Refinement to Certify Abstract Interpretations, Illustrated on Linearization for Polyhedra

Sylvain Boulmé, Alexandre Maréchal

Refinement to Certify Abstract Interpretations
Illustrated on Linearization for Polyhedra

Sylvain Boulmé and Alexandre Maréchal
Univ. Grenoble-Alpes, VERIMAG, F-38000 Grenoble, France
{sylvain.boulme,alex.marechal}@imag.fr

September 2018

Abstract

Our concern is the modular development of a certified static analyzer in the Coq proof assistant. We focus on the extension of the Verified Polyhedra Library – a certified abstract domain of convex polyhedra – with a linearization procedure to handle polynomial guards. Based on ring rewriting strategies and interval arithmetic, this procedure partitions the variable space to infer precise affine terms which over-approximate polynomials.

In order to help formal development, we propose a proof framework, embedded in Coq, that implements a refinement calculus. It is dedicated to the certification of parts of the analyzer – like our linearization procedure – whose correctness does not depend on the implementation of the underlying certified abstract domain. Like standard refinement calculi, it introduces data-refinement diagrams. These diagrams relate "*abstract states*" computed by the analyzer to "*concrete states*" of the input program. However, our notions of "*specification*" and "*implementation*" are exchanged *w.r.t.* standard uses: the "*specification*" (computing on "*concrete states*") refines the "*implementation*" (computing on "*abstract states*").

Our stepwise refinements of *specifications* hide several low-level aspects of the computations on abstract domains. In particular, they ignore that the latter may use hints from external untrusted imperative oracles (*e.g.* a linear programming solver). Moreover, refinement proofs are naturally simplified thanks to computations of weakest preconditions. Using our refinement calculus, we elegantly define our partitioning procedure with a continuation-passing style, thus avoiding an explicit datatype of partitions. This illustrates that our framework is convenient to prove the correctness of such higher-order imperative computations on abstract domains.

Keywords: Proof Assistants, Result Certification, Abstract Interpretation.

Acknowledgements This work was partially supported by French Agence Nationale de la Recherche under the VERASCO project (INS 2011) and by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".
We thank Alexis Fouilhé, Michaël Périn, David Monniaux and the other members of the Verasco project for their continuous feedback all along this work.

1 Introduction

This paper presents two contributions: first, a certified linearization procedure for a certified abstract domain of convex polyhedra; second, a refinement calculus to help in mechanizing this proof in Coq [31]. We detail below the context and features of these two contributions.

1.1 A Certified Linearization for the Abstract Domain of Polyhedra

We consider the certification of an *abstract interpreter*, which aims at ensuring absence of undefined behaviors such as division by zero or invalid memory access in an input source program. This analyzer computes for each program point an *invariant*: a property that must hold in all executions at that point. Such invariants belong to datatypes called *abstract domains* [9] which are syntactic classes of properties on memory states. For instance, in the abstract domain of *convex polyhedra* [10], invariants are conjunctions of affine constraints written $\sum_i a_i x_i \leq b$ where $a_i, b \in \mathbb{Q}$ are scalar values and x_i are integer program variables. This domain is able to capture relations between program variables (*e.g.* $x + 2 \leq y + x - 2z$). However, it cannot deal directly with non-linear invariants, such as $x^2 - y^2 \leq x \times y$. This is why linearization techniques are necessary to analyze programs with non-linear arithmetic.

Our certified linearization procedure is based on intervalization [26]. It consists in replacing some variables of nonlinear products by intervals of constants. For instance, Example 1 replaces variable x by interval $[0, 10]$ in product "$x.(y - z)$". The interval is then eliminated by analyzing the sign of $y - z$, leading to affine constraints usable by the polyhedra domain.

Example 1 (Intervalization using a sign-analysis) In any state where $x \in [0, 10]$, assignment "$r := x.(y - z) + 10.z$" is approximated by the affine program below. Here operator $:\in$ performs a non-deterministic assignment.

```
if y − z ≥ 0 then r :∈ [10.z, 10.y] else r :∈ [10.y, 10.z]
```

In other words, r is updated to any value of $[\min(10.y, 10.z), \max(10.y, 10.z)]$.

Let us clearly delimit the scope of our work. Our linearization procedure is part of the Verified Polyhedra Library (VPL) [13, 14, 22], which provides a certified polyhedra domain to Verasco [18, 19, 17], a certified abstract interpreter for CompCert C [20]. Verasco is a static analyzer that checks that C programs have no undefined behavior. Hence, our refinement calculus focuses on abstract interpretations that overapproximate sets of reachable states and that reject reachable error states. For example, our refinement calculus cannot prove the correctness of an abstract interpretation bounding execution times of programs. Second, the VPL abstract domain in Verasco is limited to integer variables and rational constants. It could also support rational variables. But supporting floating-point operators would be a non-trivial extension.

Following a design proposed in [2], the VPL is organized as a two-tier architecture: an untrusted oracle, combining OCaml and C code, performs most complex computations and outputs a Farkas certificate used by a certified front-end to build a correct-by-construction result. As oracles may have side-effects and bugs, they are viewed in Coq as non-deterministic computations of an axiomatized monad [13].

Built on a similar design, our linearization procedure invokes an untrusted oracle[1] that selects strategies for linearizing an arithmetic expression and produces a certificate that is checked by the certified part of the procedure. It leads to a correct-by-construction over-approximation of the expression. It is convenient to see such strategies as program transformations, because their correctness is independent from the implementation of the underlying abstract domain and is naturally expressed using concrete semantics of programs. Indeed, a linearization is correct if,

[1] There are several kinds of oracles in the VPL: those based on Farkas certificate for basic polyhedra computations; the linearization strategy in the linearization procedure; etc. In the Coq code, each of these oracles is declared as a "non-deterministic" function in parameter of the code (through an axiom). Here, "non-determinism" is formalized by requiring the results of such functions to inhabit a may-return monad. Section 4.1 recalls the axioms of may-return monads, initially proposed in [13].

in the current context of the analysis, any postcondition satisfied by the output program is also satisfied by the input one (see Example 1). In such a case, we say that the input program *refines* the output one. This paper aims to explain how refinement helps to develop certified procedures on abstract domains, and in particular our linearization algorithm.

1.2 Refinement to Certify Computations on Abstract Domains

Program refinement [1, 27] consists in decomposing proofs of complex programs by stepwise applications of correctness-preserving transformations. We provide a framework in Coq to apply this methodology for certifying the correctness of computations combining operators of an existing abstract domain: our goal is to compositionally build correct-by-construction abstract computations, by reasoning only on the concrete semantics of programs.

Typically, an affine program – like in Example 1 – is both interpreted in our abstract and concrete semantics. We thus reduce the proof that the abstract interpretation of this affine program computes a correct overapproximation of the input program to the proof that its concrete interpretation refines the input program. This proof may be itself composed of several stepwise refinements in the concrete semantics. Indeed, the development of our linearization procedure extends concrete semantics with *affine interval arithmetic* [26] (*i.e.* affine arithmetic where constants are replaced by intervals of constants). In this approach, refinement of Example 1 is decomposed into two refinement steps given in Example 2. Here, assumption $x \in [0, 10]$ is reflected in the input program syntax thanks to an `assume` command (formally defined in Section 2.1).

Example 2 (Refinement steps) The affine program

$\texttt{if}\ \ y - z \geq 0\ \ \texttt{then}\ \ r :\in [0 + 10.z, 10.(y - z) + 10.z]\ \ \texttt{else}\ \ r :\in [10.(y - z) + 10.z, 0 + 10.z]$

is refined by

$r :\in [0, 10].(y - z) + 10.z$

itself refined by

$\texttt{assume}\ \ x \in [0, 10]\,;$ $r := x.(y - z) + 10.z$

On Example 2, the first refinement step reduces to two properties of interval multiplication

$$\begin{array}{rcl} y - z \geq 0 & \Rightarrow & [0, 10].(y - z) = [0, 10.(y - z)] \\ y - z < 0 & \Rightarrow & [0, 10].(y - z) = [10.(y - z), 0] \end{array}$$

The program in the middle just aims at simplifying proofs. Indeed, the second refinement step reduces to

$$x \in [0, 10] \ \Rightarrow\ x.(y - z) + 10.z \in [0, 10].(y - z) + 10.z$$

This property trivally results from composition properties of interval arithmetic operators. Thus, this whole proof completely ignores that our abstract intepretation of the first program involves imperative computations using a given representation of polyhedra.

1.3 Overview of our Refinement Calculus

Our framework defines a Guarded Command Language (GCL) called $^{\dagger}\mathbb{K}$ that contains the basic operators of the abstract domain. A computation $^{\dagger}K$ in $^{\dagger}\mathbb{K}$ comes with two types of semantics: an abstract and a concrete one. Concrete semantics of $^{\dagger}K$ is a transformation on *memory states*. Abstract semantics of $^{\dagger}K$ is a transformation on *abstract states*, *i.e.* on values of the abstract domain. A $^{\dagger}\mathbb{K}$ computation also embeds a proof that abstract semantics is correct *w.r.t.* concrete one: each $^{\dagger}\mathbb{K}$ operator thus preserves correctness by definition. Moreover, an Ocaml function is extracted from abstract semantics, which is certified to be correct *w.r.t.* concrete semantics. Hence,

concrete semantics of $^\dagger K$ acts as a *specification* which is *implemented* by its abstract semantics. In the following, a transformation on abstract (resp. memory) states is called an abstract (resp. concrete) computation.

Taking a piece of code as input, our linearization procedure outputs a $^\dagger\mathbb{K}$ computation. Its correctness is ensured by proving that concrete semantics of its input refines concrete semantics of its output. This means that the output does not forget any behaviour of the input. Our procedure being developed in a modular way from small intermediate functions, its proof reduces itself to small refinement steps.[2] Each of these refinement steps is only proved by reasoning on the concrete semantics. Our framework provides a tactic simplifying such refinement proofs by computational reflection of weakest-preconditions. The correctness of abstract semantics *w.r.t.* concrete semantics is ensured by construction of $^\dagger\mathbb{K}$ operators.

Our framework supports *impure* abstract computations, *i.e.* abstract computations that invoke imperative oracles giving them hints to build their certified results. It also allows to reason conveniently about higher-order abstract computations. In particular, our linearization procedure uses a Continuation-Passing-Style (CPS) [29] in order to partition its analyzes according to the sign of affine sub-expressions. For instance in Example 2, the approximation of the non-linear assignment depends on the sign of $y - z$. In our procedure, CPS is a higher-order programming style that avoids introducing an explicit datatype handling partitions: this simplifies both the implementation and its proof. This also illustrates the expressive power of our framework, since a simple Hoare logic does not suffice to reason about such higher-order imperative programs.

Our refinement calculus could have applications beyond the correctness of linearization strategies: it could be applied for any part of the analyzer that combines computations of existing abstract domains. In particular, the top-level interpreter of the analyzer could also be proved correct in this way. Indeed, the interpreter invokes operations on abstract domains in order to over-approximate any execution of the program, but its correctness does not depend on abstract domains implementations (as soon as these implementations are themselves correct). We illustrate this claim on a toy analyzer, also implemented in Coq. Let us explain this contribution *w.r.t.* the certification of the top-level interpreter of Verasco developed by Jacques-Henri Jourdan [17].

The interpreter of Verasco analyzes `C♯minor` [20] – an intermediate structured language of CompCert frontend [20] – *w.r.t.* its small-step semantics. Actually, this small-step semantics (from CompCert) introduces many low-level details that are tedious to deal with in the proof of the analyzer.[3] Thus, Jourdan has introduced a higher-level semantics of `C♯minor` in order to simplify his proof. This semantics is a Hoare logic because such a logic is better suited to *structured* languages than usual collecting semantics which are dedicated to *Control Flow Graph* representations [17]. Hence, Jourdan's proof is realized in a framework combining a Hoare logic as concrete semantics, with a theory of abstract domains. But Jourdan's framework assumes that operators of abstract domains are pure functions. Actually, this is not the case of VPL operators.[4]

Our refinement calculus sketches an alternative to Jourdan's framework in order to support *impure* operators in abstract domains. Our toy analyzer illustrates how the refinement calculus helps to mechanize the correctness proof of the interpreter. Moreover, it also illustrates that alarm handling of Verasco is very easy to support in our framework. However, our interpreter does not support many other features of Verasco interpreter: control-flow statements such as "`break`" and "`continue`", the inference mechanism of loop invariants[5], communication between several

[2]Thus, we do not use our refinement calculus in a *decompositional* (*i.e.* "top-down") approach, that builds an implementation by stepwise derivation from a specification. On the contrary, we use our refinement calculus in a *compositional* (*i.e.* "bottom-up") approach, that builds larger "bricks" from smaller "bricks".

[3]Typically, `C♯minor` small-step semantics distinguishes infinite loops depending on whether they invoke system calls or not. But, by definition, an infinite loop cannot have runtime errors. Hence, all infinite loops are equivalent for Verasco analyzer. Even better, the analyzer can safely prune any control-flow branch where they appear, exactly like unreachable code.

[4]Thus, the embedding of VPL in Verasco coerces its imperative operators into pure ones. Logically, this coercion remains to assume that VPL oracles are observationally pure. This is potentially wrong, because of potential bugs in these untrusted oracles [13].

[5]Our toy analyzer does not infer loop invariants but requires them from the user. It does not seem too hard to extend our analyzer with inference of loop invariants since the VPL provides a standard (untrusted) widening operator. But, this feature is quite orthogonal to the certification of the analyzer itself. For example, Laporte [19]

abstract domains, etc.

1.4 Comparison with Related Works

The mathematics involved in our refinement calculus, relating operational semantics to the lattice structure of monotone predicate transformers, are well-known in abstract interpretation theory [8]. In parallel to our work, the idea to use a refinement calculus in formal proofs of abstract interpreters was proposed in [30]. Therefore, our contribution is more practical than theoretical. On the theoretical side, we propose a refinement calculus dedicated to the certification of *impure* abstract computations (*w.r.t.* big-step operational semantics). On the practical side, we show how to get a concise implementation of such a refinement in Coq and how it helps on a realistic case study: a linearization technique inspired from [26] within the abstract interpreter Verasco.

There are alternatives to our approach for computing polyhedral approximations of semi-algebraic sets. Let us briefly compare them with intervalization. A linearization procedure based on Handelman representation of polynomials [16] has also been implemented in the VPL [23]. It is more precise than intervalization, but at a high cost: it requires solving costly parametric linear problems. Albeit powerful, Handelman's linearization does not scale properly to large polynomials and polyhedra, this is why we need a cheaper algorithm such as intervalization. Another precise approach consists in converting the polynomial into Bernstein's basis and extract the generators of the resulting polyhedron from the polynomial's coefficients [12]. Like Handelman's linearization, it offers a tunable precision: either by partitioning the variable space or by elevating the degree of the Bernstein's basis considered. However, in order to ease the certification, the VPL uses a constraint-only representation of polyhedra. Using Bernstein's linearization would thus involves costly conversions from constraints into generators, and backwards [24].

There are also linearizations dedicated to other target domains. For instance, a decision procedure for arithmetic that uses affine forms instead of polyhedra has been proven in PVS [28]. In their approach, affine approximations of polynomials are combined with partitioning through a branch-and-bound algorithm. The expressiveness of affine forms is strictly between intervals and polyhedra, but our linearization procedure would probably be greatly improved by incorporating their techniques.

1.5 Overview of the Paper

Our refinement calculus is implemented in only 350 lines of Coq (proof scripts included), by a shallow-embedding of our GCL $\mathbb{tK}$ which combines computational reflection of weakest-preconditions [11] with monads [32]. However, it can be understood in a much simpler setting using binary relations instead of monads and weakest-preconditions, and classical set theory instead of Coq.

Section 2 introduces our refinement calculus in this simplified setting, where computations are represented as binary relations. Section 3 presents our certified linearization procedure and how its proof benefits from our refinement calculus. Section 4 explains how we mechanize this refinement calculus in Coq by using smart encodings of binary relations introduced in Section 2.

This paper is intended to be self-contained. Assuming that the reader is familiar with higher-order logic, big-step semantics and Hoare logic, it attempts to introduce as simply as possible all other notions: refinement, abstract interpretation, convex polyhedra, monads and weakest-preconditions. A less detailed version of this paper has been published in [5]. Our Coq sources are available as a standalone library:

- either at `http://www-verimag.imag.fr/~boulme/vpl201503` (first release)
- or at `http://github.com/VERIMAG-Polyhedra/VPL` (current release)

The version integrated with Verasco 1.3 is available at
`http://compcert.inria.fr/verasco/release/verasco-1.3.tgz`

shows how to program such an untrusted oracle, in order to produce invariants checked by the certified analyzer.

2 A Refinement Calculus for Abstract Interpretation

We consider an analyzer correct if and only if it rejects all programs that may lead to an *error state*. Due to lack of precision, it may also reject safe programs. Section 2.1 defines the notion of error state and semantics of concrete computations, which combines big-steps operational semantics with Hoare Logic. After introducing the notion of abstract computation and its correctness *w.r.t.* a concrete computation, Section 2.2 presents our refinement calculus. Section 2.3 shows how to apply refinement to the certification of higher-order abstract computations.

Notations on Relations. Although our formalization is in the intuitionistic type theory of Coq without axioms, the paper abusively uses more common notations of classical set theory. In particular, we identify the type $A \to \texttt{Prop}$ of predicates on A with the set $\mathcal{P}(A)$. Hence, we define the set of binary relations on $A \times B$ by $\mathcal{R}(A,B) \triangleq \mathcal{P}(A \times B)$. Given R of $\mathcal{R}(A,B)$, we note $x \xrightarrow{R} y$ instead of $(x,y) \in R$. We use operators on $\mathcal{R}(A,A)$ inspired from regular expressions: ε is *the identity relation* on A, $R_1 \cdot R_2$ means "*relation* R_2 *composed with* R_1" (i.e. $x \xrightarrow{R_1 \cdot R_2} z \triangleq \exists y, x \xrightarrow{R_1} y \wedge y \xrightarrow{R_2} z$) and R^* is the reflexive and transitive closure of R. Through all the paper, $A \to B$ is a type of *total* functions.

2.1 Stepwise Refinement of Concrete Computations

Given a domain D representing the type of memory states, we add a distinguished element $\lightning$ to D in order to represent the error state: we define $D_\lightning \triangleq D \uplus \{\lightning\}$.

Concrete Computations With Runtime Errors. We define the set of computations on memory states, called here *concrete computations*, as $\mathbb{K} \triangleq \mathcal{R}(D, D_\lightning)$. Hence, an element K of $\mathbb{K}$ corresponds to a (possibly) non-deterministic or non-terminating computation from an *input state* of type D to an *output state* of type $D_\lightning$. Typically, the empty relation represents a computation that loops infinitely for any input. It also represents unreachable code *i.e.* dead code (as explained in Footnote 3).

In the following, an input $d \in D$ is said to be *erroneous for a concrete computation* K if and only if $d \xrightarrow{K} \lightning$. Informally speaking, we consider that an abstract computation is correct *w.r.t.* a concrete computation K at two conditions: first, it overapproximates the set of erroneous inputs of K as a set E; second, for each input of $D \backslash E$, it overapproximates the set of its related outputs through K. Section 2.2 formalizes this notion of abstract computation. Before that, we introduce structures on concrete computations in order to use them as *specifications* of abstract computations.

Refinement Pre-order. Given K_1 and K_2 in $\mathbb{K}$, we say that "K_1 *refines* K_2" (written $K_1 \sqsubseteq K_2$) if, informally, each abstract computation correct for K_2 is also correct for K_1. Let us now formalize this refinement relation.

First, we introduce $\downarrow K$ the normalization of K that returns any output for erroneous inputs. It is defined by $d \xrightarrow{\downarrow K} d' \triangleq (d \xrightarrow{K} d' \vee d \xrightarrow{K} \lightning)$. Informally speaking, "adding" some outputs to K on its erroneous inputs does not change the set of abstract computations that are correct *w.r.t.* K. In other words, an abstract computation is correct for K *if and only if* it is correct *w.r.t.* $\downarrow K$. Moreover, $\downarrow K$ is the *maximal relation* which is equivalent to K *w.r.t.* (correct) abstract interpretation.

Then, normalization enables us to define refinement from inclusion. Formally, we define $K_1 \sqsubseteq K_2$ as $K_1 \subseteq \downarrow K_2$ (or equivalently, $\downarrow K_1 \subseteq \downarrow K_2$). Relation $\sqsubseteq$ is called *refinement* and is a pre-order on $\mathbb{K}$. The equivalence relation $\equiv$ associated with this pre-order is given by $K_1 \equiv K_2$ iff $\downarrow K_1 = \downarrow K_2$.

Hoare Specifications. Hoare logic is a standard and convenient framework to reason about imperative programs. Let us explain how computations in $\mathbb{K}$ are equivalent to specifications of Hoare logic. A computation K corresponds to a Hoare specification $(\mathfrak{p}_K, \mathfrak{q}_K)$ of $\mathcal{P}(D) \times \mathcal{R}(D,D)$,

where $\mathfrak{p}_K$ is a precondition ensuring the absence of error, and $\mathfrak{q}_K$ a postcondition relating the input state to a non-error output state[6]. They are defined by $\mathfrak{p}_K \triangleq D \setminus \{d \mid d \xrightarrow{K} \lightning\}$ and $\mathfrak{q}_K \triangleq K \cap (D \times D)$. Conversely, any Hoare specification (P, Q) corresponds to a computation $\vdash P; Q$ – defined below – such that $K \equiv \vdash \mathfrak{p}_K; \mathfrak{q}_K$. Moreover, the refinement pre-order $K_1 \sqsubseteq K_2$ is equivalent to the usual refinement of specifications in Hoare logic, which is $\mathfrak{p}_{K_2} \subseteq \mathfrak{p}_{K_1} \ \wedge \ \mathfrak{q}_{K_1} \cap (\mathfrak{p}_{K_2} \times D) \subseteq \mathfrak{q}_{K_2}$.

Algebra of Guarded Commands. Initially proposed by [11], guarded commands are also equivalent to Hoare specifications, but with an algebraic style, more suited for the methodology of stepwise refinement[1]. Inspired by this methodology, we equip $\mathbb{K}$ with an algebra of guarded commands.[7] It combines a *complete* lattice structure with operators inspired from regular expressions. Here, we present this algebra in our simplified setting, where $\mathbb{K}$ is defined as $\mathcal{R}(D, D_\lightning)$. Our Coq implementation, described in Section 4.2, has a different representation of $\mathbb{K}$ in order to mechanize refinement proofs.

First, the complete lattice structure of $\mathbb{K}$ (for pre-order $\sqsubseteq$) is given by operator $\sqcap$ defined as "$\cap$ after normalization" (*i.e.* $\bigsqcap_i K_i \triangleq \bigcap_i \downarrow K_i$) and by operator $\sqcup$ defined as $\cup$. In our context, $\sqcup$ represents alternatives that may non-deterministically happen at runtime: the analyzer must consider that each of them may happen. Symmetrically, $\sqcap$ represents some choice left to the analyzer. The empty relation $\emptyset$ is the bottom element and is written $\bot$. The relation $D \times \{\lightning\}$ is the top element. Given $d \in D_\lightning$, we implicitly coerce it as the constant relation $D \times \{d\}$. Hence, the top element of the $\mathbb{K}$ lattice is simply written $\lightning$. The notation $\uparrow f$ explicitly lifts function f from $D \to D$ to $\mathbb{K}$.

Given a relation $K \in \mathcal{R}(D, D_\lightning)$, we define its lifting $\lceil K$ in $\mathcal{R}(D_\lightning, D_\lightning)$ by $\lceil K \triangleq K \cup \{(\lightning, \lightning)\}$. This allows us to define the sequence of computations by $K_1 \,;\, K_2 \triangleq K_1 \cdot \lceil K_2$, and the unbounded iteration of this sequence (*i.e.* a loop with a runtime-chosen number of iterations) by

$$K^* \triangleq (\lceil K)^* \cap (D \times D_\lightning)$$

Given a predicate $P \in \mathcal{P}(D)$, we define the notion of *assumption* (or *guard*) as $\dashv P \triangleq (P \times D) \sqcap \varepsilon$. Informally, if P is satisfied on the current state then $\dashv P$ skips: it behaves like ε. Otherwise, $\dashv P$ produces no output: it behaves like $\bot$. We also define the dual notion of *assertion* as $\vdash P \triangleq (\dashv \neg P \,;\, \lightning) \sqcup \varepsilon$. If P is not satisfied on the current state, then $\vdash P$ produces an error. Otherwise, it skips.

With these operators, $\mathbb{K}$ provides a convenient language to express specifications: any Hoare specification (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ is expressed as the computation $\vdash P; Q$. Moreover, refinement allows to express usual Verification Conditions (VC) of Hoare Logic, for partial and total correctness. For our toy analyzer (described later), we only need VC for partial correctness. Typically, we use the usual partial correctness VC of unbounded iteration: K^* is equivalent to produce an output satisfying *every* inductive invariant I of K.

$$K^* \equiv \bigsqcap_{I \in \{I \in \mathcal{P}(D) \mid K \sqsubseteq \vdash I; D \times I\}} \vdash I; D \times I$$

In this equivalence, the $\sqsubseteq$-way corresponds to the soundness of the VC, whereas the $\sqsupseteq$-way corresponds to its completeness. In our context, such a soundness proof typically ensures that the specification of an abstract computation is refined by concrete semantics of the analyzed code. It guarantees that the analysis is correct *w.r.t.* semantics of the analyzed code.

Example on a Toy Language. Let t stands for an arithmetic term and c be a condition over numerical variables, whose syntax is $c ::= t_1 \bowtie t_2 \mid \neg c \mid c_1 \wedge c_2 \mid c_1 \vee c_2$ with $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. Semantics $[\![t]\!]$ of t and $[\![c]\!]$ of c work with a domain of integer memories $D \triangleq \mathbb{V} \to \mathbb{Z}$ where $\mathbb{V}$ is the type of variables. Hence, $[\![t]\!] \in D \to \mathbb{Z}$ and $[\![c]\!] \in \mathcal{P}(D)$. We omit their definition here.

[6] A postcondition is thus in $\mathcal{P}(D \times D)$ instead of the original $\mathcal{P}(D)$: this standard generalization avoids introducing "auxiliary variables" to represent the input state.

[7] However, in our algebra, $\sqsubseteq$ corresponds to "*refines*", whereas in standard refinement calculus it dually corresponds to "*is refined by*". Actually, our convention follows lattice notations of abstract interpretation.

Let us now introduce a small imperative programming language named $\mathbb{S}$ for which we will describe a toy analyzer in Section 2.2. The syntax of a $\mathbb{S}$ program s is described on Figure 1 together with its big-steps semantics $[\![s]\!]$ in $\mathbb{K}$. This semantics is defined recursively on the syntax of s using guarded commands derived from $\mathbb{K}$. First, we define $\dashv c \triangleq \dashv [\![c]\!]$ and $\vdash c \triangleq \vdash [\![c]\!]$. We also use command "$x := t$" defined as $\uparrow\!\lambda d.d[x := [\![t]\!](d)]$, where the memory assignment written "$d[x := n]$" – for $d \in D$, $x \in \mathbb{V}$ and $n \in \mathbb{Z}$ – is defined as the function $\lambda x' : \mathbb{V}, \texttt{if}\ x' = x\ \texttt{then}\ n\ \texttt{else}\ d(x')$.

s	$\textbf{assert}(c)$	$x \leftarrow t$	$s_1\,;s_2$	$\textbf{if}(c)\{s_1\}\textbf{else}\{s_2\}$	$\textbf{while}(c)\{s\}$
$[\![s]\!]$	$\vdash c$	$x := t$	$[\![s_1]\!]\,;[\![s_2]\!]$	$\dashv c\,;[\![s_1]\!] \sqcup \dashv\neg c\,;[\![s_2]\!]$	$(\dashv c\,;[\![s]\!])^*\,;\dashv\neg c$

Figure 1: Syntax and concrete semantics of $\mathbb{S}$

At this point, we have defined an algebra $\mathbb{K}$ of concrete computations: a language that we use to express specifications – for instance, in the form of Hoare specifications – on abstract computations. This algebra also provides denotations for defining big-steps semantics (like in Figure 1). Hence, $\mathbb{K}$ is aimed at providing an intermediate level between operational semantics of programs and their abstract interpretations (with the same purpose than the intermediate Hoare Logic in Verasco [17]). The next section defines how we certify correctness of abstract computations *w.r.t.* $\mathbb{K}$ computations.

2.2 Composing Diagrams to Certify Abstract Computations

Rice's theorem states that the property $d \overset{K}{\rightarrow} d'$ is undecidable. In the theory of abstract interpretation, we approximate K by a *computable (terminating) function* $^\sharp K$ working on an approximation $^\sharp D$ of $\mathcal{P}(D)$. Set $^\sharp D$ is called an *abstract domain* and it is related to $\mathcal{P}(D)$ by a concretization function $\gamma : {}^\sharp D \to \mathcal{P}(D)$. Function $^\sharp K$ is called an *abstract interpretation* (or *abstract computation*) of K. This paper considers two abstract domains, intervals and convex polyhedra, associated with the concrete domain $D \triangleq \mathbb{V} \to \mathbb{Z}$ involved in Figure 1.

1. Given $\mathbb{Z}_\infty \triangleq \mathbb{Z} \uplus \{-\infty, +\infty\}$, an abstract memory $^\sharp d$ of the interval domain is a finite map associating each variable x with an interval $[a_x, b_x]$ of $\mathbb{Z}_\infty \times \mathbb{Z}_\infty$. Its concretization is the set of concrete memory states satisfying the constraints of $^\sharp d$, *i.e.*

$$\gamma(^\sharp d) \triangleq \{d \in D \,|\, \forall x, a_x \leq d(x) \leq b_x\}$$

2. The concretization of a convex polyhedron $^\sharp d = \bigwedge_i \sum_j a_{ij}.x_j \leq b_i$, where a_{ij}'s and b_i's are rational constants and x_j's are integer program variables, is

$$\gamma(^\sharp d) \triangleq \{d \in D \mid \bigwedge_i \sum_j a_{ij}.d(x_j) \leq b_i\}$$

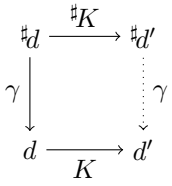
Correctness Diagrams of Impure Abstract Computations. Our framework only deals with partial correctness: we do not prove that abstract computations terminate, but only that they are a sound over-approximation of their corresponding concrete computation. Moreover, abstract computations may invoke untrusted oracles, whose results are verified by a certified checker. A bug in those oracles may make the whole computation non-deterministic or divergent. Thus, it is potentially unsound to consider abstract computations as pure functions. In this simplified presentation of our framework, we define abstract computations as relations in $^\sharp\mathbb{K} \triangleq \mathcal{R}(^\sharp D, {}^\sharp D)$. A more elaborate representation – based on monads – is defined in Section 4.1, in order to extract abstract computations from Coq to Ocaml functions.

We express correctness of abstract computations through commutative diagrams defined as follows.

Definition 1 (Correctness of abstract computations) *An abstract computation* $^\sharp K$ *of* $^\sharp\mathbb{K}$ *is correct w.r.t. a concrete computation* K *of* $\mathbb{K}$ *iff*

$\forall {}^\sharp d, {}^\sharp d' \in {}^\sharp D, \quad \forall d \in D, \forall d' \in D_{\lightning}$,

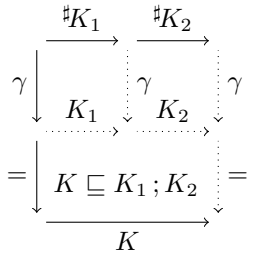
$${}^\sharp d \xrightarrow{{}^\sharp K} {}^\sharp d' \ \wedge \ d \xrightarrow{K} d' \ \wedge \ d \in \gamma({}^\sharp d) \qquad \Rightarrow \qquad d' \in \gamma({}^\sharp d')$$



Note that $d' \in \gamma({}^\sharp d')$ implies itself that $d' \neq \lightning$ because $\lightning$ is not in the image of γ.

Such a diagram thus corresponds to a pair of an abstract and a concrete computation, with a proof that the abstract one is correct *w.r.t.* the concrete one. As illustrated on the example below, these diagrams allow to build *compositional proofs* that an abstract computation, composed of several simpler parts, is correct *w.r.t.* a concrete computation. Diagrams are indeed preserved by several composition operators, and also by refinement of concrete computations.

As an example, consider two abstract computations $^\sharp K_1$ and $^\sharp K_2$ that are correct *w.r.t.* concrete K_1 and K_2. In order to show that the sequential composition $^\sharp K_1 \cdot {}^\sharp K_2$ is correct *w.r.t.* concrete K, it suffices to prove that $K \sqsubseteq K_1 ; K_2$, as illustrated on the right hand side scheme.



In the following, we introduce a datatype written $^\dagger\mathbb{K}$ to represent these diagrams: a diagram $^\dagger K \in {}^\dagger\mathbb{K}$ represents an abstract computation $^\sharp K$ which is correct *w.r.t.* its associated concrete computation K. The core of our approach is to lift guarded-commands on $\mathbb{K}$ involved in Figure 1 to guarded-commands on $^\dagger\mathbb{K}$. For instance, our toy analyzer $^\sharp[\![s]\!]$ for s in $\mathbb{S}$ is defined similarly to $[\![s]\!]$ of Figure 1, but from $^\dagger\mathbb{K}$ operators instead of $\mathbb{K}$ ones. For a given diagram $^\dagger K$, we can prove the correctness of an abstract computation $^\sharp K$ *w.r.t.* a concrete computation K' simply by proving that $K' \sqsubseteq K$. In practice, such refinement proofs are simplified using a weakest-liberal-precondition calculus (see Section 4.2).

Our Interface of Abstract Domains. The (simplified) theory of our abstract domains is defined in Figure 2. This theory is not included in Verasco's one because it allows impure operators (operators are relation, and not pure functions). Besides its concretization function γ, an abstract domain $^\sharp D$ provides constants $^\sharp\top$ and $^\sharp\bot$, representing respectively predicate true and false. It also provides abstract computations $^\sharp\dashv c$ and $x {}^\sharp{:=} t$ of $\mathcal{R}({}^\sharp D, {}^\sharp D)$, which are respectively correct *w.r.t.* concrete computations $\dashv c$ and $x := t$. It provides operator $^\sharp\sqcup$ of $\mathcal{R}({}^\sharp D \times {}^\sharp D, {}^\sharp D)$, which over-approximates the binary union on $\mathcal{P}(D)$. At last, it provides inclusion test $^\sharp\sqsubseteq$ of $\mathcal{R}({}^\sharp D \times {}^\sharp D, \texttt{bool})$.

$$D \subseteq \gamma({}^\sharp\top) \qquad \gamma({}^\sharp\bot) \subseteq \emptyset \qquad {}^\sharp d \xrightarrow{{}^\sharp\dashv c} {}^\sharp d' \ \Rightarrow \ \gamma({}^\sharp d) \cap [\![c]\!] \subseteq \gamma({}^\sharp d')$$

$${}^\sharp d \xrightarrow{x {}^\sharp{:=} t} {}^\sharp d' \wedge d \in \gamma({}^\sharp d) \Rightarrow d[x := [\![t]\!](d)] \in \gamma({}^\sharp d')$$

$$({}^\sharp d_1, {}^\sharp d_2) \xrightarrow{{}^\sharp\sqcup} {}^\sharp d' \Rightarrow \gamma({}^\sharp d_1) \cup \gamma({}^\sharp d_2) \subseteq \gamma({}^\sharp d') \qquad ({}^\sharp d_1, {}^\sharp d_2) \xrightarrow{{}^\sharp\sqsubseteq} \texttt{true} \Rightarrow \gamma({}^\sharp d_1) \subseteq \gamma({}^\sharp d_2)$$

Figure 2: Correctness specifications of our abstract domains

Abstract Computations of Guarded-commands. We derive our guarded-commands on $^\dagger\mathbb{K}$ in a generic way from any abstract domain satisfying the interface of Figure 2. As explained above, we lift each $\mathbb{K}$ guarded-command appearing in Figure 1 into a $^\dagger\mathbb{K}$ guarded-command. This lifting

is detailed in Figure 3: a $^\dagger\mathbb{K}$ operator has the same notation as its corresponding $\mathbb{K}$ operator and maps it to an abstract computation of $^\sharp\mathbb{K}$. The diagrammatic proof relating them is straightforward from correctness specifications given in Figure 2. We now detail the ideas behind this mapping.

Concrete commands $\dashv c$ and $x := t$ are trivially associated with $^\sharp\!\dashv c$ and $x\,^\sharp\!:\!=t$. Concrete command $K_1\,;K_2$ is associated with $^\sharp K_1 \cdot \underline{^\sharp K_2}$ – where $\underline{^\sharp K_2}$ returns $^\sharp\bot$ if the current abstract state is included in $^\sharp\bot$, or runs $^\sharp K_2$ otherwise. Concrete $K_1 \sqcup K_2$ is lifted by applying operator $^\sharp\!\sqcup$ to the results of $^\sharp K_1$ and $^\sharp K_2$.

Concrete assertion $\vdash c$ is associated with checking that the result of $^\sharp\!\dashv\neg c$ is *included* in $^\sharp\bot$: otherwise, the abstract computation *fails*. In our refinement proofs of abstract computations, "*to fail*" means "*to give no result*". Hence, concrete $\lightning$ is associated with abstract computation $\emptyset$ (and concrete $\bot$ is associated with $^\sharp\bot$). However, for our implementation of abstract computations in Section 4.1, "*to fail*" means "*to raise an alarm for the user*". In other words, our notion of correctness on abstract computations only gives some guarantee when no alarm is raised. In our proofs, we do not make distinction between an abstract computation that raises an alarm and an one that diverges.

At last, concrete K^* is associated with an abstract computation that invokes an untrusted oracle proposing an inductive invariant $^\sharp d_i$ of $^\sharp K$ for the current abstract state. Thus, using inclusion tests, $^\sharp(K^*)$ checks that $^\sharp d_i$ is actually an inductive invariant (otherwise, it fails), before returning it as the output abstract state.

$^\dagger\mathbb{K}$	Spec. in $\mathbb{K}$	Impl. in $^\sharp\mathbb{K}$
$\dashv c$	$\dashv c$	$^\sharp\!\dashv c$
$x := t$	$x := t$	$x\,^\sharp\!:\!=t$
$^\dagger K_1\,;{}^\dagger K_2$	$K_1\,;K_2$	$K_1.\{(^\sharp d_1, {}^\sharp d_2) \mid \exists b, (^\sharp d_1, {}^\sharp\bot) \overset{^\sharp\sqsubseteq}{\Rightarrow} b \wedge \textbf{if } b \textbf{ then } {}^\sharp d_2 = {}^\sharp\bot \textbf{ else } {}^\sharp d_1 \xrightarrow{^\sharp K_2} {}^\sharp d_2\}$
$^\dagger K_1 \sqcup {}^\dagger K_2$	$K_1 \sqcup K_2$	$\{(^\sharp d, {}^\sharp d') \mid \exists {}^\sharp d_1, \exists {}^\sharp d_2, {}^\sharp d \xrightarrow{^\sharp K_1} {}^\sharp d_1 \wedge {}^\sharp d \xrightarrow{^\sharp K_2} {}^\sharp d_2 \wedge ({}^\sharp d_1, {}^\sharp d_2) \overset{^\sharp\sqcup}{\Rightarrow} {}^\sharp d'\}$
$\vdash c$	$\vdash c$	$\{(^\sharp d, {}^\sharp d) \mid \exists {}^\sharp d', {}^\sharp d \xrightarrow{^\sharp\dashv\neg c} {}^\sharp d' \wedge ({}^\sharp d', {}^\sharp\bot) \overset{^\sharp\sqsubseteq}{\Rightarrow} \texttt{true}\}$
$^\dagger K^*$	K^*	$(^\sharp d, {}^\sharp d_i) \mid (^\sharp d, {}^\sharp d_i) \overset{^\sharp\sqsubseteq}{\Rightarrow} \texttt{true} \wedge \exists {}^\sharp d', {}^\sharp d \xrightarrow{^\sharp K} {}^\sharp d' \wedge ({}^\sharp d', {}^\sharp d_i) \overset{^\sharp\sqsubseteq}{\Rightarrow} \texttt{true}\}$

Figure 3: Guarded-commands of $^\dagger\mathbb{K}$ involved in $\mathbb{S}$ analysis

2.3 Higher-order Programming with Correctness Diagrams

Our linearization procedure detailed in Section 3.2 illustrates how we use GCL $^\dagger\mathbb{K}$ as a programming language for abstract computations. GCL $\mathbb{K}$ is our specification language. Each program $^\dagger K$ of $^\dagger\mathbb{K}$ is associated with a specification K of $\mathbb{K}$ syntactically derived from its code through mapping of Figure 3 and Figure 4. Indeed, Figure 4 details two other operators of $^\dagger\mathbb{K}$ invoked by our linearization procedure. First, operator $(\texttt{cast}\ {}^\dagger K\ K')$ casts a diagram $^\dagger K$ to a given specification K': it requires $K' \sqsubseteq K$ in order to produce a new valid $^\dagger\mathbb{K}$ diagram. This cast operator thus leads to a modular design of the certified development since it allows stepwise refinement of $^\dagger\mathbb{K}$ diagrams. Second, operator $(\pi\, {}^\dagger\!\gg\!=_Q {}^\dagger g)$ – where, for a given type A, π is of type $\mathcal{R}(^\sharp D, A)$ and $^\dagger g$ of type $A \to {}^\dagger\mathbb{K}$ – binds the results of π to $^\dagger g$. This operator requires a *concrete* postcondition Q of $A \to \mathcal{P}(D)$ on the results of π (see Figure 4).

More specifically, Section 3.2 applies our refinement calculus to certify higher-order abstract computations. Indeed, our linearization procedure *partitions* abstract states in order to increase precision. Continuation-Passing-Style (CPS) [29] is a higher-order pattern that provides a lightweight and modular style to program and certify simple partitioning strategies. Let us now detail this idea.

Given an abstract state $^\sharp d$, our linearization procedure invokes a sub-procedure $^\sharp f$ that splits $^\sharp d$ into a partition $(^\sharp d_i)_{i\in I}$ and computes a value r_i (of a given type A) for each cell $^\sharp d_i$. Then, the linearization procedure *continues* the computation from each cell $(r_i, {}^\sharp d_i)$ to finally return the join

$^\dagger\mathbb{K}$	Spec. in $\mathbb{K}$	Impl. in $^\sharp\mathbb{K}$	Under precondition
$\texttt{cast}\ {}^\dagger K\ K'$	K'	${}^\sharp K$	$K' \sqsubseteq K$
$\pi^\dagger \gg\!=_Q {}^\dagger g$	$\bigcap_x \vdash Q\,x\,;g\,x$	$\{({}^\sharp d_1, {}^\sharp d_2) \mid \exists x, {}^\sharp d_1 \overset{\pi}{\rightarrow} x \wedge {}^\sharp d_1 \xrightarrow{{}^\sharp g\,x} {}^\sharp d_2\}$	$\forall {}^\sharp d, \forall x \in A,$ ${}^\sharp d \overset{\pi}{\rightarrow} x \Rightarrow \gamma({}^\sharp d) \subseteq Q\,x$

Figure 4: $^\dagger\mathbb{K}$ operators that generate proof obligations

of all cells. In other words, from ${}^\sharp d$, ${}^\sharp f$ computes $(r_i, {}^\sharp d_i)_{i\in I}$. The main procedure finally computes ${}^\sharp\bigsqcup_{i\in I}({}^\sharp g\ r_i\ {}^\sharp d_i)$ – where ${}^\sharp g$ is a given function of $A \to {}^\sharp\mathbb{K}$. In order to avoid explicit handling of partitions, we make ${}^\sharp g$ a parameter of ${}^\sharp f$ to perform the join inside ${}^\sharp f$. In this style, ${}^\sharp f$ is of type $(A \to {}^\sharp\mathbb{K}) \to {}^\sharp\mathbb{K}$ and the parameter ${}^\sharp g$ of ${}^\sharp f$ is called their *continuation*.

However, specifying directly the correctness of computations that use CPS is not obvious because of the higher-order parameter. Actually, we define ${}^\dagger f$ of type $(A \to {}^\dagger\mathbb{K}) \to {}^\dagger\mathbb{K}$ and work with a continuation ${}^\dagger g$ of type $A \to {}^\dagger\mathbb{K}$. This allows us to specify CPS abstract computations *w.r.t.* CPS concrete computations. An example of such a specification is detailed later in Figure 8. Therefore, we keep implicit the notion of partition, both in specification and in implementation.

Similarly, CPS enables to implement some dynamic strategies of trace partitioning[25]. In abstract interpretation, "*trace partitioning*" corresponds to partition the set of all possible *execution traces* of the analyzed program in order to improve accuracy. Controlling the partitioning process is motivated by the fact that $({}^\sharp K_1 \cdot {}^\sharp K_3) {}^\sharp\sqcup ({}^\sharp K_2 \cdot {}^\sharp K_3) \ {}^\sharp\sqsubseteq\ ({}^\sharp K_1 {}^\sharp\sqcup {}^\sharp K_2) \cdot {}^\sharp K_3$, but the opposite inclusion does not hold. Hence, the left side is more precise whereas the right one is faster, as computation ${}^\sharp K_3$ is factorized. In practice, *dynamic* trace partitioning strategies select one of these two alternatives according to information of the current abstract state. The trace partitioning domain of [25] provides a functor able to extend a given abstract domain with dynamic partitioning management. More modestly, CPS allows to select some trace partitioning strategy at each function call through the choice of its continuation. For instance, we define ${}^\dagger f \triangleq \lambda {}^\dagger g, ({}^\dagger g\ {}^\dagger K_1) \sqcup ({}^\dagger g\ {}^\dagger K_2)$. Then, the precise alternative derives from ${}^\dagger f\ \lambda {}^\dagger K, ({}^\dagger K\,;{}^\dagger K_3)$ whereas the fast one derives from $({}^\dagger f\ \lambda {}^\dagger K, {}^\dagger K)\,;{}^\dagger K_3$. The CPS approach has the advantage to be very lightweight: there is no need to define and certify a data-structure to manage partitions. But it is less expressive than a trace partitioning domain. Indeed, a trace partitioning domain provides two kinds of partitioning operations: one to split partitions and one to merge partitions. Thus, the decision of merging partitions is quite independent of the decision to split partitions. On the contrary, with CPS, there is a single decision for each split/merge pair. Hence, a trace partitioning domain enables more dynamic merging strategies.

3 Interval-based Linearization Strategies for Polyhedra

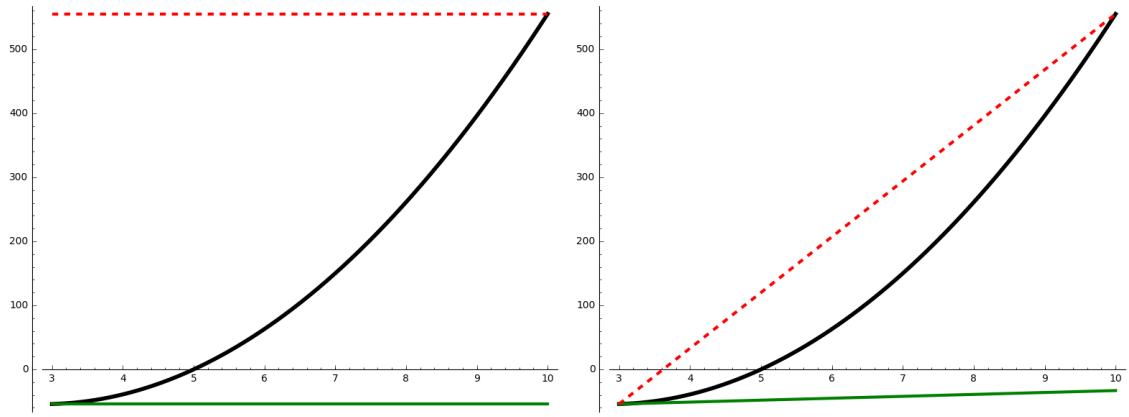
The VPL works with affine terms given by the abstract syntax $t ::= n \mid x \mid t_1 + t_2 \mid n.t$ where x is a variable and n a constant of $\mathbb{Z}$ [13]. We now extend VPL operators of Figure 2 to support polynomial terms, where the product "$n.t$" is generalized into "$t_1 \times t_2$".

The VPL derives assignment operator ${}^\sharp{:=}$ from guard ${}^\sharp{\dashv}$and two low-level operators: projection and renaming. It also derives the guard operator from a restricted one where conditions have the form $0 \bowtie t$ with $\bowtie \in \{\leq, =, \neq\}$. Hence, we only need to linearize the restricted guard ${}^\sharp{\dashv}0 \bowtie p$, where p is a polynomial. Below, we use letter p for polynomials and only keep letter t for affine terms.

Roughly speaking, we approximate a guard ${}^\sharp{\dashv}0 \bowtie p$ by guards ${}^\sharp{\dashv}0 \bowtie [t_1, t_2]$ – where t_1 and t_2 are affine or infinite bounds – such that, in the current abstract state, $p \in [t_1, t_2]$. Approximated guards ${}^\sharp{\dashv}0 \bowtie [t_1, t_2]$ are defined by cases on $\bowtie$:

$\bowtie$	$\leq$	$=$	$\neq$
${}^\sharp{\dashv}\ 0 \bowtie [t_1, t_2]$	${}^\sharp{\dashv}\ 0 \leq t_2$	${}^\sharp{\dashv}\ 0 \leq t_2 \wedge t_1 \leq 0$	${}^\sharp{\dashv}\ 0 < t_2 \vee t_1 < 0$

Affine intervals are computed using heuristics inspired from [26], except that in order to increase



(a) Constant intervalization **(b)** Focusing & affine intervalization

Figure 5: Two intervalizations of $p = (3.x-15)\times(4.x-3)$ with $x \in [3, 10]$. Constant intervalization leads to $p \in [-54, 555]$, whereas focusing gives $p \in [3.x - 63, 87.x - 315]$.

precision, we dynamically partition the abstract state according to the sign of some affine subterms. This process will be detailed further.

Our certified linearization is built on a two-tier architecture: an untrusted oracle uses heuristics to select linearization strategies and a certified procedure applies them to build a correct-by-construction result. These strategies, which are listed in Section 3.1, allow to finely tune the precision-versus-efficiency trade-off of the linearization. Section 3.2 details the design of our oracle and illustrates our lightweight handling of partitions using CPS in our certified procedure.

3.1 Our List of Interval-Based Strategies

Constant Intervalization. Our fastest strategy applies a constant intervalization operator of the abstract domain. Given a polynomial p, this operator, written $\sharp\pi(p)$, over-approximates p by an interval where affine terms are reduced to constants. More formally, $\sharp\pi(p)$ is a computation of $\mathcal{R}(\sharp D, \mathbb{Z}^2_\infty)$ such that if $\sharp d \xrightarrow{\sharp\pi(p)} [n_1, n_2]$, then $\gamma(\sharp d) \subseteq \{d \mid n_1 \leq [\![p]\!]d \leq n_2\}$. It uses a naive interval domain, where arithmetic operations $+$ and $\times$ are approximated by their correspondence on intervals:
$[n_1, n_2] + [n_3, n_4] \triangleq [n_1 + n_3, n_2 + n_4]$, and
$[n_1, n_2] \times [n_3, n_4] \triangleq [\min(E), \max(E)]$ where $E = \{n_1.n_3, n_1.n_4, n_2.n_3, n_2.n_4\}$.

Example 3 (Constant intervalization) On $x \in [3, 10]$, constant intervalization of $(3.x - 15) \times (4.x - 3)$ gives $(3.[3, 10] - 15) \times (4.[3, 10] - 3) = ([9, 30] - 15]) \times ([12, 40] - 3) = [-6, 15] \times [9, 37] = [-54, 555]$, as shown on Figure 5(a).

Ring Rewriting. A weakness of operator $\sharp\pi$ is its sensitivity to ring rewriting. For instance, consider a polynomial p_1 such that $\sharp\pi(p_1)$ returns $[0, n]$, $n \in \mathbb{N}^+$. Then $\sharp\pi(p_1 - p_1)$ returns $[-n, n]$ instead of the precise result 0. Such imprecision occurs in barycentric computations such as $p_2 \triangleq p_1 \times t_1 + (n - p_1) \times t_2$ where affine terms t_1, t_2 are bounded by $[n_1, n_2]$. Indeed $\sharp\pi(p_2)$ returns $2n.[n_1, n_2]$ instead of $n.[n_1, n_2]$. Moreover, if we rewrite p_2 into an equivalent polynomial $p'_2 \triangleq p_1 \times (t_1 - t_2) + n.t_2$, then $\sharp\pi(p'_2)$ returns $n.[2.n_1 - n_2, 2.n_2 - n_1]$. If $n_1 > 0$ or $n_2 < 0$, then $\sharp\pi(p'_2)$ is strictly more precise than $\sharp\pi(p_2)$. The situation is reversed otherwise. Consequently, our oracle begins by simplifying the polynomial before trying to factorize it conveniently. But as illustrated above, it is difficult to find a factorization minimizing $\sharp\pi$ results. We give more details on the ring rewriting heuristics of our oracle in the following.

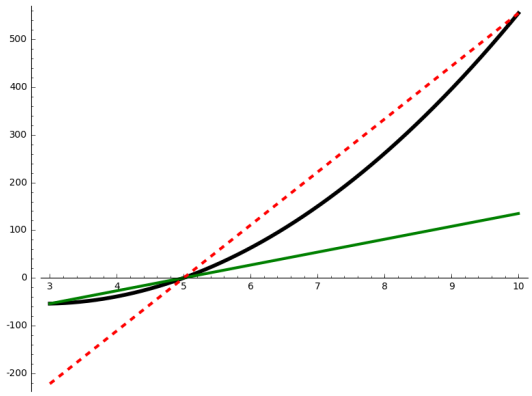


Figure 6: A wrong affine intervalization of $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$.

Sign Partitioning. In order to find more precise bounds of polynomial p than those given by $\sharp\pi(p)$, we look for an interval of two affine terms $[t_1, t_2]$ bounding p. Assume p is of the form $p' \times t$, with t an affine term and p' a polynomial. Let $[n'_1, n'_2]$ be the constant intervalization of p' obtained from $\sharp\pi(p')$. Depending on the sign of t, we deduce affine bounds of p in the following way:

- if $0 \leq t$, then $p' \times t \in [n'_1.t, n'_2.t]$
- if $t < 0$, then $p' \times t \in [n'_2.t, n'_1.t]$

When the sign of t is known, we discard one of these two cases and thus have a fast affine approximation of $p' \times t$. This is the case in Figure 5(b) (the underlying computations are detailed in Example 6). When the sign of t is unknown, we split the analysis for each sign of t.

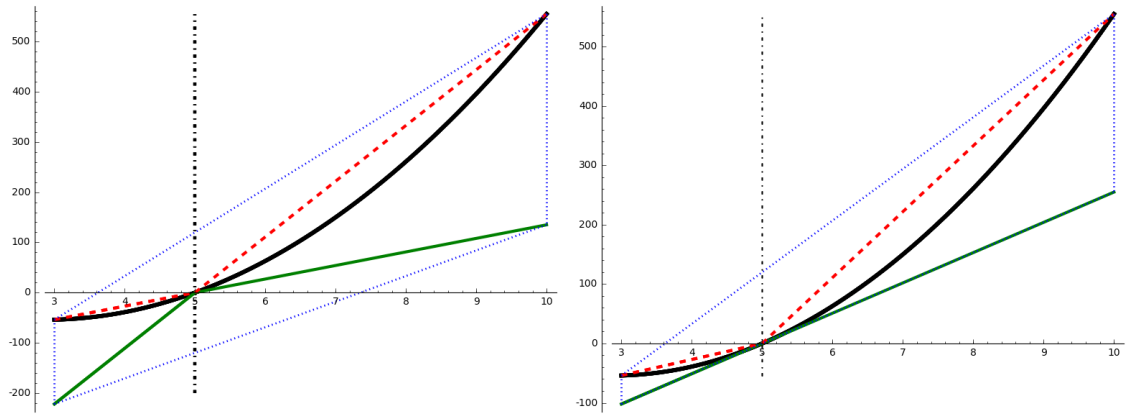
More generally, we split the current abstract state $\sharp d$ into a partition $(\sharp d_i)_{i \in I}$ according to the sign of some affine subterms of polynomial p, such that each cell $\sharp d_i$ leads to its own affine interval $[t_{i,1}, t_{i,2}]$. Finally, $\sharp\dashv 0 \bowtie p$ is over-approximated by computing the join of all $\sharp\dashv 0 \bowtie [t_{i,1}, t_{i,2}]$. The main drawback of sign partitioning is a worst-case exponential blow-up if applied systematically.

Example 4 (Sign partitioning) Consider $p = (4.x - 3) \times (3.x - 15)$ with $x \in [3, 10]$, as in Example 3. First, we compute the constant intervalization of the left term $(4.x-3)$, which gives $p = (4.[3, 10]-3)\times(3.x-15) = ([12, 40]-3)\times(3.x-15) = [9, 37]\times(3.x-15) = [9.(3.x-15), 37.(3.x-15)]$. We obtain the two affine terms $9.(3.x - 15)$ and $37.(3.x - 15)$. But as shown on Figure 6, for $x \in [3, 10]$, these two terms are not comparable. Indeed, $9.(3.x - 15)$ is not always lower than $37.(3.x - 15)$ on $x \in [3, 10]$. Thus, $[9.(3.x - 15), 37.(3.x - 15)]$ is not a well-defined interval of affine forms. In order to get an affine interval bounding p, we need to partition the space at the point where these two terms are equal, *i.e.* at the point where $3.x - 15 = 0$ which is $x = 5$. Then, by intervalizing in both cells $3.x - 15 < 0$ and $3.x - 15 \geq 0$, we get:

$$p = \begin{cases} \left[\ 37.(3.x - 15)\ ,\ \ 9.(3.x - 15)\ \right] & \text{if } 3.x - 15 < 0 \\ \left[\ 9.(3.x - 15)\ \ ,\ 37.(3.x - 15)\ \right] & \text{if } 3.x - 15 \geq 0 \end{cases}$$

The result is shown on Figure 7(a). To obtain the final result of the linearization, it is necessary to compute the convex hull of both sides. Here, the result is $p \in [51.x - 375, 87.x - 315]$, and it appears as the blue dotted polyhedron on the figure.

Let us also illustrate sign partitioning for the previous barycentric-like computation of p'_2. By convention, our certified procedure partitions the sign of right affine subterms (here, the sign of $t_1 - t_2$). Hence, it founds $p'_2 \in [n.t_2, n.t_1]$ in cell $0 \leq t_1 - t_2$, and $p'_2 \in [n.t_1, n.t_2]$ in cell $t_1 - t_2 < 0$.



(a) Static partitioning **(b)** Dynamic partitioning

Figure 7: Sign partitioning of $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$. Static partitioning gives $p \in [51.x - 375, 87.x - 315]$, whereas dynamic one gives $p \in [51.x - 255, 87.x - 315]$.

When it joins the two cells, $\sharp\!\dashv\!0 \bowtie p'_2$ is computed as $\sharp\!\dashv\!0 \bowtie n.[n_1, n_2]$ as we expect for such a barycentre. Note that sign partitioning is also sensitive to ring rewriting. In particular, the oracle may rewrite a product of affine terms $t_1 \times t_2$ into $t_2 \times t_1$, in order to discard t_1 instead of t_2 by sign partitioning.

Static vs Dynamic Intervalization During Partitioning. Computing the constant bounds of an affine term inside a given polyhedron invokes a costly linear programming procedure. Hence, for a given polynomial p to approximate, we start by computing an environment σ that associates each variable of p with a constant interval: as detailed later, this environment is indeed used by heuristics of our oracle. By default, operator $\sharp_\pi$ is called in *dynamic* mode, meaning that each bound is computed dynamically in the current cell – generated from sign partitioning – using linear programming. If one wants a faster use of operator $\sharp_\pi$, he may invoke it in *static* mode, where bounds are computed using σ.

For instance, let us consider the sign partitioning of $p \triangleq t_1 \times t_2$ in the context $0 < n_1, n_2$ and $-n_1 \leq t_2 \leq t_1 \leq n_2$. In cell $0 \leq t_2$, static mode bounds p by $[-n_1.t_2, n_2.t_2]$, whereas dynamic mode bounds p by $[0, n_2.t_2]$. In cell $t_2 < 0$, both modes bound p by $[n_2.t_2, -n_1.t_2]$. On the join of these cells, both modes give the same upper bound. But the lower bound is $-n_1.n_2$ for static mode, whereas it is $\frac{n_1.n_2}{n_1+n_2}(t_2 + n_1) - n_1.n_2$ for dynamic mode, which is strictly more precise.

Example 5 (Static vs Dynamic Intervalization) In Example 4, we saw that partitioning on the sign of $(3.x - 15)$ gave

$$\begin{array}{rcl} p & = & (4.x - 3) \times (3.x - 15) \\ & = & [9, 37] \times (3.x - 15) \\ & = & \begin{cases} \left[\begin{array}{ll} 37.(3.x - 15) \ , & 9.(3.x - 15) \end{array}\right] & \text{if } 3.x - 15 < 0 \\ \left[\begin{array}{ll} 9.(3.x - 15) \ , & 37.(3.x - 15) \end{array}\right] & \text{if } 3.x - 15 \geq 0 \end{cases} \end{array}$$

This intervalization is in fact a *static* one because $(4.x-3)$ was intervalized in the same way in both cells, using $x \in [3, 10]$. Instead, using *dynamic* intervalization during partitioning will improve the precision by finding better bounds of $(4.x - 3)$. Indeed, building on the fact that $x \in [3, 5]$ on cell $(3 \leq x \wedge 3.x - 15 < 0)$, intervalizing $(4.x - 3)$ gives $(4.[3, 5] - 3) = ([12, 20] - 3) = [9, 17]$. Similarly, on cell $(x \leq 10 \wedge 3.x - 15 \geq 0)$, $x \in [5, 10]$ hence an intervalization of $(4.x - 3)$ by

$(4.[5,10]-3) = ([20,40]-3) = [17,37]$. Thus,

$$\begin{aligned} p &= \begin{cases} [9,17] \times (3.x-15) & \text{if } 3.x-15 < 0 \\ [17,37] \times (3.x-15) & \text{if } 3.x-15 \geq 0 \end{cases} \\ &= \begin{cases} \left[\ 17.(3.x-15)\ ,\ \ 9.(3.x-15)\ \right] & \text{if } 3.x-15 < 0 \\ \left[\ 17.(3.x-15)\ ,\ 37.(3.x-15)\ \right] & \text{if } 3.x-15 \geq 0 \end{cases} \end{aligned}$$

As explained before, the final result is obtained by computing the convex hull of both cells. Here, we get $p \in [51.x - 255, 87.x - 315]$. The difference between static and dynamic partitioning is shown on Figure 7. We can see that the lower bound of p has been significantly improved by the dynamic partition. The upper bound resulting from static partitioning was already optimal.

Focusing. Focusing is a ring rewriting heuristic that may increase the precision of sign partitioning. Given a product $p \triangleq t_1 \times t_2$, we define the *focusing* of t_2 in center n as the rewriting of p into $p' \triangleq n.t_1 + t_1 \times (t_2 - n)$. Thanks to this focusing, the affine term $n.t_1$ appears whereas t_1 would otherwise be discarded by sign partitioning. Let us simply illustrate the effect of this rewriting when $0 \leq n \leq n'_1$ with t_1 (resp. t_2) bounded by $[n_1, n_2]$ (resp. $[n'_1, n'_2]$). Sign partitioning bounds p in affine interval $[n_1.t_2, n_2.t_2]$ whereas p' is bounded by interval $[n_1.t_2 + n.(t_1 - n_1), n_2.t_2 - n.(n_2 - t_1)]$. The former contains the latter since $t_1 - n_1$ and $n_2 - t_1$ are nonnegative. Under these assumptions, the precision is maximal when $n = n'_1$.

Applied carelessly, focusing may also decrease the precision. Consequently, on products $p'' \times t_2$, our oracle uses the following heuristic, which cannot decrease the precision: if $0 \leq n'_1$, then focus t_2 in center n'_1; if $n'_2 \leq 0$, then focus t_2 in center n'_2; otherwise, do not change the focus of t_2.

Example 6 (Focusing) Consider $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$, as in previous examples. The focusing of term $(4.x - 3)$ on $4.3 - 3 = 9$ is $p' = 9.(3.x - 15) + (3x - 15) \times (4.x - 12)$. Affine intervalization of p' is done by sign partitioning of $(4.x - 12)$, where cell $4.x - 12 < 0$ is empty. Finally, by intervalization,

$$\begin{aligned} p &= 9.(3.x - 15) + (3x - 15) \times (4.x - 12) \\ &= (27.x - 135) + (3.[3, 10] - 15) \times (4.x - 12) \\ &= (27.x - 135) + [-6, 15] \times (4.x - 12) \\ &= (27.x - 135) + [-24.x + 72, 60.x - 180] \\ &= [3.x - 63, 87.x - 315] \end{aligned}$$

Figure 5(b) shows its result. Intervalizations of Figure 5(a) and of Figure 5(b) have similar running times, but this latter gives strictly more precise results. Intervalizations of Figures 7(b) and 5(b) are not comparable: Figure 7(b) is more precise on a significative part of the domain $x \in [3, 10]$, but Figure 5(b) is better around the lower-left corner. The precision of Figure 7(b) comes at a cost: it requires two constant intervalizations and a convex-hull instead of one single constant intervalization.

Conjunction of strategies. As we saw by comparing Figures 5(b) and 7(b), two distinct linearization strategies may lead to incomparable polyhedra. Here, we can improve precision by computing the intersection of these polyhedra. In our stepwise refinement approach, this corresponds indeed to remark that $\dashv c \sqsubseteq (\dashv c\,; \dashv c)$, and to implement each of these guards $\dashv c$ with a distinct linearization strategy. Let us remark here that a sequence of two strategies gives more precise results than intersecting independent runs of these strategies: the second one may benefit from informations discovered by the first one. This is illustrated in Example 7 below. We use this trick in order to ensure that our linearization necessarily improves and benefits from results of a naive but quick constant intervalization.

Given ${}^{\dagger}\pi\, p$ of $(\mathbb{Z}^2_\infty \to {}^{\dagger}\mathbb{K}) \to {}^{\dagger}\mathbb{K}$ defined by ${}^{\dagger}\pi\, p\, {}^{\dagger}g_0 \triangleq {}^{\sharp}\pi(p)\, {}^{\dagger}\gg\!=_{\lambda[n_1,n_2],\{d \mid n_1 \le [\![p]\!]d \le n_2\}} {}^{\dagger}g_0$ the ${}^{\dagger}\mathbb{K}$ program on the right-hand side satisfies the specification below:

$$\bigsqcap_{[t_1,t_2]} \vdash \{d \mid t_1 \le [\![p \times t]\!]d \le t_2\}\,;g[t_1,t_2]$$

if static **then**
  ${}^{\dagger}\pi\ p\ \ (\lambda[n_1,n_2],\ \ (\dashv 0 \le t\,;{}^{\dagger}g[n_1.t, n_2.t])$
  $\sqcup\ \ (\dashv t < 0\,;{}^{\dagger}g[n_2.t, n_1.t]))$
else
  $(\dashv 0 \le t\,;{}^{\dagger}\pi\ p\ \lambda[n_1,n_2], {}^{\dagger}g[n_1.t, n_2.t])$
$\sqcup\ \ (\dashv t < 0\,;{}^{\dagger}\pi\ p\ \lambda[n_1,n_2], {}^{\dagger}g[n_2.t, n_1.t])$

Figure 8: Sign partitioning for $p \times t$ with continuation ${}^{\dagger}g$

3.2 Design of Our Implementation

We now decribe our procedure in detail. For a guard ${}^{\sharp}\dashv 0 \bowtie p$, our certified procedure first rewrites p into $p' + t$ where t is an affine term and p' a polynomial. This may keep the non-affine part p' small compared to the affine one t. Typically, if p' is syntactically equal to zero, we simply apply the standard affine guard ${}^{\sharp}\dashv 0 \bowtie t$. Otherwise, we compute environment σ for p' variables. Then, we compute ${}^{\sharp}\dashv 0 \bowtie [n_1 + t, n_2 + t]$ where $[n_1, n_2]$ is the result of ${}^{\sharp}\pi(p')$ for static environment σ. As mentioned earlier, this ensures that the resulting linearization necessarily improves and benefits from this first constant intervalization. In particular, if this guard is unsatisfiable at this point, the rest of the procedure is skipped. Otherwise, we invoke our external oracle on p' and σ. This oracle returns a polynomial p'' enriched with tags on subexpressions. We handle three tags to direct the intervalization: `AFFINE` expresses that the subexpression is affine; `STATIC` expresses that the subexpression has to be intervalized in static mode; `INTERV` expresses that intervalization is done using only ${}^{\sharp}\pi$ (instead of sign partitioning). At last, a special tag `SKIP_ORACLE` inserted at the root of p'' indicates that it is not worth attempting to improve naive constant intervalization (*e.g.* because p' is a too big polynomial, any attempt would be too costly). After that, when this special tag is absent, our certified procedure checks that $p' = p''$ using a normalization procedure defined in the standard distribution of Coq [15]. If $p' \neq p''$, our procedure simply raises an error corresponding to a bug in the oracle. If $p' = p''$, it invokes a CPS affine intervalization of p'' for continuation $\lambda[t_1, t_2], \dashv 0 \bowtie [t_1 + t, t_2 + t]$. The next paragraphs detail this certified CPS intervalization and then, our external oracle.

Certified CPS Affine Intervalization. We implement and prove our affine intervalization using the CPS technique described in Section 2.3. On polynomial p'' and continuation ${}^{\dagger}g$, the specification of our CPS intervalization is

$$\varepsilon \sqcap \bigsqcap_{[t_1,t_2]} \vdash \{d \mid t_1 \le [\![p'']\!]d \le t_2\}\,;g[t_1,t_2]$$

The ε case corresponds to a failure of our procedure: typically, a subexpression is not affine as claimed by the external oracle. In case of success, the procedure selects non-deterministically some affine intervals $[t_1, t_2]$ bounding p'' before merging continuations on them. The procedure is implemented recursively over the syntax of p''. Figure 8 sketches the implementation and the specification of the sign partitioning subprocedure. The figure deals with a particular case where p'' is a polynomial written $p \times t$ with t affine. In the implementation part, boolean `static` indicates the mode of ${}^{\sharp}\pi$. In static mode, we indeed factorize the computation of ${}^{\sharp}\pi$ on both cells of the partition.

Our linearization procedure is written in around 2000 Coq lines, proofs included. Among them, the CPS procedure and its subprocedures take only 200 lines. The bigger part – around 1000 lines – is thus taken by arithmetic operators on interval domains (constant and affine intervals).

Design of Our External Oracle. Our external oracle ranks variables according to their priority to be discarded by sign partitioning. Then, it factorizes variables with the highest priority. The

priority rank is mainly computed from the size of intervals in the precomputed environment σ: unbounded variables must not be discarded whereas variables bounded by a singleton are always discarded by static intervalization. Our oracle also tries to minimize the number of distinct variables that are discarded: variables appearing in many monomials have a higher priority. The oracle also interleaves factorization with focusing. Our oracle is written in 1300 lines of OCaml code.

Example 7 (A full run of the certified procedure) Let us consider the effect of our linearization procedure on guard $\dashv x \times (y-2) \leq z$ in a context where $(0 \leq x) \wedge (x+1 \leq y \leq 1000) \wedge (z \leq -2)$. First, note that a constant intervalization of $z - x \times (y-2)$ would bound it in $]-\infty, 997]$, and thus would not deduce anything useful from this guard.

Instead, our procedure rewrites the guard into $\dashv 0 \leq p' + t$ with $p' \triangleq -x \times y$ and $t \triangleq z + 2x$. Then, it computes environment $\sigma \triangleq \{x \mapsto [0, 999], y \mapsto [1, 1000]\}$ and applies constant intervalization on p', leading to $p' \in]-\infty, 0]$. As you may notice, approximating this guard requires only an upper-bound on p', and our procedure does not compute the useless lower bound. From this first approximation of $\dashv 0 \leq p' + t$, it deduces $0 \leq t$.

Then, our oracle, invoked on p' and σ, decides to focus y in center 1 and thus rewrites p' as $p'' \triangleq x \times (1-y) - x$. Here, our CPS subprocedure only intervalizes the nonlinear part $x \times (1-y)$ using sign partitioning on $1-y$. Because we know $1 \leq x$ from $2 \leq -z \leq 2x$, we deduce $1-y \leq -x \leq -1$. Therefore, because $1-y < 0$ and $1 \leq x$, sign partitioning on $1-y$ bounds $x \times (1-y)$ by $]-\infty, 1-y]$. At last, CPS intervalization now approximates $\dashv 0 \leq p' + t$ in $\dashv 0 \leq 1 - y - x + t$. In fact, this implies $0 \leq z$ which contradicts $z \leq -2$. Hence, our polyhedral approximation of $\dashv x \times (y-2) \leq z$ detects that this guard is unsatisfiable in the given context.

As a conclusion, let us remark that the first approximation leading to $0 \leq t$ is necessary to the full success of the second one.

4 A Lightweight Refinement Calculus in Coq

Our implementation in Coq reformulates Section 2 with more computational representations of binary relations. Section 4.1 presents the representation change of impure computations, which include abstract computations. Section 4.2 presents that of concrete ones. Finally, Section 4.3 presents our datatypes for correctness diagrams of abstract computations. Sections 4.1 and 4.3 also detail how the framework is adapted in order to handle alarms during the analysis.

4.1 Monadic Representations of Impure (and Abstract) Computations

In VPL, all computations involving oracles (including abstract computations) are "impure" – in the sense defined in [13]. This section recalls the representation of impure computations from [13] and links it their representation by relations from Section 2.

A relation R of $\mathcal{R}(A, B)$ can be equivalently seen as the function $\lambda x, \{y \mid x \overset{R}{\to} y\}$ of $A \to \mathcal{P}(B)$. This curryfied representation is the basis of VPL representation for impure computations. Indeed, this latter aims to provide a Coq representation of relations that can be turned into an OCaml function at extraction. To this end, type "$\mathcal{P}(B)$" is axiomatized in Coq as type "$??B$" where "$??$" is a notation for the type transformer of may-return monads introduced in [13] and recalled below. Hence, all impure computations of $\mathcal{R}(A, B)$ in Figure 2 are actually expressed in our Coq development as functions of $A \to ??B$ in a given may-return monad. Indeed, the interface of may-return monads also allows to hide data-structure details – such as handling of alarms – for the correctness proof of abstract computations. The next paragraphs detail these ideas.

Definition 2 (May-return monad) *For any type A, type $??A$ represents impure computations returning values of type A. Type transformer "$??.$" is equipped with a monad [32] providing a may-return relation [13].*

- *Operator $\gg\!=_{A,B} : ??A \to (A \to ??B) \to ??B$ encodes an impure sequence "***let** $x = k_1$ **in** k_2*" as "$k_1 \gg\!= \lambda x, k_2$".*

$$^{c}??A \triangleq A \qquad k_1 \,^{c}\!\equiv k_2 \triangleq k_1 = k_2 \qquad k \,^{c}\!\rightsquigarrow a \triangleq k = a \qquad ^{c}\varepsilon\, a \triangleq a \qquad k_1 \,^{c}\!\gg\!= k_2 \triangleq k_2\, k_1$$

Figure 9: Identity implementation of the core monad

$$^{c}??A \triangleq S \to A \to S \to \mathrm{Prop} \qquad k_1 \,^{c}\!\equiv k_2 \triangleq \forall s_0, \forall a, \forall s_1,\ k_1\, s_0\, a\, s_1 \leftrightarrow k_2\, s_0\, a\, s_1$$

$$k \,^{c}\!\rightsquigarrow a \triangleq \exists s_0, \exists s_1,\ k\, s_0\, a\, s_1 \qquad ^{c}\varepsilon\, a \triangleq \lambda s_0, \lambda a', \lambda s_1,\ a = a' \wedge s_0 = s_1$$

$$k_1 \,^{c}\!\gg\!= k_2 \triangleq \lambda s_0, \lambda b, \lambda s_1,\ \exists a, \exists s_1, k_1\, s_0\, a\, s_1 \wedge k_2\, a\, s_1\, b\, s_2$$

Figure 10: A model of the core monad as big-steps over a global state S

- *Operator* $\varepsilon_A : A \to ??A$ *lifts a pure computation to an impure one.*
- *Relation* $\equiv_A : ??A \to ??A \to \mathrm{Prop}$ *is a congruence (w.r.t.* $\gg\!=$*) representing equivalence of semantics between impure computations. Moreover, operator* $\gg\!=$ *is associative and admits* ε *as neutral element (w.r.t.* $\equiv$*).*
- *Relation* $\rightsquigarrow_A : ??A \to A \to \mathrm{Prop}$*, where "*$k \rightsquigarrow a$*" means that "*$k$ may return a*". This relation must be compatible with* $\equiv_A$ *and satisfies the axioms*

$$\varepsilon\, a_1 \rightsquigarrow a_2 \ \Rightarrow\ a_1 = a_2 \qquad k_1 \gg\!= k_2 \rightsquigarrow b \ \Rightarrow\ \exists a, k_1 \rightsquigarrow a \wedge k_2\, a \rightsquigarrow b$$

Correspondence with Set Theory Notations of Section 2. Abstraction of set "$\mathcal{P}(A)$" as type "$??A$" is given by the following definitions:

$$??A \triangleq \mathcal{P}(A) \qquad k_1 \equiv k_2 \triangleq \forall x, x \in k_1 \Leftrightarrow x \in k_2 \qquad k \rightsquigarrow a \triangleq a \in k \qquad \varepsilon\, a \triangleq \{a\}$$

$$k_1 \gg\!= k_2 \triangleq \bigcup_{a \in k_1} (k_2\, a)$$

Conversely, for any may-return monad, a computation k of $A \to ??B$ represents a relation of $\mathcal{R}(A, B)$ defined by $d \overset{k}{\to} d' \triangleq k\, d \rightsquigarrow d'$. Given two abstract computations k_1 and k_2 in $^{\sharp}D \to ??^{\sharp}D$, then "$\lambda x, (k_1\ x) \gg\!= k_2$" corresponds to a subrelation of "$k_1 \cdot k_2$". Actually, our Coq implementation generalizes Section 2 when we compose abstract computations, because we use operator "$\gg\!=$" instead of the less precise "$\cdot$".

Impure Computations of the Core May-return Monad. The VPL is parametrized by a *core* may-return monad that axiomatizes external computations. This monad avoids a potential unsoundness by expressing that external oracles are not pure functions, but encode relations. It is instantiated at extraction by providing the identity implementation given in Figure 9.

Of course, this implementation of the core monad remains hidden for our Coq proofs: they are thus valid for any instance of a may-return monad. As an example, big-step semantics of imperative computations over a global state S induces such an instance, defined in Figure 10. Actually, our Coq proofs are sound under the hypothesis that there exists a may-return monad able to denote any typesafe OCaml computation. But, formalizing the soundness of Coq-certified code embedding untrusted OCaml oracles through may-return monads is still an open issue: see [4, 6].

Alarm Handling in the Analyzer. Our toy analyzer, specified in Figure 1, handles alarms in the style of Verasco [17]. On a potential error, it does not stop its analysis, but writes an

$$^{w}??A \triangleq {}^{c}??(A \times \texttt{bool}) \qquad k_1 \,{}^{w}\!\equiv k_2 \triangleq k_1 \,{}^{c}\!\equiv k_2 \qquad k \,{}^{w}\!\rightsquigarrow a \triangleq k \,{}^{c}\!\rightsquigarrow (a, \texttt{true}) \qquad {}^{w}\varepsilon\, a \triangleq {}^{c}\varepsilon\,(a, \texttt{true})$$

$$k_1 \,{}^{w}\!\gg\!= k_2 \;\triangleq\; k_1 \,{}^{c}\!\gg\!= \lambda(a_1, l_1), (k_2\, a_1)\,{}^{c}\!\gg\!= \lambda(a_2, l_2), {}^{c}\varepsilon\,(a_2, l_1 \wedge l_2)$$

$$\texttt{lift}\, k \triangleq k \,{}^{c}\!\gg\!= \lambda a,\, {}^{c}\varepsilon\,(a, \texttt{true}) \qquad {}^{w}\texttt{write}\, m\, a \triangleq {}^{c}\texttt{write}\, m \,{}^{c}\!\gg\!= \lambda\_,\, {}^{c}\varepsilon\,(a, \texttt{false})$$

Figure 11: Alarm writer monad and its specific operators

alarm – represented here as a value of type `alarm` – and continues the analysis. Technically, this corresponds to lifting the core monad through a *writer monad transformer* [21]. Actually, we assume that the core monad has already an operation to write alarms ${}^{c}\texttt{write} : \texttt{alarm} \to {}^{c}??\texttt{unit}$, which is efficiently extracted as OCaml external code. On the Coq side, our *alarm writer monad* thus only encodes the underlying list of alarms as a boolean: `true` corresponds to an empty list of alarms. It is defined in Figure 11 where alarm writer (resp. core) constructs are prefixed by a "^{w}" (resp. "^{c}"). The implementation of ${}^{w}\!\rightsquigarrow$ means that the formal correctness of abstract computations with at least one alarm holds trivially. Hence, on a ${}^{\dagger}\mathbb{K}$ diagram, an abstract computation fails (*i.e.* produces no result) as soon as it produces an alarm. On the contrary, in the actual implementation, it produces a result that may be used to find more alarms (without formal guarantee on their meaning).

Figure 11 also defines operator $\texttt{lift}_A : {}^{c}??A \to {}^{w}??A$. Using `lift`, it is straightforward to lift VPL abstract domains with computations in the core monad to abstract domains with computations in the alarm writer monad. At last, operator ${}^{w}\texttt{write}_A : \texttt{alarm} \to A \to {}^{w}??A$ encodes that ${}^{w}\texttt{write}\, m\, a$ writes alarm m and returns value a. It is invoked in the implementation of ${}^{\dagger}\mathbb{K}$ commands that may *fail*: assert (operator "$\vdash .$") and loop (operator "$.^*$").

For example, let us assume here that function ${}^{\sharp}\!\dashv\neg c : {}^{\sharp}D \to {}^{c}??{}^{\sharp}D$ and function ${}^{\sharp}\!\sqsubseteq : {}^{\sharp}D \to {}^{\sharp}D \to {}^{c}??\texttt{bool}$ are VPL operators from the *core monad* corresponding to those of Figure 2. Operator ${}^{\sharp}\!\vdash c$, described in Figure 3, is implemented in the *alarm writer monad* by the function of type ${}^{\sharp}D \to {}^{w}??{}^{\sharp}D$ given below:

$$\begin{array}{ll} \lambda {}^{\sharp}d, & (\texttt{lift}\ ({}^{\sharp}\!\dashv\neg c\ {}^{\sharp}d))\ {}^{w}\!\gg\!= \lambda {}^{\sharp}d', \\ & (\texttt{lift}\ ({}^{\sharp}\!\sqsubseteq\ {}^{\sharp}d'\ {}^{\sharp}\!\bot))\ {}^{w}\!\gg\!= \lambda b, \\ & \textbf{if}\ b\ \textbf{then}\ {}^{w}\varepsilon\, {}^{\sharp}d\ \textbf{else}\ ({}^{w}\texttt{write}\ \texttt{"}\textit{assert failure}\texttt{"}\ {}^{\sharp}d) \end{array}$$

In order to prove that operator ${}^{\sharp}\!\vdash c$ is correct *w.r.t.* its specification $\vdash c$, it suffices to prove the property "${}^{\sharp}\!\vdash c\ {}^{\sharp}d \rightsquigarrow {}^{\sharp}d' \Rightarrow \gamma({}^{\sharp}d) \subseteq [\![c]\!] \wedge {}^{\sharp}d = {}^{\sharp}d'$". The proof that this property implies a correct abstraction of $\vdash c$ is independent of the underlying monad.

In summary, the alarm writer monad instantiates our notion of analyzer correctness into "*if the analyzer terminates without raising any alarm*[8]*, then the analyzed program has no runtime error*". Thanks to our compositional design through monads, reasonings on alarm handling appear only in the implementation of the alarm writer monad. Indeed, "raising an alarm" is logically equivalent to a computation that never returns. Actually, Verasco also manages alarms through a writer monad [17]. We have just shown that this feature is very easy to deal with in our framework.

4.2 Representation of Concrete Computations

We consider the issue of mechanizing refinement proofs of $\mathbb{K}$ computations. Definition of $\mathbb{K}$ in Section 2.1 uses operators inspired from regular expressions. Formally, $\mathbb{K}$ embeds the Kleene algebra[9] of $\mathcal{R}(D, D)$: if K_1 and K_2 are in $\mathcal{R}(D, D)$, then $K_1 ; K_2 = K_1 \cdot K_2$. However, $\mathbb{K}$ does

[8] Formally, the status "no alarm is raised" is given by the boolean of our alarm writer monad

[9] A Kleene algebra is an idempotent (and thus partially ordered) semiring endowed with a closure operator. It generalizes the operations known from regular expressions: the set of regular expressions over an alphabet is a *free* Kleene algebra.

not satisfy itself all properties of a Kleene algebra. In particular, " ;" has two distinct left-zeros $\bot$ and $\lightning$. Thus, it has no right-zero. This forbids applying directly existing Coq tactics for Kleene algebras[7].

Like in standard refinement calculus [1], we simplify refinement proofs by computations of weakest-preconditions [11]. More exactly, we use weakest-*liberal*-preconditions (WLP) because they appear naturally in correctness diagrams of abstract computations (illustrated by Figure 14 below). Fundamentally, this comes from the fact that weakest-liberal-preconditions do not aim to ensure termination of programs – like our static analyzes – contrary to original weakest-preconditions of Dijkstra.

Definition 3 (Weakest-liberal-preconditions) *Given $K \in \mathcal{R}(D, D_\lightning)$, the WLP of K, written here $[K]$, is a function of $\mathcal{P}(D) \to \mathcal{P}(D)$ defined by*

$$[K]P \triangleq \{d \in D \,|\, \forall d' \in D_\lightning, d \xrightarrow{K} d' \Rightarrow d' \in P\}$$

To Simplify Refinement Goals by WLP. The main benefit of WLP is to propagate function computations through sequences of relations. Indeed, WLP transforms a sequence into a function composition: $[K_1\,;K_2]P = [K_1]([K_2]P)$. And, given f a function of type $D \to D$, $[\uparrow f\,]P = \{d \,|\, f(d) \in P\}$. This allows for instance to compute $[\uparrow f_1\,;\uparrow f_2]P$ as $\{d \,|\, f_2(f_1(d)) \in P\}$. To understand the benefit of WLP, let us compare this to the direct definition of "$x \xrightarrow{K_1\,;\,K_2} z$". It induces a formula with an existantial quantifier "$\exists y, x \xrightarrow{K_1} y \wedge y \xrightarrow{\uparrow K_2} z$", which, when K_1 is $\uparrow f_1$, can be simplified into a formula without existential quantifier "$f_1(x) \xrightarrow{K_2} z$". In a sense, WLP computations achieve such a simplification for free. Another benefit of WLP is to perform an implicit normalization of computations, in the sense that $[K]P = [\downarrow K]P$ holds.

We embed WLP computations in refinement proofs using the equivalence between $K_1 \sqsubseteq K_2$ and $\forall P, [K_2]P \subseteq [K_1]P$. We list below WLP of main guarded-commands:

$$[\bot]P = D \qquad [\lightning]P = \emptyset \qquad [\varepsilon]P = P$$

$$[\vdash P']P = P' \cap P \qquad [\dashv P']P = (D \setminus P') \cup P$$

$$\left[\bigsqcup_{a \in A} K_a\right]P = \bigcap_{a \in A} [K_a]P \qquad \left[\bigsqcap_{a \in A} K_a\right]P = \bigcup_{a \in A} [K_a]P$$

The methodology of stepwise refinement relies on the fact that $K_1 \sqsubseteq K_2$ implies $K_1\,;K \sqsubseteq K_2\,;K$ and $K\,;K_1 \sqsubseteq K\,;K_2$. While trying to prove these two properties only from WLP properties above, we observe that the first one derives from $\forall P, [K_2]([K]P) \subseteq [K_1]([K]P)$, itself implied by $K_1 \sqsubseteq K_2$. However, in order to prove the second one, we need to establish an additional property on $[K]$: it is a *monotone predicate transformer*. This means that if $P_1 \subseteq P_2$ then $[K]P_1 \subseteq [K]P_2$.

A Shallow Embedding of WLP Computations. In the style of [3], we use a shallow embedding of WLP computations, meaning that we avoid the introduction of abstract syntax trees for $\mathbb{K}$ computations, which would induce many difficulties because of binders in $\bigsqcup$ and $\bigsqcap$ operators. Instead, we represent $\mathbb{K}$ computations directly as monotone predicate transformers. In other words, our syntax for $\mathbb{K}$ guarded commands is directly provided by a given set of Coq operators on monotone predicate transformers (corresponding to some WLP computations).

Actually, by exploiting type isomorphism $\mathcal{P}(D) \to \mathcal{P}(D) \simeq D \to \mathcal{P}(\mathcal{P}(D))$, we encode monotone predicate transformers as functions $D \to \mathbb{P}(D)$ where $\mathbb{P}$ is the monad of *monotone predicates of predicate* (that we define below). Indeed, they are simpler and more general than monotone predicate transformers: all composition operators of predicate transformers can be derived by combining only *atomic* operators with the $\gg\!=$ operator of monad $\mathbb{P}$. For instance, in Figure 13, the A-indexed meet operator of $\mathbb{K}$ is derived from the atomic operator $\overset{A}{\sqcap}$ of $\mathbb{P}$.

Figure 12 sketches the Coq definitions of monad $\mathbb{P}$. An element of type $(\mathbb{P}\,A)$ is a record with two fields: a field `app` representing a predicate of $\mathcal{P}(\mathcal{P}(A))$, and a field `app_monot` that is a

```
Record ℙ(A:Type) := {
  app:> (A → Prop) → Prop;
  app_monot(P Q:A → Prop): app P → (∀ d, P d → Q d) → app Q}.
```

$$k_1 \,{}^{\mathbb{P}}\!\sqsubseteq k_2 \triangleq \forall P, (k_2\, P) \to (k_1\, P)$$

$${}^{\mathbb{P}}\varepsilon\, a \triangleq \{\mathtt{app} := \lambda P, (P\, a)\} \qquad k_1 \,{}^{\mathbb{P}}\!\gg\!= k_2 \triangleq \{\mathtt{app} := \lambda P, (k_1\, \lambda a, (k_2\, a\, P))\}$$

$${}^{\mathbb{P}}\!\vdash P' \triangleq \{\mathtt{app} := \lambda P, P' \wedge (P\ \mathtt{tt})\} \qquad {}^{\mathbb{P}}\!\dashv P' \triangleq \{\mathtt{app} := \lambda P, P' \to (P\ \mathtt{tt})\}$$

$$\overset{A}{\sqcup} \triangleq \{\mathtt{app} := \lambda P, \forall a\!:\!A, (P\, a)\} \qquad \overset{A}{\sqcap} \triangleq \{\mathtt{app} := \lambda P, \exists a\!:\!A, (P\, a)\}$$

Figure 12: Coq definitions for main operators of monad $\mathbb{P}$

$$\mathbb{K} \triangleq D \to \mathbb{P}\, D \qquad K_1 \sqsubseteq K_2 \triangleq \forall d, (K_1\, d)\ {}^{\mathbb{P}}\!\sqsubseteq (K_2\, d)$$

$$\uparrow f \triangleq \lambda d, {}^{\mathbb{P}}\varepsilon\, (f\, d) \qquad K_1\,;K_2 \triangleq \lambda d, (K_1\, d)\ {}^{\mathbb{P}}\!\gg\!= K_2$$

$$\vdash P' \triangleq \lambda d, {}^{\mathbb{P}}\!\vdash (P'\ d)\ {}^{\mathbb{P}}\!\gg\!= \lambda\_, ({}^{\mathbb{P}}\varepsilon\, d) \qquad \dashv P' \triangleq \lambda d, {}^{\mathbb{P}}\!\dashv (P'\ d)\ {}^{\mathbb{P}}\!\gg\!= \lambda\_, ({}^{\mathbb{P}}\varepsilon\, d)$$

$$\bigsqcup_{a:A} K_a \triangleq \lambda d, \overset{A}{\sqcup}\ {}^{\mathbb{P}}\!\gg\!= \lambda a\!:\!A, (K_a\, d) \qquad \bigsqcap_{a:A} K_a \triangleq \lambda d, \overset{A}{\sqcap}\ {}^{\mathbb{P}}\!\gg\!= \lambda a\!:\!A, (K_a\, d)$$

Figure 13: Coq definitions for main $\mathbb{K}$ operators

proof that `app` is monotone. Here, elements of $(\mathbb{P}\, A)$ are implicitly coerced into functions through field `app`. In Figure 12, each record definition generates a proof obligation for the missing field `app_monot`. Assert (resp. assume) operator of $\mathbb{P}$ monad is written ${}^{\mathbb{P}}\!\vdash P'$ (resp. ${}^{\mathbb{P}}\!\dashv P'$) where P' is of type `Prop`. These operators are of type "$\mathbb{P}$ `unit`" where `unit` is a singleton type which inhabitant is `tt`. Operators $\overset{A}{\sqcap}$ and $\overset{A}{\sqcup}$ are of type $\mathbb{P}\, A$.

A Lightweight Formalization of $\mathbb{K}$ in Coq. Figure 13 illustrates how we derive guarded-commands of $\mathbb{K}$ from operators of $\mathbb{P}$ monad.

With this representation change, a relation Q in $\mathcal{R}(D, D)$ is now embedded in $\mathbb{K}$ as $\overline{Q} \triangleq \bigsqcup_{d' \in D} \dashv \{d \mid d \xrightarrow{Q} d'\}\,; d'$. We can thus still express Hoare specifications (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ by $\vdash P; \overline{Q}$. Hence, we express unbounded iteration by a meet over inductive invariants as explained in Section 2.1.

On the contrary to [3], we have not proved in Coq the properties of $\mathbb{K}$ algebra. On refinement goals, we let Coq compute weakest-preconditions and simply solve the remaining goal with standard Coq tactics. This gives us well-automated proof scripts in practice. Thus, Coq code for $\mathbb{K}$ operators (with $\mathbb{P}$ included) remains very small (around 150 lines, proofs and comments included).

4.3 Representations of Correctness Diagrams

The Coq definition of ${}^{\dagger}\mathbb{K}$ datatype, sketched in Figure 14, is actually parametrized by a structure of may-return monad: abstract computations are functions of ${}^{\sharp}D \to ??{}^{\sharp}D$. Here, ${}^{\sharp}D$ equipped with its operators (satisfying the interface given at Figure 2) is also a parameter of the definition. Thus, our modular design allows to have abstract computations that do handle alarms, like in our toy analyzer, or that do not, like in our linearization procedure. Indeed, in abstract interpreters, detection of runtime errors (and handling of alarm) is generally done at the top-level interpreter

```
Record ⁺𝕂: Type := {
  impl:♯D →??♯D; spec:𝕂;
  impl_correct: ∀♯d♯d′, (impl ♯d) ⇝♯d′ → ∀d, d ∈ γ(♯d) → (spec d γ(♯d′)) }.
```

Figure 14: Sketch of the Coq definition for $^+\mathbb{K}$ datatype

of the analyzer, but not in the internal levels. Our notion of diagram can handle both cases in a generic way.

Therefore, Figure 14 defines values of $^+\mathbb{K}$ as triples with a field `impl` being an abstract computation, a field `spec` being a concrete computation and a field `impl_correct` being a proof that `impl` is correct w.r.t `spec`. Such proofs are simplified by applying together the WLP embedded in `spec` and the WLP already designed by [13]. The latter indeed simplifies reasonings with $\rightsquigarrow$ relation. At last, `impl` being the only informative[10] field of $^+\mathbb{K}$ record, type $^+\mathbb{K}$ is extracted as Ocaml type $^\sharp D \to {}^\sharp D$. Similarly, a $^+\mathbb{K}$ command is extracted exactly as its underlying abstract computation. Here again, the Coq code for $^+\mathbb{K}$ operators (diagrammatic proofs included) is small (around 200 lines, without the implementation of the alarm writer monad).

5 Conclusion & Perspectives

We extended the VPL with certified handling of non-linear multiplications by a modular and novel design. Our computations are performed by an untrusted oracle delivering a certificate to a certified front-end. Our proofs use diagrammatic constructs based on stepwise refinement calculus. Refinement proofs are finally made clear and concise by the computations of Weakest-Liberal-Preconditions.

Our linearization procedure is able to give a fast over-approximation of integer polynomials thanks to variable intervalization. The precision is increased by domain partitioning (implicitly done with a Continuation-Passing-Style design) and the dynamic computation of bounding affine terms, enabling to finely tune the precision-versus-efficiency trade-off in the oracle.

Because floating arithmetic requires to explicitly handle error terms at each operation, VPL does not currently support floating points variables, and our linearization neither. Most non-linear arithmetic used in real-life programs involve floating points. Therefore, it is hard to evaluate our method on real-life programs. Hence, our experiments are limited to small handmade examples inspired by polynomials often encountered in real-life code, such as parabola or barycentres. On these cases, our oracle is able to give much more precise approximations than the Verasco interval domain.

Our linearization procedure needs also to be extended with others arithmetic operators like integer division and integer modulo. A simple approach in this direction would: 1) replace each call to these operators by a fresh temporary variable; 2) express the meaning of these operations as non-deterministic assignments of their corresponding variables, using only polynomials, *i.e.* if t_1 and t_2 are positive then $q := t_1/t_2$ is replaced by $q :\in \{x \mid t_1 - t_2 < x \times t_2 \leq t_1\}$; 3) eliminate temporary variables out of approximated guards. The VPL already provides the bricks for such an approach.

At last, we certified a toy analyzer from big-steps semantics of Figure 1, by interpreting the operators of concrete semantics in abstract semantics, according to the correspondence of Figure 3. We detailed how this toy analyzer handles alarms in the style of Verasco. This could give some hints to adapt Jourdan's framework for Verasco [17] with impure operators on abstract domains and some dynamic strategies of trace partitioning.

[10] In Coq jargon, something is "informative" if it is "not a piece of proof" (thus, it remains at extraction).

References

[1] Back, R.J., von Wright, J.: Refinement calculus - a systematic introduction. Graduate texts in computer science. Springer (1999)

[2] Besson, F., Jensen, T.P., Pichardie, D., Turpin, T.: Certified result checking for polyhedral analysis of bytecode programs. In: TGC, pp. 253–267 (2010)

[3] Boulmé, S.: Intuitionistic Refinement Calculus. In: TLCA, *LNCS*, vol. 4583. Springer (2007)

[4] Boulmé, S.: What is the Foreign Function Interface of the Coq Programming Language? Talk at the Coq Workshop 2018 (2018). URL https://coqworkshop2018.inria.fr/files/2018/07/coq2018_talk_boulme.pdf

[5] Boulmé, S., Maréchal, A.: Refinement to Certify Abstract Interpretations, Illustrated on Linearization for Polyhedra. In: ITP, *LNCS*, vol. 9236. Springer (2015)

[6] Boulmé, S., Maréchal, A.: Toward Certification for Free! Preprint on HAL (2017). URL https://hal.archives-ouvertes.fr/hal-01558252

[7] Braibant, T., Pous, D.: Deciding Kleene Algebras in Coq. Logical Methods in Computer Science **8**(1) (2012)

[8] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. TCS **277**(1-2) (2002)

[9] 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)

[10] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. ACM (1978)

[11] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)

[12] Farouki, R.T.: The Bernstein polynomial basis: A centennial retrospective. Computer Aided Geometric Design **29**(6) (2012)

[13] Fouilhé, A., Boulmé, S.: A certifying frontend for (sub)polyhedral abstract domains. In: VSTTE, *LNCS*, vol. 8471. Springer (2014)

[14] Fouilhé, A., Monniaux, D., Périn, M.: Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In: SAS, vol. 7935. Springer (2013)

[15] Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in Coq. In: TPHOL, *LNCS*, vol. 3603, pp. 98–113. Springer (2005)

[16] Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. Pacific Journal of Mathematics **132**(1) (1988)

[17] Jourdan, J.H.: Verasco: a Formally Verified C Static Analyzer. Theses, Universite Paris Diderot-Paris VII (2016). URL https://hal.archives-ouvertes.fr/tel-01327023

[18] Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL. ACM (2015)

[19] Laporte, V.: Verified static analyzes for low-level languages. Theses, Université Rennes 1 (2015). URL https://tel.archives-ouvertes.fr/tel-01285624

[20] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009)

[21] Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: ESOP, vol. 1058, pp. 219–234. Springer (1996)

[22] Maréchal, A.: New algorithmics for polyhedral calculus via parametric linear programm ing. Ph.D. thesis, Université Grenoble Alpes (2017)

[23] Maréchal, A., Fouilhé, A., King, T., Monniaux, D., Périn, M.: Polyhedral approximation of multivariate polynomials using Handelman's theorem. In: VMCAI, pp. 166–184 (2016)

[24] Maréchal, A., Périn, M.: Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra. Tech. Rep. TR-2014-7, Verimag Research Report (2014)

[25] Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP'05, *LNCS*, vol. 3444 (2005)

[26] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI, *LNCS*, vol. 3855. Springer (2006)

[27] Morgan, C.: Programming from specifications, 2nd Edition. Prentice Hall International series in computer science. Prentice Hall (1994)

[28] Moscato, M.M., Muñoz, C.A., Smith, A.P.: Affine arithmetic and applications to real-number proving. In: ITP, *LNCS*, vol. 9236. Springer (2015)

[29] Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation **6**(3-4) (1993)

[30] Spiwack, A.: Abstract interpretation as anti-refinement. CoRR **abs/1310.4283** (2013). URL http://arxiv.org/abs/1310.4283

[31] The Coq Development Team: The Coq proof assistant reference manual – version 8.4. INRIA (2012-2014)

[32] Wadler, P.: Monads for functional programming. In: AFP, *LNCS*, vol. 925. Springer (1995)