Symbolic derivation of abstract interpreter in Coq

Dominique Barbe

***Abstract*—Software reliability is a sensitive problem, from the economic point of view (development costs) and from the human point of view (embedded computing). There exists two methods to certify a software. Programs can be verified by hand, using a proof assistant. Programs can also be verified automatically by a dedicated tool. During this internship, we studied the merging of these two methods. In particular, we tried to completely use the theoretical framework used for static analysis.**

INTRODUCTION

How can we trust the programs incorporated in the objects around us ? Bugs are frequent : they can be seen in cell phones, hallway's screens, printer's terminals, etc. But every program can undergo bugs. In particular, the ones deployed within nuclear plants or planes. Any flaw in one of these sensitive software can have serious consequences. Thus, these kind of programs are costly to develop because a lot of means are deployed to avoid any mistake.

Let's take this C program as an example :

```
int t[25] = {0} ;
int x = y = 0 ;
while (y < 10) {
  y = y + 1 ;
  if (x < 25) {
    x = x + y ;
  } else {
    x = x - y
  }
}
t[x] = 1 ;
```

To be sure that this program is sound, a few questions must be answered : do the program halt ? Does the last command is legal (is x a valid array index) ? Different methods exists to answer these question. For example, proving manually the correctness, making tests, etc. But it would be useful to answer automatically to these kind of questions. The goal is to scale and analyse larger programs. Large programs can't be verified by hand, either because it is unsafe, or too much time and money would be spent.

The general problem here is the one of program semantics. Given the text of a program (its syntax) and a runtime environment, determine its possible executions (its semantic). We want to address this problem algorithmically. Does a program capable of computing the semantic of any program exists ? Such a program would be used for goals such as :

- determine if a program halt ;
- determine if a program computes the expected values ;
- determine the areas of the memory reserved by the program (to free them later).

The halting problem is undecidable. A program that could state for any program and for any input if it stops does not exist. The first goal can not be achieved . But a more general theorem exists. Rice's theorem states no non-trivial properties can be computed from the source code of a program. This theorem applies to programming languages that are as expressive as a Turing machine. Thus, there exists no algorithm to compute program's semantics.

The problem of program's semantics can not be completely solved. Two tracks are possible. Either abandoning the idea of a complete automation. Or inventing algorithms to analyse a fraction of the programs.

For the first track, one can use a proof assistant, such as Coq or Isabelle, to certify some aspects of a program. A proof assistant is a tool allowing formalization and proof of mathematical theorems. In particular, this applies to theorems relatives to the execution of programs. Such methods are not automatic : one must guide the proof assistant though proofs. But proof assistant are considered as reliable. They often rely on small kernels, based upon mathematical theories. Thus, they can ensure that no flaws lie within the reasoning made. Moreover, some proof assistants still offer automation for precise settings.

For the second track, one searches a tool capable of analysing automatically programs. Two families of such tools exist. In dynamic analysis, programs are executed and its effects analysed. This method has the advantage of being easy to put in place. But it can't address every problem (as the halting one), because exhaustive testing is almost impossible. In static analysis, programs are read and properties are computed automatically. The program is never executed. Here, we are interested in

static analysis by abstract interpretation, that we will detail later.

A static analyser is a program that read a code and try to compute its semantic. But the analyser is a program too, sometimes complex. Who will verify the analyser ? We must find a starting point to put our trust. To solution explored here is to merge the two methods : using a proof assistant to build a static analyser. Here, we refer to the proof assistant Coq, that will be introduced further. The goal is to have an analyser correct by construction. We use the proof assistant to prove correct each of its mechanisms.

To build the analyzer, we use the following theoretical framework : abstract interpretation. We use this theory to approximate program's semantics. In abstract interpretation, two distinguished worlds exist. The concrete world is used to express properties relatives to programs. The abstract world approximate such properties. Two functions link these worlds : α and γ. α is the abstraction function and γ is the concretization one. They can be used to express what a correct approximation is. Actually, abstract interpretation is already used to run static analysis. But the two function can also be used to express what an optimal approximation is. The main goal of this internship was to explore this idea.

I. Preliminaries

A. The proof assistant Coq

Coq is a proof assistant developed since the eighties at INRIA. It is a tool allowing to write and proof mathematical theorems. It offers a small degree of automation. Coq is based on a small kernel using the theory of the calculus of inductive constructions. Its development is part of a broader logic of systematic verification of IT tools and mathematical proofs. For example, is was used to develop CompCert, a certified C compiler. Another example is the implementation and proof of the four colors theorem.

We present here the specificities of Coq. Unlike other languages, it do not provide usual data types (integers, booleans, etc). In place, it offers a system of inductive types allowing to redefine the ones needed. Coq also manipulate predicates, functions and theorems. Proofs are interactive in Coq. For each theorem Coq shows the associated goal. As long as the goal is not solved, "tactics" are applied to transform the goal, solving it or dividing it.

Coq offers an extraction mechanism : it can generate Ocaml code from Coq code. Thus, when we finish the analyzer, we will export it to a more realistic language (performance-speaking).

B. Abstract interpretation

1) Introduction: Abstract interpretation is a theory introduced by Patrick and Radhia Cousot in the seventies. It offers a theoretical framework to formalize methods of static analysis. The main idea is to approximate program's semantics. Indeed, as it is in general not computable, program's operations are simplified. By loosing precision, one seeks to gain decidability. The approximated semantic must remain precise enough to answer interesting questions.

Abstract interpretation allows to compute, at different points of program, properties on its variables. Such properties can be "x is even", "$0 \leq x < 100$" or "x is an imaginary root $x^7 - x^2 + 1$". Generally, for $x \in \mathbb{Z}$, a property on x is an element of $\wp(\mathbb{Z})$. As the worlds of properties is too complex ($\wp(\mathbb{Z})$ is too big), a simpler one must be found.

An example often used is the abstraction by sign. Instead of considering every properties on integers, only the ones relatives to the sign are kept. The abstract world is $\{\emptyset, \{0\}, \mathbb{Z}_-, \mathbb{Z}_+, \mathbb{Z}\}$ (we recall that properties are elements of $\wp(\mathbb{Z})$. Now, for every concrete property, an abstract representative must be found. Such a representative is an approximation of the original property. In the context of static analysis, over-approximations must be done. The goal is to certify that programs are corrects. With this idea in mind, one had better be too careful than not enough. False positives (the analyzer finds an error when there is none) are preferable to false negatives (the analyzer does not find an error when there is one). Thus, over-approximation, *i.e* too generic properties, must be done. Each element of $\wp(\mathbb{Z})$ is abstract by an element of $\{\emptyset, \{0\}, \mathbb{Z}_-, \mathbb{Z}_+, \mathbb{Z}\}$ containing it. Such an abstraction should be kept as small as possible, to stay precise.

Abstract interpretation is based upon the theory of Galois connections, explained further.

2) Galois connections:

a) Definition: a **lattice** is a quadruplet $(E, \sqsubseteq, \sqcup, \sqcap)$ where

- E is a set
- $\sqsubseteq$ is a partial on E
- $\sqcup : E \times E \longrightarrow E$ is a least upper bound :
  - $\forall (x, y) \in E^2$:
    $$x \sqsubseteq x \sqcup y \quad \text{and} \quad y \sqsubseteq x \sqcup y$$
  - $\forall (x, y, m) \in E^3$,
    $$(x \sqsubseteq m \wedge y \sqsubseteq m) \Rightarrow x \sqcup y \sqsubseteq m$$

- $\sqcup : E \times E \longrightarrow E$ is a greatest lower bound :
  - $\forall (x, y) \in E^2$:

$$x \sqcap y \sqsubseteq x \quad \text{and} \quad x \sqcap y \sqsubseteq y$$

  - $\forall (x, y, m) \in E^3$,

$$(m \sqsubseteq x \wedge m \sqsubseteq y) \Rightarrow m \sqsubseteq x \sqcap y$$

b) Example: $\wp(\mathbb{Z})$ can be provided with a structure of lattice. The order is the usual inclusion. It represents the implication (between properties). It is partial, as two properties can't be always compared. The least upper bound is the usual union. It represents the disjunction between hypothesis. The greatest lower bound is the usual intersection. It represents the conjunction between hypothesis.

c) Definition: given two lattices, $(E, \sqsubseteq, \sqcup, \sqcap)$ and $\left(E^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#\right)$, and two functions, $\alpha : E \longrightarrow E^\#$ and $\gamma : E^\# \longrightarrow E$. The pair (α, γ) is said to be a **Galois connection** when :

$$\forall (x, x^\#) \in E \times E^\#, \quad \alpha(x) \sqsubseteq^\# x^\# \Leftrightarrow x \sqsubseteq \gamma(x^\#)$$

E is the concrete world and $E^\#$ the abstract world. α is the abstraction function. For each concrete property P, it gives its best abstraction $P^\#$. γ is the concretization function. For each abstract property $P^\#$, it gives the concrete property that is best approximated by it.

We try to explain the given property. Let x be an element of E, a property. $\alpha(x)$ is its abstraction, an abstract property. Let $x^\#$ be an abstract property approximating x. Stating that α is the best abstraction is formalized as follows : $\alpha(x) \sqsubseteq^\# x^\#$. Another way to express this optimality is to use the concretization function. Stating that γ finds the most general property is formalized as follows : $x \sqsubseteq \gamma(x^\#)$.

d) Theorems: given a galois connection (α, γ) between two lattices $(E, \sqsubseteq, \sqcup, \sqcap)$ and $\left(E^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#\right)$, the following properties hold:

- α and γ are monotonous ;
- $\forall x \in E$, $x \sqsubseteq \gamma(\alpha(x))$;
- $\forall x^\# \in E^\#$, $\alpha(\gamma(^\#)) \sqsubseteq^\# x^\#$.

Informally, these properties must be understood as such :

- α and γ preserve the order, *i.e* the implications between informations. Let's take an example with the abstraction by sign. α is the best approximation, so $\alpha(\{x | x\, is\, prime\}) = \mathbb{Z}_+$ and $\alpha(\{x | 2 \leq x\}) = \mathbb{Z}_+$. The following property holds : $(x\, is\, prime) \Rightarrow (2 \leq x)$, or in the conrete world : $\{x | x\, is\, prime\} \sqsubseteq \{x | 2 \leq x\}$. And indeed, in the abstract world : $\mathbb{Z}_+ \sqsubseteq^\# \mathbb{Z}_+$;
- α over-approximates (an element taken to the abstract world and then back to the concrete world would be bigger than at the start) ;
- γ do not introduce loss of information.

The second and the third properties reflect the qualities required for an abstract analyzer. The first function over-approximates, to be sure that no execution, no state, no computation of the program is missed. The second introduces no loss of informations, as computation in the abstract world must remain correct.

Here, we did not presented the whole theory. Usually, complete lattices are used. A complete lattice is a lattice where the least upper bound and greatest lower bound operations apply to any subset instead of two elements. Such operations are used to define fixpoints. We started by using this simpler framework ; the main goal of this internship was to study the optimality of the analyser. For such an objectif, using simple lattices was enough.

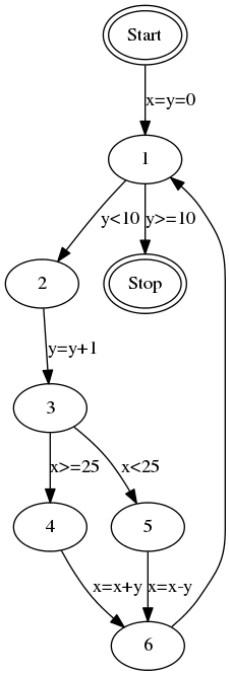
II. Building a static analyser with Coq : forward collecting semantics of arithmetic expressions

The idea of building a static analyzer, based on the theory of abstract interpretation, in Coq, have already been done [1]. However, the theoritical framework was not completly used. The abstraction and concretization functions α and γ allow studying the precision of the approximations. Given a concrete operator f, its closest abstract approximation is charaterized by $\alpha \circ f \circ \gamma \sqsubseteq f^\#$. By a sequence of derivation this specification, it is possible to build an explicit operator. Implementing such derivations in Coq was the goal of this internship.

To build the analyser, we used a document from Patrick Cousot [2]. It details each step of the construction of a static analyser.

A static analyser based on abstract interpretation is made of different parts. One of them is the abstraction used. We detail the other parts here.

Here is the first example we gave, represented by its flow control graph.



In a control flow graph, nodes represent different points of the program. Eedges are labelled with instructions executed between points or conditions needed to go from one point to another. Instructions and conditions are concrete concepts. To perform a static analysis, they need an abstract equivalent.

We give an example to illustrate the reasonnings done with instructions, sill with the abstraction by signs. If on point 2 y is postitive, it will still be positive on point 3, after executing `y = y+1`. Moreover, this is the most precise informations that gan be given with this abstract lattice. However, if on point 2 we know that y was negative, we would have deduced on point 3 that y is anywhere (it can have any sign).

A static analyser reason as such automatically. The choice of the abstraction determine the precision of its reasonnings.

A. The lattice of intervals

In this internship, we tried to build a specific analyser using the lattice of intervals as the abstract world. We supposed that integers are not bounded : the concrete world was $\wp(\mathbb{Z})$. Thus, the lower and upper bounds of intervals can be finite or infinite :

$$E^{\#} = \{\emptyset\} \cup \left\{ \ [a, b] \mid a \in \overline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \leq b \ \right\}$$

where $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{\infty\} \cup \{-\infty\}$

B. Concrete forward collecting semantics of arithmetic expressions

We examine the following instruction of the example code :

`x = x + y`

Knowing the intervals where x and y are "living", what can be deduced about the interval where x now lives ? Can more complex expressions be analysed ? Is the new interval always exact (is it always possible to compute the exact interval where x lives) ? We will answer these questions one by one.

The **forward/bottom-up collecting semantics** of an expression is a relation that, given a set of valuations (or "environments"), defines the possible values that it can evaluate to.

Mathematicaly, we write $\rho \vdash A \Rightarrow v$ if the arithmetic expression A can evaluate to v in the environment ρ. Thus, for an arithmetic expression A and a set of environments R, the forward collecting semantics if defined as such :

$$Faexp[\![A]\!]R = \{ \ v \mid \exists \, \rho \in R, \ \rho \vdash A \Rightarrow v \ \}$$

The forward collecting semantics is concrete : the abstract notion associated must be defined.

C. Abstract forward collecting semantics of arithmetic expressions

The "abstract" forward collecting semantics whould work as such : given an arithmetic expression and an abstract environment (a valuation form the set of variables to the set of intervals), it computes the interval that the expression can evaluate to. Or at least it computes an over-approximation : an interval too big, containing the one that the expression can evaluate to.

The concrete forward collecting semantics is an operator on the set of environments (or valuations). It has an abstract operator associated, and the theory of abstract interpretation gives a specification of it. If f is a concrete operator and (α, γ) a galois connection, the closest correct approximation of f is $f^{\#} = \alpha \circ f \circ \gamma$.

However, this specification is not yet an implementation. An over-approximation of this abstract operator that can be computed is required. To be an over-approximation, this computable operator $Faexp^{\#}$ must verify :

$$\forall A, \forall R, \quad \alpha \left(Faexp[\![A]\!](\gamma'(R)) \right) \sqsubseteq Faexp^{\#}[\![A]\!](R)$$

where γ' is the pointwise application of γ.

To define this function, an induction over the structure of A is employed. We give an example with the sum (as

our precedent example involve the sum of two variables). In abstract interpretation, computations are moved to an abstract world. We defined an operator $+^{\#}$ that mimics what happens in the concrete world. Given two intervals I_1 and I_2 and two integers x_1 and x_2, the following property must hold : if $x_1 \in \gamma(I_1)$ and $x_2 \in \gamma(I_2)$, then $x_1 + x_2 \in \gamma(I_1 +^{\#} I_2)$. Here, we have :

$$\forall (a_1, b_1, a_2, b_2) \in \overline{\mathbb{Z}}^4,$$

$$[a_1, b_1] +^{\#} [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$$

e) Remark :: the sum over $\mathbb{Z}$ isn't yet defined. This can lead to problems, such as "what is $\infty - \infty$?". This problem can be postponed, as it shouldn't appear if the bounds of the intervals are in the right order.

D. Problems of precision

The way components of expressions are abstracted is crucial, as it determines the precision of the informations computed. For example, the way we defined the sum over intervals is optimal. Let $\mathbb{I}$ be the set of intervals ; the following properties hold :

$$\begin{cases} (1) & \forall (i_1, i_2) \in \mathbb{I}^2,\ \forall (x_1, x_2) \in \mathbb{Z}^2, \\ & (x_1 \in i_1 \wedge x_2 \in i_2) \Rightarrow x_1 + x_2 \in i_1 +^{\#} i_2 \\ & \text{(the abstract sum respects the concrete sum)} \\ (2) & \forall (i_1, i_2) \in \mathbb{I}^2,\ \forall x) \in \mathbb{Z}, \\ & x \in i_1 +^{\#} i_2 \Rightarrow \exists\, (x_1, x_2) \in \mathbb{Z}^2, \\ & \quad x = x_1 + x_2 \wedge x_1 \in i_1 \wedge x_2 \in i_2 \\ & \text{(the sum over intervals correspond to the} \\ & \text{sum of sets)} \end{cases}$$

But noting prevented us from defining the sum of intervals from always being $[\infty, \infty]$. Such a definition would have respected the concrete sum. It would only have been less precise.

We managed to find the best abstraction for the sum. However, for some combination of absract world and operator, it can be impossible. For example : the multiplication of intervals. This is one of the spots where precision is lost.

f) Remark: the word "optimal" is clear. The sens we gives here is "that don't lead to a loss of precision". In reality, the sum we defined is not optimal with this definition, as it doesn't take into account the relations between variables. For example, if x in $[0, 1]$, then $-x$ is in $[-1, 0]$. With the sum we defined, the information computed is that $x + (-x)$ is in $[-1, 1]$, wich is correct, but not precise enough. Some abstract domains can adress this issue, but this is not the point here.

III. Implementing in Coq the forward collecting semantics

A. Arithmetic expressions

Our static analyzer was a particular case, as we discussed earlier. We choose to study only the abstraction by intervals. Following the same reasonning, we choose a specific definition of the arithmetic expressions. For our case, an expression is either a constant integer, a variable, the sum of two expression or a random number (to handle non deterministic expressions). Such a definition is enough for what we wanted to study. But it would be easy to add more cases to the definition.

B. Concrete forward collecting semantics

In the concrete world :

- a constant expression can evaluate into the constant it represents ;
- a variable expression can evaluate into values given bu the environment (an environment being a valuation) ;
- the sum of two expressions can evaluate into every sum of evaluations of the two expressions composing it ;
- a random expression can evaluate into any integer.

C. Abstract forward collecting semantics

We write again the specification of the abstract forward collecting semantics :

$$\forall A, \forall R, \quad \alpha\left(Faexp[\![A]\!](\gamma'(R))\right) \sqsubseteq Faexp^{\#}[\![A]\!](R)$$

To implement $Faexp^{\#}$, we worked by induction of the structure of arithmetic expressions. For each case, we started from the left member of the specification, $\alpha\left(Faexp[\![A]\!](\gamma'(R))\right)$. Then, by a sequence of derivations, we tried to make appear an explicit interval i. Thus, we defined $Faexp^{\#}[\![A]\!](R) = i$ for this specific case.

A "sequence of derivations" is a sequence of transformations of the object we are working on. Such transformations can be making definitions explicit, or over-approximations of the oject.

The goal is to make these sequences of derivations inside Coq to proove correct each part of the analyser, and then to extract the corresponding Ocaml code. Thus, we would have built and verified at the same time a part of the analyzer.

g) Example: with the random expression, written "?" and the abstraction by signs. R is supposed to be not empty. Then :

$$
\begin{aligned}
& \alpha\left(Faexp[\![?]\!](\gamma'(R))\right) \\
= \; & \alpha\left(\{v \mid \exists \rho \in \gamma'(R),\ \rho \vdash ? \Rightarrow v\}\right) \\
= \; & \text{(by definition of } . \vdash . \Rightarrow .) \\
& \alpha\left(\{v \mid v \in \mathbb{Z}\}\right) \\
= \; & \text{(this term can be computed)} \\
& \mathbb{Z} \\
= \; & \text{(by defining } Faexp^{\#}[\![A]\!](R) = \mathbb{Z}) \\
& Faexp^{\#}[\![A]\!](R)
\end{aligned}
$$

Such resonnings are given by Cousot. By reproduce them in Coq to ensure that they are correct. Having defined the abstract forward collecting semantics in Coq, we used the extraction mechanism to have the corresponding Ocaml's code :

```
let interpa exp v =
  match enva_bottom_dec v with
  | Left ->
    let rec f = function
    | Rand_expr -> Int_pair (Zf_min, Zf_max)
    | C_expr z0 -> Int_pair ((Zf_Z z0), (Zf_Z
    z0))
    | Sum_expr (e0, e1) -> int_sum (f e0) (f
    e1)
    | Var_expr n0 -> er_env v n0
    in f exp
  | Right -> Int_pair (Zf_max, Zf_min)
```

h) Remarks:

- `interpa` is the name we gave to $Faexp^{\#}$ insode the code ;
- `enva_bottom_dec` decides if the abstract environment is sound or not (it checks if there exists variables that can't evaluate to non-empty intervals).

REFERENCES

[1] D. Cachera and D. Pichardie. A certified denotational abstract interpreter (proof pearl).

[2] 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.