Refinement calculus for a simple certification of static polyhedral analysis with code transformations

Sylvain Boulmé, Michaël Périn

Verimag Research Report n^o ?

January 18, 2013

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Refinement calculus for a simple certification of static polyhedral analysis with code transformations

Sylvain Boulmé, Michaël Périn

January 18, 2013

Abstract

A static analyzer such as ASTRÉE [CCF+05, BCC+10] is able to ensure *safety* of critical software, i.e. the absence of runtime *overflows*. But ASTRÉE is itself a very complex software and its full formal verification seems currently impossible. A more feasible alternative might be to make the analyzer produce a *formally verifiable certificate*. Such a certificate would summarize the proof of safety found by the analyzer.

As a preliminary step to address this challenge, we experiment in COQ with the design of a tiny language of certificates, called "SCAT"[1]. We believe that instrumented analyzers could produce SCAT certificates when analysis is successful. Roughly, a SCAT certificate annotates the source with *loop invariants* that are hard to re-infer and also with *code transformations* used during the analysis. These code transformations come typically from trace-partitioning (loop unrolling, etc) [MR05] and linearization of arithmetic expressions [Min06].

Hence, this paper presents the SCAT language and an automatic checker of SCAT certificates which is formally verified in COQ [The12]. Our main COQ theorem ensures that if the SCAT certificate is accepted by the checker then the original source is safe.

Keywords: Abstract Interpretation, Certification, Coq, Hoare Logic

Reviewers: David Monniaux, Marie-Laure Potet

How to cite this report:

```
@techreport {?,
    title = {Refinement calculus for a simple certification of static polyhedral analysis with code
transformations},
  author = {Sylvain Boulmé, Michaël Périn},
  institution = {{Verimag} Research Report},
  number = {?},
  year = {}
}
```

[1] SCAT is the acronym of "*Simple Certificates for Analysis with code Transformations*"

1 Introduction

This paper explores how to *formally and automatically* verify the *safety* of source programs by *certifying in* COQ the results of a static analyzer. Such a source is said to be safe if and only if none of its executions raises a memory or arithmetic *overflow*. Whereas our current tool only applies to a toy imperative language involving only integer computations, our long-term goal is to support COMPCERT sources. COMPCERT is an optimizing C-compiler which is formally verified in COQ [Ler09]. More precisely, it is proved to produce an executable that behaves as authorized by the semantics of the source program *only if* this latter is safe. Hence, certifying the safety of a COMPCERT source program will ensure that it is correctly compiled (and that the resulting executable is also safe).

Related work. The principle of *a posteriori* verifying the results of a static analyzer consists in instrumenting an analyzer in order to produce a *certificate* that proves the validity of the discovered properties in all possible executions of the program under analysis. *A posteriori* certification of static analyzers has been studied with the motivation of Proof-Carrying Code [SYY03, Cha06, GS07, BJP06, BJPT10] or in order to free the user from specifying full contracts in Hoare logic [MM10]. In these propositions except [BJP06, BJPT10], the certificate checker generates Verification Conditions (VC) in a logic more expressive than the abstract domains of the analyzer. Typically, the certificate is the original source annotated with assertions discovered during analysis. The certificate is checked using a Hoare logic based on a general theorem prover. However, in order to plug our verifier on COMPCERT, we aim to have a *fully automated certification process* which is itself *formally verified*. Generating VCs in an expressive logic seems currently incompatible with this goal. In our proposal, like in [BJPT10], all VCs are implications in the abstract domain of convex polyhedra introduced in [CH78]. Hence, they might be automatically discharged using the COQ certified implementation of polyhedral inclusion introduced in [BJPT10].

Contributions. Building on [BJPT10], we address the certification of analyzers performing program transformations such as *trace partitioning* and *linearization* whose soundness itself *depends on properties found during analysis*. Indeed, these transformations overcome two main limitations of the convex polyhedra domain: the loss of precision due to the approximation of disjunctions and the treatment of expressions which have no best abstraction. We formalize these transformations using refinement calculus[BvW99] because this framework is *simple to formalize* in COQ. Moreover, although we leave the issue of certificate generation for further work, we also take care to design a language allowing *small and fast-checking certificates*. Hence, this paper introduces our language of certificates, called SCAT, and the COQ certified SCAT-CHECKER.

A running example. We consider source programs like the one of figure 1 that contain two kinds of assertions:

- Assertion "$\lceil c \rceil$" means that the condition c is *required* by execution: if c does not hold then $\lceil c \rceil$ raises a runtime error. Typically, this corresponds to the precondition of an array access that requires the index to belong to the static bounds of the array. Arithmetic expressions in our simple semantics are pure and total: if necessary, division by zero and arithmetic overflows must be explicitly avoided through a preceding *require* assertion.

- Assertion "$\lfloor c \rfloor$" is a *guard*: if c does not hold then execution is safely blocked. Otherwise, the condition c is *ensured* and can be used as a hypothesis.

The program of figure 1 illustrates that the join operator (i.e. the least upper bound) of convex abstract domains may be too imprecise to discharge certain safety requirements. Indeed, the join of interval abstractions from the "then-branch" and from the "else-branch" on line 5 is $1 \leq x, y \leq 10$ which does not suffice to prove requirement of line 7. In order to discharge this requirement, this join must be delayed after line 6 where both branches entail $6 \leq r \leq 50$. Delaying join according to heuristics is a particular case of *trace partitioning* [MR05].

The delay of a join after an "if-then-else" must appear in SCAT certificates as an explicit program transformation noted by $\triangleleft/\triangleright$ brackets. Roughly, a certificate $\mathbb{p}$ of the form "$\triangleleft$ **if** c **then** $\mathbb{p}_1$ **else** $\mathbb{p}_2$ **fi ;** $\mathbb{p}_3$ $\triangleright$" has two interpretations:

$$\begin{array}{rcl} \mathcal{C}(\mathbb{p}) & = & \text{"}\mathbf{if}\ c\ \mathbf{then}\ \mathcal{C}(\mathbb{p}_1)\ \mathbf{else}\ \mathcal{C}(\mathbb{p}_2)\ \mathbf{fi}\ \mathbf{;}\ \mathcal{C}(\mathbb{p}_3)\text{"} \\ \mathcal{A}(\mathbb{p}) & = & \text{"}\mathbf{if}\ c\ \mathbf{then}\ \mathcal{A}(\mathbb{p}_1)\ \mathbf{;}\ \mathcal{A}(\mathbb{p}_3)\ \mathbf{else}\ \mathcal{A}(\mathbb{p}_2)\ \mathbf{;}\ \mathcal{A}(\mathbb{p}_3)\ \mathbf{fi}\text{"} \end{array}$$

The *concrete program* $\mathcal{C}(\mathbb{p})$ gives back the source code whereas the *abstract program* $\mathcal{A}(\mathbb{p})$ is the code that must be checked by a polyhedral analysis. Hence, the transformation $\triangleleft/\triangleright$ forces here the analyzer to delay the join after $\mathbb{p}_3$.

```
1  ⌊1≤x≤10⌋ ;
2  if x≤5
3  then y←x+5  (* 1≤x≤5 ; 6≤y≤10 *)
4  else y←x-5  (* 6≤x≤10 ; 1≤y≤5 *)
5  fi ;        (* 1≤x,y≤10 *)
6  r←x*y ;
7  ⌈6≤r≤50⌉ ;
8  r←r+x+y ;
9  ⌈13≤r≤65⌉
```

Annotations inside "(* ... *)" typically result from a naive interval analysis, where "**fi**" is analyzed as a join of intervals. The postcondition on line 5 authorizes the unreachable state x=7 and y=8. Hence, because 7×8=56>50, the requirement of line 7 can not be proved. This unreachable state is also authorized using any other convex overapproximation (octagons, polyhedra, polynomial inequalities, etc). Indeed, it belongs to the convex-hull of the strongest-postconditions of branch "then" and branch "else": this convex-hull is computed at figure 4.

Figure 1: Two arithmetic computations after an "if-then-else"

Figure 2 gives an overview of our SCAT-CHECKER. We assume an analyzer instrumented to return SCAT certificates. Given a certificate $\mathbb{p}$, the checker extracts $\mathcal{C}(\mathbb{p})$ (which is checked against the expected *source*) and $\mathcal{A}(\mathbb{p})$. By construction, $\mathcal{C}(\mathbb{p})$ *refines* $\mathcal{A}(\mathbb{p})$. This means that "*if* $\mathcal{A}(\mathbb{p})$ *is safe then any behavior of* $\mathcal{C}(\mathbb{p})$ *is a behavior of* $\mathcal{A}(\mathbb{p})$": in particular, if $\mathcal{A}(\mathbb{p})$ is safe, then so is $\mathcal{C}(\mathbb{p})$. Indeed, $\mathbb{p}$ is syntactically built from operators preserving refinement (like $\triangleleft/\triangleright$) which are themselves certified once and for all in COQ. Last, the absence of unsafe behavior is proved on $\mathcal{A}(\mathbb{p})$ which generates only VCs that are discharged using the certified polyhedral inclusion of [BJPT10].

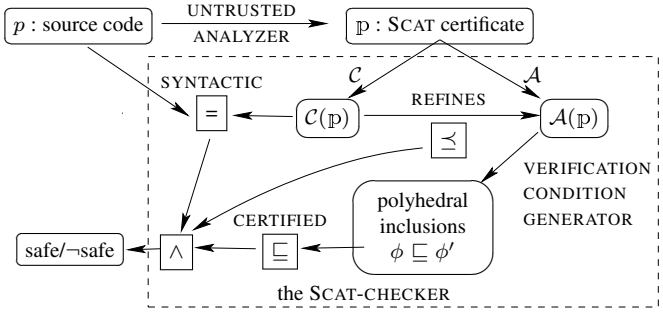


Figure 2: Architecture of the SCAT-CHECKER

Overview of the paper. Section 2 briefly introduces our vision of abstract interpretation. Section 3 is a short tutorial to our SCAT certificate language, based on examples. Section 4 gives a "paper" formalization of SCAT. Section 5 explains how this formalization is implemented in COQ. Section 6 concludes and opens on perspectives.

Contents

1 Introduction **1**

2 Our vision of polyhedral analysis with trace partitioning **4**
2.1 Preliminary notations. . . . 4
2.2 The abstract domain of convex polyhedra. . . . 4
2.3 Abstraction of non-polyhedral conditions 5
2.4 Trace partitioning 6

3 Examples of SCAT certificates **8**
3.1 Primitive programming statements of SCAT syntax 8
3.2 A certificate for the code of figure 1 (illustrating delay of a join) 8
3.3 Certificate for source of figure 7 (illustrating loop unrolling & invariant) 9
3.4 Certificates for source of figure 5 (illustrating composition of Hoare-terms) 9

3.5 Linear interpolation (illustrating full unrolling of loops) . . . 10
3.6 Exhaustive exploration by trace partitioning . . . 11

4 Formalizing SCAT **13**
4.1 Syntax and operational semantics of SCAT programs . . . 13
4.2 Abstract semantics and VC-Generator . . . 14
4.3 The SCAT certificate language . . . 16

5 Implementation in COQ **18**
5.1 Limitations of implementation w.r.t section 4 . . . 18
5.2 Most notable improvements of implementation w.r.t section 4 . . . 18
5.2.1 Checking efficiently non-influence of partition labels on concrete code . . . 18
5.2.2 VCG of loop unrolling . . . 20
5.2.3 Coding a frame as an integer . . . 21
5.2.4 Reflecting concrete wp-calculus to automate (meta)proofs of refinement . . . 21
5.3 Presentation of the current SCAT prototype in COQ . . . 22

6 Conclusion **24**

2 Our vision of polyhedral analysis with trace partitioning

This section presents only the concepts of abstract interpretation that are relevant for the remainder of the paper. For example, as our checker does not infer loop invariants, but performs only a kind of postfixpoint verification, we do not introduce widenings and narrowings that make abstract interpreters so smart. Our first objective here is to present the main design choices leading to SCAT. Moreover, this section introduces examples for which SCAT certificates are given at section 3. At last, it also introduces the technical notations about polyhedra domain that are used in the formalization of section 4. Hence, it presents a reformulation of some basic notions of abstract interpretation, which are both adapted to COQ logic (not founded on set-theory, but on type-theory), and "simplified" according to our needs.

2.1 Preliminary notations.

- We use an ambient higher-order meta-logic very close to the one of COQ. We note $\mathbf{Prop}$ the type of (meta)propositions. We also assume a type $\mathbf{FSet}(A)$ to represent finite subsets of type A. Similarly, we assume a type $\mathbf{FMap}(A, B)$ to represent finite maps from A to B. We use notations of set-theory to handle elements of these two types.

- We note $\mathbf{X}$ the countable type of programming variables; we use $x, x_1 : \mathbf{X}$. We note $\mathbf{V}$ the type of values. In our toy language, we simply take $\mathbf{V} \triangleq \mathbb{Z}$. We use $v, v_1 : \mathbf{V}$.

- We note $\mathbf{S} \triangleq \mathbf{X} \to \mathbf{V}$ the type of *memory states*: it is the type of *total* functions modulo extensionality from variables into values. We use $s, s_1 : \mathbf{S}$. Hence, implicitly, every variable is initially assigned to an arbitrary value. Our notion of safety means that the program can not have wrong behavior for any initial state.

- Given $f : \mathbf{FMap}(\mathbf{X}, \mathbf{V})$, we note "$s \oplus f$" the operation that returns a copy of state s where the old values associated to $\mathrm{dom}\ f$ have been replaced by their image by f.
$$s \oplus f \triangleq \lambda x.\texttt{if}\ x \in \mathrm{dom}(f)\ \texttt{then}\ f(x)\ \texttt{else}\ s(x)$$
We note $\{x \mapsto v\}$ the singleton map associating x to value v. And, we write "$\{x_1 \mapsto v_1 \mid \ldots \mid x_n \mapsto v_n \mid \_ \mapsto v\}$" as a shortcut for "$\lambda\_.v \oplus \{x_n \mapsto v_n\} \oplus \ldots \oplus \{x_1 \mapsto v_1\}$". We use these notations for any type $\mathbf{X}$ with a decidable equality $\overset{dec}{=}$.

- We note $\mathbf{C}$ the type of logical *conditions*. They are quantifier-free first-order formula used as boolean expressions of programs. Their definition is precised in section 4.1. We use $c, \ldots : \mathbf{C}$. Below, we note $[\![.]\!] : \mathbf{C} \to \mathbf{S} \to \mathbf{Prop}$ the *semantics* of conditions, and we note $|.| : \mathbf{C} \to \mathbf{FSet}(\mathbf{X})$ the *frame* function returning the set of variables that are syntactically constrained by a condition. They satisfy[2]
$$x \notin |c| \quad \Rightarrow \quad [\![c]\!](s) = [\![c]\!](s \oplus \{x \mapsto v\})$$

2.2 The abstract domain of convex polyhedra.

We focus on the case where abstract states – over-approximating the reachable memory states – are represented by polyhedra. Hence, the type $\mathbf{S}^\sharp$ of abstract states is the type of convex polyhedra.

Definition 1 (convex polyhedron)**.** *A convex polyhedron $\phi : \mathbf{S}^\sharp$ is a finite conjunction of affine inequalities of the form "$v_1.x_1 + \ldots + v_n.x_n \leq v$".*

Basically, computations on $\mathbf{S}^\sharp$ allow to approximate logical reasonings on formula of type $\mathbf{C}$. As explain later in the paper, they are employed to overapproximate strongest-postconditions of programs. Hence, along the paper, we coerce *implicitly* $\mathbf{S}^\sharp$ into $\mathbf{C}$. The semantics $[\![\phi]\!](s)$ of a polyhedron ϕ is the conjunction of the semantics of ϕ inequalities. Similarly, its frame $|\phi|$ is included in the union of the frames of its inequalities.

$$[\![\Sigma_{i=1}^n v_i.x_i \leq v]\!](s) \ = \ \Sigma_{i=1}^n v_i.s(x_i) \leq v \qquad\qquad |\Sigma_{i=1}^n v_i.x_i \leq v| \subseteq \{x_1, \ldots, x_n\}$$

Actually, in abstract interpretation, the semantics $[\![.]\!] : \mathbf{S}^\sharp \to \mathbf{S} \to \mathbf{Prop}$ is traditionally called *concretization* of $\mathbf{S}^\sharp$, and is rather noted γ. Hence, we use γ instead of $[\![.]\!]$ when applying to $\mathbf{S}^\sharp$.

The geometric nature of polyhedra makes them a powerful data-structure with efficient algorithms from linear programming. In this paper, we do not use a particular implementation of $\mathbf{S}^\sharp$. Instead, we only use the basic

[2] As usual, all our formula are implicitly universally quantified.

operators specified below. However, often, our safety theorem requires only a weak version of the specification (whereas the stronger version is only needed for the precision of the result). These weak specifications of operators are formalized on figure 3: given a particular implementation of $\mathbf{S}^\sharp$, they must be proved in COQ. But, these weak specifications allow *result certification* of most operators below (see [BJPT10]). Hence, their core implementation (which may use a complex procedure based on Simplex algorithm) is not directly proved in COQ. Instead, they generate a witness that their result is correct. And, only the witness checker is proved in COQ. When the witness checker fails, a default result satisfying the weak specification can be returned. Here, these witnesses may typically exploit *Farkas lemma* in order to reduce witness checking to matrix multiplications (see [BJPT10]).

The basic operators on $\mathbf{S}^\sharp$ used by our VCG are the following:

- the decidable inclusion relation $\sqsubseteq$: $\mathbf{S}^\sharp \times \mathbf{C} \to \mathbf{bool}$ [3] such that $\phi_1 \sqsubseteq \phi_2$ is exactly equivalent to $\gamma(\phi_1)(s) \Rightarrow \gamma(\phi_2)(s)$. But, $\phi \sqsubseteq c$ may return false, if our abstract domain does not to deal with condition c.
- the empty polyhedron $\bot$ (e.g. $0 \leq -1$) and the full polyhedron $\top$ (e.g. $0 \leq 1$).
- the join operator $\sqcup : \mathbf{S}^\sharp \times \mathbf{S}^\sharp \to \mathbf{S}^\sharp$. Actually, $\phi_1 \sqcup \phi_2$ is the smallest polyhedron containing ϕ_1 and ϕ_2, or in other words, their *convex-hull* (see example of figure 4). But this property is not needed for safety.
- a guard operator $\sqcap : \mathbf{S}^\sharp \times \mathbf{C} \to \mathbf{S}^\sharp$ such that polyhedron $\phi \sqcap c$ *abstracts* condition c in the context of ϕ. In the particular case where c is in $\mathbf{S}^\sharp$, then $\phi \sqcap c$ is exactly the intersection of c and ϕ (which remains a polyhedron). Otherwise, c may be simply ignored: in this case, $\phi \sqcap c$ is equivalent to ϕ. Let us remark here, that, if c is not a polyhedron, then $\phi \sqsubseteq c$ may be implemented as "$(\phi \sqcap \neg c) \sqsubseteq \bot$" (hence, it gives a exact answer when $\neg c$ is a polyhedron).
- a renaming operator $.[.] : \mathbf{S}^\sharp \times \mathbf{FMap}(\mathbf{X}, \mathbf{X}) \to \mathbf{S}^\sharp$ such that if σ is an involution [4], then $\phi[\sigma]$ is the result of applying this permutation to ϕ.
- a projection operator $.\backslash.\ : \mathbf{S}^\sharp \times \mathbf{FSet}(\mathbf{X}) \to \mathbf{S}^\sharp$ such that $\phi_{\backslash\chi}$ is a simplified version of ϕ using the fact that variables χ are useless. Actually, $\phi_{\backslash\chi}$ may be the projection of ϕ on $|\phi| \backslash \chi$, but may also be not. This operator is only used in order to reduce the size of polyhedra when possible. But, eliminating some variables may also lead to a growth of the resulting polyhedron. Moreover, from the VCG point of view, invoking $\phi_{\backslash\chi}$ in unappropriate circumstances will only result in a loss of precision, not in a loss of safety.

A polyhedral analyzer over-approximates the strongest-postcondition of programs using computations on $\mathbf{S}^\sharp$. Typically, strongest-postcondition of assignment are approximated by combining a renaming of variables, a guard and a projection. The strongest-postcondition of if-then-else is approximated using *convex-hull* $\sqcup$. See example of figure 4. These ideas are formalized in section 4.2.

2.3 Abstraction of non-polyhedral conditions

A non-polyhedral condition c may be abstracted according to the current postcondition ϕ by a smart implementation of the guard operator $\phi \sqcap c$. However, delegating to the guard operator all the job to find sufficient abstractions of non-polyhedral conditions is clearly not a *complete* approach. For instance, if c is "$x \times x + y \times y \leq 100$" (e.g. it corresponds geometrically to a disk), then it has no *best* abstraction as a polyhedron: given any polyhedron ϕ containing c, there is a polyhedron ϕ' containing c and strictly contained in ϕ (otherwise quadrature of circle would be possible). In practice, analyzers employ complex heuristics to find good abstractions which may consider the remaining code depending on c. They may also apply some symbolic transformation of the code before linearization techniques [Min06].

In this last approach, linearization consists in approximating non-linear code by *interval affine forms* that are kind of affine expressions where "scalars" are generalized into "intervals of scalar". Typically, a multiplication between two linear terms "$t_1 \times t_2$" is linearized by approximating one of the two terms t_1 or t_2 as an interval of scalars. The choice between t_1 and t_2 itself relies on several strategies: one of them could be "*try both choices and select the one that gives the most precise result*". See figure 5.

In conclusion, it seems interesting to get hints from the analyzer in order to avoid to replay the search of good abstractions during the verification of certificates. The instrumented analyzer may generate a SCAT certificate according to flexible strategies: either implicitly use the "default abstraction strategy" implemented in $\sqcap$, or

[3] In all this paper, booleans of **bool** are implicitly coerced as propositions of **Prop**. However, **bool** indicates that the underlying proposition is decidable.

[4] We say "$\sigma : \mathbf{FMap}(\mathbf{X}, \mathbf{X})$ is an *involution*" iff $\mathrm{dom}(\sigma) \subseteq \mathrm{image}(\sigma)$ and $\sigma \circ \sigma = \mathrm{id}_{\mathrm{dom}(\sigma)}$

$$\phi \sqsubseteq c \wedge \gamma(\phi)(s) \Rightarrow [\![c]\!](s) \qquad \neg\gamma(\bot)(s) \qquad \gamma(\top)(s) \qquad |\bot| = |\top| = \emptyset$$

$$\gamma(\phi_1)(s) \vee \gamma(\phi_2)(s) \Rightarrow \gamma(\phi_1 \sqcup \phi_2)(s) \qquad |\phi_1 \sqcup \phi_2| \subseteq |\phi_1| \cup |\phi_2| \qquad \gamma(\phi)(s) \wedge [\![c]\!](s) \Rightarrow \gamma(\phi \sqcap c)(s)$$

$$\gamma(\phi_1 \sqcap \phi_2)(s) \Rightarrow \gamma(\phi_1)(s) \wedge \gamma(\phi_2)(s) \qquad |\phi \sqcap c)| \subseteq |\phi| \cup |c| \qquad \gamma(\phi)(s) \Rightarrow \gamma(\phi_{\setminus\chi})(s) \qquad \left|\phi_{\setminus\chi}\right| \subseteq |\phi|$$

$$\sigma \text{ involution} \wedge \gamma(\phi)(s \oplus (s \circ \sigma)) \Rightarrow \gamma(\phi[\sigma])(s) \qquad |\phi[\sigma]| \subseteq (|\phi| \setminus \mathrm{dom}(\sigma)) \cup \sigma[|\phi| \cap \mathrm{dom}(\sigma)]$$

Figure 3: Properties of $\mathbf{S}^\sharp$ required for safety checking

On figure 1, postcondition of line 5 is approximated by

$$\bigsqcup \begin{array}{lcll} 1 \leq x \leq 5 & \wedge & y = x + 5 & \text{(post of } \textbf{then}\text{)} \\ 6 \leq x \leq 10 & \wedge & y = x - 5 & \text{(post of } \textbf{else}\text{)} \end{array}$$

This convex-hull (pictured on the right) results in

$$-5 \leq x - y \leq 5 \quad \wedge \quad 7 \leq x + y \leq 15$$

Whereas each points of initial segments satisfy $x \times y \leq 50$, the points of their convex-hull only satisfy $x \times y \leq 56.25$

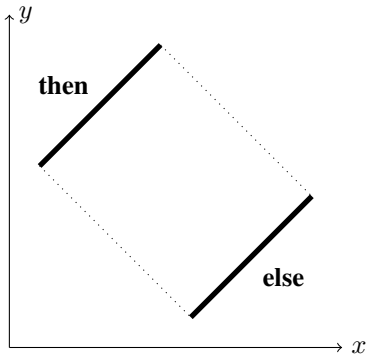


Figure 4: Abstract postcondition of if-statement in source of figure 1

```
⌊1≤x≤5 ∧ 1≤y≤3 ∧ 1≤z≤3⌋ ;
r←y*x+z*(-x)+z*5 ;
⌈1≤r≤27⌉
```

Here, intervalizing `z` leads to reject the program (proves only $-9 \leq r \leq 29$). But, "homogeneity strategy" [Min06] is successful by intervalizing `x`. Indeed, this strategy intervalizes the smallest set of variables that makes the expression homogeneous (such that arguments of $+$ and $-$ operators have the same degree).

Figure 5: Example of a non-linear assignment

explicitly specify a transformation on the original code before to apply the default strategy. This transformation may be compositionally built from elementary ones, themselves proved once for all in a Coq library or with an automatically generated Coq proof, in parallel of the certificate.

2.4 Trace partitioning

Trace partitioning [MR05] is a technique to associate a disjunction of abstract postconditions to a given point of control. Hence, it allows to compensate the lack of precise disjunction in convex abstract domains. However, this increased precision comes at the price of analyzing the same piece of code several times in different contexts of execution.

For example, on the source of figure 1, performing the convex-hull at line 5 forbids to prove the safety of line 7 (because it would authorize unreachable state x=7 and y=8, as illustrated by figure 4). Hence, a polyhedral analyzer must linearize line 6 for each branch of if-then-else: it attaches two postconditions on line 6. Assuming that the analyzer decides to approximate y in both cases (it may however apply a different strategy in each branch), we get those on figure 6. However, systematically avoiding convex-hulls leads to an exponential blow-up. If the analyzer performs one just after line 6, it gets a sufficient condition to discharge requirements of line 7 and line 9.

Figure 7 gives a slightly more complex example where a control-point is associated to 4 postconditions. This last example also illustrates that loops often need to be unrolled at least once (see figure 7). Hence, partial or full loop unrolling is another frequent case of trace partitioning. For instance, Mauborgne and Rival [MR05] identifies some patterns like "linear interpolation" which are often used in embedded software, and that are efficiently analyzed by a full unrolling of loops.

- Post of line 6 after branch "then"
  $1 \leq x \leq 5 \wedge y = x + 5 \wedge 6x \leq r \leq 10x$
- Post of line 6 after branch "else"
  $6 \leq x \leq 10 \wedge y = x - 5 \wedge x \leq r \leq 5x$

Their convex-hull
$-5 \leq x - y \leq 5 \ \wedge \ 7 \leq x + y \leq 15$
$\wedge\, x + y + 5 \leq 2r \leq 5(x + y) + 25$
$\wedge\, x + 5y - 25 \leq r \leq 7x + 3y - 15$

Figure 6: Two postconditions for line 6 of figure 1 with their convex hull

```
1  ⌊x≤10⌋ ;
2  if x≤5
3  then y←x+5
4  else y←x-5
5  fi ;
6  i←0 ;  r←0 ;
7  while i≤x-1 do
8    i←i+1 ;  r←r+y
9  done ;
10 ⌈0≤r≤50⌉
```

We consider a variant of figure 1 where multiplication is computed in a loop and x has an infinite lower bound. Here also, the required precision obliges to approximate multiplication on variable `r` independently in each branch of the if-then-else. Hence, the concrete loop invariant "$r = y.i$" is abstracted as "$r \leq 10.i$" in the first branch, and "$r \leq 5.i$" in the second one.
Moreover, we need to unroll the loop once, in order to express the following reasoning: "*If the body of the loop has been run at least once, then at the end of the loop* `i=x`. *Otherwise, the required property holds because* `r=0`."
Hence, line 9 is associated to 4 postconditions. Their convex-hull is sufficient to prove the required property.

Figure 7: Partitioning of loop invariant & unrolling

At last, another case of trace partitioning in [MR05] replaces complex linearization strategies by a simple exhaustive (but costly) test. On source of figure 5, as "x" is an integer known to be between 1 and 5, the analyzer may compute the postcondition on `r` as the convex-hull of the five cases corresponding to replace x by a value between 1 and 5. Hence, it is able to prove the very precise postcondition "$5 \leq r \leq 15$".

Actually, on this source, a smarter analysis than the two previous ones first factorizes x (in order to avoid twice independant intervalizations on x): "$y \times x + z \times (-x) + z \times 5$" is rewritten "$(y - z) \times x + z \times 5$". Then, trace partitioning is used on two cases $y - z \geq 0$ and $y - z < 0$: this allows to keep precision while intervalizing on x. Hence, we get $y + 4.z \leq r \leq 5.y$ for first branch, and $5.y \leq r \leq y + 4.z$ for second branch. The convex-hull of this two branches implies "$5 \leq r \leq 15$". As we see, such an analysis is able to discover that r is a barycentre of $5.y$ and $5.z$.

In [MR05], trace partitioning is formalized as the abstract domain of mappings from *finite partitions of traces*[5] to postconditions. During the analysis, splitting or merging partitions is dynamically guided by heuristics, according to the informations already discovered by the analyzer. For instance on figure 7, the analysis starts with a singleton partition. It is split in two partitions at line 2. Each one is then split at line 7, and all 4 are merged at line 9. Line 10 is analyzed with a singleton partition.

Hence, at the end of the analysis, we assume that the analyzer is able to annotate the source program with the chosen partitions and their scope, using split/merge blocks. SCAT syntax provides a family of operators "$\triangleleft_{v \leq \ell \leq v'}\ \mathbb{p} \triangleright$" where $\mathbb{p}$ is a piece of certificate which precise syntax and semantics are specific to each operator of the family. Here ℓ is a "ghost constant" called *a partition label* and bound in $\mathbb{p}$. Its value is an integer of the range $v \ldots v'$ that represents exactly one branch of the partition. Hence, such a statement corresponds to split the analysis of $\mathbb{p}$ in $\max(v' - v + 1, 0)$ sub-branches. The abstract program $\mathcal{A}(\mathbb{p})$ can be adapted for each branch by case analysis on ℓ value. However, $\mathcal{C}(\mathbb{p})$ must not depend on this value, otherwise the certificate is rejected by the VCG.

[5] A trace is the sequence of full states (i.e. memory & control) in a given execution of the small-step semantics.

3 Examples of SCAT certificates

This section presents SCAT certificates for the analysis described on previous examples. In these examples, we assume that the guard operator $\sqcap$ does not well find good abstractions of non-polyhedral conditions. Hence, certificates makes explicit the linearization strategies. First, we introduce a bit of SCAT syntax.

3.1 Primitive programming statements of SCAT syntax

The primitive statements of SCAT are inspired from guarded commands of refinement calculus [BvW99]. As detailed on figure 8, if-then-else and while-loop are macros defined from non-deterministic constructs. When necessary, this makes easy to express that condition c and its negation $\neg c$ are abstracted by different strategies (e.g. if t_1 and t_2 are two linear expressions, then $t_1 = t_2$ is directly a polyhedron, whereas $t_1 \neq t_2$ is often imprecisely abstracted).

$$\text{“}\mathbf{if}\ c\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{fi}\text{”} \triangleq (\lfloor c\rfloor; p_1)\ \amalg\ (\lfloor \neg c\rfloor; p_2) \qquad \text{“}x \leftarrow t\text{”} \triangleq x \leftarrow \lfloor x = \mathbf{old}(t)\rfloor$$

$$\text{“}\mathbf{while}\ c\ \mathbf{do}\ p\ \mathbf{done}\text{”} \triangleq (\lfloor c\rfloor; p)^*; \lfloor \neg c\rfloor \qquad \text{“}p_1\ \amalg\ p_2\text{”} \triangleq \coprod_{0\le\ell\le 1} \{0 \mapsto p_1 \mid \_ \mapsto p_2\}(\ell)$$

Figure 8: Definition of macros from primitive statements

Here, statement “$p_1\ \amalg\ p_2$” runs non-deterministically p_1 or p_2: this operator is a *join* in the lattice of statements for order $p \preceq p'$ read as *“p refines p'”*. Statement “p^*” iterates statement p either a non-deterministic number of times or infinitely. [6] Actually, the binary join in “$p_1\ \amalg\ p_2$” is derived from a more general bounded-join operator $\coprod_{v\le\ell\le v'}$. At last, assignment is itself derived from the general guarded assignment “$x \leftarrow \lfloor c\rfloor$” which non-deterministically assigns a value to x such that condition c is satisfied. In c, all occurrences of x under a “**old**” subterm refer to the value of x in the prestate of the assignment. All other occurrences of x refer to its final value. Actually, if c does not contain an occurrence of x under a **old**, then “$x \leftarrow \lfloor c\rfloor$” is exactly equivalent to “$x \leftarrow \lfloor \top\rfloor; \lfloor c\rfloor$” (which first assigns x to an arbitrary value, and then ensures c).

3.2 A certificate for the code of figure 1 (illustrating delay of a join)

Below, we present a certificate for source of figure 1 that corresponds to the analysis of page 6. First, we focus on the linearization of statement “$r \leftarrow x \times y$” in the “then-branch” : y is abstracted as interval $6..10$. This is expressed by the statement “$r \leftarrow$ `mpi` $x\ y\ (6, 10)$” where “`mpi`” is a function defined in the SCAT library: “`mpi`” stands for *multiplication of a positive by an interval*. This function is parameterized by two arithmetic terms t_1 and t_2 and a pair of scalars (v_1, v_2). It returns a “Hoare-term” that associates the concrete term “$t_1 \times t_2$” to an abstraction in the form of a pre/post specification, accompanied by a COQ proof-term that the concrete term satisfies the specification. In the case of `mpi`, the precondition is “$0 \le t_1 \wedge v_1 \le t_2 \le v_2$” and the postcondition is “$\lambda x'.\, \mathbf{old}(t_1) \times v_1 \le x' \le \mathbf{old}(t_1) \times v_2$” where x' represents the result of the concrete term “$t_1 \times t_2$”. This allows to define “$r \leftarrow$ `mpi` $x\ y\ (6, 10)$” as the certificate $\mathbb{p}$ such that

$$\mathcal{C}(\mathbb{p}) = r \leftarrow x \times y \qquad \mathcal{A}(\mathbb{p}) = \begin{array}{l} \lceil 0 \le x \wedge 6 \le y \le 10\rceil\ \mathbf{;} \\ r \leftarrow \lfloor \mathbf{old}(x) \times 6 \le r \le \mathbf{old}(x) \times 10\rfloor \end{array}$$

The safety of source of figure 1 is now established by certificate at figure 9. This certificate uses a statement of the form “$\triangleleft_{0\le\ell\le 1}\ (\mathbb{p}_1\ \amalg\ \mathbb{p}_2)\ \mathbf{;}\ \mathbb{p}_3\ \triangleright$” expressing that the concrete code is the one of “$(\mathbb{p}_1\ \amalg\ \mathbb{p}_2)\ \mathbf{;}\ \mathbb{p}_3$”, but, that it must be analyzed as “$(\mathbb{p}_1; \mathbb{p}_3)\ \amalg\ (\mathbb{p}_2; \mathbb{p}_3)$”. Here ℓ is the *partition label* that is bound to $\mathbb{p}_3$, that must be not used by its concrete generated code, and that equals 0 if the left branch of the join as been executed or 1 otherwise.

[6] SCAT statements have *almost* the structure of a Kleene algebra, with “$(\amalg, \lfloor\bot\rfloor)$” as the additive monoid, with “$(;, \lfloor\top\rfloor)$” as the multiplicative one, and with “$.^*$” as the star operator. The only difference is that $\lfloor\bot\rfloor$ is not right-annihilator of “;” (because error $\lceil\bot\rceil$ is also a left-annihilator).

Below, the left side is the result from unfolding "**if**" macro in source of figure 1. The corresponding certificate is given on the right side.

$$\begin{array}{l}\lfloor 1 \le x \le 10 \rfloor\ ;\\ \left(\amalg\ \begin{array}{l}\lfloor x \le 5 \rfloor ; y \leftarrow x+5\\ \lfloor \neg x \le 5 \rfloor ; y \leftarrow x-5\ ;\end{array}\right)\\ r \leftarrow x \times y\ ;\\ \lceil 6 \le r \le 50 \rceil\ ;\\ r \leftarrow r+x+y\ ;\\ \lceil 13 \le r \le 65 \rceil\end{array} \qquad \begin{array}{l}\lfloor 1 \le x \le 10 \rfloor\ ;\\ \underset{0\le\ell\le 1}{\triangleleft}\left(\begin{array}{l}\left(\amalg\ \begin{array}{lll}\lfloor x \le 5 \rfloor & ; & y \leftarrow x+5\\ \lfloor \neg x \le 5 \rfloor & ; & y \leftarrow x-5\end{array}\right)\ ;\\ r \leftarrow \texttt{mpi}\ x\ y\ \{0 \mapsto (6,10)\,|\,\_ \mapsto (1,5)\}(\ell)\end{array}\right)\triangleright\ ;\\ \lceil 6 \le r \le 50 \rceil\ ;\\ r \leftarrow r+x+y\ ;\\ \lceil 13 \le r \le 65 \rceil\end{array}$$

Figure 9: "Desugared" code and certificate of source in figure 1

3.3 Certificate for source of figure 7 (illustrating loop unrolling & invariant)

The strategy described in figure 7 to analyze the source in the same figure, is now expressed by the certificate at figure 10. This certificate uses a statement "$\triangleleft_{0\le\_\le n}\ \mathbb{p}_1{}^{*\phi};\mathbb{p}_2\ \triangleright$" meaning that

$$\begin{array}{l}\lfloor x \le 10 \rfloor\ ;\\ \underset{0\le\ell\le 1}{\triangleleft}\left(\begin{array}{l}\left(\amalg\ \begin{array}{lll}\lfloor x \le 5 \rfloor & ; & y \leftarrow x+5\\ \lfloor \neg x \le 5 \rfloor & ; & y \leftarrow x-5\end{array}\right)\ ;\\ i \leftarrow 0\ ; r \leftarrow 0\ ;\\ \underset{0\le\_\le 1}{\triangleleft}\left(\begin{array}{l}(\lfloor i \le x-1 \rfloor ; i \leftarrow i+1 ; r \leftarrow r+y)^{*\phi}\ ;\\ \lfloor \neg i \le x-1 \rfloor\end{array}\right)\triangleright\end{array}\right)\triangleright\ ;\\ \lceil 0 \le r \le 50 \rceil\end{array}$$

where invariant $\phi \triangleq i \le x \wedge 0 \le r \wedge\ \{0 \mapsto r \le 10.i\,|\,\_ \mapsto r \le 5.i\}(\ell)$

Figure 10: Certificate for source of figure 7

- the concrete code is the one of "$\mathbb{p}_1{}^*;\mathbb{p}_2$" ;
- the abstract code is analyzed as a join of $n+1$ branches (n being a natural number), where
  - for all k in $0\ldots n-1$, the branch k (if any) is analyzed as "$\mathbb{p}_1^k;\mathbb{p}_2$". In other words, $\mathbb{p}_1$ is iterated k times, before $\mathbb{p}_2$ is run.
  - last branch is analyzed as "$\mathbb{p}_1^n;\mathbb{p}_1{}^{*\phi};\mathbb{p}_2$" using loop-invariant "$\phi$" in analysis of loop "$\mathbb{p}_1{}^*$".

3.4 Certificates for source of figure 5 (illustrating composition of Hoare-terms)

First, we consider the "homogeneity strategy" described at section 2.3 to analyze certificate of source 5. This illustrates the composition of Hoare-terms. The certificate below means to approximate the term "`y*x+z*(-x)+z*5`" by intervalizing the first occurrence of `x` as $1\ldots5$ and $-$`x` as $-5\ldots-1$, and to compose these two "Hoare-terms" using **let-in**.

$$\begin{array}{l}\lfloor 1 \le x \le 5 \wedge 1 \le y \le 3 \wedge 1 \le z \le 3 \rfloor;\\ r \leftarrow\ \ \textbf{let}\ y_0 = \texttt{mpi}\ y\ x\ (1,5)\ \textbf{in}\\ \qquad\quad \textbf{let}\ z_0 = \texttt{mpi}\ z\ (-x)\ (-5,-1)\ \textbf{in}\\ \qquad\qquad y_0 + z_0 + z \times 5\ ;\\ \lceil 1 \le r \le 27 \rceil\end{array}$$

Here, the corresponding concrete term is the addition of the concrete terms of y_0 and z_0, plus $z \times 5$. Similarly, its abstract code expresses that the result is the addition of a possible result of y_0 with a possible result of z_0, plus $z \times 5$.

Now, let us consider in figure 11 a variant of this example using the last trace partitioning technique exposed in section 2.4. It uses a certificate of the form "$\triangleleft_{0\le\ell\le 1}\ t_1 \le^? t_2 :\ \mathbb{p}\ \triangleright$" expressing that the concrete code is the one

of "$\mathbb{p}$", but, that it must be analyzed using a case analysis based on $t_1 \leq t_2$ or $t_2 + 1 \leq t_1$. Hence, ℓ is a partition label that is bound to $\mathbb{p}$ and that equals 0 if $t_1 \leq t_2$ or 1 otherwise.

In figure 11 "rew" names a lemma expressing equality of terms $(y-z)\times x+z\times 5 = y\times x+z\times(-x)+z\times 5$. Currently, our verifier does not embed itself ring rewriting. But such a rewriting lemma can be generated in Coq by the analyzer and proved automatically in Coq using `ring` tactic. Then, this "rew" lemma may be used as a certificate of term of the form "rew[$\mathbb{t}$]". This generates the same abstract code than $\mathbb{t}$ but with concrete term $y \times x + z \times (-x) + z \times 5$, at the condition – verified during typechecking – that the concrete term generated from $\mathbb{t}$ is $(y - z) \times x + z \times 5$.

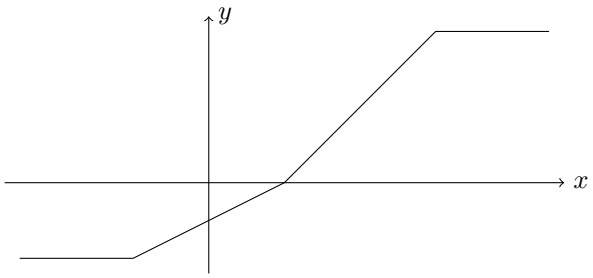
At last `mni` stands for *multiplication of a negative by an interval*. Thus, on branch "$0 \leq (y - z)$", we have $y - z \leq r_0 \leq 5 \times (y - z)$. Whereas on branch "$y - z + 1 \leq 0$", we have $5 \times (y - z) \leq r_0 \leq y - z$.

$$\begin{array}{l} \lfloor 1 \leq x \leq 5 \wedge 1 \leq y \leq 3 \wedge 1 \leq z \leq 3 \rfloor; \\ \underset{0\leq\ell\leq 1}{\triangleleft} \left(\begin{array}{l} 0 \leq^? y - z : \\ r \leftarrow \text{rew}\,[\textbf{let}\ r_0 = \{0 \mapsto \texttt{mpi}\ (y-z)\ x\ (1,5) \mid \_ \mapsto \texttt{mni}\ (y-z)\ x\ (1,5)\}(\ell)\ \textbf{in} \\ \qquad\qquad r_0 + z \times 5] \end{array} \right) \triangleright; \\ \lceil 5 \leq r \leq 15 \rceil \end{array}$$

Figure 11: Mixing intervalization & rewriting & case analysis (trace partitioning)

3.5 Linear interpolation (illustrating full unrolling of loops)

This example below is adapted from [MR05] in order to have only integer computations. The purpose of the source (given figure 13) is to compute output variable y in function of input variable x as defined by the piecewise function at figure 12.



range of x			equation of y
$-\infty$	...	-1	-6
-1	...	$+1$	$3.x - 3$
$+1$	...	$+3$	$6.x - 6$
$+3$	...	$+\infty$	12

Figure 12: A piecewise function

This computation is implemented as given in figure 13. Roughly, an array tx stores the abscissa range of the $i + 1$-th "piece" as "$tx(i)$ ... $tx(i + 1)$" (for i in $0 \ldots 2$). Thus, an initial loop search the value of i corresponding to x. Then, the $i + 1$-th piece is simply defined as the straight line of slope "$tc(i)$" and containing the point of coordinate "$(tx(i), ty(i))$".

i	0	1	2	3
tx	?	-1	1	3
tc	0	3	6	0
ty	-6	-6	0	12

```
i←0 ;
while i≤2 ∧ tx(i+1) < x do
  i←i+1
done ;
y←tc(i)*(x-tx(i))+ty(i) ;
⌈-6≤y ∧ y≤12⌉
```

Figure 13: Concrete computation of the piecewise function

In order to deal with this source, we have thus extended our formalism to deal with constant arrays. Actually, they are only interpreted as unary symbol functions in the language of terms. In other words, the semantics of our toy language does not require that array accesses are inside array bounds.

Now, let us apply the strategy promoted by Mauborgne and Rival [MR05] on this source. It simply consists in unrolling completely the loop in 3 iterations. Indeed, after 3 iterations, the guard of the loop body "$i \leq 2$" becomes false. Then, for each i in $0..2$, we can abstract term "$tc(i)$" as its value. Hence, we avoid the need to introduce an explicit loop invariant and a complex abstraction of the multiplication.

In certificate of figure 14, we use a variant of loop-unrolling noted "$\triangleleft_{0\leq\ell\leq n+1}\ \mathbb{p}_1{}^*; \mathbb{p}_2\ \triangleright$". Here, the lack of loop-invariant indicates that the loop is fully unrolled. Hence, it means that:

- the concrete code is the one of "$\mathbb{p}_1{}^*; \mathbb{p}_2$"
- the abstract code is analyzed as a join of $n+2$ branches where
  - for k in $0 \ldots n$, the k-th branch is analyzed as "$\mathbb{p}_1^k; \mathbb{p}_2$".
  - the last branch with $k = n+1$ is analyzed as "$\mathbb{p}_1^k; \lceil \bot \rceil$": its safety implies that $\mathbb{p}_1$ is iterated at most n times.
  - ℓ is a partition label allowing to access the value of k both in $\mathbb{p}_1$ and $\mathbb{p}_2$.

In other words, in the certificate of figure 14, the value of ℓ is exactly the one of i. This allows to abstract term $tc(i)$ by case analysis on ℓ using `mlc` (for "*multiplication at left by a constant*"). Hence, in figure 14, "`mlc` $t_1\ t_2\ v$" is a Hoare-term such that its corresponding concrete term is $t_1 \times t_2$, and under precondition $t_1 = v$ the abstract result is $v \times t_2$. Notice that the statement "$\lfloor \neg(i \leq 2 \wedge tx(i+1) < x) \rfloor$" is here always analyzed in a context where $i \leq 2$, and thus, this condition reduces to ensure $x \leq tx(i+1)$.

$$
\begin{array}{l}
i \leftarrow 0\ ; \\
\overset{\triangleleft}{_{0\leq\ell\leq 2+1}} \left(\begin{array}{l}
(\lfloor i \leq 2 \wedge tx(i+1) < x \rfloor\ ; i \leftarrow i+1)^*\ ; \\
\lfloor \neg(i \leq 2 \wedge tx(i+1) < x) \rfloor\ ; \\
y \leftarrow \textbf{let}\ y_0 = \texttt{mlc}\ \ tc(i)\ \ x - tx(i)\ \ \{1 \mapsto 3 \,|\, 2 \mapsto 6 \,|\, \_ \mapsto 0\}(\ell)\ \ \textbf{in} \\
\qquad y_0 + ty(i)
\end{array} \right) \triangleright\ ; \\
\lceil -6 \leq y \wedge y \leq 12 \rceil .
\end{array}
$$

Figure 14: Certificate for source of figure 13

3.6 Exhaustive exploration by trace partitioning

The source of figure 15 is also adapted from Barycentre example of [MR05]. As our prototype does not know yet handling floating points, we use an euclidean division on integers (noted by operator $/$) instead of a floating-point division. However, this does not change too much the global idea of the example.

Here successive approximations of $\times$ and $/$ over an interval would be too imprecise. The strategy promoted by Mauborgne and Rival [MR05] is here to test $-10 \leq x \leq 10$ for all possible values of r in the interval $0 \ldots 5$. Hence, linearization is simply done by abstracting r as a constant. This kind of strategy is expressed on certificate of figure 16 using a certificate of the form "$\triangleleft_{v\leq\ell\leq v'}\ \textbf{explore}\ t\ \textbf{in}\ \mathbb{p}\ \triangleright$" meaning that

- the concrete code is the one of $\mathbb{p}$;
- the abstract code requires first "$v \leq t \leq v'$", then "$\lfloor t = \ell \rfloor\ ; \mathcal{A}(\mathbb{p})$" is analyzed for each value ℓ in "$v \ldots v'$".

$$
\begin{array}{l}
\lfloor 0 \le r \le 5 \wedge -10 \le x, y \le 10 \rfloor; \\
x \leftarrow (x \times r + y)/(r+1)\ ; \\
\lceil -10 \le x \le 10 \rceil
\end{array}
$$

Figure 15: Computation of a barycentre

$$
\begin{array}{l}
\lfloor 0 \le r \le 5 \wedge -10 \le x, y \le 10 \rfloor; \\
\underset{0 \le \ell \le 5}{\triangleleft} \left(\begin{array}{l} \textbf{explore}\ r\ \textbf{in} \\ \quad x \leftarrow \textbf{let}\ x_0 = \texttt{mrc}\ \ x\ \ r\ \ \ell\ \ \textbf{in} \\ \qquad\quad \texttt{dbpc}\ \ x_0{+}y\ \ r{+}1\ \ \ell{+}1 \end{array} \right) \triangleright\ ; \\
\lceil -10 \le x \le 10 \rceil
\end{array}
$$

Here, `mrc` is similar to previous `mlc` but for "*multiplication at right by a constant*". Function `dbpc` stands for "*division by a positive constant*". Hence, "`dbpc` $t_1\ t_2\ v$" is a Hoare-term such that its concrete term is "t_1/t_2", its preconditon is "$t_2 = v \wedge 0 \le v$" and its result x satisfies postcondition

"$x \times v \le \mathbf{old}(t_1) \le x \times v + (v-1)$".

Figure 16: Certificate for source of figure 15

4 Formalizing SCAT

This section presents SCAT formalization with slight differences with respect to the current implementation in COQ. See discussion in section 5. The subset of the SCAT certificates language presented here is sufficient to express most of the examples given in section 3.

This section first defines the syntax and the semantics of SCAT programs. Then, it specifies the VCG (VC-Generator) and its correctness with respect to operational semantics. This VCG is very close to a computation of strongest-postcondition, but approximates it using postconditions in $\mathbf{S}^\sharp$. One of its main feature is to work with a *weak* projection operator $.\backslash.$: it still works even if this operator is identity, that is, when it does not perform a projection[7]. Actually, this approach is inspired from [BJPT10]. Another main feature is the support of *partition labels*, through a mechanism of constant propagation. At last, this section provides the definition of SCAT certificates and their verification. It states our safety theorem with respect to the original source. The proofs can be found in our COQ code [Bou13].

4.1 Syntax and operational semantics of SCAT programs

We first define the abstract syntax $\mathbf{T}$ of terms. We use $t, t' : \mathbf{T}$. The main originality of $\mathbf{T}$ comes from the "**old**" modality (inspired from [Fil98]) that allows to refer to an implicit previous state of the variables. This is expressed by extending the usual semantics with an additional parameter for the previous state. Hence, $[\![t]\!] : \mathbf{S} \times \mathbf{S} \to \mathbf{V}$ expects the previous state as first parameter and the current one as second parameter. The frame of terms satisfies

$$x \notin |t| \;\Rightarrow\; [\![t]\!](s_0, s_1) = [\![t]\!](s_0 \oplus \{x \mapsto v_0\}, s_1 \oplus \{x \mapsto v_1\})$$

Constructors of $\mathbf{T}$ are given figure 17 with their associated frame and semantics. They are dotted when they need to be distinguished from notations of the meta-language. The abstract syntax $\mathbf{C}$ of conditions introduced at section 2.1, is also defined in figure 17. Its semantics is similar to the one of $\mathbf{T}$.

When using **old** is irrelevant (there is no implicit previous state), by convention, it is interpreted as identity. More formally, we define the unary versions of $[\![t]\!]$ and $[\![c]\!]$ with the meaning $[\![t]\!](s) \triangleq [\![t]\!](s, s)$ and $[\![c]\!](s) \triangleq [\![c]\!](s, s)$. Hence, we have $[\![\mathbf{old}(t)]\!](s_0, s_1) = [\![t]\!](s_0)$ and $[\![\mathbf{old}(t)]\!](s) = [\![t]\!](s)$.

t	$\triangleq$	$\dot{x}$	$\mid$	$\mathbf{old}(t_0)$	$\mid$	$\dot{v}$	$\mid$	$t_1 \mathbin{\dot{\diamondsuit}} t_2$
$[\![t]\!](s_0, s_1)$	$\triangleq$	$s_1(x)$	$\mid$	$[\![t_0]\!](s_0, s_0)$	$\mid$	v	$\mid$	$[\![t_1]\!](s_0, s_1) \diamondsuit [\![t_2]\!](s_0, s_1)$
$\lvert t \rvert$	$\triangleq$	$\{x\}$	$\mid$	$\lvert t_0 \rvert$	$\mid$	$\emptyset$	$\mid$	$\lvert t_1 \rvert \cup \lvert t_2 \rvert$

where $\diamondsuit : \mathbf{V} \times \mathbf{V} \to \mathbf{V}$ with $\diamondsuit \in \{+, *, -, /, \mathrm{mod}\}$

c	$\triangleq$	$\top$	$\mid$	$\bot$	$\mid$	$t_1 \mathbin{\dot{\mathcal{R}}} t_2$	$\mid$	$\dot{\neg}\, c$	$\mid$	$c_1 \mathbin{\dot{\heartsuit}} c_2$
$[\![c]\!](s_0, s_1)$	$\triangleq$	**True**	$\mid$	**False**	$\mid$	$[\![t_1]\!](s_0, s_1) \mathcal{R} [\![t_2]\!](s_0, s_1)$	$\mid$	$\neg [\![c]\!](s_0, s_1)$	$\mid$	$[\![c_1]\!](s_0, s_1) \heartsuit [\![c_2]\!](s_0, s_1)$
$\lvert c \rvert$	$\triangleq$	$\emptyset$	$\mid$	$\emptyset$	$\mid$	$\lvert t_1 \rvert \cup \lvert t_2 \rvert$	$\mid$	$\lvert c \rvert$	$\mid$	$\lvert c_1 \rvert \cup \lvert c_2 \rvert$

where $\mathcal{R} : \mathbf{V} \times \mathbf{V} \to \mathbf{Prop}$ with $\mathcal{R} \in \{=, \leq, <, >, \geq, \neq\}$
and $\heartsuit : \mathbf{Prop} \times \mathbf{Prop} \to \mathbf{Prop}$ with $\heartsuit \in \{\wedge, \vee\}$

Figure 17: Syntax and semantics of terms and conditions

The abstract syntax $\mathbf{P}$ of SCAT programs includes *primitive constructs* and *derived constructs*. The latter have been defined in figure 8. The former are given in figure 18 together with their natural semantics (big-step operational semantics) : syntactic elements appear at the bottom of inference rules on the left of $\vdash$ symbol. This syntax is actually a higher-order abstract syntax: in constructor "$\coprod_{v \leq \ell \leq v'} p(\ell)$" ℓ is a metavariable and p is a function of type $\mathbf{V} \to \mathbf{P}$. As formalized by the semantics, here instances of ℓ only range between constants v and v' (thus p is only used as a finite map). Statement "**clone** x **in** p" runs p on a local copy of variable x. It typically allows to encode "**let** $x = t$ **in** p" as the more primitive "**clone** x **in** $(x \leftarrow t; p)$".

The natural semantics of $\mathbf{P}$ is a ternary relation denoted "$p \vdash s_0 \to s_1$" where $s_0 : \mathbf{S}$ is the initial state, $p : \mathbf{P}$ is a concrete program, and $s_1 : \mathbf{S} \uplus \{\lightning\}$ is a final state or an error $\lightning$. Here, operator $\oplus$ is extended on its left operand such that $\lightning \oplus f \triangleq \lightning$. This relation is defined by inductive rules of figure 18 and uses definition 2 below.

[7] Projection approximates elimination of existential quantifier.

Definition 2. *Given $\chi \subseteq \mathbf{X}$, we say that "$s_1$* coincides with *$s_2$ on χ" and we note $s_1 \equiv_\chi s_2$ if and only if $\forall x \in \chi, s_1(x) = s_2(x)$.*
This defines $\equiv_\chi$ as an equivalence relation.

$$\frac{[\![c]\!](s)}{\lfloor c \rfloor \vdash s \to s} \qquad \frac{[\![c]\!](s)}{\lceil c \rceil \vdash s \to s} \qquad \frac{\neg\,[\![c]\!](s)}{\lceil c \rceil \vdash s \to \lightning} \qquad \frac{s_1 \equiv_{\mathbf{X}\setminus\{x\}} s_0 \qquad [\![c]\!](s_0, s_1)}{x \leftarrow \lfloor c \rfloor \vdash s_0 \to s_1}$$

$$\frac{p_1 \vdash s \to \lightning}{p_1; p_2 \vdash s \to \lightning} \qquad \frac{p_1 \vdash s_0 \to s_1 \qquad s_1 \neq \lightning \qquad p_2 \vdash s_1 \to s_2}{p_1; p_2 \vdash s_0 \to s_2} \qquad \frac{}{p^* \vdash s \to s} \qquad \frac{p\,; p^* \vdash s_0 \to s_1}{p^* \vdash s_0 \to s_1}$$

$$\frac{v \leq v_0 \leq v' \qquad p(v_0) \vdash s_0 \to s_1}{\coprod_{v \leq \ell \leq v'} p(\ell) \vdash s_0 \to s_1} \qquad \frac{p \vdash s_0 \to s_1}{\mathbf{clone}\ x\ \mathbf{in}\ p \vdash s_0 \to s_1 \oplus \{x \mapsto s_0(x)\}}$$

Figure 18: Syntax and natural semantics of primitive programs

$$x \leftarrow t \vdash s \to s \oplus \{x \mapsto [\![t]\!](s)\} \qquad p_1 \vdash s_0 \to s_1 \vee p_2 \vdash s_0 \to s_1 \;\Rightarrow\; p_1 \amalg p_2 \vdash s_0 \to s_1$$

$$([\![c]\!](s_0) \wedge p_1 \vdash s_0 \to s_1) \vee (\neg\,[\![c]\!](s_0) \wedge p_2 \vdash s_0 \to s_1) \;\Rightarrow\; \mathbf{if}\ c\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{fi} \vdash s_0 \to s_1$$

$$\neg\,[\![c]\!](s) \Rightarrow \mathbf{while}\ c\ \mathbf{do}\ p\ \mathbf{done} \vdash s \to s$$

$$([\![c]\!](s_0) \;\wedge\; p; \mathbf{while}\ c\ \mathbf{do}\ p\ \mathbf{done} \vdash s_0 \to s_1) \;\Rightarrow\; \mathbf{while}\ c\ \mathbf{do}\ p\ \mathbf{done} \vdash s_0 \to s_1$$

Figure 19: Derived rules of macros

4.2 Abstract semantics and VC-Generator

Our VCG is specified through an abstract semantics. It needs some new definitions summarized below:

- In the abstract semantics, each loop "p^*" must be annotated by the loop-invariant ϕ discovered by the analysis. We write "$p^{*\phi}$" for $p^{*\phi} \triangleq \lceil \phi \rceil; (p; \lceil \phi \rceil)^*$.
- We note $|p| : \mathbf{FSet}(\mathbf{X})$ the frame of program p. It has a quite straightforward definition. We only give the less obvious cases:

$$|x \leftarrow \lfloor c \rfloor| \triangleq \{x\} \cup |c| \qquad |\mathbf{clone}\ x\ \mathbf{in}\ p| \triangleq \{x\} \cup |p| \qquad \left|\coprod_{v \leq \ell \leq v'} p(\ell)\right| \triangleq \bigcup_{v \leq \ell \leq v'} |p(\ell)|$$

  It satisfies $\qquad x \notin |p| \;\wedge\; p \vdash s_0 \to s_1 \;\Rightarrow\; p \vdash s_0 \oplus \{x \mapsto v\} \to s_1 \oplus \{x \mapsto v\}$

- The modifying frame of programs $\overleftarrow{|.|} : \mathbf{P} \to \mathbf{FSet}(\mathbf{X})$, returning the set of potentially modified variables, is defined in a quite similar way:

$$\overleftarrow{|x \leftarrow \lfloor c \rfloor|} \triangleq \{x\} \qquad \overleftarrow{|\mathbf{clone}\ x\ \mathbf{in}\ p|} \triangleq \overleftarrow{|p|} \setminus \{x\} \qquad \left|\coprod_{v \leq \ell \leq v'} p(\ell)\right| \triangleq \bigcup_{v \leq \ell \leq v'} \overleftarrow{|p(\ell)|}$$

  Hence, $\overleftarrow{|p|} \subseteq |p|$ and $x \notin \overleftarrow{|p|} \;\wedge\; p \vdash s_0 \to s_1 \;\wedge\; s_1 \neq \lightning \implies s_1(x) = s_0(x)$.
  This operator is used to authorize "partial" loop-invariant ϕ: the "full" invariant ϕ_1 is obtained by extending ϕ with the part of the loop-precondition remaining unassigned by the loop (see figure 20).

$$\frac{}{\langle \chi\,,\,\phi\rangle\ \lfloor c\rfloor\ \langle \chi\,,\,\phi\sqcap c\rangle} \qquad \frac{\phi\sqsubseteq c}{\langle \chi\,,\,\phi\rangle\ \lceil c\rceil\ \langle \chi\,,\,\phi\rangle}$$

$$\frac{(\chi_1,\sigma)\triangleq \mathtt{fresh}_{\chi_0}(\{x\}) \qquad \phi_1\triangleq \phi_0[\sigma] \qquad c_1\triangleq c[x\xleftarrow{\mathbf{old}}\sigma(\dot{x})]}{\langle \chi_0\,,\,\phi_0\rangle\ x\leftarrow\lfloor c\rfloor\ \left\langle \chi_1\,,\,(\phi_1\sqcap c_1)_{\setminus\{\sigma(x)\}}\right\rangle}$$

$$\frac{\langle \chi_0\,,\,\phi_0\rangle\ p_1\ \langle \chi_1\,,\,\phi_1\rangle \qquad \langle \chi_1\,,\,\phi_1\rangle\ p_2\ \langle \chi_2\,,\,\phi_2\rangle}{\langle \chi_0\,,\,\phi_0\rangle\ p_1;p_2\ \langle \chi_2\,,\,\phi_2\rangle}$$

$$\frac{\phi_0\sqsubseteq\phi \qquad (\chi_1,\sigma)\triangleq \mathtt{fresh}_{\chi_0}(\overleftarrow{|p|}) \qquad \phi_1\triangleq (\phi_0[\sigma]\sqcap\phi)_{\setminus\sigma[\overleftarrow{|p|}]} \qquad \langle \chi_1\,,\,\phi_1\rangle\ p\ \langle \chi_2\,,\,\phi_2\rangle \qquad \phi_2\sqsubseteq\phi}{\langle \chi_0\,,\,\phi_0\rangle\ p^{*\phi}\ \langle \chi_1\,,\,\phi_1\rangle}$$

$$\frac{n\triangleq \max(0,v'-v+1) \qquad \langle \chi_0\,,\,\phi_0\rangle\ p(v)\ \langle \chi_1\,,\,\phi_1\rangle \qquad \dots \qquad \langle \chi_0\,,\,\phi_0\rangle\ p(v')\ \langle \chi_n\,,\,\phi_n\rangle}{\langle \chi_0\,,\,\phi_0\rangle\ \coprod_{v\le\ell\le v'} p(\ell)\ \left\langle \bigcup_{0\le i\le n}\chi_i\,,\,\bigsqcup_{1\le i\le n}\phi_i\right\rangle}$$

$$\frac{(\chi,\sigma)\triangleq \mathtt{fresh}_{\chi_0}(\{x\}) \qquad \langle \chi\,,\,\phi_0[\sigma]\sqcap x=\sigma(x)\rangle\ p\ \langle \chi_1\,,\,\phi_1\rangle}{\langle \chi_0\,,\,\phi_0\rangle\ \mathbf{clone}\ x\ \mathbf{in}\ p\ \left\langle \chi_1\,,\,\phi_1[\sigma]_{\setminus\{\sigma(x)\}}\right\rangle}$$

Figure 20: Abstract semantics of programs

- We define an operation "$\mathtt{fresh}_{\chi_1}(\chi_0)$" with a result $(\chi_2,\sigma)\ :\ \mathbf{FSet}(X)\times\mathbf{FMap}(X,X)$ such that if $\chi_0\subseteq\chi_1$, then σ is an involution and $\chi_0\subseteq\mathrm{dom}(\sigma)$ and $\sigma[\chi_0]\cap\chi_1=\emptyset$ and $\chi_2=\sigma[\chi_0]\cup\chi_1$. Hence, variables in $\chi_2\backslash\chi_1$ are "fresh" renamings of those in χ_0 through σ.

- At last, we define an operation "$c[x\xleftarrow{\mathbf{old}}t]$" (of result in $\mathbf{C}$), that substitutes variable x by term t *only in subterms under a "*$\mathbf{old}$*"* of condition c, and then removes "$\mathbf{old}$" modalities of c. We also define a more standard substitution operation "$t_1[x\leftarrow t_0]$" (removing also $\mathbf{old}$ from t_1). Formally, they satisfy:

$$\left[\!\left[c[x\xleftarrow{\mathbf{old}}t]\right]\!\right](s)=[\![c]\!](s\oplus\{x\mapsto[\![t]\!](s)\}\,,\,s) \qquad \left|c[x\xleftarrow{\mathbf{old}}t]\right|\subseteq|c|\cup|t|$$

$$[\![t_1[x\leftarrow t_0]]\!](s)=[\![t_1]\!](s\oplus\{x\mapsto[\![t_0]\!](s)\})$$

Below $\dot{x_1}[x\leftarrow t_0]$ is t_0 if $x_1=x$ or $\dot{x_1}$ otherwise.

t	$=$	$\dot{x_1}$	$\mid\ \mathbf{old}(t_1)\ \mid\ \dot{v}\ \mid$	$t_1\ \dot{\Diamond}\ t_2$	
$t[x\leftarrow t]$	$\triangleq$	$\dot{x_1}[x\leftarrow t_0]$	$\mid\ t_1[x\leftarrow t_0]\ \mid\ \dot{v}\ \mid$	$t_1[x\leftarrow t_0]\ \dot{\Diamond}\ t_2[x\leftarrow t_0]$	
$t[x\xleftarrow{\mathbf{old}}t]$	$\triangleq$	$\dot{x_1}$	$\mid\ t_1[x\leftarrow t_0]\ \mid\ \dot{v}\ \mid$	$t_1[x\xleftarrow{\mathbf{old}}t_0]\ \dot{\Diamond}\ t_2[x\xleftarrow{\mathbf{old}}t_0]$	

c	$=$	$t_1\ \dot{\mathcal{R}}\ t_2$	$\mid\ c_1\ \dot{\heartsuit}\ c_2$	$\mid\ \dots$
$c[x\xleftarrow{\mathbf{old}}t]$	$\triangleq$	$t_1[x\xleftarrow{\mathbf{old}}t]\ \dot{\mathcal{R}}\ t_2[x\xleftarrow{\mathbf{old}}t]$	$\mid\ c_1[x\xleftarrow{\mathbf{old}}t]\ \dot{\heartsuit}\ c_2[x\xleftarrow{\mathbf{old}}t]$	$\mid\ \dots$

- The abstract semantics is defined as a ternary relation inspired from Hoare triples with an explicit management of fresh variables using the χ sets. The triple "$\langle \chi_0\,,\,\phi_0\rangle\ \ p\ \ \langle \chi_1\,,\,\phi_1\rangle$" relates program p to precondition ϕ_0 and postcondition ϕ_1. Valid triples are defined by induction on figure 20. They satisfy $|p|\cup|\phi_0|\subseteq\chi_0\Rightarrow\chi_0\cup|\phi_1|\subseteq\chi_1$.

Given a precondition $\langle\chi_0,\phi_0\rangle$ and a program p such that $|p|\cup|\phi_0|\subseteq\chi_0$, our VCG either fails, or computes a postcondition $\langle\chi_1,\phi_1\rangle$ such that triple $\langle \chi_0\,,\,\phi_0\rangle\ \ p\ \ \langle \chi_1\,,\,\phi_1\rangle$ is valid. Note that the premises of the rules of figure 20 involve only definitions and inclusion test $\sqsubseteq$.

Definition 3 (VCG checker). *Using our VCG, we define* $.^{\checkmark} : \mathbf{P} \to \mathbf{bool}$ *such that* $p^{\checkmark} \Rightarrow \exists\chi\exists\phi.\, \langle |p| \,,\, \top\rangle \;\; p \;\; \langle \chi \,,\, \phi\rangle$.

Our main technical lemma is given below. Here p is a statement inside a bigger program which global set of variables is χ. And, χ_0 is an extension of χ where new names correspond to old values of some variables in χ: the relation between χ_0 variables are specified by abstract precondition ϕ_0. Hence, given an initial state s_0, we also assume s'_0 which coincides with s_0 on χ, but which gives also the required old values of variables such that concrete precondition $\gamma(\phi_0)(s'_0)$ holds. This lemma claims that for any final state s_1, there exists s'_1 which coincides with s_1 on χ, but still coincides with s'_0 on $\chi_0 \backslash \chi$ and satisfies postcondition $\gamma(\phi_1)$.

Lemma 1. *For all* s_0, s_1, p, χ, χ_0, ϕ_0, χ_1, ϕ_1 *and* s'_0,
If $p \vdash s_0 \to s_1$ *and* $\langle \chi_0 \,,\, \phi_0\rangle \;\; p \;\; \langle \chi_1 \,,\, \phi_1\rangle$ *and* $|p| \subseteq \chi$ *and* $\chi \cup |\phi_0| \subseteq \chi_0$ *and* $s'_0 \equiv_\chi s_0$ *and* $\gamma(\phi_0)(s'_0)$
Then, $s_1 \neq \lightning$ *and there exists* s'_1 *such that* $s'_1 \equiv_\chi s_1$ *and* $s'_1 \equiv_{\chi_0 \backslash \chi} s'_0$ *and* $\gamma(\phi_1)(s'_1)$.

In particular, taking $\chi = \chi_0 = |p|$ and $s'_0 = s_0$, we deduce:

Theorem 1 (VCG safety). *For any program* p, *if* $p^{\checkmark}$ *returns true then* p *can not reach error state from any initial state,* i.e. *we have* $p^{\checkmark} \Rightarrow \neg\, p \vdash s \to \lightning$.

4.3 The SCAT certificate language

The syntax $\mathbb{P}$ of SCAT certificates extends syntax $\mathbf{P}$ with additional constructs. In definition 4, we do not define $\mathbb{P}$ as an inductive type, but by its interpreters (i.e. as a *shallow* embedding). This allows an incremental construction of $\mathbb{P}$ language.

Definition 4. *The type* $\mathbb{P}$ *of certificates for programs is defined as a pair with projections* $\mathcal{C} : \mathbb{P} \to \mathbf{P}$ *and* $\mathcal{A} : \mathbb{P} \to \mathbf{P}$ *such that for all* $\mathbb{p} : \mathbb{P}$, *"*$\mathcal{C}(\mathbb{p})$ *refines* $\mathcal{A}(\mathbb{p})$*"* i.e.

$$(\neg\, \mathcal{A}(\mathbb{p}) \vdash s_0 \to \lightning) \;\Rightarrow\; (\mathcal{C}(\mathbb{p}) \vdash s_0 \to s_1 \Rightarrow \mathcal{A}(\mathbb{p}) \vdash s_0 \to s_1)$$

From the preceding definition and theorem 1, we immediately get theorem 2

Theorem 2 (SCAT safety). *For all* $\mathbb{p} : \mathbb{P}$, $\mathcal{A}(\mathbb{p})^{\checkmark} \;\Rightarrow\; \neg\, \mathcal{C}(\mathbb{p}) \vdash s \to \lightning$

Below, we build all syntactic constructs of $\mathbb{P}$ language as a pair $\mathcal{C}/\mathcal{A}$ as specified in definition 4. We note $\overset{dec}{=}$ the decidable syntactic equality on $\mathbf{P}$ (after normalization in the submonoid of sequence). Its result is implicitly coerced in $\mathbf{C}$ as $\bot$ (false) or $\top$ (true). First, all construct of $\mathbf{P}$ are trivially lift to $\mathbb{P}$. We detail only some of the constructs introduced in section 3.

- Certificate $\mathbb{p}$ for "$\triangleleft_{0\le\ell\le 1}\, (\mathbb{p}_1 \;\amalg\; \mathbb{p}_2)$ **;** $\mathbb{p}_3(\ell) \triangleright$" – where ℓ is a metavariable and $\mathbb{p}_1, \mathbb{p}_2 : \mathbb{P}$ and $\mathbb{p}_3 : \mathbf{V} \to \mathbb{P}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \begin{array}{l}(\mathcal{C}(\mathbb{p}_1) \;\amalg\; \mathcal{C}(\mathbb{p}_2))\;; \\ \mathcal{C}(\mathbb{p}_3(0))\end{array} \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l}\left\lceil \mathcal{C}(\mathbb{p}_3(1)) \overset{dec}{=} \mathcal{C}(\mathbb{p}_3(0)) \right\rceil\;; \\ \coprod\limits_{0\le\ell\le 1} \begin{array}{l}\{0 \mapsto \mathcal{A}(\mathbb{p}_1) \mid \_ \mapsto \mathcal{A}(\mathbb{p}_2)\}(\ell)\;; \\ \mathcal{A}(\mathbb{p}_3(\ell))\end{array}\end{array}$$

- Certificate $\mathbb{p}$ for "$\triangleleft_{0\le\_\le n}\, \mathbb{p}_1{}^{*\phi}; \mathbb{p}_2 \triangleright$" – where $\mathbb{p}_1, \mathbb{p}_2 : \mathbb{P}$ and $n : \mathbb{N}$ and $\phi : \mathbf{S}^\sharp$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \begin{array}{l}\mathcal{C}(\mathbb{p}_1)^*\;; \\ \mathcal{C}(\mathbb{p}_2)\end{array} \qquad \mathcal{A}(\mathbb{p}) \triangleq \coprod_{0\le\ell\le n} \begin{array}{l}\mathcal{A}(\mathbb{p}_1)^\ell\;; \\ \left\{n \mapsto \mathcal{A}(\mathbb{p}_1)^{*\phi} \mid \_ \mapsto \lfloor \top \rfloor\right\}(\ell)\;; \\ \mathcal{A}(\mathbb{p}_2)\end{array}$$

  where VCG of "$\mathcal{A}(\mathbb{p}_1)^\ell$" iterates exactly ℓ times VCG of $\mathcal{A}(\mathbb{p}_1)$.

- Certificate $\mathbb{p}$ for "$\triangleleft_{0\le\ell\le 1}\; t_1 \le^? t_2 :\; \mathbb{p}_0 \triangleright$" – where $t_1, t_2 : \mathbf{T}$ and ℓ is a metavariable and $\mathbb{p}_0 : \mathbf{V} \to \mathbb{P}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_0(0)) \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l}\left\lceil \mathcal{C}(\mathbb{p}_0(1)) \overset{dec}{=} \mathcal{C}(\mathbb{p}_0(0)) \right\rceil\;; \\ \coprod_{0\le\ell\le 1} \{0 \mapsto \lfloor t_1 \le t_2 \rfloor \mid \_ \mapsto \lfloor t_2 + 1 \le t_1 \rfloor\}(\ell); \mathcal{A}(\mathbb{p}_0(\ell))\end{array}$$

- Certificate $\mathbb{p}$ for "$\triangleleft_{v_1 \le \ell \le v_2}$ **explore** t **in** $\mathbb{p}_0(\ell) \triangleright$" – where $t : \mathbf{T}$ and $v_1, v_2 : \mathbb{Z}$ and ℓ is a metavariable and $\mathbb{p}_0 : \mathbf{V} \to \mathbb{P}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_0(v_1))) \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l} \lceil v_1 \le t \le v_2 \rceil \,; \\ \left\lceil \bigwedge_{v_1+1 \le \ell \le v_2} \mathcal{C}(\mathbb{p}_0(\ell)) \overset{dec}{=} \mathcal{C}(\mathbb{p}_0(v_1)) \right\rceil \,; \\ \coprod_{v_1 \le \ell \le v_2} \lfloor t = \ell \rfloor ; \mathcal{A}(\mathbb{p}_0(\ell)) \end{array}$$

- Certificate $\mathbb{p}$ for "$\triangleleft_{0 \le \ell \le n+1} \mathbb{p}_1{}^* ; \mathbb{p}_2 \triangleright$" – where ℓ is a metavariable and $\mathbb{p}_1, \mathbb{p}_2 : \mathbf{V} \to \mathbb{P}$ and $n : \mathbb{N}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \begin{array}{l} \mathcal{C}(\mathbb{p}_1(0))^* \,; \\ \mathcal{C}(\mathbb{p}_2(0)) \end{array} \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l} \left\lceil \bigwedge_{1 \le \ell \le n+1} \mathcal{C}(\mathbb{p}_1(\ell)) \overset{dec}{=} \mathcal{C}(\mathbb{p}_1(0)) \right\rceil \,; \\ \left\lceil \bigwedge_{1 \le \ell \le n+1} \mathcal{C}(\mathbb{p}_2(\ell)) \overset{dec}{=} \mathcal{C}(\mathbb{p}_2(0)) \right\rceil \,; \\ \coprod_{0 \le \ell \le n+1} \ \mathbf{giter}\ \ell\ (\lambda \ell'. \mathcal{A}(\mathbb{p}_1(\ell'))) \,; \\ \qquad \{n+1 \mapsto \lceil \bot \rceil \mid \_ \mapsto \mathcal{A}(\mathbb{p}_2(\ell))\}(\ell) \end{array}$$

  where VCG of "$\mathbf{giter}\ \ell\ p$" is VCG of "$p(0); \ldots ; p(\ell - 1)$".

Actually $\mathbb{P}$ describes "abstraction patterns" for $\mathbf{P}$. We also need to describe abstraction patterns for $\mathbf{T}$ and for $\mathbf{C}$. We limit here to the former case. Below, we abstract a term by a function $\mathbf{X} \to \mathbf{P}$, where the variable in parameter is used to store the result of the evaluation of the term.

Definition 5. *The type $\mathbb{T}$ of certificates for term is as a pair with projections $\mathcal{C} : \mathbb{T} \to \mathbf{T}$ and $\mathcal{A} : \mathbb{T} \to \mathbf{X} \to \mathbf{P}$ such that for all $\mathbb{t} : \mathbb{T}$,*

$$\neg\, \mathcal{A}(\mathbb{t})(x) \vdash s \to \lightning \ \Rightarrow\ \mathcal{A}(\mathbb{t})(x) \vdash s \to s \oplus \{x \mapsto [\![\mathcal{C}(\mathbb{t})]\!](s)\}$$

Below, we give basic operators about $\mathbb{T}$:

- Certificate $\mathbb{p} : \mathbb{P}$ for "$x \leftarrow \mathbb{t}$" – where $x : \mathbf{X}$ and $\mathbb{t} : \mathbb{T}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq x \leftarrow \mathcal{C}(\mathbb{t}) \qquad \mathcal{A}(\mathbb{p}) \triangleq \mathcal{A}(\mathbb{t})(x)$$

- Certificate $\mathbb{t} : \mathbb{T}$ for "$\mathbf{let}\ x_1 = \mathbb{t}_1\ \mathbf{in}\ \mathbb{t}_2$" – where $x_1 : \mathbf{X}$ and $\mathbb{t}_1, \mathbb{t}_2 : \mathbb{T}$ – is

$$\mathcal{C}(\mathbb{t}) \triangleq \mathcal{C}(\mathbb{t}_2)[x_1 \leftarrow \mathcal{C}(\mathbb{t}_1)] \qquad \mathcal{A}(\mathbb{t}) \triangleq \begin{array}{l} \lambda x_2.\ \mathbf{clone}\ x_1\ \mathbf{in} \\ \qquad (\mathcal{A}(\mathbb{t}_1)(x_1); \mathcal{A}(\mathbb{t}_2)(x_2)) \\ \oplus \{x_1 \mapsto \mathcal{A}(\mathbb{t}_1)(x_1); \mathcal{A}(\mathbb{t}_2)(x_1)\} \end{array}$$

  For example, given $x_2 \neq x_1$, certificate $\mathbb{p}$ for "$x_2 \leftarrow \mathbf{let}\ x_1 = \mathbb{t}_1\ \mathbf{in}\ \mathbb{t}_2$" is

$$\mathcal{C}(\mathbb{p}) = x_2 \leftarrow \mathcal{C}(\mathbb{t}_2)[x_1 \leftarrow \mathcal{C}(\mathbb{t}_1)] \qquad \mathcal{A}(\mathbb{p}) = \mathbf{clone}\ x_1\ \mathbf{in}\ (\mathcal{A}(\mathbb{t}_1)(x_1); \mathcal{A}(\mathbb{t}_2)(x_2))$$

- Given a term $t : \mathbf{T}$, we embed it as a certificate $\dot{t} : \mathbb{T}$ by

$$\mathcal{C}(\dot{t}) \triangleq t \qquad \mathcal{A}(\dot{t}) \triangleq \lambda x.\ x \leftarrow t$$

- At last, we define a *Hoare-term* as a triple (c_0, t, c_1) where $c_0 : \mathbf{C}$, $t : \mathbf{T}$ and $c_1 : \mathbf{X} \to \mathbf{C}$ satisfy

$$[\![c_0]\!](s) \Rightarrow [\![c_1(x)]\!](s, s \oplus \{x \mapsto [\![t]\!](s)\}) \qquad (1)$$

  Hence, we embed such a Hoare-term as the certificate $\mathbb{t}$ defined by

$$\mathcal{C}(\mathbb{t}) \triangleq t \qquad \mathcal{A}(\mathbb{t}) \triangleq \lambda x.\ \lceil c_0 \rceil \,; x \leftarrow \lfloor c_1(x) \rfloor$$

As shown in section 3, parameterized Hoare-terms can be defined in a library. Each definition of a Hoare-term leads to a proof obligation of implication (1) above.

5 Implementation in Coq

This section presents the Coq implementation of our current Coq prototype (downloadable on [Bou13]). In particular, subsection 5.3 presents our Coq sources and their usage. Other subsections introduce the main differences between formalization of section 4 and the current Coq prototype. These differences lie in two categories: improvements of Coq implementation with respect to formalization of section 4 (in subsection 5.2), and conversely, limitations of Coq implementation w.r.t. section 4 (in subsection 5.1).

5.1 Limitations of implementation w.r.t section 4

A poor's man polyhedra domain Our VCG is not yet connected to a Coq implementation of polyhedra domain. Instead, polyhedra are represented as (deeply-embedded) logical formulas: intersection of two polyhedra is simply implemented as a conjunction. Inclusions tests are solved using a decision procedure for Presburger's arithmetic (called `omega`). Convex-hull are computed by "hand" and inserted in certificates.

No partial invariants The implementation does not support partial loop-invariants described in page 14.

5.2 Most notable improvements of implementation w.r.t section 4

5.2.1 Checking efficiently non-influence of partition labels on concrete code

The definition of "$\triangleleft_{0\leq\ell\leq 1}\,(\mathbb{p}_1 \amalg \mathbb{p}_2)$ **;** $\mathbb{p}_3(\ell)\triangleright$" construct in section 4.3, expresses that the value of partition label ℓ must have no influence on the concrete code generated from $\mathbb{p}_3$. Indeed, this is expressed through the following requirement in the generated abstract code

$$\mathcal{C}(\mathbb{p}_3(1)) \equiv \mathcal{C}(\mathbb{p}_3(0))$$

Checking non-influence of partition labels in this way is rather costly: the comparison may involve huge portions of concrete code. And, in many cases – example of figure 11 is the only example of this report that is not in this case – it can be avoided. For instance, on certificate of figure 10, the only part depending on the value of "ℓ" is the invariant, which is obviously eliminated in concrete code. In order to overcome such inefficiencies, our Coq implementation handles partition label in a more subtle way than section 4.

Basically, we change syntax of Scat certificates, such that dependencies on labels in certificates can be more finely expressed by the certificate producer. Typically, dependencies on a label can now only appear under an operator called "**branch**". All operators defined in section 4.3 that introduces an explicit dependencies over a label (like "$\triangleleft_{0\leq\ell\leq 1}\,(\mathbb{p}_1 \amalg \mathbb{p}_2)$ **;** $\mathbb{p}_3(\ell)\triangleright$", "$\triangleleft_{v_1\leq\ell\leq v_2}$ **explore** t **in** $\mathbb{p}_0(\ell)\triangleright$", etc) are now redefined in such way that these dependencies disappear. When needed, a given dependency can be reintroduced by an explicit use of "**branch**". For example, the invariant certificate of figure 10 should now be written as in figure 21. Figure 25 (page 23) illustrates how **branch** is introduced for Hoare-terms using Coq syntax.

$$i \leq x \wedge 0 \leq r \wedge (\mathbf{branch}\ \ell\ \{0 \mapsto r \leq 10.i \mid \_ \mapsto r \leq 5.i\})$$

Figure 21: Reformulation of invariant of figure 10 with explicit dependencies on labels

Operator "**branch**" is defined for most syntactic categories of the certificate language : terms, invariants, programs, etc. For instance, at the program level, "**branch** $\dot{\ell}$ $\mathbb{p}$" – where $\dot{\ell}$ is deep partition label (see below) and $\mathbb{p} : \mathbf{V} \to \mathbb{P}$ – returns a certificate of type $\mathbb{P}$. Hence, non-influence checking is only localized under a "**branch**" operation. Sometimes, it is even not necessary: this is the case for invariants, which simply disappear in concrete code. We formalize these ideas in the next paragraphs.

Deep partition labels in program statements First, at the level of $\mathbf{P}$ statements, we introduce a deeper embedding of partition labels. We note $\mathbf{L}$ a countable type of label names. We use $\dot{\ell}, \dot{\ell}_1$ for elements of $\mathbf{L}$. We impose that such a partition label is associated with static bounds (fixed in the declaration of the label). Hence, we assume two functions $\min, \max : \mathbf{L} \to \mathbf{V}$ such that values v associated to name $\dot{\ell}$ are expected to satisfy $\min(\dot{\ell}) \leq v \leq \max(\dot{\ell})$. These "deep" partition labels are handled by two additional constructs of $\mathbf{P}$ statements with the following abstract semantics

- VCG of "$\mathbf{deflab}\ \dot{\ell}\ v\ p$" – where $p : \mathbf{P}$ – performs the VCG of p by associating value v to label $\dot{\ell}$.
- VCG of "$\mathbf{branch}\ \dot{\ell}\ p$" – where $p : \mathbf{V} \to \mathbf{P}$ – fails if $\dot{\ell}$ has no associated value (there is no enclosing "$\mathbf{deflab}\ \dot{\ell}$") or performs the VCG of $p(v)$ where v is the value associated to $\dot{\ell}$.

The concrete semantics of **deflab** and **branch** do not depend on the values associated to $\dot{\ell}$ (see figure 22). Hence, whereas non-determinism of ". $\amalg$." and ". $\leftarrow \lfloor . \rfloor$" is *intern* (or angelic), non-determinism of **branch** is *extern* (or demonic) see [BvW99].

$$\frac{p \vdash s_0 \to s_1}{\mathbf{deflab}\ \dot{\ell}\ v\ p \vdash s_0 \to s_1} \qquad \frac{\forall v, \min(\dot{\ell}) \leq v \leq \max(\dot{\ell}) \Rightarrow p(v) \vdash s_0 \to s_1}{\mathbf{branch}\ \dot{\ell}\ p \vdash s_0 \to s_1}$$

Figure 22: Concrete semantics of partition labels handling

In order to formalize the abstract semantics, we extend abstract judgment of section 4.2 with a notion of *environment* for labels. As usual, such an environment is a finite map from labels to value $\delta : \mathbf{FMap}(\mathbf{L}, \mathbf{V})$. We use $\oplus$ notation to express overriding of definition in environment.

Ideally, the verification that the value $\delta(\dot{\ell})$ fits in the bounds of $\dot{\ell}$ could be done at **deflab** operation. However, this makes safety theorem requiring the (meta)invariant $\dot{\ell} \in \mathrm{dom}(\delta) \Rightarrow \min(\dot{\ell}) \leq \delta(\dot{\ell}) \leq \max(\dot{\ell})$. In order to avoid this difficulty, we choose a weaker (but less efficient) semantics : the verification of the bounds on $\delta(\dot{\ell})$ is done in **branch** operation.

Hence, the rules of abstract semantics extends the rules of figure 20 – where judgment "$\langle \chi_0 , \phi_0 \rangle\ p\ \langle \chi_1 , \phi_1 \rangle$" are extended as "$\delta \vdash \langle \chi_0 , \phi_0 \rangle\ p\ \langle \chi_1 , \phi_1 \rangle$" – with the ones of figure 23.

In this extended framework, lemma 1 and definition 3 of VCG checker are simply extended by lemma 2 and definition 6 below. This allows to prove theorem 1 in the extended framework.

Lemma 2. *For all s_0, s_1, p, χ, χ_0, ϕ_0, χ_1, ϕ_1, s'_0 and δ*
If $p \vdash s_0 \to s_1$ and $\delta \vdash \langle \chi_0 , \phi_0 \rangle\ p\ \langle \chi_1 , \phi_1 \rangle$ and $|p| \subseteq \chi$ and $\chi \cup |\phi_0| \subseteq \chi_0$ and $s'_0 \equiv_\chi s_0$ and $\gamma(\phi_0)(s'_0)$
Then, $s_1 \neq \lightning$ and there exists s'_1 such that $s'_1 \equiv_\chi s_1$ and $s'_1 \equiv_{\chi_0 \setminus \chi} s'_0$ and $\gamma(\phi_1)(s'_1)$.

Definition 6 (VCG checker)**.** *Using our VCG, we define $.^\checkmark : \mathbf{P} \to \mathbf{bool}$ such that*

$$p^\checkmark \Rightarrow \exists \chi \exists \phi.\ \emptyset \vdash \langle |p| , \top \rangle\ p\ \langle \chi , \phi \rangle$$

Deep partition labels in assertions In order to support loop invariants like those of figure 21, we introduce a new syntactic category $\mathbf{A}$ for "assertions". These assertions extend polyhedral conditions with the ability to read the value of deep labels through assertion "$\mathbf{branch}\ \dot{\ell}\ a$" where $a : \mathbf{V} \to \mathbf{A}$. Hence, the semantics of an assertion "$a : \mathbf{A}$" is a *total* function $[\![a]\!] : \mathbf{FMap}(\mathbf{L}, \mathbf{V}) \to \mathbf{S}^\sharp$. If the context is not consistent with the assertion (using an undefined label) or with the label (its associated value is out of its bounds), then the semantics returns $\bot$. This may only lead VCG to fail in proving safety. The syntax and semantics of assertions is defined below together

$$\frac{\delta \oplus \left\{ \dot{\ell} \mapsto v \right\} \vdash \langle \chi_0 , \phi_0 \rangle\ p\ \langle \chi_1 , \phi_1 \rangle}{\delta \vdash \langle \chi_0 , \phi_0 \rangle\ \mathbf{deflab}\ \dot{\ell}\ v\ p\ \langle \chi_1 , \phi_1 \rangle}$$

$$\frac{\dot{\ell} \in \mathrm{dom}(\delta) \qquad \min(\dot{\ell}) \leq \delta(\dot{\ell}) \leq \max(\dot{\ell}) \qquad \delta \vdash \langle \chi_0 , \phi_0 \rangle\ p(\delta(\dot{\ell}))\ \langle \chi_1 , \phi_1 \rangle}{\delta \vdash \langle \chi_0 , \phi_0 \rangle\ \mathbf{branch}\ \dot{\ell}\ p\ \langle \chi_1 , \phi_1 \rangle}$$

Figure 23: Abstract semantics of partition labels handling

with the usual notion of frame function.

$$a[\delta, \dot{\ell}] \triangleq \texttt{if } \dot{\ell} \notin \mathrm{dom}(\delta) \texttt{ or } \delta(\dot{\ell}) \notin \min(\dot{\ell})..\max(\dot{\ell}) \texttt{ then } \bot \texttt{ else } a(\delta(\dot{\ell}))$$

a	$\triangleq$	ϕ	$\mid$	$a_1 \dot{\wedge} a_2$	$\mid$	$\mathbf{branch}\ \dot{\ell}\ a$
$[\![a]\!](\delta)$	$\triangleq$	ϕ	$\mid$	$[\![a_1]\!](\delta) \sqcap [\![a_2]\!](\delta)$	$\mid$	$\left[\!\left[a[\delta, \dot{\ell}]\right]\!\right](\delta)$
$\lvert a \rvert$	$\triangleq$	$\lvert\phi\rvert$	$\mid$	$\lvert a_1 \rvert \cup \lvert a_2 \rvert$	$\mid$	$\bigcup_{\min(\dot{\ell}) \leq v \leq \max(\dot{\ell})} \lvert a(v) \rvert$

Hence, loop statements in abstract code actually uses *assertions* as invariants. Whereas invariants are ignored in concrete semantics, abstract semantics of loops is now given by the following rule (with no support for partial invariants as explained in section 5.1):

$$\frac{\phi \triangleq [\![a]\!](\delta) \qquad \phi_0 \sqsubseteq \phi \qquad \delta \vdash \langle \chi_0 \,,\, \phi \rangle \ p \ \langle \chi_2 \,,\, \phi_2 \rangle \qquad \phi_2 \sqsubseteq \phi}{\delta \vdash \langle \chi_0 \,,\, \phi_0 \rangle \ p^{*a} \ \langle \chi_0 \,,\, \phi \rangle}$$

Deep partition labels in certificates As explained above, all statements defined in section 4.3 that introduces an explicit dependencies over a label are now redefined in such way that these dependencies disappear. Non-influence checking is removed from generated abstract code. But, in generated abstract code, dependencies on labels remain under the form of a "**deflab**" statement. We limit below to detail the redefinition of statements except loops: redefinition of statements for loop are given section 5.2.2.

- Certificate $\mathbb{p}$ for "$\triangleleft_{0 \leq \dot{\ell} \leq 1} (\mathbb{p}_1 \amalg \mathbb{p}_2)\ ;\ \mathbb{p}_3 \triangleright$" – where $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_1, \mathbb{p}_2, \mathbb{p}_3 : \mathbb{P}$ – is now defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \begin{array}{l} \mathcal{C}(\mathbb{p}_1) \amalg \mathcal{C}(p_2)\ ; \\ \mathcal{C}(p_3) \end{array} \qquad \mathcal{A}(\mathbb{p}) \triangleq \coprod_{0 \leq v \leq 1} \begin{array}{l} \{0 \mapsto \mathcal{A}(\mathbb{p}_1) \mid \_ \mapsto \mathcal{A}(\mathbb{p}_2)\}(v)\ ; \\ \mathbf{deflab}\ \dot{\ell}\ v\ \mathcal{A}(\mathbb{p}_3)) \end{array}$$

- Certificate $\mathbb{p}$ for "$\triangleleft_{0 \leq \dot{\ell} \leq 1}\ t_1 \leq^? t_2 :\ \mathbb{p}_0 \triangleright$" – where $t_1, t_2 : \mathbf{T}$ and $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_0 : \mathbb{P}$ – is now defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_0) \qquad \mathcal{A}(\mathbb{p}) \triangleq \coprod_{0 \leq v \leq 1} \{0 \mapsto \lfloor t_1 \leq t_2 \rfloor \mid \_ \mapsto \lfloor t_2 + 1 \leq t_1 \rfloor\}(v); \ \mathbf{deflab}\ \dot{\ell}\ v\ \mathcal{A}(\mathbb{p}_0)$$

- Certificate $\mathbb{p}$ for "$\triangleleft_{v_1 \leq \dot{\ell} \leq v_2}$ **explore** t **in** $\mathbb{p}_0 \triangleright$" – where $t : \mathbf{T}$ and $v_1, v_2 : \mathbb{Z}$ and $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_0 : \mathbb{P}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_0) \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l} \lceil v_1 \leq t \leq v_2 \rceil\ ; \\ \coprod_{v_1 \leq v \leq v_2} \lfloor t = v \rfloor ; \mathbf{deflab}\ \dot{\ell}\ v\ \mathcal{A}(\mathbb{p}_0) \end{array}$$

Dependencies in certificates can still be reintroduced using **branch**. Certificate $\mathbb{p}$ for "$\mathbf{branch}\ \dot{\ell}\ \mathbb{p}_0$" – where $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_0 : \mathbf{V} \to \mathbb{P}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_0(\min(\dot{\ell}))) \qquad \mathcal{A}(\mathbb{p}) \triangleq \begin{array}{l} \left\lceil \bigwedge_{\min(\dot{\ell})+1 \leq v \leq \max(\dot{\ell})} \mathcal{C}(\mathbb{p}_0(v)) \equiv \mathcal{C}(\mathbb{p}_0(\min(\dot{\ell}))) \right\rceil\ ; \\ \mathbf{branch}\ \ \dot{\ell}\ \ \lambda v.\mathcal{A}(\mathbb{p}_0(v)) \end{array}$$

5.2.2 VCG of loop unrolling

The abstract code generated from certificates of loop unrolling in section 4.3 is very inefficient. Indeed, the number of VCG iterations on the loop body is quadratic in the number of unrolling steps, whereas it is linear in our Coq implementation. This is achieved by introducing a dedicated construct in the syntax of abstract programs: "$\mathbf{le\_iter}\ n\ \dot{\ell}\ p_1\ p_2\ p_3$" – where $n : \mathbb{N}$, $\dot{\ell} : \mathbf{L}$ and $p_0, p_1, p_2 : \mathbf{P}$ – is equivalent to

$$\mathbf{le\_iter}\ n\ \dot{\ell}\ p_1\ p_2\ p_3 \triangleq \coprod_{0 \leq v \leq n+1} \begin{array}{l} \mathbf{giter}\ v\ (\lambda v'.\mathbf{deflab}\ \dot{\ell}\ v'\ p_1)\ ; \\ \mathbf{deflab}\ \dot{\ell}\ v\ (\{n+1 \mapsto p_3 \mid \_ \mapsto p_2\}(v)) \end{array}$$

But VCG of "$\mathbf{le\_iter}\ n\ \dot{\ell}\ p_1\ p_2\ p_3$" performs only $n+1$ iterations of p_1 and p_2, plus one iteration of p_3.

For full unrolling of loops "$\triangleleft_{0\leq\dot{\ell}\leq n+1}\ \mathbb{p}_1{}^*; \mathbb{p}_2\ \triangleright$" – where $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_1, \mathbb{p}_2 : \mathbb{P}$ and $n : \mathbb{N}$ – is now defined as the certificate $\mathbb{p}$ such that

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_1)^* ; \mathcal{C}(\mathbb{p}_2) \qquad\qquad \mathcal{A}(\mathbb{p}) \triangleq \mathbf{le\_iter}\ n\ \dot{\ell}\ \mathcal{A}(\mathbb{p}_1)\ \mathcal{A}(\mathbb{p}_2)\ \lceil\bot\rceil$$

At last, partial unrolling statement of section 4.3 is now generalized with partition labels in the following way: certificate $\mathbb{p}$ for "$\triangleleft_{0\leq\dot{\ell}\leq n+1}\ \mathbb{p}_1{}^{*a}; \mathbb{p}_2\ \triangleright$" – where $\dot{\ell} : \mathbf{L}$ and $\mathbb{p}_1, \mathbb{p}_2 : \mathbb{P}$ and $n : \mathbb{N}$ and $a : \mathbf{A}$ – is defined by

$$\mathcal{C}(\mathbb{p}) \triangleq \mathcal{C}(\mathbb{p}_1)^* ; \mathcal{C}(\mathbb{p}_2) \qquad\qquad \mathcal{A}(\mathbb{p}) \triangleq \mathbf{le\_iter}\ \ n\ \ \dot{\ell}\ \ \mathcal{A}(\mathbb{p}_1)\ \ \mathcal{A}(\mathbb{p}_2)\ \ (\mathcal{A}(\mathbb{p}_1)^{*a}\ ;\ \mathcal{A}(\mathbb{p}_2))$$

5.2.3 Coding a frame as an integer

In our implementation, the type $\mathbf{X}$ is defined as `positive`, that is the type of positive integers. Instead to represent frames as sets of variables, we overapproximate a frame by its upper bound: more precisely, a frame χ is encoded as the lowest positive p such that

$$x \in \chi \Rightarrow x < p$$

This approximation both improves efficiency of frames computations (union between sets is efficiently approximated by maximum of integers), and simplifies metaproofs (reasoning on set inclusions is safely approximated by reasoning on integer comparisons). This approximation is sufficient in most of the formalization (which deals about "fresh" variables).

However, this approach is not sufficient for *modifying frames* involved in *partial invariants*. An other notion of frame, similar to the one of the paper, must be used instead. Hence, our implementation does not yet support partial invariants. They do not seem to raise any kind of difficulty: supporting them is only a matter of development time (using two notions of frames instead of a single one).

5.2.4 Reflecting concrete wp-calculus to automate (meta)proofs of refinement

SCAT theory can be understood using two levels of refinement. The first level is that concrete semantics of abstract programs refine their abstract semantics. The second level is that in certificates, generated concrete programs refine generated abstract programs. Until here, we have expressed these two levels of refinement using operational semantics for concrete semantics. Our implementation uses instead a much more effective semantics in COQ proofs: a weakest-precondition calculus. This weakest-precondition calculus – which computes directly on COQ propositions – allows to simplify proofs about concrete semantics. Similar ideas have been already detailed in [Bou07].

However, we believe that "natural semantics" is the most simple semantics of concrete programs: hence it should be used as the COQ specification of programs behavior. Thus, we have established in COQ a proof of equivalence between the natural semantics presented at figure 18 and our weakest-precondition calculus.

Below, we rephrase the global picture of the formalization, using weakest-precondition calculus instead of natural semantics. Actually, theorem 2, our main theorem, is unchanged. Only intermediate refinement steps are rephrased. Hence, our wp-calculus is a function $\mathrm{wlp} : \mathbf{P} \to (\mathbf{S} \to \mathbf{Prop}) \to \mathbf{S} \to \mathbf{Prop}$ such that

$$\mathrm{wlp}(p)(Q)(s_0) \ \Leftrightarrow\ (\forall s_1,\ p \vdash s_0 \to s_1\ \Rightarrow\ s_1 \neq \lightning \wedge Q(s_1))$$

We do not detail its inductive definition: it is standard and can be found in our COQ sources [Bou13]. Lemma 2, and theorem 1 are actually expressed under the following form below:

Lemma 3. *For all s_0, p, χ, χ_0, ϕ_0, χ_1, ϕ_1, s'_0 and δ*
If $\delta \vdash \langle \chi_0 , \phi_0 \rangle\ p\ \langle \chi_1 , \phi_1 \rangle$ and $|p| \subseteq \chi$ and $\chi \cup |\phi_0| \subseteq \chi_0$ and $s'_0 \equiv_\chi s_0$ and $\gamma(\phi_0)(s'_0)$
Then, we have

$$\mathrm{wlp}(p)(\lambda s_1.\exists s'_1.s'_1 \equiv_\chi s_1 \wedge s'_1 \equiv_{\chi_0 \setminus \chi} s'_0 \wedge \gamma(\phi_1)(s'_1))(s_0)$$

Theorem 3 (VCG safety)**.** $p^\checkmark\ \Rightarrow\ \mathrm{wlp}(p)(\lambda s_1.\mathbf{True})(s_0)$.

Definition 4 is rephrased by definition 7 below. This allows much automated proofs that $\mathbb{P}$ statements are wellformed.

Definition 7. *The type $\mathbb{P}$ of certificates for concrete programs is defined as a pair with projections $\mathcal{C} : \mathbb{P} \to \mathbf{P}$ and $\mathcal{A} : \mathbb{P} \to \mathbf{P}$ such that for all $\mathbb{p} : \mathbb{P}$, $\mathrm{wlp}(\mathcal{A}(\mathbb{p}))(Q)(s)\ \Rightarrow\ \mathrm{wlp}(\mathcal{C}(\mathbb{p}))(Q)(s)$.*

Similarly, definition 5 is rephrased using wp-calculus by definition 8 below. Actually, this definition uses a dual version of wlp at kernel level, as a function angel_wlp : $\mathbf{P} \to (\mathbf{S} \to \mathbf{Prop}) \to \mathbf{S} \to \mathbf{Prop}$ such that

$$\text{angel\_wlp}(p)(Q)(s_0) \;\Rightarrow \neg\text{wlp}(p)(\lambda s_1.\neg Q(s_1))(s_0)$$

But, this dual version handles double negation of $\neg\text{wlp}(p)(\lambda s_1.\neg Q(s_1))(s_0)$ in a more effective way. Its main interest comes from the property that using excluded-middle[8], we have

$$\neg\text{wlp}(p)(\lambda s_1.\neg Q(s_1))(s_0) \;\Leftrightarrow\; \exists s_1,\ p \vdash s_0 \to s_1 \ \wedge (s_1 \neq \lightning \Rightarrow Q(s_1))$$

Definition 8. *The type $\mathbb{T}$ of certificates for term is as a pair with projections $\mathcal{C} : \mathbb{T} \to \mathbf{T}$ and $\mathcal{A} : \mathbb{T} \to \mathbf{X} \to \mathbf{P}$ such that for all $\mathbb{t} : \mathbb{T}$,*

$$\text{angel\_wlp}\ (\mathcal{A}(\mathbb{t})(x))\ (\lambda s_1.s_1 = s_0 \oplus \{x \mapsto [\![\mathcal{C}(\mathbb{t})]\!](s_0)\})\ (s_0)$$

5.3 Presentation of the current SCAT prototype in COQ

The COQ sources of our SCAT prototype are downloadable on [Bou13]. A good entry point is file `Examples.v` containing all the examples of section 3. Let us now present the verification process of a given source program. Given a program called `source` and a certificate called `annotated`, the COQ proof that `source` is safe is given by the small script below.

```
Goal is_ok source.
Proof.
   apply vcg_correctness with (cert:=annotated).
   (* 1st subgoal *) vm_compute ; auto.
   (* 2nd subgoal *) simplify_vc_eval ; try omega.
Qed.
```

The goal to prove is "`is_ok source`" which exactly means

$$\forall s \forall s',\ \texttt{source} \vdash s \to s' \;\Rightarrow\; s' \neq \lightning$$

The script of the proof first calls our main theorem `vcg_correctness` (roughly theorem 2) using `annotated` as witness. This splits the goal in two subgoals: first "$\texttt{source} \overset{dec}{=} \mathcal{C}(\texttt{annotated})$" and second "$\mathcal{A}(\texttt{annotated})^{\checkmark}$". The first goal is simply proved by computational reflection (tactic `vm_compute`). The second goal is proved by calling our VCG which returns a big conjunction of VC, this conjunct is then split (tactic `simplify_vc_eval`), and each VC is proved using tactic `omega` of Presburger's arithmetic.

Figure 24 gives the main correspondences between paper notations and our COQ sources. On the contrary to the paper presentation, our implementation distinguishes between syntax of concrete code (called `prog`) and between syntax of abstract code (called `kernel`). This allows to oblige loops of abstract code to be annotated with invariants. And certain constructs such as `clone` or `branch` that are not intended to be used in programs (concrete code), appear only in kernel syntax. Hence, our COQ implementation does not define an operational semantics for these constructs: their concrete semantics is instead directly provided by the concrete wp-calculus presented in section 5.2.4. At last, the subcategory of atomic statements is shared between concrete syntax and kernel syntax through a type called `atom`.

At last, figure 25 details the correspondence for Hoare-terms. It shows also how **branch** is made explicit in Hoare-terms using COQ syntax. We do not describe here the structure of our COQ sources: see instead the documentation extracted from the sources.

[8] On the contrary to the framework of [Bou07], expressing duality between wlp and angel_wlp requires here excluded-middle. This due to the fact that these predicate transformers are neither conjunctive nor disjunctive, because of the presence of both external non-determinism (e.g. **branch**) or internal non-determinism (e.g. "$\cdot \leftarrow \lfloor . \rfloor$") in the concrete semantics.

Types

$\mathbf{V}$	`value`
$\mathbf{X}$	`var`
$\mathbf{T}$	`term`
$\mathbf{S}$	`state`
$\mathbf{C}$	`cond`
$\mathbf{S}^\sharp$	abusively `cond` (see limitations of section 5.1)
$\mathbf{A}$	`assertion`
$\mathbf{P}$	`atom` or `prog` or `kernel`
$\mathbb{T}$	`abs_term`
$\mathbb{P}$	`certif`

Terms and Conditions

Paper and Coq roughly coincide except:

$\mathbf{old}(t)$	`told t`

Statements in `atom`

$\mathbf{old}(t)$	`told t`
$\lfloor c \rfloor$	`ensure c`
$\lceil c \rceil$	`require c`
$x \leftarrow t$	`assign x t`
$x \leftarrow \lfloor c \rfloor$	`guassign [x] c` (NB: `guassign` accepts a list of variable to assign)

Statements in `prog`

Extend syntax of `atom` with

$p; p'$	`seq p p'` or `p -; p'`
p^*	`loop p` or `bloop p done`
$p \amalg p'$	`join p p'`
if c **then** p **else** p' **fi**	`ifc c then p else p' fi`
while c **do** p **done**	idem

Statements in `kernel`

Extend syntax of `atom` with

$p; p'$	`Kseq p p'`
p^{*a}	`Kloop p a`
$p \amalg p'$	`Kjoin p p'`
$\coprod_{v \le \ell \le v'} p(\ell)$	`Kgjoin v v' (fun l => p l)`
clone x **in** p	`Kclone x p`
deflab ℓ v p	`Kdeflab l v p`
branch ℓ p	`Kbranch l p`

Certificates in $\mathbb{T}$

Extend Hoare-terms of figure 25 with

$\dot{t}$	`t`
let $x = \mathbb{t}$ **in** $\mathbb{t}'$	`let_term x := t in t'`
$\mathbf{r}[\mathbb{t}]$	`abs_rewrite t r`

Certificates in `certif`

Extend `prog` syntax with

$x \leftarrow \mathbb{t}$	`abs_assign x t`
$\triangleleft_{0 \le \ell \le 1} (\mathbb{p} \amalg \mathbb{p}')\,;\,\mathbb{p}'' \triangleright$	`joinseq p p' l p''` or with explicit convex-hull `c` as `joinchseq p p' l p'' c`
$\triangleleft_{0 \le \ell \le 1}\ t \le^? t' :\ \mathbb{p} \triangleright$	`casescmp t t' l p`
$\triangleleft_{v \le \ell \le v'}$ **explore** t **in** $\mathbb{p} \triangleright$	`explore t v v' l p`

$\mathbb{p}^{*a}$	`bloop p`
	`invariant a`
	`done`

$\triangleleft_{0 \le - \le n} \mathbb{p}^{*a}; \mathbb{p}' \triangleright$	`bloop p`
	`fwd n`
	`invariant a`
	`then p'`
	`done`

$\triangleleft_{0 \le \ell \le n+1} \mathbb{p}^*; \mathbb{p}' \triangleright$	`bloop`
	`split l`
	`p`
	`all n+1`
	`then p'`
	`done`

Operations involved in semantics

$[\![t]\!](s)$	`teval t s`
$[\![t]\!](s, s')$	`oteval t s s'`
$[\![c]\!](s)$	`eval c s`
$[\![c]\!](s, s')$	`oeval c s s'`
$\lightning$	`None`
$s \oplus \{x \mapsto v\}$	`add (vid x) v s`
$p \vdash s \to s'$	`sem p s s'` (for `p:prog`) or `asem p s s'` (for `p:atom`)

Checker

$\mathcal{A}(\mathbb{p})$	`abstract p`
$\mathcal{C}(\mathbb{p})$	`concrete p`
$\mathcal{A}(\mathbb{t})$	`tabs t`
$\mathcal{C}(\mathbb{t})$	`tconc t`
$p \overset{dec}{=} p'$	`prog_eq (normalize p) (normalize p')` NB: for `p,p': prog`, syntactic equality after normalization
$\mathcal{A}(\mathbb{p})^\checkmark$	`vc_eval (vcg p)`
Thm 2	`vcg_correctness`

Figure 24: Main correspondences between paper notations and Coq sources

Names of Hoare-term functions in Coq

`mpi`	`mult_pos_interv`	`mlc`	`mult_left_cte`	`dbpc`	`Zdiv_by_pos_cte`
`mni`	`mult_neg_interv`	`mrc`	`mult_right_cte`		

Application of Hoare-term functions (with branch) in Coq examples

`mlc` $t\ t'\ 3$	`mult_left_cte t t' 3`
`mlc` $t\ t'\ (\ell{+}1)$ (with ℓ label)	`ht_branch (mult_left_cte t t') l (fun v => v+1)`
`mlc` $t\ t'\ \{1 \mapsto 3 \mid 2 \mapsto 6 \mid \_ \mapsto 0\}(\ell)$	`ht_branch (mult_left_cte t t') l (fun v => match v with 1 => 3 | 2 => 6 | _ => 0 end)`
`mpi` $t\ t'\ (6, 10)$	`mult_pos_interv t t' (6,10)`
`mpi` $t\ t'\ \{0 \mapsto (6,10) \mid \_ \mapsto (1,5)\}(\ell)$	`ht_branch (mult_pos_interv t t') l (fun v => match v with 0 => (6,10) | _ => (1,5) end)`

Figure 25: Correspondence between paper and Coq for Hoare-terms

6 Conclusion

This paper proposes a certificate format called SCAT to formally verify the results of static analyzers. Given a source code and a SCAT-certificate, the SCAT-CHECKER produces a list of polyhedral inclusion tests that are sufficient to ensure the safety of the source code. Our tool is formally certified in COQ with respect to the operational semantics of the source code [Bou13].

SCAT aims to certify analyzers which perform dynamic program transformations in order to increase the precision of their results. In our COQ development, we paid specific attention to easing the extension of SCAT with new transformations. This requires defining the transformation in COQ and proving that its input (i.e. the concrete code) *refines* its output (i.e. the abstract code). In order to provide automation for such a refinement proof, we have developed a *concrete weakest-precondition calculus* which is reflected in COQ and inspired from [Bou07]. Hence, tedious proofs on the operational semantics are avoided. This paper presents two significant patterns of such transformations: linearization – for approximating non-affine code – and trace partitioning – that increases precision with convex abstract domains.

In order to simplify the formalization and limit the development effort in COQ, we choose to restrict our SCAT-CHECKER to the domain of polyhedra. However, the SCAT format of certificates can be used for analyzers based on any combination of abstract domains as long as they are special cases of polyhedra (e.g. intervals, octagons,...). Note that only the last iteration of the analysis must be replayed with the polyhedra domain in order to generate the certificates which report the linearization of expressions.

Our work is dedicated to *a posteriori* certification of static analysis. However, as a conclusion, let us rephrase our contributions as a variant of the proof-carrying code architecture of [BJP06, BJT07]. On the code-producer side, the proof of safety is assigned to a static analyzer that produces a certificate $\mathbb{p}$ if it succeeds in discarding all the safety requirements, or it just fails. On the code-consumer side, the certificate $\mathbb{p}$ is used to extract the source program, $\mathcal{C}(\mathbb{p})$, and a Hoare proof sketch, $\mathcal{A}(\mathbb{p})$, that is played in our SCAT-CHECKER. Thanks to Theorem 2 the validity of $\mathcal{A}(\mathbb{p})$ implies the safety of $\mathcal{C}(\mathbb{p})$. The Hoare proof is designed to exactly stick to the behavior of the analyzer on its last iteration so that the implications required at the core of the proof can be decided by the inclusion test ($\sqsubseteq$) of the abstract domain. Hence, we guarantee that a success of the analyzer leads to a proof without resorting to more powerful logics and tools (such as SMT solvers or general theorem provers).

Perspectives. In future work, we plan to instrument an analyzer so that it returns SCAT certificates. This will allow us to evaluate the efficiency of certificate generation and verification with the OCAML SCAT-CHECKER extracted from our COQ development. Then, if it reveals that the size of certificates must be reduced, we will use the technique of [BJT07] to reduce the size of our loop invariants. Moreover, the SCAT-CHECKER could use a combination of strongest post-condition and weakest pre-condition calculus in order to be able to chose the way that needs fewer external calls to basic polyhedra operations.

Our work is part of the VERASCO project on the development of certified analysis for the COMPCERT compiler. Thus, the following step is to extend the SCAT format to address programs written in the Clight[9] language. Since the architecture of our checker establishes a clear separation of treatments of source code and abstract code, and since in the context of critical software, we can assume non-recursive functions and no dynamic allocation of memory, we have good hope that the addition of the remaining constructions of Clight (function call, arrays, pointers, floating-point numbers) will retain the simplicity of our initial development.

Acknowledgments We thank David Monniaux, Alexis Fouilhe and Marie-Laure Potet for useful remarks on this paper. This work is supported by French ANR-VERASCO.

References

[BCC+10] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *Proc. of AIAA Infotech@Aerospace*, 2010. (document)

[BJP06] Frédéric Besson, Thomas P. Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3), 2006. 1, 6

[9] One possible input language of the COMPCERT compiler

[BJPT10] Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *TGC'10*, volume 6084 of *LNCS*, 2010. 1, 1, 1, 2.2, 4

[BJT07] Frédéric Besson, Thomas P. Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In *ESOP'07*, volume 4421 of *LNCS*, 2007. 6, 6

[Bou07] Sylvain Boulmé. Intuitionistic refinement calculus. In *TLCA'07*, volume 4583 of *LNCS*, 2007. 5.2.4, 8, 6

[Bou13] Sylvain Boulmé. SCAT sources, 2013. http://www-verimag.imag.fr/~boulme/scat/. 4, 5, 5.2.4, 5.3, 6

[BvW99] Ralph-Johan Back and Joakim von Wright. *Refinement calculus - a systematic introduction*. Graduate texts in computer science. Springer, 1999. 1, 3.1, 5.2.1

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP'05*, volume 3444 of *LNCS*, 2005. (document)

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*. ACM Press, 1978. 1

[Cha06] Amine Chaieb. Proof-producing program analysis. In *ICTAC'06*, volume 4281 of *LNCS*, 2006. 1

[Fil98] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *TYPES'98*, volume 1657 of *LNCS*, 1998. 4.1

[GS07] Benjamin Grégoire and Jorge Luis Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In *TGC'07*, volume 4912 of *LNCS*, 2007. 1

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), 2009. 1

[Min06] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, volume 3855 of *LNCS*, 2006. (document), 2.3, 2.3

[MM10] Yannick Moy and Claude Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11), 2010. 1

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05*, volume 3444 of *LNCS*, 2005. (document), 1, 2.4, 2.4, 2.4, 3.5, 3.5, 3.6

[SYY03] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *APLAS'03*, volume 2895 of *LNCS*, 2003. 1

[The12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. TypiCal Project – INRIA, 2004-2012. http://coq.inria.fr. (document)