A certifying frontend for (sub)polyhedral abstract domains

Alexis Fouilhe, Sylvain Boulmé

A certifying frontend for (sub)polyhedral abstract domains*

Alexis Fouilhe and Sylvain Boulmé

Univ. Grenoble-Alpes, VERIMAG, F-38000 Grenoble, France
{alexis.fouilhe,sylvain.boulme}@imag.fr

Abstract. Convex polyhedra provide a relational abstraction of numerical properties for static analysis of programs by abstract interpretation. We describe a lightweight certification of polyhedral abstract domains using the Coq proof assistant. Our approach consists in delegating most computations to an untrusted backend and in checking its outputs with a certified frontend. The backend is free to implement relaxations of domain operators (i.e. a *subpolyhedral* abstract domain) in order to trade some precision for more efficiency, but must produce hints about the soundness of its results. Previously published experimental results show that the certification overhead with a full-precision backend is small and that the resulting certified abstract domain has comparable performance to non-certifying state-of-the-art implementations.

Keywords: abstract interpretation, abstract domain of polyhedra, program verification in Coq

1 Introduction

Astrée [1] is a major success of semantics-based static analysis of programs: it is capable proving the absence of runtime undefined behaviours in large scale real world C programs from avionics. Abstract interpretation [2], on which Astrée is based, formalizes the state analysis of programs and guarantees that the analyzer soundly over-approximates the behaviours of the program under analysis. However, Astrée is itself a complex piece of software. Despite the care put in its development, it may contain bugs. One possible solution consists in proving that the analyzer implementation is sound and having this proof mechanically checked by a proof assistant. Trusting the result of the analyzer is thereby reduced to trusting the proof checker and answering the question: Is what has been proved what we want to prove?

This question is especially relevant in the context of automated C program manipulation, where the semantics of the C programming language are part of the specification. The CompCert C compiler [3] is a successful project built with the Coq proof assistant. The Verasco project aims at building an abstract interpreter in a similar manner, reusing the CompCert infrastructure: mainly the formalized semantics of the C programming language and the frontend. Once

* This work was partially supported by ANR project VERASCO (INS 2011).

the semantics of the program under analysis are defined, the correctness proof is composed of two components: the abstract domain must be shown to soundly over-approximate manipulations of sets of reachable states, and the link between the program semantics and the abstract domain must be proved correct.

Our work addresses the problem of proving correct in Coq an implementation of the abstract domain of polyhedra [4], which capture linear relationships between program variables. The abstract domain we built is similar both in features and performance to the core of the polyhedra library in the PPL [5] and Apron [6]. We adopted the same two tier architecture as Besson et al. [7]: an untrusted Ocaml backend performs most of the computations and outputs proof hints for the results it produces, which are used by a frontend developed in Coq to build trustworthy results.

The efficient generation of proof hints, which we call *certificates*, is described elsewhere [8], along with an experimental evaluation of the overall abstract domain. The main contribution of the work described here is the design of the link between the Coq frontend and the untrusted backend. It avoids the conversion and transfer of polyhedra. This makes the coupling between the frontend and the backend very loose. As a result, building other certificate-producing backends is easy and has no impact on the Coq frontend code. Complete freedom is given on the choice of data structures: a backend could use constraint or double representation for polyhedra. Furthermore, since the backend does not give formal precision guarantees, a backend could implement relaxations of domain operators [9,10], trading precision for efficiency.

We also present a lightweight method to ensure the soundness of Ocaml code extracted from the Coq frontend, even when the backend has an internal state, or when the functional purity of the backend is not trusted. Last, we describe the architecture of the frontend as a collection of functors which extends a bare metal abstract domain interface. This approach makes the proof modular: it is simpler and more robust to change.

2 A certified interface of polyhedral abstract domains

Let us introduce an small imperative programming language, named *PL*. The syntax of *PL* programs is described on figure 1. Letter t stands for an affine term and c is a condition over numerical variables with the following syntax:

$$c ::= t_1 \bowtie t_2 \mid \neg c \mid c_1 \wedge c_2 \mid c_1 \vee c_2$$

with $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. All numbers are rationals.

s	$x := t$	$s_1; s_2$	**if**$(c)\{s_1\}$**else**$\{s_2\}$	**while**$(c)\{s : p_i\}$
$\langle p \rangle s$	$p[x := t]$	$\langle \langle p \rangle s_1 \rangle s_2$	$\langle p \sqcap c \rangle s_1$ $\sqcup$ $\langle p \sqcap \neg c \rangle s_2$	$\begin{cases} p_i \sqcap \neg c & \text{if } p \sqsubseteq p_i \wedge \langle p_i \sqcap c \rangle s \sqsubseteq p_i \\ \top \sqcap \neg c & \text{otherwise} \end{cases}$

Fig. 1. Syntax and postcondition computation of *PL*

Let us now sketch how to build a "*sound-by-construction*" static analyzer performing a value analysis for this toy language. This will also introduce our logical interface of abstract domains. For simplicity, we assume here that *PL* programs are annotated, by an *untrusted* analyzer, with candidate invariants p_i where they are hard to infer: at the loop headers. Hence, we only have to prove the soundness of a postcondition computation, described on figure 1, which checks whether the candidate invariants are inductive. Given a precondition p, the postcondition $\langle p \rangle s$ of a statement s is computed using recursion on the syntax of s. If a candidate invariant cannot be shown to be inductive, it is replaced by $\top$, which is always safe. This happens when the candidate invariant does not include the postcondition of the loop body and may have two causes. Either the candidate invariant *is not* inductive, or the abtract domain used for checking inductiveness is not precise enough.

The postcondition computation relies on the operators of the abstract domain. Let us introduce them on the example of the abstract domain of polyhedra. Their Coq formal specifications are presented on figure 2. A polyhedron p encodes a formula $\bigwedge_i \boldsymbol{a_i}.\boldsymbol{x} \leq b_i$, where $\boldsymbol{a_i}$ is a row vector of rational constants, b_i is a rational constant and $\boldsymbol{x}$ is a column vector of numerical variables of the program. Its semantics, or concretization, is the *predicate* $[\![p]\!]$ defined as $\lambda m. \bigwedge_i \sum_j a_{ij}.m(x_j) \leq b_i$, where m is a total map from variables to rationals representing a memory state. We omit the definitions of the semantics $[\![t]\!]$ of t and $[\![c]\!]$ of c, as they are standard.

- Polyhedron $\top$ corresponds to the predicate `True`.
- Polyhedron $\bot$ corresponds to the predicate `False`.
- Polyhedron $p \sqcap c$ over-approximates the conjunction of $[\![p]\!]$ and $[\![c]\!]$ (hence, the forward predicate transformer for *guard*).
- Polyhedron $p_1 \sqcup p_2$ over-approximates the disjunction of $[\![p_1]\!]$ and $[\![p_2]\!]$ (hence, the forward predicate transformer for *join*).
- Given a term t and a variable x, polyhedron $p[x := t]$ over-approximates the result of applying the forward predicate transformer for $x := t$ on $[\![p]\!]$.
- Boolean $p_1 \sqsubseteq p_2$ over-approximates the inclusion of predicates: if it is true, then $[\![p_2]\!]$ is a logical consequence of $[\![p_1]\!]$.

Although we have omitted them here, the Coq code of the invariant checker needs to formalize the semantics of *PL* and prove that the reachable states of a *PL* program are soundly captured by the postcondition computation defined on figure 1. This relies on the abstract domain operations satisfying the specifications on figure 2.

These specifications are weak: they only enforce that the operators of the abstract domain perform safe over-approximations. They give no information on the precision of the results. Building an abstract domain satisfying these specifications is our focus in this paper.

$$[\![\top]\!]m \qquad \neg\,[\![\bot]\!]m \qquad [\![p]\!]m \wedge [\![c]\!]m \;\Rightarrow [\![p \sqcap c]\!]m \qquad [\![p_1]\!]m \vee [\![p_2]\!]m \;\Rightarrow [\![p_1 \sqcup p_2]\!]m$$

$$[\![p]\!]m \Rightarrow [\![p[x := t]]\!](m[x := [\![t]\!]m]) \qquad p_1 \sqsubseteq p_2 \wedge [\![p_1]\!]m \;\Rightarrow [\![p_2]\!]m$$

Fig. 2. Correctness specifications of our main polyhedral operations

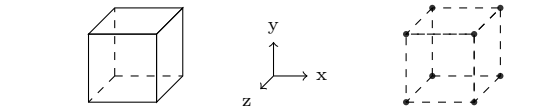

Fig. 3. Constraint and generator representations of the 3-dimensional hypercube

3 Result certification of polyhedral abstract domains

While using Coq enhances the reliability of software, it sets a number of restrictions on the programs which can be reasoned about. First, the Coq programming language is restricted to pure functions that must be shown to terminate. The algorithms used by the abstract domain of polyhedra are complex to implement and these Coq requirements would have made their implementation even harder. The most representative example is the simplex algorithm.

Furthermore, Coq programs cannot use native machine arithmetic for computing. Instead, numbers are represented as lists of bits. The algorithms operating on polyhedra being arithmetic intensive, this suggests carrying out as much computation outside Coq as possible. Again, the simplex algorithm is the most representative example.

The arguments required to prove the correctness of the operators of the abstract domain of polyhedra make it convenient to offload much computation to an untrusted oracle and keep only a small amount of code to be proved correct in Coq. We back up this claim with some background on polyhedra.

3.1 Representing polyhedra for certification

A polyhedron can be represented in two ways: as a conjunction of constraints (i.e. affine inequalities) or as a set of generators, as illustrated on figure 3.

When working with generator representation, proving correctness of the polyhedral operations specified on figure 2 requires proving completeness results. Indeed, forgetting one vertex of the hypercube yields an under-approximation of this hypercube. Correctness of static forward analysis is not preserved through under-approximation, but through over-approximation.

Proving correctness of polyhedral operations in constraint representation is easier, as forgetting one constraint of the result produces a safe over-approxi-

$$[\![p]\!]m \Rightarrow \forall a, [\![p\backslash x]\!](m[x := a]) \qquad [\![p]\!](m[x_1 := m(x_2)]) \Rightarrow [\![p[x_1 \leftarrow x_2]]\!]m$$

Fig. 4. Correctness specifications of the projection and renaming operators

mation. The proof can be built incrementally by proving that each produced constraint includes the exact result.

3.2 Expressing correctness as inclusions of polyhedra

The correctness of each operation reduces to inclusions of polyhedra. However, this reduction requires to first break the complex operations given in figure 2 into simpler ones, which compose a *low level interface* of polyhedral abstract domains.

This low level interface has the same inclusion test and join operators as before. The guard is restricted to one affine constraint. There is no forward predicate transformer for assignment, but projection and renaming operators are provided, from which it can be built (see §6.3). The specifications for projection and renaming are given in figure 4. Polyhedron $p\backslash x$ results from the projection of p on the space of variables where dimension x has been removed. Renaming $p[x_1 \leftarrow x_2]$ over-approximates the renaming of x_1 as x_2 in p. Variable x_2 is required to be fresh, but this precondition is not formalized in the Coq specification as it is not needed for our correctness proofs. A violation of this precondition may result in a precision bug, see §4.1.

Correctness of the operators of the low level interface can now be reduced to inclusions of polyhedra, with the exception of renaming. However, under the freshness precondition, renaming is a purely syntactic transformation. Three operations remain: the guard of a polyhedron p with an affine constraint c, the projection of a variable x from a polyhedron p and the join of two polyhedra p_1 and p_2. Each constraint c' of their result must be shown to satisfy the inclusion properties specified below.

guard. $\bigwedge_i c_i \wedge c \sqsubseteq c'$, with $p \triangleq \bigwedge_i c_i$
projection. $p \sqsubseteq c'$ (and x should have a nil coefficient in c')
join. $p_1 \sqsubseteq c'$ and $p_2 \sqsubseteq c'$

3.3 Checking inclusion of polyhedra

The correctness of the complex low level operations reduces to inclusions of polyhedra. Farkas's lemma further reduces polyhedra inclusion to a linear programming problem on constraint representation of polyhedra. Below, we say that "constraint $\boldsymbol{a_1}.\boldsymbol{x} \leq b_1$ *syntactically entails* $\boldsymbol{a_2}.\boldsymbol{x} \leq b_2$" if and only if $\boldsymbol{a_1} = \boldsymbol{a_2}$ and $b_1 \leq b_2$.

Farkas's lemma. A polyhedron $p \triangleq \bigwedge c_i$ is included in a one-constraint polyhedron c' if and only if there exists $\lambda_i \geq 0$, such that $\sum_i \lambda_i.c_i$ syntactically entails c'.

```
type poly

val freshId : poly -> positive
val top : poly
val isEmpty : poly -> (cert option)
val isIncl : poly * poly -> (cert option)
val guard : poly * affineConstraint -> (poly option) * cert
val join : poly * poly -> poly * cert
val project : poly * var -> poly * cert
val rename : var * var * poly -> poly
```

Fig. 5. OCaml interface of the backend

Given $\boldsymbol{\lambda}$, a vector of λ_i, checking that p is included in c' is straightforward: build the linear combination $c \triangleq \lambda.p$ and check that c syntactically entails c'. This generalizes to a polyhedron $p' \triangleq \bigwedge c'_j$ by supplying a vector of coefficients $\boldsymbol{\lambda}_j$ for each constraint c'_j of p'. The vectors $\boldsymbol{\lambda}_j$ form a matrix Λ such that $p_s \triangleq \Lambda.p$ and the constraints of p_s entail those of p' syntactically. The matrix Λ can be used by a checker to validate the result of an operator: we call Λ an *inclusion certificate*. Nothing is proved when the check fails, however.

3.4 Core architecture of the abstract domain

Farkas's lemma makes result verification cheap. Moreover, it guarantees that producing a certificate to justify an inclusion property is always possible. This motivates the two tier architecture we have chosen for our polyhedral abstract domain.

The abstract domain is split in an untrusted OCaml backend and a frontend which is developed in Coq. The backend performs most complex computations of the low level interface. Its interface is given on figure 5. The backend provides certificates of type `cert` that allow the frontend to produce certified results. Type `poly` is the internal representation of polyhedra used by the backend: it remains opaque for the frontend. The functions `isEmpty` and `isIncl` produce a certificate only when inclusion in $\bot$ or in another polyhedron holds. Other operations produce both polyhedra and certificates, except for renaming where a certificate is not needed.

The communication protocol between the backend and the frontend is detailed in next section. Section 5 describes the formalization in Coq of the backend functions. Last, section 6 describes how complex polyhedra operations are built from the low level interface.

4 Using certificates as build instructions

Three polyhedra operators use a certificate from the backend and produce a polyhedron: the guard, join and projection operators. What we have presented

```
Definition project (pF, pB) x :=
   let (pB', ce) := Backend.project pB x in
   let pF' := projectUsing ce pF x in
   (pF', pB')
```

Fig. 6. The implementation of the projection in the frontend

leads naturally to a pattern of algorithms for the frontend, which we illustrate for the projection operator. First, polyhedron $p_s \triangleq \Lambda.p$ is built, using the certificate Λ provided by the backend. Syntactic entailment is then checked with the result $p\backslash x$ actually provided by the backend. An extra check is specific to the projection: verifying that x is free in p.

Checking syntactic entailment is actually unnecessary: p' can be used as a result of the projection operator. It satisfies the inclusion property, by construction. On top of sparing the entailment check, this approach removes the need for the backend to communicate its result to the frontend. The certificate is sufficient. This remark applies to the projection operator, as well as to the guard and join operators.

As a result, the operators follow a simpler pattern, illustrated for the projection operator on figure 6. The backend and the frontend both have their own representation of a polyhedron, which we call `pB` and `pF`, respectively.

That `pB` and `pF` represent the same polyhedron is an invariant property. An operator of the abstract domain consists in invoking the corresponding operator of the backend, thereby obtaining the backend representation `pB'` for the resulting polyhedron. The backend also produces a certificate `ce`, from which the frontend computes its representation `pF'` of the result of the operator, along with a proof that it is correct. This restores the synchronisation between the frontend and backend: `pF'` and `pB'` represent the same polyhedron.

4.1 The impact of bugs

Previous discussion makes the assumption that all goes well: the certificate is well-formed and yields a representation of the result computed by the backend. However, bugs might lurk in the backend, leading to incorrect results or erroneous certificates. Two possible effects can be observed by the user of the abstract domain.

- If the certificate is well-formed but yields a result different from that of the backend, synchronization is lost and the results built by the abstract domain are likely to be wildly over-approximated, yet correct.
- If an ill-formed certificate (e.g. refers to nonexistent constraints) is output by the backend, the frontend will report a failure. Two failure modes are supported: abort or return a correct $\top$ result.

Unless the backend aborts, the frontend returns correct results in all cases: soundness bugs in the backend induce precision bugs of the abstract domain. These bugs are uncovered using standard software engineering methods.

4.2 The certificate language

```
Inductive cert :=
| Implies : list (positive * consCert) → cert
| Empty : linComb → cert
| Bind : positive → consCert → cert → cert.
```

Fig. 7. Coq definition of polyhedron build instructions

The frontend builds correct by construction results using certificates provided by the backend. The type `cert` of the certificates is given in figure 7. We will describe the design of the certificates from the ground up on the example of a projection $p\backslash x$ for which the backend has produced a certificate `Implies l`.

From a high level of abstraction, `Implies l` is the sparse representation of a matrix Λ which defines the result $p' \triangleq \Lambda.p$ of the projection. In order to make the certificate compact, the constraints of p are identified by positive numbers and the descriptions of linear combinations, the type `linComb`, refer to constraints by their identifier. Identifier generation is handled by the backend: the frontend requests `freshId pB` when it needs a constraint identifier that does not appear in polyhedron `pB`. The frontend does not check the freshness of identifiers: as described in §4.1, invalid identifiers may result in precision bugs.

Type `consCert` describes the various ways to build one constraint of p'_1. Its definition appears in figure 8. The `Direct` construct is the standard application of Farkas's lemma. For efficiency reasons, a backend may handle equality constraints specially, instead of representing them as pairs of inequalities. Two applications of Farkas's lemma are necessary to build an equality $\boldsymbol{a}.\boldsymbol{x} = b$ from $p \triangleq \bigwedge_i c_i$. One builds $\boldsymbol{a}.\boldsymbol{x} \leq b$ and the other builds $\boldsymbol{a}.\boldsymbol{x} \geq b$. The equality follows from their conjunction and we introduced the `SplitEq` construct to handle this case.

The join operator requires a special construct, `JoinCons`. For each constraint c of the result of $p_1 \sqcup p_2$, it must be shown that $p_1 \sqsubseteq c$ and $p_2 \sqsubseteq c$. To this end, a `JoinCons` certificate contains one linear combination to build $c_1 \triangleq \boldsymbol{a_1}.\boldsymbol{x} \leq b_1$ such that $p_1 \sqsubseteq c_1$ and another for $c_2 \triangleq \boldsymbol{a_2}.\boldsymbol{x} \leq b_2$ such that $p_2 \sqsubseteq c_2$. The frontend checks that $\boldsymbol{a_1} = \boldsymbol{a_2}$ and then chooses c_1 as the resulting constraint if $b_1 \geq b_2$, or c_2 otherwise. If $\boldsymbol{a_1} \neq \boldsymbol{a_2}$, the certificate is considered ill-formed.

Type `cert` also provides a construct to build $\bot$, as the result of a guard for example. An `Empty l` certificate is used for this purpose, where the linear combination `l` yields a trivially contradictory constraint, like $0 \leq -1$.

Let us motivate the last construct of type `cert` through a glimpse of the redundancy elimination behind a backend implementation of the guard $p \sqcap c$, with $p \triangleq \bigwedge_i c_i$. Constraint c is rewritten using the equality constraints in p, so as to lower the number of variables involved. The result c' could then be involved in proving that the system of inequalities $\bigwedge_i c_i \wedge c'$ hides an implicit equality e.

The new equality e could then used for further rewriting. Building a certificate in that setting is hard. The construct `Bind j cc ce` helps by introducing a new constraint resulting from the linear combination `cc` and giving it identifier `j`. The remainder of the inclusion certificate, `ce`, may then use it.

5 Formalizing the backend in Coq

Our abstract domain is split in two components: the frontend, which is developped in Coq, and the backend, which is written in OCaml. In order to execute the code, the Coq frontend must be extracted to OCaml code through Coq extraction mechanism. Extraction roughly consists in removing all the proof-related information from a Coq development, as OCaml type system is not powerful enough to represent it.

Once extracted, the frontend calls to the backend appear as function calls in the operators code. For the extraction to generate these calls, the backend functions must be declared to Coq as axioms. Let $\underline{f}$ be an external function of OCaml type $\underline{A} \to \underline{B}$. It is declared to Coq as a function f, of Coq type $A \to B$ and the extractor is instructed to replace calls to f with calls to $\underline{f}$. Types $\underline{A}$ and $\underline{B}$ must be the extracted versions of A and B. The OCaml compiler will report an error otherwise.

These declarations prevent the execution of the Coq development in Coq virtual machine: the body of the backend functions is not available to Coq. Furthermore, this process of linking certified Coq code to untrusted OCaml code may lead to a number of serious pitfalls.

Inconsistency. An axiom like `failwith` : $\forall B, \texttt{string} \to B$ introduces inconsistency as it builds a proof of any B from a `string`. In particular, `failwith False ""` gives a proof of `False`. This pitfall is avoided by providing a model of axioms in Coq: a proof that their Coq type is inhabited.

Implicit axioms. Inductive type B (e.g. $\{x : \mathbb{Z} \mid x < 5\}$) may be extracted into a strictly larger extracted type $\underline{B}$ (e.g. $\mathbb{Z}$). This introduces an implicit requirement on $\underline{f}$ (i.e. its results are lower than 5) that OCaml typechecker cannot ensure. For our frontend, we have thus carefully checked that Coq inductive types involved in backend functions are identical to their OCaml extraction.

```
Inductive consCert :=
| Direct : linComb → consCert
| SplitEq : linComb → linComb → consCert
| JoinCons : linComb → linComb → consCert.
```

Fig. 8. Coq definition of constraint build instructions

$$?A \triangleq S \to A \times S \qquad k_1 \equiv k_2 \triangleq \forall s, (k_1\ s) = (k_2\ s) \qquad k \rightsquigarrow a \triangleq \exists s, \mathrm{fst}\,(k\ s) = a$$

$$\texttt{unit}\ a \triangleq \lambda s.(a, s) \qquad \texttt{bind}\ k_1\ k_2 \triangleq \lambda s_0.\mathbf{let}\,(a, s_1) = (k_1\ s_0)\,\mathbf{in}\,(k_2\ a\ s_1)$$

Fig. 9. The state-transformer model of may-return monads

Memory corruption. Our backend uses the GMP [11] C library. A bug in GMP or its OCaml frontend, zarith [12], may corrupt arbitrary memory locations. However, it seems unlikely that such a bug breaks soundness silently.

Implicit purity axiom. Semantics of $\to$ are different in Coq and in OCaml. In Coq, f is implicitly a pure function: hence $\forall x, f\ x = f\ x$ is provable. On the contrary, $\underline{f}$ in OCaml may use an implicit state such that, for a given x two distinct calls $\underline{f}\ x$ give different results. In other word, axiomatizing f as $A \to B$ in Coq introduces an implicit functional requirement: $\underline{f}$ is observationally pure. Having an implicit state is allowed only if the effect of this implicit state remains hidden (e.g. for memoization). See [13] for details.

However, it may be difficult to ensure that a backend has no observable side effects. In ours, a bug in GMP or zarith may break this requirement. Furthermore, our proofs do not rely on the purity of backend functions. The following describes the theory of impure computations we have formalized in Coq in order to declare the backend functions as potentially impure. This theory is inspired by *simulable monads* [14], but from which we drop the notion of prophecy, because we are not interested in generating the backend from Coq.

5.1 *May-return* monads: a simple theory of impure computations

Impure computations are Coq computations that may use external computations in OCaml. For any Coq type A, we assume a type $?A$ to denote impure computations returning values of type A. Type transformer " $?.$ " is equipped with a monad:

- Operator $\texttt{bind}_{A,B} : ?A \to (A \to ?B) \to ?B$ encodes OCaml "$\mathbf{let}\ x = k_1\ \mathbf{in}\ k_2$" as "$\texttt{bind}\ k_1\ \lambda x.k_2$".
- Operator $\texttt{unit}_A : A \to ?A$ lifts a pure computation as an impure one.
- Relation $\equiv_A : ?A \to ?A \to$ Prop is a congruence (w.r.t. `bind`) which represents equivalence of semantics between OCaml computations. Moreover, operator `bind` is associative and admits `unit` as neutral element.

Last, we assume a relation $\rightsquigarrow_A : ?A \to A \to$ Prop and write "$k \rightsquigarrow a$" to denote the property that "computation k may return a". This relation is assumed to be compatible with $\equiv_A$ and to satisfy the following axioms:

$$\texttt{unit}\ a_1 \rightsquigarrow a_2 \Rightarrow a_1 = a_2 \qquad \texttt{bind}\ k_1\ k_2 \rightsquigarrow b \Rightarrow \exists a, k_1 \rightsquigarrow a \wedge k_2\ a \rightsquigarrow b$$

$$?A \triangleq A \qquad k_1 \equiv k_2 \triangleq k_1 = k_2 \qquad k \rightsquigarrow a \triangleq k = a \qquad \texttt{unit}\, a \triangleq a \qquad \texttt{bind}\, k_1\, k_2 \triangleq k_2\, k_1$$

Fig. 10. A trivial implementation of the may-return monad

The theory of may-return monads is a very abstract axiomatization of impurity: it does not provide any information about *effects* of impure computations. However, as our frontend only cares about *results* of backend functions, this theory suffices to our needs. Hence, backend functions $\underline{f}$ of type $\underline{A} \to \underline{B}$ are simply axiomatized in Coq as $f : A \to ?B$.

Our frontend is parameterized by an implementation of may-return monads: it does not depend on a particular model. Simple transformers over a global state have a denotation in the state monad defined in figure 9, using S as type of states. Even if building a model where any OCaml computation is denoted is complex [15] and beyond the scope of this work, this gives us confidence in our frontend being correct when used with a side-effecting backend.

5.2 Extraction of impure computations

The may-return monad is useful to control Coq assumptions that would otherwise be left implicit. However, it is of no other practical interest and is removed at extraction time by providing the trivial implementation given on figure 10. The extractor inlines these definitions so that the monad has no runtime overhead.

The trivial implementation of the may-return monad is also used to provide a pure Coq interface to our abstract domain, by exposing that $\rightsquigarrow$ is equality. Although this partly puts the backend in the trusted computing base (TCB), this was actually required to plug our library as an abstract domain of the analyzer developed as part of the Verasco project [16].

5.3 Backward reasoning on impure computations

Having introduced axioms for impure computations in section 5.1, we sketch below how we automate Coq reasonings about such computations, by using a weakest-precondition calculus programmed as a Ltac tactic.

First, we define in Coq an operator $\texttt{wlp}_A : ?A \to (A \to \text{Prop}) \to \text{Prop}$ such that $\texttt{wlp}\, k\, P \triangleq \forall a, k \rightsquigarrow a \Rightarrow (P\, a)$ expresses the weakest precondition ensuring that any result returned by computation k satisfies postcondition P.

For example, let us consider a Coq function g that first calls an external f returning a natural number of $\mathbb{N}$ and second, increments its result. We define $g\, x \triangleq \texttt{bind}\ (f\, x)\ \lambda n.(\texttt{unit}\ n{+}1)$ and express the property that "*g returns only strictly positive naturals*" as the goal "$\forall x, \texttt{wlp}\ (g\, x)\ \lambda n.n \neq 0$". Our Ltac tactic simplifies this goal into a trivial consequence of "$\forall n : \mathbb{N}, n{+}1 \neq 0$".

This tactic proceeds backward on `wlp`-goals, by applying repeatedly lemmas which are represented below as rules. It first tries to apply backward a *decomposition* rule: one for UNIT or BIND below, or one for pattern-matching over some usual types (booleans, option types, product types, etc.). When no decomposition applies, the tactic applies CUT&UNFOLD. Actually, it tries to discharge the left premise using existing lemmas; if this fails, the definition of `wlp` is simply unfolded; otherwise, the goal is replaced using the right premise: the unfolding is thus performed with a lemma injection in hypothesis.

$$\textsc{Decomp-unit}\ \frac{P\,a}{\texttt{wlp}\ (\texttt{unit}\ a)\ P} \qquad \textsc{Decomp-bind}\ \frac{\texttt{wlp}\ k_1\ \lambda a.(\texttt{wlp}\ (k_2\,a)\ P)}{\texttt{wlp}\ (\texttt{bind}\ k_1\,k_2)\ P}$$

$$\textsc{Cut\&Unfold}\ \frac{\texttt{wlp}\ k\ P_1 \qquad \forall a, k \rightsquigarrow a \wedge P_1\ a \Rightarrow P_2\ a}{\texttt{wlp}\ k\ P_2}$$

In our COQ development, this tactic automates most of the bureaucratic reasoning on first-order impure computations. For higher-order impure computations (e.g. invoking a list iterator), equational reasoning is also needed.

6 Modular construction of the abstract domain

We have described in the last two sections a basic interface to the abstract domain of polyhedra. It is a restricted version of the interface described in section 2: the forward predicate transformer for assignment is missing, for example. The gap between the low level interface, closer to what the backend provides, and the fully-fledged interface, that our abstract domain offers to the user, is bridged entirely in the COQ frontend. The extra functionality is provided through the use of functors. Each functor takes an abstract domain and builds a richer one while lifting the proofs as necessary. This decomposition makes the proofs more manageable and modular.

The overall architecture of the abstract domain is pictured on figure 11. The shaded left-hand side is the COQ frontend. Each of the pictured layers represents a functor. The untrusted backend stands on the right-hand side. While communication between the two is represented by arrows, it reduces to function calls in the extracted frontend code.

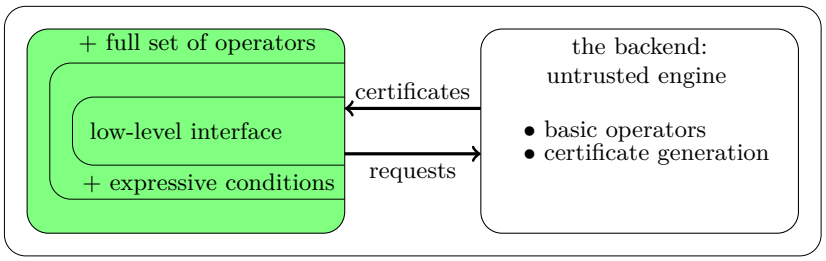

Fig. 11. Overview of the domain architecture

6.1 Building the guard operator

A first example of our modular construction of some abstract domain features is the guard operator presented in section 3. This operator $p \sqcap c$ accepts an arbitrary propositional formula as constraint c. However, the backend guard operator takes only constraints of the form $t \bowtie 0$, with $\bowtie \in \{=, \leq, <\}$ and t an affine term. The transformation from the more expressive guards to the basic guards is performed by the frontend by the following steps:

1. Negations are eliminated using De Morgan's laws on binary operators to push negation inwards, eliminating double negations, and taking the dual comparison on atomic formula.
2. Comparison $\neq$ is rewritten as a disjunction of strict inequalities.
3. On $\mathbb{Z}$, $t_1 < t_2$ is rewritten as $t_1 - t_2 + 1 \leq 0$. This increases precision of our polyhedra computations where all variables are in $\mathbb{Q}$.
4. Disjunctions are over-approximated by joins.

For a guard $p \sqcap c$, our algorithm performs a number of polyhedra operations that is linear in the number of operations in c. The functor which provides this extended guard operator to an abstract domain featuring only a basic one also contains the proof that the algorithm described above is sound.

6.2 Framing potentially constrained variables

Generating fresh variables has many applications for program verification: handling local variables, parameter passing during function inlining, implementing the forward predicate transformer for assignment, etc. However, our Coq specifications of abstract domains do not provide any information about the set of variables constrained by a polyhedron. Indeed, these loose specifications allow a modular management of fresh variables: in particular, certification of low-level operations presented at section 3 is not intricated with fresh variables handling.

Hence, this section and the next one introduce generic abstract domains (functors) gluing additional data about constrained variables to the value of an underlying abstract domain. We certify these functors by expressing the invariant of theses additional data through the *concretization function* of the newly introduced domain. We now illustrate the necessity of this trick on a functor that simply frames the variables constrained by an abstract value (e.g. a polyhedron).

Formally, if F is a set of variables, we note $m \equiv_F m'$ if and only if memories m and m' coincide on F. Then, given a polyhedron p, we say that F *frames* p if and only if $\forall m_1 \forall m_2,\ m_1 \equiv_F m_2 \ \Rightarrow\ (\llbracket p \rrbracket m_1 \Leftrightarrow \llbracket p \rrbracket m_2)$, and that x *is free in* p if and only if $\{x' \mid x' \neq x\}$ frames p. An operator $\mathrm{bnd}(p)$ can then be defined such that $\{x \mid x \leq \mathrm{bnd}(p)\}$ frames p (variables are represented by positive integers). This operator returns an upper bound β on the variables constrained by p: we say that β *bounds* p. These definitions also apply to conditions and terms.

Operator bnd is provided by a new abstract domain $\mathbb{P}^{\mathrm{bnd}}$ wrapping each element p of the underlying domain into a pair (p, β) such that β bounds (p, β). Operations of $\mathbb{P}^{\mathrm{bnd}}$ are given figure 12.

Naive definitions of $\mathbb{P}^{\text{bnd}}$ fail to make provable the correctness of operations: property "$\forall (p,\beta) \in \mathbb{P}^{\text{bnd}}, \beta$ bounds p" may not be preserved by the operations of figure 12. For example, let us consider $(p_1,\beta_1) \sqcup^{\text{bnd}} (p_2,\beta_2)$. We expect $(p_1 \sqcup p_2, \max(\beta_1,\beta_2))$ to be a good candidate implementation, since if β_1 bounds p_1 and β_2 bounds p_2, then $\max(\beta_1,\beta_2)$ bounds both p_1 and p_2. However, $\max(\beta_1,\beta_2)$ may not bound $p_1 \sqcup p_2$, as this somewhat contorted, yet correct, implementation of $p_1 \sqcup p_2$ shows when x is chosen above the bound 1:

$$p_1 \sqcup p_2 \triangleq \begin{cases} x \leq 0 & \text{if } p_1 = p_2 = \bot \\ \top & \text{otherwise} \end{cases}$$

Our solution consists in keeping the definitions given in figure 12, but changing that of $[\![(p,\beta)]\!]$ so that it implies the property "β bounds (p,β)". Given a concrete memory m, we impose that variables above β are free in $[\![(p,\beta)]\!]m$ by quantifying over any abstract memory $m_\sharp$ that results from the arbitrary update of m on these variables:

$$[\![(p,\beta)]\!]m \triangleq \forall m_\sharp,\ m_\sharp \equiv_{\{\, x \mid x \leq \beta \,\}} m \ \Rightarrow\ [\![p]\!]m_\sharp$$

6.3 Assignment with buffered renaming

The $\mathbb{P}^{\text{bnd}}$ functor can be used to over-approximate the forward predicate transformer of assignment. Indeed, it allows to introduce an auxiliary fresh variable x_0 which names the value of variable x before the assignment:

$$p[x := t] \triangleq (p[x \leftarrow x_0] \sqcap x = t[x \leftarrow x_0]) \setminus x_0 \quad \text{where} \quad x_0 \triangleq \max(\text{bnd}(t), \text{bnd}(p)) + 1$$

However, our abstract domain uses the $\mathbb{P}^{:=}$ functor described below instead, because it performs a lower *amortized number of polyhedra renamings*.

Functor $\mathbb{P}^{:=}$ makes it possible to express relations between memory states in the intermediary computations of the operators. This achieved by duplicating the set of variable names: each variable x can be represented as $x@0$ or $x@1$. Of these two representatives of x, the concretization imposes that exactly one refers to a concrete memory cell. Similarly to what is done in $\mathbb{P}^{\text{bnd}}$, the other representative is arbitrarily updated in abstract memory $m_\sharp$. The concretization involves a function σ that associates its current representative to each variable. It also

$$\text{bnd}((p,\beta)) \triangleq \beta \qquad \top^{\text{bnd}} \triangleq (\top, 1) \qquad \bot^{\text{bnd}} \triangleq (\bot, 1)$$

$$(p_1,\beta_1) \sqcup^{\text{bnd}} (p_2,\beta_2) \triangleq (p_1 \sqcup p_2, \max(\beta_1,\beta_2))$$

$$(p,\beta) \sqcap^{\text{bnd}} c \triangleq (p \sqcap c, \max(\beta, \text{bnd}(c))) \qquad (p_1,\beta_1) \sqsubseteq^{\text{bnd}} (p_2,\beta_2) \triangleq p_1 \sqsubseteq p_2 \wedge \beta_1 \leq \beta_2$$

Fig. 12. Main operators of $\mathbb{P}^{\text{bnd}}$

involves a function π that associates concrete x to both abstract variables $x@0$ and $x@1$, for all x.

$$[\![(p,\sigma)]\!]m \triangleq \forall m_\sharp,\ m_\sharp \equiv_{\{\, x_\sharp \mid x_\sharp = \sigma(\pi(x_\sharp))\,\}} m \circ \pi \ \Rightarrow\ [\![p]\!]m_\sharp$$

In the $\mathbb{P}^{:=}$ functor, assignment to x switches the representative of x, instead of renaming the variable in the underlying polyhedron as with $\mathbb{P}^{\mathrm{bnd}}$. Renamings from assignments are buffered until joins or inclusions, where they may eventually be performed (only when representatives of identical variables need to be unified). Furthermore, two successive renamings on the same variable in the buffer annihilate (by involution of representative switch).

This functor could be extended so as to buffer projections, which can then be reordered to get smaller intermediate results (in terms of size of representation). The decision to apply projections is delegated to the backend. In this version, the functor introduces a unique representative at each assignment: a kind of SSA form is thus computed on-the-fly in the abstract domain. This extension is not implemented yet.

In conclusion, our modular treatments of assignment depart from [7]: this results in more manageable proofs. In [7], projections are systematically delayed until inclusion tests: we believe that the choice of when to apply projections should be delegated to the backend.

7 Conclusion

We presented one solution to prove the correctness of an implementation of the abstract domain of polyhedra using the Coq proof assistant. In this setting, correctness reduces to inclusions of polyhedra which, through Farkas's lemma, makes a posteriori verification of results a convenient approach. As a result, our domain is composed of an untrusted backend, to which most of the complex computations are offloaded, and a Coq frontend which validates the results produced by the backend. This work makes two main contributions.

On one hand, we consider the implicit requirements set when linking certified code to untrusted external code in the Coq proof assistant. This delicate issue might be carefully considered by certification authorities for e.g. avionics. We partly address it through a lightweight method for declaring the backend functions to Coq in such a way that the proofs remain trustworthy even when the backend is not functionally pure.

On the other hand, we show how communication between the frontend and the backend can be reduced to certificates, which serve as build instructions for the frontend. The certificate language induces a low coupling between the frontend and the backend: the latter could implement relaxations of some operators [9,10] or use entirely different data structures without requiring changes to the frontend. Although it does not make abstract domain development easier, our approach reduces the impact of bugs.

The complete domain further distinguishes itself from previous work by integrating certificate generation to the backend and by a more modular proof

architecture. Experiments shows that it has comparable performance to non-certifying state-of-the-art implementations [8].

The complete code is available on the Web, along with a demonstration application, from www-verimag.imag.fr/~boulme/vstte2014.html.

Acknowledgements. We would like to thank Michaël Périn and David Monniaux for their continuous feedback all along this work. We also thank the members of the Verasco project for their motivating interaction.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, ACM (2003)
2. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM (1977)
3. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, ACM (1978)
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1–2) (2008)
6. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. (2009)
7. Besson, F., Jensen, T., Pichardie, D., Turpin, T.: Result certification for relational program analysis. Technical Report RR-6333, INRIA (2007)
8. Fouilhe, A., Monniaux, D., Périn, M.: Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In: SAS. Volume 7935., Springer (2013)
9. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: VMCAI. Volume 5403 of LNCS., Springer (2009) 229–244
10. Sankaranarayanan, S., Colón, M., Sipma, H., Manna, S.: Efficient strongly relational polyhedral analysis. In: VMCAI. Volume 3855 of LNCS., Springer (2006) 111–125
11. Free Software Foundation: The GNU Multiple Precision Arithmetic Library. 5.0 edn. (2012)
12. Miné, A., Leroy, X.: ZArith. http://forge.ocamlcore.org/projects/zarith
13. Pottier, F.: Syntactic soundness proof of a type-and-capability system with hidden state. JFP **23**(1) (January 2013)
14. Claret, G., Gonzalez Huesca, L.D.C., Régis-Gianas, Y., Ziliani, B.: Lightweight proof by reflection using a posteriori simulation of effectful computation. In: ITP. (July 2013)
15. Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., Yang, H.: Step-indexed kripke models over recursive worlds. In: POPL, ACM (2011)
16. Blazy, S., Laporte, V., Maroneze, A., Pichardie, D.: Formal Verification of a C Value Analysis Based on Abstract Interpretation. In: Static Analysis Symposium (SAS). Volume 7935 of LNCS., Springer (2013)