c) Accelerated iteration

Fig. 6. Fixpoint positions in a complete lattice.

Interval lattice example The interval lattice obviously does not fulfill the ascending chain condition. For instance, $\bot_{Int} \sqsubseteq_{Int} [0,0] \sqsubseteq_{Int} [0,1] \sqsubseteq_{Int} [0,2] \sqsubseteq_{Int}$ $\ldots \sqsubseteq_{Int} [0,n] \sqsubseteq_{Int} \ldots$ is an infinite increasing chain. We will thus define a widening operator ∇_{Int}. In the infinite chain above, we would like to replace interval $[0,n]$ by $[0,+\infty]$ after a finite (preferably small) number of iterations. We would like for instance that $[0,1]\nabla_{Int}[0,2] = +\infty$. The general definition of ∇_{Int} is as follows.

$$[a,b]\nabla_{Int}[a',b'] = [\text{if } a' < a \text{ then } -\infty \text{ else } a,$$
$$\text{if } b' > b \text{ then } +\infty \text{ else } b]$$
$$\bot_{Int}\nabla_{Int}[a,b] = [a,b]$$
$$I\nabla_{Int}\bot_{Int} = I$$

As an example, let us take a program composed of one single loop decrementing a counter x, starting from value 100 and stopping when x reaches 0, which control flow graph and abstract semantics are displayed on Figure 7. The abstract semantics is written as a set of inequations, where X_i denotes the abstract value of variable x at program point i.

$$[100,100] \sqsubseteq X_1$$
$$\left(X_2 -^\sharp [1,1]\right) \sqsubseteq X_1$$
$$[1,+\infty] \sqcap_{Int} X_1 \sqsubseteq X_2$$
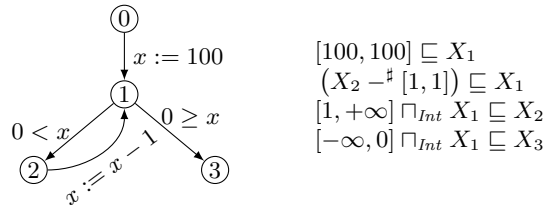$$[-\infty,0] \sqcap_{Int} X_1 \sqsubseteq X_3$$

Fig. 7. A program and the specification of its abstract semantics

We now want to compute the result of the analysis, *i.e.* a solution to this set of inequations. Equivalently, we are looking for a post-fixpoint of the fonction F constructed as the composition of elementary functions $F_{i,j} : X \mapsto X[j \mapsto X_j \sqcup_{Int} f(X_i)]$ for each constraint of the form $f(X_i) \sqsubseteq X_j$. We then have to choose an iteration strategy for evaluating these elementary functions. We take for instance a round-robin scheme, computing X_1, X_2 and X_3 in turn and iterating over the three equations defining these variables. We thus compute the successive iterates of the following sequences.

$$X_1^0 = \bot \quad X_1^{n+1} = [100,100] \sqcup_{Int} \left(X_2^n -^\sharp [1,1]\right)$$
$$X_2^0 = \bot \quad X_2^{n+1} = [1,+\infty] \sqcap_{Int} X_1^{n+1}$$
$$X_3^0 = \bot \quad X_3^{n+1} = [-\infty,0] \sqcap_{Int} X_1^{n+1}$$

The result of this computation is given below.

Iteration	0	1	2	3	4		100	101
X_1	$\bot$	$[100,100]$	$[99,100]$	$[98,100]$	$[97,100]$	$\cdots$	$[1,100]$	$[0,100]$
X_2	$\bot$	$[100,100]$	$[99,100]$	$[98,100]$	$[97,100]$	$\cdots$	$[1,100]$	$[1,100]$
X_3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$[0,0]$

Even if on this example the iteration converges in finite time, the length of the iteration sequence depends on the concrete value of the loop bound. If we now introduce widening operations at each control point, we get the following equations

$$X_1^0 = \bot \qquad X_1^{n+1} = X_1^n \nabla_{Int} \left([100,100] \sqcup_{Int} \left(X_2^n -^\sharp [1,1]\right)\right)$$
$$X_2^0 = \bot \qquad X_2^{n+1} = X_2^n \nabla_{Int} \left([1,+\infty] \sqcap_{Int} X_1^{n+1}\right)$$
$$X_3^0 = \bot \qquad X_3^{n+1} = X_3^n \nabla_{Int} \left([-\infty,0] \sqcap_{Int} X_1^{n+1}\right)$$

resulting in a much shorter iteration sequence.

Iteration	0	1	2
X_1	$\bot$	$[100,100]$	$[-\infty,100]$
X_2	$\bot$	$[100,100]$	$[-\infty,100]$
X_3	$\bot$	$\bot$	$[-\infty,0]$

As explained above, a subsequent iteration using a narrowing operator can be used to refine the post-fixpoint computed by the widening iteration sequence. The general definition of the narrowing operator we use for intervals is as follows.

$$[a,b]\Delta_{Int}[c,d] = [\text{if } a = -\infty \text{ then } c \text{ else } a \;\; ; \;\; \text{if } b = +\infty \text{ then } d \text{ else } b]$$
$$I \;\; \Delta_{Int} \bot_{Int} = \bot_{Int}$$
$$\bot_{Int} \Delta_{Int} \;\; I \;\; = \bot_{Int}$$

The intuition behind this definition is that only infinite bounds, that may have been too widely extrapolated, are refined down to finite ones. In practice, a few iterations already notably improve the result that has been obtained after widening. If we come back to our previous example and introduce a narrowing operation for each control point, starting from the already computed result we get convergence after only 2 steps, once again with a round-robin strategy.

$$Y_1^0 = X_1^2 \qquad Y_1^{n+1} = Y_1^n \Delta_{Int} \left([100,100] \sqcup_{Int} \left(Y_2^n -^\sharp [1,1]\right)\right)$$
$$Y_2^0 = X_2^2 \qquad Y_2^{n+1} = Y_2^n \Delta_{Int} \left([1,+\infty] \sqcap_{Int} Y_1^{n+1}\right)$$
$$Y_3^0 = X_3^2 \qquad Y_3^{n+1} = Y_3^n \Delta_{Int} \left([-\infty,0] \sqcap_{Int} Y_1^{n+1}\right)$$

Iteration	0	1	2
Y_1	$[-\infty,100]$	$[-\infty,100]$	$[0,100]$
Y_2	$[-\infty,100]$	$[1,100]$	$[1,100]$
Y_3	$[-\infty,0]$	$[-\infty,0]$	$[0,0]$

Besides the introduction of widening operators, the order in which equations are selected in the fixpoint computation has a noteworthy influence on the convergence speed of the iteration and on the precision of final result. On the other hand, widening operators induce strong over-approximations and must be used as less as possible. A balanced solution consists in introducing widening operations only at a subset of the control points, and selecting an appropriate iteration order. If we start from the following equation system

$$\begin{cases} X_1 = f_1(X_1, \ldots, X_n) \\ \quad \vdots \\ X_n = f_n(X_1, \ldots, X_n) \end{cases}$$

it is sufficient (and more precise) to use ∇_{Int} (and Δ_{Int}) for a selection of indices W such that each dependence cycle in the control flow graph of the system goes through at least one point in W.

$$\forall k = 1..n, \ X_k^{i+1} = \begin{array}{l} X_k^i \nabla_{Int} f_k(X_1^i, \ldots, X_n^i) \text{ if } k \in W \\ f_k(X_1^i, \ldots, X_n^i) \qquad \text{otherwise} \end{array}$$

Constructivity issues The classical definition of ACC, as well as those of widening or narrowing operators, are not well suited to constructive proofs, even if it is possible to give a constructive proof of the Knaster-Tarski theorem for complete lattices. In order to implement fixpoint computations in Coq, it is possible to modify the definitions of ACC, widening and narrowing, using the notion of well-founded relation. Note that these definitions are equivalent to the initial ones.

Widening and narrowing in the lattice library The above definitions are included in a Coq module signature `Lattice` described in Section 5. The difficulty consists in effectively constructing complex abstract domains. Defining widening and narrowing operators can indeed be tricky, since they have to be accompagnied by their termination proof. The functors provided by the Coq library alleviate this problem, since for each functor such as `ListLattice` or `ArrayBinLattice`, a proof of termination for widening and narrowing operators is directly constructed from the termination proofs included in the lattice arguments of the functor. One important technical point concerns the domain of key that is used in functor `ArrayBinLattice`. As we have seen before, keys has of type `word`. In order to prove termination of point-wise widening on functions we need to consider a finite number of keys. We hence define `word` as a *finite* type.

Definition word : **Set** := {p:positive | inf_log_bool p 32 = true}.

Such a type reads as follow: an element of type `word` is a couple `(p,h)` such that `p` inhabits the type `positive` of binary numbers and `h` is a proof that `p` as at most 32 bits. It ensures we have only a finite number of keys in our maps.

This modular construction allows us to construct a generic post-fixpoint solver based on widening/narrowing iterations. It takes the form of module functor that, given a lattice module `L`, implements a function `pfp` that computes a post-fixpoint of any operator on `L.t`.

Module PostFixPointSolver (L:Lattice).
  Definition pfp (f : L.t → L.t) : { x:L.t | L.order (f x) x }
    (* ... omitted ... *)
End PostFixPointSolver.

We use this functor on the lattice module `GlobalLat` in order to find a post-fixpoint of the operator build in the last Section. Combined with the lemmas `spec_correct` of Section 8 and `global_fun_correct` of Section 9 we obtain a certified analyzer of type

```
analyzer : ∀ p:program,
   { F:GlobalLat.t | ∀ st, ReachableStates p st → gamma p F st}
```

11 Abstract checking

The combination of all the previous parts has allowed us to obtain a provably safe
over-approximation of the reachable states. We will now use this approximation
to check if a program is safe.

Given an abstract domain $(A^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$ we provide an abstract safety check
function $check^\sharp \in A^\sharp \to bool$ that checks, in the abstract world, if a given safety
property is satisfied. For a given program, if the abstract check succeeds then
the program should be safe. It is does not succeed, we can not conclude anything
because of the over-approximation.

For our case study, our safety property, for a program p has been defined as:

```
∀ st, ReachableStates p st → Safe st
```

We first introduce an intermediate safety property `PreSafe`, using a characteri-
zation of states that definitively lead to error states.

```
Inductive pre_state_error (p:program) : state → Prop :=
| no_pre_state_error_newarray : ∀ pc n s l h,
  instr_at p pc = Some Newarray →
  n<0 →
  pre_state_error p (St pc (Num n :: s) l h)
| no_pre_state_error_iaload : ∀ pc loc s l h i a,
  instr_at p pc = Some Arrayload →
  h loc = Array a →
  ¬ (0 <= i < array_length a) →
  pre_state_error p (St pc (Num i::Ref loc :: s) l h)
| pre_state_error_iastore : ∀ pc loc s l h i n a,
  instr_at p pc = Some Arraystore →
  h loc = Array a →
  ¬ (0 <= i < array_length a →
  pre_state_error p (St pc (Num n :: Num i :: Ref loc :: s) l h).

Definition PreSafe (p:program) (st:state) : Prop :=
  ¬ pre_state_error p st.
```

Any successor of a *pre-safe* state is safe.

```
Lemma pre_safe_implies_safe_step : ∀ p,
  ∀ st1 st2, PreSafe p st1 → st1 -[p]-> st2 → Safe st2.
```

As a consequence, it is sufficient to check if all reachable states satisfy the
`PreSafe` property.

```
Lemma pre_safe_implies_safe_reach : ∀ p,
  (∀ pc st, ReachableStates p st → PreSafe p st) →
  ∀ st, ReachableStates p st → Safe st.
```

For all instruction that may lead to the error state, the abstract checker verifies if the inferred abstract property enforces the `PreSafe` property.

```
Definition check_instr
  (F:GlobalLat.t) (i:pc) (ins:instruction) : bool :=
  match ins with
    | Newarray ⇒
      match GlobalLattice.get F i with
        | (Top,_) ⇒ false
        | (Base (Top::_),L) ⇒ false
        | (Base (Base e::_),L) ⇒ Interval.check_ge L e (Const 0)
        | _ ⇒ true
      end
    | Arrayload ⇒
      match GlobalLat.get F i with
        | (Top,_)
        | (Base (Top::_::_),_)
        | (Base (_::Top::_),_) ⇒ false
        | (Base (Base e_i::Base e_a::_),L) ⇒
          Interval.check_ge L e_i (Const 0)
            && Interval.check_gt L e_a e_i
        | _ ⇒ true
      end
    | Arraystore ⇒
      match GlobalLattice.get F i with
        | (Top,_)
        | (Base (_::Top::_::_),_)
        | (Base (_::_::Top::_),_) ⇒ false
        | (Base (_::Base e_i::Base e_a::_),L) ⇒
          Interval.check_ge e_i (Const 0)
            && Interval.check_gt e_a e_i
        | _ ⇒ true
      end
    | _ ⇒ true
  end.

Fixpoint check (l:program) (F:GlobalLattice.t) : bool :=
  match l with
    | nil ⇒ true
    | (pc,ins)::q ⇒
      if check_instr F pc ins then check q F else false
  end.
```

The function `Interval.check_ge` (resp. `Interval.check_gt`) checks if an expression is greater (resp. strictly greater) than an other one in an abstract set of local variables (here denoted by `L`).

We then prove the soundness of the check function with respect to the `PreSafe` property.

```
Lemma check_true_implies_PreSafe : ∀ p F,
  (∀ st, ReachableStates p st → gamma p F st) →
```

```
  check p F = true →
  ∀ st, ReachableStates p st → PreSafe st.
```

Combined with the previous result `pre_safe_implies_safe_reach`, this gives us the final soundness property:

```
Lemma check_true_implies_PreSafe : ∀ p F,
  (∀ st, ReachableStates p st → gamma p F st) →
  check p F = true →
  ∀ st, ReachableStates p st → Safe st.
```

12   Conclusions

We can conclude our development by defining a verifier that runs the analyzer and then makes an abstract check on it. To vary a little our Coq style we will perform a proof-as-programing construction that is permitted by the Curry-Howard correspondence underlying the Coq system.

```
Definition verifier (p:program) :
  { b:bool | b = true → ∀ st, ReachableStates p st → Safe st}.
Proof.
  intros p.
  destruct (analyzer p) as [F h].
  exists (check p F).
  exact (check_true_implies_PreSafe p F h).
Defined.
```

This reads as follow:

– First we state that we want to construct a function `verifier` that takes a program `p` as argument and returns a dependent pair `(b,h)` such that if the boolean `b` is equal to `true` then the program `p` is safe;
– We then start an interactive proof in order to progressively prove this type is inhabited. What we have to prove is mainly a statement like:

  ```
  ∀ p, ∃ b, b = true → ∀ st, ReachableStates p st → Safe st
  ```

– We first introduce `p` in our context and then destruct the result of `analyzer p` as a pair `(F,h)`. By typing of `analyzer`, the term `F` has type `GlobalLat.t` and `h` has type `∀ st, ReachableStates p st  → gamma p F st`;
– The first part of the result by giving the boolean value `(check p F)`;
– The Coq system now ask us to fill the last missing part and prove

  ```
  check p F = true → ∀ st, ReachableStates p st → Safe st
  ```

  i.e. give a term of the corresponding type;
– We conclude using the term `(check_true_implies_PreSafe p F h)` which is exactly of the right type.

This last function and the rest of the development can be automatically extracted into a Ocaml program. We obtain for example the following Ocaml code for for the `verifier` function.

```
let verifier p = check p (analyzer p)
```

Note that all the logical elements have been discarded. When launched on our bubble sort program, the extracted analyser computes the result given in Figure 2 and successfully apply all the abstract checks.

The results presented in this tutorial papers demonstrates how to construct a non-trivial, provably correct abstract interpreter inside the Coq. Using the extraction mechanism we can extract a certified array-bound verifier and bridges the gap that often exists between a paper-specification of an analysis and the analyser that is actually implemented. Thanks to the general methodology we follow and the lattice library we provide, our approach is applicable to a large variety of program analyses for different language paradigms.

References

[BD04]   G. Barthe and G. Dufay. A tool-assisted framework for certified bytecode verification. In *Proc. of FASE'04*, volume 2984 of *LNCS*, pages 99–113. Springer, 2004.

[BDHdS01] G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: a toolset for reasoning about JavaCard. In *Proc. of e-SMART'01*. Springer LNCS vol. 2140, 2001.

[BDJ⁺01] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the JavaCard platform. In *Proc. ESOP'01*, number 2028 in LNCS. Springer-Verlag, 2001.

[Ber08]   Yves Bertot. Structural abstract interpretation, a formal study using Coq. In *LERNET Summer School*, LNCS. Springer, 2008.

[BGL06]   Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES'04*, pages 66–81. Springer LNCS vol. 3839, 2006.

[CC77]   P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[CC79]   P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL'79*, pages 269–282. ACM Press, 1979.

[CC92a]   P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b]   P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CGD06]   S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES'04*, pages 115–137. Springer LNCS vol. 3839, 2006.

[CJPR05] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[CJPS05]   D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *Procóf FM'05*, pages 91–106. Springer LNCS vol. 3582, 2005.

[Coq09]   Coq development team. The Coq proof assistant reference manual V8.2. Technical report, INRIA, France, 2009. http://coq.inria.fr/doc/main.html.

[Cou99]   P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[KN02]   G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[KN06]   G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[LCH07]   D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ml. In *Proc. of POPL'07*, pages 173–184. ACM Press, 2007.

[Ler06]   X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of POPL'06*, pages 42–54. ACM Press, 2006.

[MF99]   G. McGraw and E. W. Felten. *Securing Java: getting down to business with mobile code*. John Wiley & Sons, Inc., 1999.

[Mon98]   D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. In french.

[Pic05]   D. Pichardie. *Interprtation abstraite en logique intuitionniste : extraction d'analyseurs Java certifis*. PhD thesis, Universit Rennes 1, 2005. In french.

[Pic08]   D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proc. of FICS'08*, volume 212 of *Electronic Notes in Theoretical Computer Science*, pages 225–239, 2008.