Toward Certification for Free!

Sylvain Boulmé, Alexandre Maréchal

Toward Certification for Free!*

Correct-By-Construction ML Oracles with Polymorphic LCF Style

SYLVAIN BOULMÉ and ALEXANDRE MARÉCHAL, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

How can we reduce the required effort to develop certified programs in proof assistants such as Coq? A major trend is to introduce untrusted *oracles* able to justify their answers by producing a *certificate*, i.e. a witness of their computations. A trustworthy result is then built from this certificate by a certified checker. This alleviates the burden of proof. However, generating certificates may lead to obfuscate the oracle code.

We present *Polymorphic Factory Style*, a design pattern that removes the need for certificates while preserving confidence in the computations: oracles directly compute a trusted result by invoking certified operators and datastructures extracted from Coq. ML oracles only handle these datastructures as polymorphic values, which forbids them to forge incorrect results. Part of the certification relies on parametricity and is delegated to the ML typechecker. We demonstrate the relevance of Polymorphic Factory Style for the certification of a realistic library: an abstract domain of convex polyhedra.

CCS Concepts: • **Software and its engineering** → **Polymorphism**; • **Theory of computation** → *Program verification*; *Invariants*; *Type theory*; • **Mathematics of computing** → Solvers;

Additional Key Words and Phrases: Abstract Domain of Polyhedra, Coq, Linear Programming, Parametricity.

1 CONTEXT AND CONTRIBUTIONS

Our concern is the formal certification of programs in a proof assistant like Coq. We propose a new programming style which is particularly adapted for certifying the results of solvers which solutions are hard to discover but easy to verify. Our approach reduces the development effort in Coq by delegating part of the verification to the OCaml typechecker. We will illustrate our design on the development of a realistic library: a certified abstract domain of polyhedra. First, let us summarize the three major trends for certifying programs within Coq: autarkic, skeptical with certificates, and skeptical LCF style. We evaluate their Trusted Computing Base (TCB), i.e. the volume of code they need to trust. We also describe how they combine with OCaml and their ability to use imperative datastructures for better performance.

Autarkic Approach. A program can be directly implemented and proved correct in Coq following the *autarkic* approach [Barendregt and Barendsen 2002]. This approach provides safety without runtime verifications and reduces the TCB to the Coq proof-checker only (see Figure 1(*a*)). The program can furthermore be used outside of Coq thanks to a built-in Coq process named *extraction*, which exports the Coq code into OCaml– at the price of adding both the extraction process and the OCaml compiler into the TCB. The development of an efficient implementation following the autarkic approach is both very development-time consuming and restrictive, since it forbids the use of efficient imperative datastructures.

*In reference to the seminal "*Theorems for Free!*" of Wadler [1989].

This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR", and by the French national research organization ANR through "Vocal" (grant ANR-15-CE25-0008).

Authors' address: Sylvain Boulmé, sylvain.boulme@univ-grenoble-alpes.fr; Alexandre Maréchal, alex.marechal@univ-grenoble-alpes.fr, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France.

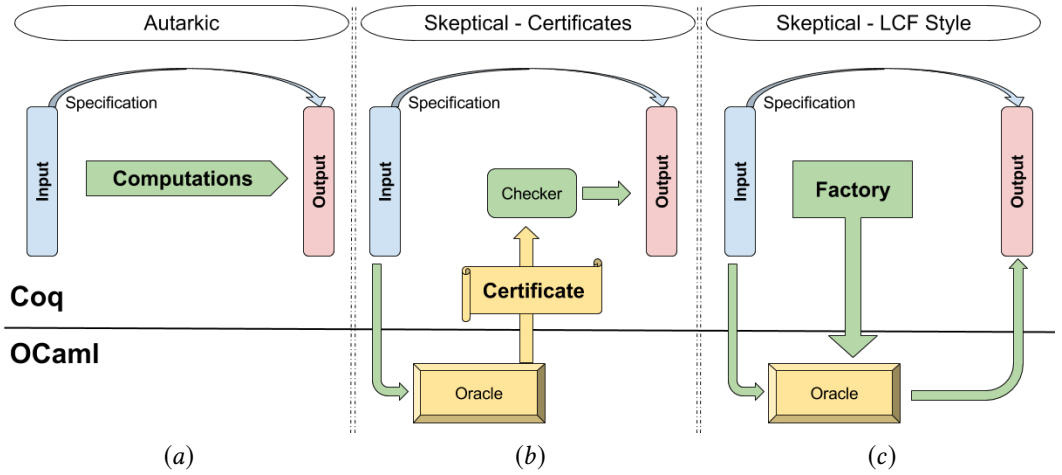


Fig. 1. Three methods of certified computations: autarkic, certificate-driven and with LCF Style.

Skeptical Approach with Certificates. The program can rather be implemented in a two-tier architecture: (1) an OCaml backend provides an untrusted oracle that performs efficient but unproved computations, and generates *certificates* driving the reconstruction of a trusted result; (2) a Coq frontend uses the backend oracle and certificates to provide the certified result (see Figure 1(*b*)). The frontend is then extracted from Coq into OCaml and the whole software – backend and frontend – is finally compiled by OCaml into binaries. Therefore, the TCB of this approach contains the Coq proof-checker, its extraction process, and the OCaml compiler.

This approach requires the definition of a Coq certificate format and a Coq checker able to interpret and verify it. Certificate generation does not guarantee the absence of bugs in the oracle. This approach ensures however a *partial correctness* property: when the oracle terminates and provides a well-formed certificate, the frontend can build a certified result that satisfies the *formal specification* of the program. On the other hand, if the oracle provides an ill-formed certificate, it will not pass the check and the frontend can fail or return a trivially correct – but weak – result.

Skeptical Method with LCF Style. For complex datastructures, certificates can be tricky to manipulate, in particular for nested computations. In order to completely avoid the handling of certificates, we are tempted by another style of skeptical certification, called LCF style. The name "LCF" stands for *Logic for Computable Functions* – a prover at the origin of ML where theorems were handled through an abstract datatype [Gordon et al. 1978]. This LCF style is still at the heart of HOL provers. It is much lighter than the preceding one, because it avoids the introduction of a certificate format for representing certified computations. Instead, the OCaml oracle directly performs trusted computations by using a *factory* of operations (the "Factory" of Figure 1(*c*)) that are implemented and certified in Coq before extraction. The key idea is that such a factory can only build logical consequences of some given set of axioms.

This style of certification relies on one assumption: the frontend can trust results produced by an external oracle that uses its certified operators. However, in presence of references, type abstraction is generally not sufficient on its own to ensure this property: for example, nothing prevents an imperative OCaml program from cheating by returning a result obtained from a previous run. This naive LCF style could therefore be unsound for certification (wait Section 3.1 for details), but introducing parametric polymorphism will elegantly avoid this flaw.

A Sound LCF Style by Parametric Polymorphism. This paper builds upon the work of Boulmé and Vandendorpe [2019] that uses LCF style to safely embed imperative programs into a Coq development. Their approach allows the developer to never formally reason about effects of imperative functions, but only about their results. Formal guarantees are obtained by combining *parametric reasoning over polymorphic functions* with *verified defensive programming*. This follows the work from Wadler [1989] who showed that "theorems for free" are derivable from the type of polymorphic functions. For instance, in the purely functional setting of Wadler [1989], a function of type `'a -> 'a` is necessarily the identity. In the imperative setting of Boulmé and Vandendorpe [2019], if a function of type `'a -> 'a` returns a result, then this result equals the input parameter; but, the function may also not terminate normally, or produce arbitrary side-effects while returning its result.

Boulmé and Vandendorpe aimed at certifying a SAT-solver, a program that checks the satisfiability of a Boolean formula in conjunctive normal form (a conjunction of *clauses*, i.e. disjunctions of literals). Modern SAT-solvers are often required to produce proofs of unsatisfiability, which can be shown by a succession of resolutions yielding $\bot$. The resolution operator combines two clauses containing complementary literals ($C_1 \vee l,\ C_2 \vee \neg l$) to produce the logical consequence $C_1 \vee C_2$. In particular, ($l,\ \neg l$) entails $\bot$. In order to certify unsatisfiability while still benefiting from complex solver optimizations, the authors implemented a Coq type for clauses and a resolution operation[1] proved to build only logical consequences of its operands. The solver can then perform the difficult part, that is finding the good resolutions that yields $\bot$, and replay those with the extracted certified resolution operator. Finally, the unsatisfiability is verified if the certified clause built by the oracle is syntactically $\bot$.

Contributions and Overview. In this paper, we report on the application of the LCF style of Boulmé and Vandendorpe [2019] to the development of a realistic library: a certified abstract domain of polyhedra for static analysis. Through this case study, the paper presents a new programming style, called *Polymorphic Factory Style* (PFS), for developing correct-by-construction oracles in a skeptical approach. We will show that polymorphism is a simple and efficient way to solve the soundness issue of naive LCF style. Results produced by oracles are correct once the operators of the factory are proved to preserve the desired property. [2]

After introducing the domain of polyhedra, the paper presents PFS on increasingly complex operators. We will begin in Section 2 with a simple application of PFS on a Boolean operator checking the emptiness of a polyhedron. This case is very similar to the unsatisfiability test of Boolean formulas mentioned previously. Section 3 details PFS on the projection operator, which transforms a polyhedron into another one by eliminating some variables from it. This illustrates that PFS is not limited to unsatisfiability tests: it can also transform a datastructure by applying operations that preserve a given invariant. This will be our archetype example, on which we will show why naive LCF style could be unsound and compare PFS to various alternatives. We will explain how correctness proofs are expressed in Coq from the type of PFS oracles in Section 4. Section 5 illustrates the expressiveness of polymorphic factories on a binary operator that computes the convex union of two polyhedra. Finally, Section 6 shows that using an adequate factory, it is possible to generate compact certificates for embedding our oracles within Coq tactics.

Through the application of polymorphic factories to four examples, we hope to demonstrate that the modular and expressive design of PFS simplifies certified developments while allowing optimization and instrumentation.

[1] To get smaller proofs, they actually use a more complex backward n-ary resolution rule, but the proof principle is similar.
[2] Our source code is available at http://github.com/VERIMAG-Polyhedra/VPL.

2 INTRODUCTORY EXAMPLE: EMPTINESS TEST

This section presents our case study, the certification of an abstract domain of polyhedra. Then, it introduces PFS on the emptiness test operator (`is_empty`) of the domain. This is a direct adaptation of the unsatisfiability check of Boolean formulas mentioned in Section 1 to convex polyhedra. We will certify `is_empty` by exhibiting a combination of constraints that yields a contradiction, in the same way that clauses are combined by resolution to obtain $\bot$.

2.1 Certifying an Abstract Domain of Polyhedra

A static analyzer may be used to prove the absence of runtime errors – such as arithmetic or memory overflows – in all possible executions of a program. In abstract interpretation [Cousot and Cousot 1977], the analyzer attaches to each program point an *invariant*, which is a property satisfied by all reachable states at this point. These invariants belong to classes of predicates called *abstract domains* that must provide operators for computing the disjunction of two invariants (`join`), their conjunction (`meet`), and the existential quantification of a variable in an invariant (`proj`). They must also provide tests for implication between invariants (`is_included`) and unsatisfiability (`is_empty`).

A static analyzer is *correct*[3] provided that it does not miss any reachable state. This requirement has repercussions on abstract domains: to formally ensure the analyzer correctness, all operators on abstract domains must be proved to compute overapproximations. In particular, we do not need to prove that operators are precise (i.e. they compute tight results), even though they are in practice.

In the following, we focus on the abstract domain of convex polyhedra on $\mathbb{Q}$ [Cousot and Halbwachs 1978], which is able to handle linear relations between numerical variables $\boldsymbol{x} \triangleq (x_1, \ldots, x_n) \in \mathbb{Q}^n$. For simplicity, we do not consider integer or floating point variables in this paper. A *convex polyhedron* is a conjunction of linear constraints of the form $\sum_i a_i x_i \bowtie b$ where a_i, b are constants in $\mathbb{Q}$ and $\bowtie$ is $\geq$, $>$ or $=$. A polyhedron is represented as a list of `Cstr.t`, which is the type of linear constraints.[4]

A lot of libraries feature polyhedral calculus, but only a few certify their results. The Coq-Polyhedra library [Allamigeon and Katz 2017] follows the autarkic approach; according to their authors, the goal of this library is not to perform efficient computations, but to formalize a large part of the convex polyhedra theory by using reflexive proofs. Fouilhé et al. [2013] developed the Verified Polyhedra Library (VPL), an abstract domain for the certified Verasco static analyzer [Jourdan et al. 2015], by using uncertified code through a *skeptical* approach. Unlike most polyhedra libraries, VPL uses the constraints-only representation of polyhedra in order to ease its certification in Coq.[5] Implementing this skeptical approach requires first to introduce a certificate format that captures the information needed to prove the correctness of the polyhedral operators. Fortunately, proving their correctness reduces to verifying implications[6] between polyhedra, in conjunction with other simple verifications that depend on the operator. For example, polyhedron P is empty iff $P \Rightarrow P_\emptyset$, where $P_\emptyset$ is a single contradictory constant constraint such as $0 \geq 1$. The emptiness of $P_\emptyset$ is thus itself checkable by a simple rational comparison.

Farkas' lemma gives a simple way to prove polyhedral implications [Farkas 1902]. It states that any nonnegative linear combination of the constraints of a polyhedron P is an obvious logical

[3] We make a distinction between correctness and soundness. We say that a program or an operator is *correct* w.r.t. a given specification. We rather use *soundness* to qualify the sanity of an approach. In particular, a correction proof can be unsound, as we will see in Section 3.1.

[4] As we only deal with convex polyhedra, the adjective convex is often omitted in the remaining of the paper.

[5] Most polyhedra libraries maintain a double representation of polyhedra as *constraints* and as *generators*, i.e. vertices and rays. Certifying them would require to prove the correctness of Chernikova's conversion algorithm. Instead, Fouilhé looked for efficient polyhedra operators in constraints-only representation.

[6] Note that a polyhedral implication $P_1 \Rightarrow P_2$ is geometrically an inclusion between polyhedra $P_1 \subseteq P_2$.

consequence of P. For instance, $x \geq 3 \wedge y \geq 0$ implies $2 \cdot (x \geq 3) + 1 \cdot (y \geq 0) = 2x + y \geq 6$, meaning that any point satisfying $x \geq 3 \wedge y \geq 0$ also satisfies $2x + y \geq 6$. Moreover, Farkas' lemma states that any polyhedral implication can be proved thanks to such simple computations on constraints and thus provides a theoretical foundation for designing the certificate format of polyhedral operators [Besson et al. 2010]. The formulation below is restricted to polyhedra with non strict inequalities only, and Section 5.1 will provide a generalization to polyhedra with equalities and strict inequalities.

Lemma 2.1 (Farkas 1902). *Let P_1 and P_2 be two polyhedra containing only non strict inequalities. Let us call* Farkas combination *of P_1 any nonnegative linear combination of P_1 constraints. Any* Farkas combination *of P_1 is a logical consequence of P_1. Moreover, if $P_1 \Longrightarrow P_2$ then*

- *either P_1 is empty and there exists a Farkas combination of P_1 producing the contradictory constraint $0 \geq 1$,*
- *or each constraint of P_2 is a Farkas combination of P_1.*

For instance, the polyhedron $x \geq 3 \wedge y \geq 0 \wedge -2x - y \geq -5$ is empty, as shown by the combination $2 \cdot (x \geq 3) + 1 \cdot (y \geq 0) + 1 \cdot (-2x - y \geq -5) = 0 \geq 1$. In general, finding the right combination requires a Linear Programming (LP) solver [Chvatal 1983].

2.2 Emptiness Test Certification

We now illustrate our method for developing a certified abstract domain on operator `is_empty` of type `Cstr.t list -> bool`. This operator checks the existence of a point of $\mathbb{Q}^n$ satisfying the constraints of the input polyhedron. First, let us see how it is implemented in A. Fouilhé's VPL, following the skeptical approach with certificates.

Here, the OCaml oracle for `is_empty` returns a certificate as the list of coefficients of a Farkas combination that gives $0 \geq 1$. The OCaml type of the oracle is thus

```
Back.is_empty: Cstr.t list -> Cert.t option
```

where the `None` answer means that the input polyhedron is not empty, and a `Some` answer gives a certificate of type `Cert.t` allowing the frontend to establish the polyhedron emptiness. From `Cert.t`, the frontend computes the result of the combination with its own certified Coq datastructures of constraints and obtains $0 \geq 1$ (if nothing went wrong in the oracle).

In actual fact, the VPL certificates of Fouilhé and Boulmé [2014] are more complex than sketched above (wait Section 3.2 for more details). And, in an informal discussion, A. Fouilhé stated that the code generating them was particularly difficult to develop and debug. He concluded that simplifying this process would be helpful. Hence, let us try to implement our operator in LCF style.

In LCF style, the oracle has two versions of each constraint: (1) an untrusted one of type `BackCstr.t`, combining complex datastructures and GMP rationals; (2) a second one of type `FrontCstr.t`, extracted from Coq, on which the oracle can only apply factory operators. Given an empty polyhedron, the backend uses an untrusted solver to efficiently find the contradictory Farkas combination and then builds a certified combination of type `FrontCstr.t` using the factory operators extracted from Coq. Then, the frontend only has to check that the resulting constraint is syntactically $0 \geq 1$. In the backend, `is_empty` has the type

```
Back.is_empty: (BackCstr.t * FrontCstr.t) list -> FrontCstr.t option
```

However, this naive LCF operator is not safe. An imperative OCaml program could cheat by returning a contradictory constraint obtained from a previous run, and making `FrontCstr.t` opaque from the oracle point of view is not sufficient to prevent this. Such an unsoundness is not present in standard LCF provers. They avoid it at the price of a greater complexity in the implementation of the

"*factory of theorems*", where they represent theorems by sequents (thus, recording the hypotheses of the theorem in addition to its conclusion). This discussion will be detailed in Section 3.3.

Alternatively, we introduce PFS, which consists in abstracting the certified datatype `FrontCstr.t` by a polymorphic type `'c` in the oracle. Roughly speaking, the key idea is that the polymorphic nature of this type prevents a result from being stored and reused later: it is forbidden by the OCaml typechecker (more details in Section 3.4). The oracle is provided with operators on this datatype grouped in a factory of type `'c lcf`. In our case, this factory contains what is necessary to combine constraints, namely an addition between two constraints and a multiplication of a constraint by a coefficient. Polymorphism ensures that the oracle can only produce correct results by combining its inputs using the operators of the factory. The type of the `is_empty` oracle becomes

```
Back.is_empty: 'c lcf -> (BackCstr.t * 'c) list -> 'c option
```

Polymorphism of PFS is a simple and efficient way to solve the soundness issue of naive LCF style. Results produced by oracles are correct by construction provided that the operators of the factory preserve the desired correctness property. In the following, we detail the content of the factory and how it applies on the projection of polyhedra.

3 PFS ORACLES EXPLAINED WITH A DETAILED EXAMPLE: THE PROJECTION

This section gives a tutorial on PFS oracles, illustrated on operator `proj` of the abstract domain of polyhedra. This operator performs the elimination of existential quantifiers on polyhedra. More precisely, given a polyhedron P and a variable x, ($\texttt{proj}\, P\, x$) computes a polyhedron P' such that $P' \Leftrightarrow \exists x, P$. Let us consider the example of Figure 2. Predicate P_0 expresses that q is the result of the Euclidean division of x by 3, with r as remainder. Predicate P_1 "instantiates" P_0 with $x = 15$. Then, predicate P'_1 corresponds to the computation of $\exists r, P_1$ (as a polyhedron on $\mathbb{Q}$).

$$P_0 \triangleq \begin{cases} & x = 3 \cdot q + r & [C_1] \\ \wedge & r \geq 0 & [C_2] \\ \wedge & r < 3 & [C_3] \end{cases} \qquad P'_1 \triangleq \begin{cases} & x - 15 = 0 & [C'_1] \\ \wedge & q - 4 > 0 & [C'_2] \\ \wedge & 5 - q \geq 0 & [C'_3] \end{cases}$$

$$P_1 \triangleq P_0 \wedge x = 15 \qquad [C_4]$$

Fig. 2. Computation of P'_1 as "$\texttt{proj}\, P_1\, r$"

Geometrically, `proj` $P\, x$ represents the orthogonal projection of a polyhedron P according to direction x. The standard algorithm for computing this projection is Fourier-Motzkin elimination [Fourier 1827]. Ongoing research is trying to improve efficiency with alternate algorithms [Howe and King 2012; Maréchal et al. 2017]. But in our two-tier approach, the correctness proof of `proj` does not need to consider these implementation details.

As mentioned in Section 2.1, we assume that for proving the correctness of our surrounding software (typically, a static analyzer), we do not need to prove $P' \Leftrightarrow \exists x, P$ but only $(\exists x, P) \Rightarrow P'$. Thus, we only want to prove the correctness of `proj` as defined below.

Definition 3.1 (Correctness of `proj`*).* Function `proj` is correct iff any result P' for a computation ($\texttt{proj}\, P\, x$) satisfies $(P \Rightarrow P') \wedge x \notin V(P')$ where $V(P')$ is the set of variables appearing in P' with a non-null coefficient.

The condition $x \notin V(P')$ ensures that variable x is no longer bounded in P'. As dynamic checking of this condition is fast and easy, we only look for a way to build P' from P which ensures by construction that $P \Rightarrow P'$. For this purpose, we exploit Farkas' lemma as follows. Internally, we

handle constraints in the form "$t \bowtie 0$" where t is a linear term and $\bowtie \in \{=, \geq, >\}$. Hence, each input constraint "$t_1 \bowtie t_2$" is first normalized as "$t_1 - t_2 \bowtie 0$". Then, we generate new constraints using only the two operations of Definition 3.2. Obviously, such constraints are necessarily implied by P.

Definition 3.2 (Linear Combinations of Constraints). We define operations + and · on normalized constraints by

- $(t_1 \bowtie_1 0) + (t_2 \bowtie_2 0) \triangleq (t_1 + t_2) \bowtie 0$
  where $\bowtie \triangleq \max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $=, \geq, >$.
- $n \cdot (t \bowtie 0) \triangleq (n \cdot t) \bowtie 0$
  under preconditions $n \in \mathbb{Q}$ and, if $\bowtie \in \{\geq, >\}$ then $n \geq 0$.

For example, P'_1 is generated from P_1 by the script on the right hand-side. Here `tmp` is an auxiliary constraint, where variable x has been eliminated from C_1 by rewriting using equality C_4.

$$\begin{array}{rcl} \texttt{tmp} & \leftarrow & C_4 + -1 \cdot C_1 \\ C'_1 & \leftarrow & C_4 \\ C'_2 & \leftarrow & \frac{1}{3} \cdot (C_3 + \texttt{tmp}) \\ C'_3 & \leftarrow & \frac{1}{3} \cdot (C_2 + -1 \cdot \texttt{tmp}) \end{array}$$

In the following, we study how to design – in OCaml– a certified frontend `Front.proj` that monitors Farkas' combinations produced by an untrusted backend `Back.proj`. Section 4 will then formalize `Front.proj` in Coq.

3.1 Naive but Unsound LCF Style

In a first step, we follow the LCF style introduced in Section 2.2. We thus consider two datatypes for constraints: modules `BackCstr` and `FrontCstr` define respectively the representation of constraints for the backend and the frontend.

Each module is accessed both in the backend and in the frontend, but the frontend representation is abstract for the backend. Hence, the visible interface of `FrontCstr` for the backend is given on the right-hand side. Type `Rat.t` represents set $\mathbb{Q}$, and `add` and `mul` represent respectively operators + and · on constraints.

```
module FrontCstr: sig
  type t
  val add: t -> t -> t
  val mul: Rat.t -> t -> t
end
```

Going back to our example, P'_1 is firstly computed from P_1 using backend constraints. Indeed, with its own representation, the backend finds the solution by efficient computations, combining complex datastructures, GMP rationals and even floating-point values. On the contrary, the frontend representation is based on certified code extracted from Coq. In particular, it uses internally the certified rationals of the Coq standard library, where integers are represented as lists of bits. Once a solution is found, the backend thus rebuilds this solution in the frontend representation. The easiest way is to make `Back.proj` compute the certified constraints (of type `FrontCstr.t`) in parallel of its own computations. Hence, we propose a first version of `Back.proj`, called `Back.proj0`, with the following type.

```
Back.proj0: (BackCstr.t * FrontCstr.t) list -> Var.t -> FrontCstr.t list
```

Let us define two functions: a certified function `occurs: Var.t -> FrontCstr.t -> bool` such that `occurs x c` tests whether $x \in V(c)$ and an untrusted function `export: FrontCstr.t -> BackCstr.t` that converts a frontend constraint into a backend one. Then, we implement `Front.proj` as follows:

```
let Front.proj (p: FrontCstr.t list) (x: Var.t): FrontCstr.t list =
  let bp = List.map (fun c -> (export c, c)) p in
  let p' = List.map snd (Back.proj0 bp x) in
  if List.exists (occurs x) p'
  then failwith "oracle error"
```

```
  else p'
```

`Front.proj` only dynamically checks that $x \notin P'$. In particular, it does not verify that $P \Longrightarrow P'$ holds, because it should follow directly from the correctness of `FrontCstr.add` and `FrontCstr.mul`. Ideally – mimicking a LCF-style prover – function `Back.proj0` uses type `FrontCstr.t` as a type of theorems. It derives logical consequences of a list of constraints (of type `FrontCstr.t`) by combining them with `FrontCstr.mul` and `FrontCstr.add`. Like in a LCF-style prover, there is no explicit "proof object" as value of this theorem type.

Unfortunately, this approach is unsound. We now provide an example which only involves two input polyhedra that are reduced to a single constant constraint. Let us imagine an oracle wrapping function `memofst` given below. Assuming that it is first applied to the unsatisfiable constraint $0 \geq 1$, this first call returns $0 \geq 1$, which is a correct answer. However, when it is then applied to the satisfiable constraint $2 \geq 0$, this second call still returns $0 \geq 1$, which is now incorrect! This unsoundness is severe, because even a faithful programmer could, by mistake, implement such a behavior while handling mutable datastructures.

```
let memofst:FrontCstr.t -> FrontCstr.t =
  let first = ref None in
  fun c ->
    match !first with
    | None -> (first := Some c); c
    | Some c' -> c'
```

3.2 Generating an Intermediate Certificate

In order to avoid the unsoundness issue of the naive LCF style, we could instead introduce an intermediate datastructure representing a trace of the backend computation. Then, the frontend would use this trace to rebuild the certified result using its own certified datastructures. Such a trace has the form of an Abstract Syntax Tree (AST) and is called a *certificate*. This approach was used to design the first version of the VPL [Fouilhé and Boulmé 2014; Fouilhé et al. 2013]. In the following, we detail the process of certificate generation and why we prefer avoiding it.

We define below a certificate type named `pexp`. It represents a type of polyhedral computations, and depends on type `fexp` that corresponds to Farkas combinations. Constraints are identified by an integer. Type `pexp` provides a `Bind` construct for computing auxiliary constraints like `tmp` in the example of P'_1.

```
type fexp =
  | Ident of int
  | Add of fexp * fexp
  | Mul of Rat.t * fexp
```

```
type pexp =
  | Bind of int * fexp * pexp
  | Return of fexp list
```

Figure 3 gives an example of certificate for P'_1, where each input constraint C_i is represented by "`Ident` i". The intermediate constraint `tmp` is bound to identifier 5.

```
Bind (5, Add (Ident 4, Mul (−1, Ident 1)),
      Return [ Ident 4;
               Mul (1/3, Add (Ident 3, Ident 5));
               Mul (1/3, Add (Ident 2, Mul (−1, Ident 5))) ])
```

Fig. 3. A certificate for P'_1

Next, we easily implement in Coq a `Front.run` interpreter of `pexp` certificates (corresponding to the "checker" part of Figure 1) and prove that it only outputs a logical consequence of its input polyhedron.

```
Front.run: pexp -> (FrontCstr.t list) -> (FrontCstr.t list)
```

Let us precise that when a `pexp` uses certificate identifiers that have no meaning w.r.t. `Front.run`, this latter fails. For the following, we do not need to precise how identifiers are generated and attached to constraints. We let this implementation detail under-specified.

Now, we need to turn `Back.proj0` into a function `Back.proj1` where each `BackCstr.t` constraint in input is associated to a unique identifier.

```
Back.proj1: (BackCstr.t * int) list -> Var.t -> pexp
```

However, `Back.proj1` is more complex to program and debug than `Back.proj0`. Indeed, in LCF-style, certified operations run in "parallel" of the oracle. On an oracle bug (for instance, if the oracle multiplies an inequality by a negative scalar), the LCF-checker raises an error right at the point where the bug appears in the oracle: this makes debugging of oracles much easier. On the contrary, in presence of an ill-formed certificate, the developer has to find out where does the ill-formness come from in its oracle. Moreover, an oracle like `Back.proj1` needs to handle constraint identifiers for `Bind` according to their semantics in `Front.run`. As detailed in Section 5, this is particularly painful on Fouilhé's implementation of the `join` operator, because a binary operator involves two spaces of constraint names (one for each "implication proof"). In the following, we present two solutions that fix the issue of naive LCF style.

3.3 Standard LCF Style

Intuitively, the lying function `memofst` of Section 3.1 exploits the fact that constraints of the result P' are typed with a single type of "theorems", whereas these "theorems" are relative to a given set of axioms: the input constraints of P. The standard LCF style fixes this issue by memorizing in the type of "theorems" the set of axioms in which these theorems has been derived. In other words, in standard LCF style, a Farkas combination is encoded by a sequent "$P \vdash C$" where P is a polyhedron and C a constraint: P is the polyhedron to which the Farkas combination is applied and C is the result of the Farkas combination. This enables the front-end to dynamically check that oracles do not mix these sequents in an unsound way.

Figure 4 sketches a standard LCF style implementation in OCaml. For the sake of simplicity, this implementation uses `BackCstr.t` as an internal representation of constraints: a sequent "$P \vdash C$" is a encoded as a pair (P, C) where P is a list of constraints and C a constraint. This implementation thus wraps operations of `BackCstr` module, but with defensive verifications ensuring that such a pair (P, C) does always satisfy the invariant "$P \Rightarrow C$".

Finally, some additional checks are necessary in front-end operations. For example, for the `is_empty` operation of Section 2, `FrontCstr` provides an operation `proves_unsat` enabling to prove that a given polyhedra P_1 is UNSAT from a sequent "$P_2 \vdash C$" where C is trivially UNSAT: to this end, `proves_unsat` additionally checks that P_1 and P_2 are syntactically equal.

In standard LCF style, an oracle can still use the `memofst` function. But this will be detected at runtime and rejected by the frontend. As explained below, Polymorphic Factory Style (PFS) improves this by preventing the cheating `memofst` at compile-time (by static typechecking). Moreover it makes the implementation of the front-end even more lightweight, since some defensive checks are removed.

```
module FrontCstr: sig

  (* restricted interface for Backend computations *)
  type t
  val add: t -> t -> t
  val mul: Rat.t -> t -> t

  (* extended interface for Frontend computations *)
  val import: BackCstr.t list -> t list
  val export: t list -> BackCstr.t list
  val proves_unsat: BackCstr.t list -> t -> bool
  ...
```

```
module FrontCstr = struct

  type t = BackCstr.t list * BackCstr.t
  let add (p1,c1) (p2,c2) =
      assert (p1 == p2); (p1, BackCstr.add c1 c2)
  let mul r (p,c) =
      assert (Rat.is_nonnegative r); (p, BackCstr.mul r c)

  let import p = List.map (fun c -> (p,c)) p
  let export p = List.map snd p
  let proves_unsat p1 (p2,c) =
      assert (p1 == p2); BackCstr.is_unsat c
  ...
```

Fig. 4. Interface and (Sound) Implementation of `FrontCstr` in Standard LCF Style

3.4 Polymorphic Factory Style

The principle of PFS is very simple: instead of abstracting the "type of theorems" (i.e. type `FrontCstr.t`) using an ML abstract datatype, we abstract it using ML polymorphism. As explained above, the lying function `memofst` of Section 3.1 exploits the fact that we have a *static* type of theorems, whereas when we interpret constraints of the result P' as theorems, they are relative to the input constraints of P. Hence, this issue would disappear by using instead a *dynamic* type, generated at each call to the oracle. Using ML polymorphism, we actually express that our oracle is parameterized by any of such dynamic type of theorems.

In practice, the type `FrontCstr.t` used in backend oracles – e.g. `Back.proj` – is replaced by `'c`. In order to allow the backend to build new "theorems" – i.e. Farkas combinations – we introduce a polymorphic record type `lcf` (acronym of *Logical Consequences Factory*).

```
type 'c lcf = {
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c
}
```

Then, the previous oracle `Back.proj0` that we defined for the simple LCF style is generalized into

```
val Back.proj: 'c lcf -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

Intuitively, function `Back.proj0` could now be redefined as `(Back.proj {add=FrontCstr.add; mul=FrontCstr.mul})`.

Let us point out here that the type of `Back.proj` implementation must generalize this signature, and not simply unify with it. This directly forbids `memofst` trick: if we remove the type coercion from the code of `memofst`, the type system infers `memofst: '_a -> '_a` where `'_a` is an existential type variable introduced for a sound typing of references [Garrigue 2002; Wright 1995]. Hence, a cheating use of `memofst` would prevent oracles (like `Back.proj`) from having an acceptable type.

In other words, the unsoundness of `memofst` is detected *statically*, at compile-time. This is a first significant advantage over standard LCF style, where it was only detected at runtime. Moreover, in PFS, Farkas combinations do not need to track the list of axioms P, because this is only necessary for the defensive checks of standard LCF style. PFS is slightly simpler and more efficient than standard LCF style. This is a second advantage over standard LCF style. Beyond these two advantages, Section 5 and Section 6 illustrate that the polymorphic style provides interesting opportunities to reuse oracles for free, whereas, in the style based on type abstraction, this would require a refactorization of oracles with explicit functors.

4 FORMALIZING PROJ FRONTEND IN COQ

In order to program and prove `Front.proj` in Coq, we need to declare `Back.proj` and its type in Coq. This is achieved by turning `Back.proj` into a Coq axiom, itself replaced by the actual OCaml function at extraction. However, such an axiom may be unsound w.r.t a runtime execution. In particular, a Coq function f satisfies $\forall x, (f\ x) = (f\ x)$. But, an OCaml function may not satisfy this property, because of side-effects or because of low-level constructs distinguishing values considered equal in the Coq logic. Section 4.1 recalls the may-return monad introduced by Fouilhé and Boulmé [2014] to overcome this issue. Section 4.2 explains how PFS oracles are embedded in this approach.

4.1 Coq Axioms for External OCaml Functions

Let us consider the Coq example on the right hand-side. It first defines a constant `one` as the Peano's natural number representing 1. Then, it declares an axiom `test` replaced at extraction by a function `oracle`. At last, a lemma `congr` is proved, using the fact that `test` is a function. The following OCaml implementation of `oracle` makes the lemma `congr` false at runtime:

```
Definition one: nat := (S O).

Axiom test: nat → bool.
Extract Constant test ⇒ "oracle".

Lemma congr: test one = test (S O).
  auto.
Qed.
```

```
let oracle x = (x == one)
```

Indeed `(oracle one)` returns **`true`** whereas `(oracle (S O))` returns **`false`**, because == tests the equality between *pointers*. Hence, the Coq axiom is unsound w.r.t this implementation. A similar unsoundness can be obtained if `oracle` uses a reference in order to return **`true`** at the first call, and **`false`** at the following ones. Fouilhé and Boulmé [2014] solve this problem by axiomatizing OCaml functions using a notion of non-deterministic computations. For example, if the result of `test` is declared to be non-deterministic, then the property `congr` is no more provable. For a given type A, type $??A$ represents the type of non-deterministic computations returning values of type A: type $??A$ can be interpreted as $\mathcal{P}(A)$. Formally, the type transformer " $??$. " is axiomatized as a monad that provides a *may-return* relation $\rightsquigarrow_A : ??A \to A \to \text{Prop}$. Intuitively, when "$k : ??A$" is seen as "$k \in \mathcal{P}(A)$", then "$k \rightsquigarrow a$" means that "$a \in k$". At extraction, $??A$ is extracted like A, and its binding operator is efficiently extracted as an OCaml let-in.

For example, replacing the `test` axiom by "**Axiom** `test: nat → ??bool`" avoids the above unsoundness w.r.t the OCaml `oracle`. The lemma `congr` can still be expressed as below, but it is no longer provable.

```
∀ b b', (test one)⇝b → (test (S O))⇝b' → b=b'.
```

The soundness of this approach has been investigated by Boulmé and Vandendorpe [2019]. However, it is still an open conjecture.

4.2 Reasoning on PFS oracles in Coq

Let us now sketch how the frontend is formalized in Coq. We define the type `Var.t` as `positive` – the Coq type for binary positive integers. We build the module `FrontCstr` of constraints encoded as radix trees over `positive` with values in `Qc`, which is the Coq type for $\mathbb{Q}$. Besides operations `add` and `mul`, module `FrontCstr` provides two predicates: `(sat c m)` expresses that a model `m` satisfies the constraint `c` ; and `(noccurs x c)` expresses that variable `x` does not occur in constraint `c`.

```
sat: t → (Var.t → Qc) → Prop.
noccurs: Var.t → t → Prop.
```

We also prove that `sat` is preserved by functions `add` and `mul`. Then, these predicates are lifted to polyhedra `p` of type `(list FrontCstr.t)`.

```
Definition sat p m := List.Forall (fun c ⇒ FrontCstr.sat c m) p.
Definition noccurs x p := List.Forall (FrontCstr.noccurs x) p.
```

Because `front_proj` invokes a non-deterministic computation (the external oracle as detailed below), it is itself a non-deterministic computation. Here is its type and its specification:

```
front_proj: list FrontCstr.t → Var.t → ??(list FrontCstr.t).
Lemma front_proj_correctness: ∀ p x p',
   (front_proj p x) ⇝ p' → (∀ m, sat p m → sat p' m) ∧ noccurs x p'.
```

We implement `front_proj` in PFS, as explained in Section 3.4. First, we declare a `lcf` record type containing operations for frontend constraints. These operations do not need to be declared as non-deterministic: in the Coq frontend, they will be only instantiated by pure Coq functions. Then, `back_proj` is defined as a non-deterministic computation. The type of `back_proj` is given uncurried in order to avoid nested "??" type transformers. At extraction, this axiom is replaced by a wrapper of `Back.proj` from Section 3.4.

```
Record lcf A := { add: A → A → A; mul: Qc → A → A }.
Axiom back_proj: ∀ {A},
   ((lcf A) * (list (FrontCstr.t * A))) * Var.t → ??(list A).
```

Like in Section 3.4, `back_proj` receives each constraint in two representations: an opaque one of polymorphic type `A` and a clear one of another type. For simplicity, this paper uses `FrontCstr.t` as the clear representation on the Coq side. [7]

Now, let us sketch how we exploit our polymorphic `back_proj` to implement `front_proj` and prove its correctness. For a given `p: (list FrontCstr.t)`, parameter `A` of `back_proj` is instantiated with `wcstr(sat p)` where `wcstr(s)` is the type of constraints satisfied by any model

[7] In order to avoid unnecessary conversions from `FrontCstr.t` to `BackCstr.t` (that would be hidden in `back_proj` wrapper), our actual implementation uses instead an axiomatized type which is replaced by "`BackCstr.t`" at extraction: this is similar to the implementation of Fouilhé and Boulmé [2014].

satisfying `s`. In other words, `wcstr(sat p)` is the type of logical consequences of `p`, i.e. the type of its Farkas combinations. Hence, instantiating parameter `A` of `back_proj` by this dependent type expresses that combinations from the input `p` and from the `lcf` operations are satisfied by models of `p`. Concretely, `(front_proj p x)` binds the result of `(back_proj ((mkInput p), x))` to a polyhedron `p'` and checks that `x` does not occur in `p'`.

```
Record wcstr(s: (Var.t → Qc) → Prop) :=
  { rep: FrontCstr.t; rep_sat: ∀ m, s m → FrontCstr.sat rep m }.

mkInput: ∀ p, lcf(wcstr(sat p)) * list(FrontCstr.t * wcstr(sat p)).
```

Actually, `rep_sat` above can be seen as a data-invariant attached to a `rep` value. This invariant is trivially satisfied on the input values, i.e. the constraints of `p`. And, it is preserved by `lcf` operations. These two properties are reflected in the type of `mkInput`. The polymorphism of `back_proj` is a way to ensure that `back_proj` preserves any data-invariant like this one, on the output values. Hence, the fact that `(back_proj ((mkInput p), x))` preserves `wcstr(sat p)` is a kind of "theorems for free" a la Wadler [1989] and [Reynolds 1983], resulting from a *parametricity* property over *unary* relations of the underlying Coq+OCaml type-system. See [Boulmé and Vandendorpe 2019] for more details.

5 THE FLEXIBLE POWER OF PFS ILLUSTRATED ON CONVEX HULL

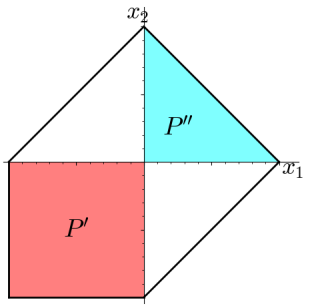
This section provides an advanced usage of polymorphic factories through the `join` operator. It first gives another non-trivial example of *correct-by-construction* oracle. It also illustrates the flexible power of PFS, by deriving `join` from the projection operator of Section 3.4. On this `join` oracle, PFS induces a drastic simplification by removing many cumbersome rewritings on certificates. Indeed, we simply derive the certification of the `join` operator by invoking the projection operator on a *direct product* of factories. As we detail below, such a product computes two independent polyhedral inclusions, in parallel.

In abstract interpretation, `join` approximates the disjunction of two invariants. For the abstract domain of polyhedra, this disjunction geometrically corresponds to the union of two polyhedra $P' \cup P''$. However, in general, such a union is not a convex polyhedron. Operator `join` thus overapproximates this union by the convex hull $P' \sqcup P''$ that *we define* as the smallest convex polyhedron containing $P' \cup P''$. For instance, given

$$P' \triangleq \{x_1 \leq 0,\ x_2 \leq 0, x_1 \geq -1,\ x_2 \geq -1\}$$

$$P'' \triangleq \{x_1 \geq 0,\ x_2 \geq 0, x_1 + x_2 \leq 1\}$$

$P' \sqcup P'' \triangleq \{x_1 \geq -1,\ x_2 \geq -1,\ x_1{+}x_2 \leq 1,\ x_2{-}x_1 \geq -1,\ x_2{-}x_1 \leq 1\}$ as represented on the right hand side figure as the black outline.



The correctness of `join`, given in Definition 5.1, is reduced to two implications themselves proved by Farkas' lemma. More precisely, on a computation $(\texttt{join}\, P'\, P'')$, the oracle produces internally two lists of Farkas combinations that build a pair of polyhedra (P_1, P_2) satisfying $P' \Rightarrow P_1$ and $P'' \Rightarrow P_2$. Then, the frontend checks that P_1 and P_2 are syntactically equal. If the check is successful, it returns polyhedron P_1.

Definition 5.1 (Correctness of `join`*).* Function `join` is correct iff any result P for a computation $(\texttt{join}\, P'\, P'')$ satisfies $(P' \Rightarrow P) \wedge (P'' \Rightarrow P)$.

5.1 Extended Farkas Factories

The factory operations of Definition 3.2 are sufficient to compute any result of a projection, but they do not suffice for the convex-hull and more generally for proving all kinds of polyhedra inclusions. The definition 5.2 given here completes this set of operations. The following lemma ensures its completeness for proving polyhedra inclusions. It extends Lemma 2.1 for polyhedra with equalities and strict inequalities.

Definition 5.2 (Extended Farkas Combination). An extended Farkas combination may invoke one of the five operations:

- $(t_1 \bowtie_1 0) + (t_2 \bowtie_2 0) \triangleq (t_1 + t_2) \bowtie 0$
  where $\bowtie \triangleq \max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $=, \geq, >$.
- $n \cdot (t \bowtie 0) \triangleq (n \cdot t) \bowtie 0$
  under preconditions $n \in \mathbb{Q}$ and, if $\bowtie \in \{\geq, >\}$ then $n \geq 0$.
- $\texttt{weaken}((t \bowtie 0)) \triangleq (t \geq 0)$, for all linear term t and $\bowtie \in \{=, \geq, >\}$.
- $\texttt{cte}(n, \bowtie) \triangleq (n \bowtie 0)$ assuming $n \in \mathbb{Q}$ and $n \bowtie 0$.
- $\texttt{merge}((t \geq 0), (-t \geq 0)) \triangleq (t = 0)$, for all linear term t.

Besides the operations of Definition 5.2, we also define $\top$ as a shortcut for $\texttt{cte}(0, =)$ that thus corresponds to constraint $0 = 0$. Hence, $\top$ is neutral for operations $+$ and $\cdot$ on constraints. It is thus a very convenient default value in our oracles.

Lemma 5.3 (Extended Farkas Lemma). *Let P_1 and P_2 be two convex polyhedra on $\mathbb{Q}$ such that $P_1 \Rightarrow P_2$. Then,*

- *either P_1 is empty and a contradictory constant constraint (e.g. $0 > 0$) is a Farkas combination of P_1,*
- *or each constraint of P_2 is an extended Farkas combination of P_1.*

Proof. Our proof has three cases.

(1) If P_1 is unsatisfiable, then we build the expected contradictory constant constraint using Fourier-Motzkin elimination, i.e. by successive projection of each variable of P_1. Actually, this contradictory constant is built by using only operations from Definition 3.2.

(2) Otherwise, let "$t \bowtie 0$" be a constraint of P_2 such that $\bowtie \in \{\geq, >\}$. By hypothesis, $P_1 \Rightarrow P_2$, so in particular $P_1 \Rightarrow t \bowtie 0$. By defining the complementary of $\bowtie$ (written $\overline{\bowtie}$) as $\{\geq, >\} \setminus \{\bowtie\}$, we get that polyhedron $P_1 \wedge -t \,\overline{\bowtie}\, 0$ is unsatisfiable. By the proof of case (1), there exists a contradictory constant constraint $-\lambda_0 \bowtie' 0$ where $\bowtie' \in \{\geq, >, =\}$ such that $-\lambda_0 = \sum_{i=1}^{k} \lambda_i.t_i - \lambda_{k+1}.t$ and for all i, $\lambda_i \geq 0$ and $P_1 \Rightarrow t_i \geq 0$. Moreover $\lambda_{k+1} > 0$, otherwise P_1 would be unsatisfiable. Thus, we have $t = \frac{1}{\lambda_{k+1}} \cdot (\lambda_0 + \sum_{i=1}^{k} \lambda_i.t_i)$.
Hence, constraint $t \geq 0$ is generated by combining only constraints of P_1 and constraint $\texttt{cte}(\lambda_0, \geq)$ with operators $+$ and $\cdot$, and possibly a final $\Downarrow$. If $\lambda_0 > 0$, then constraint $t > 0$ is also generated in a similar way but from $\texttt{cte}(\lambda_0, >)$ (and avoiding $\Downarrow$). Let us consider the case where $\bowtie$ is $>$ and $\lambda_0 = 0$. In this case, $\bowtie'$ is $>$ (because $0 \bowtie' 0$ is contradictory) whereas $\overline{\bowtie}$ is $\geq$. Thus, there exists $i \in [1, k]$ such that $\lambda_i > 0$ and $t_i > 0$ is a constraint of P_1. Hence, $t > 0$ is generated from P_1 constraints using only operations of Definition 3.2.

(3) At last, let "$t = 0$" be a constraint of P_2 when P_1 is satisfiable. We build this constraint as the result of operator "&" on the two extended Farkas combinations associated to inclusions $P_1 \Rightarrow t \geq 0$ and $P_1 \Rightarrow -t \geq 0$, themselves built like case (2) above.

□

From now on, we only consider extended Farkas combinations and omit the adjective "extended". Definition 5.2 leads to extend our factory type as given on the right-hand side.
Fields `top`, `weaken` and `merge` correspond respectively to $\top$, $\Downarrow$ and $\&$. Type `cmpT` is our enumerated type of comparisons representing $\{\geq, >, =\}$.

```
type 'c lcf =
{ top: 'c;
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c;
  weaken: 'c -> 'c;
  cte: Rat.t -> cmpT -> 'c;
  merge: 'c -> 'c -> 'c }
```

5.2 Encoding `join` as a Projection

Most polyhedra libraries use the double representation of polyhedra, as constraints and as generators. Computing the convex hull $P' \sqcup P''$ using generators is easy. It consists in computing the union of generators and in removing the redundant ones. In constraints-only, the convex hull is computed as a projection problem, following the algorithm of Benoy et al. [2005]. The convex hull is the set of convex combinations of points from P' and P'', i.e.

$$\left\{\boldsymbol{x} \mid \boldsymbol{x}' \in P',\ \boldsymbol{x}'' \in P'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\right\} \tag{1}$$

To express that a point belongs to a polyhedron in a more computational way, we introduce the following matrix notation. We denote $\boldsymbol{x}' \in P'$ by $\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$, where each line of this system represents one constraint of P'. Similarly, $\boldsymbol{x}'' \in P''$ is rewritten into $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$. The previous set of points (1) becomes

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\} \tag{2}$$

Then, by eliminating variables α', α'', $\boldsymbol{x}'$ and $\boldsymbol{x}''$, we obtain $P' \sqcup P''$. Note that we cannot use directly operator `proj` to compute this projection because the set of points (2) is defined with a nonlinear constraint $\boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''$. We go back to linear constraints by applying the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$. By multiplying matrix $\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$ by α' and $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$ by α'', we obtain equivalent systems $\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}'$ and $\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}''$. The set of points (2) is now described as

$$P_H \triangleq \{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''\} \tag{3}$$

For our previous example, P_H is the set of points $\boldsymbol{x} \triangleq (x_1, x_2)$ that satisfy

$$\begin{cases} -y_1' \geq 0,\ -y_2' \geq 0,\ y_1' \geq -\alpha',\ y_2' \geq -\alpha' & (\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}') \\ y_1'' \geq 0,\ y_2'' \geq 0,\ -y_1'' - y_2'' \geq -\alpha'' & (\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}'') \\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1 & \\ x_1 = y_1' + y_1'',\ x_2 = y_2' + y_2'' & \text{(Encoding convex combinations)} \end{cases}$$

Operator `join` finally consists in eliminating variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$ from P_H. The presence of equalities or strict inequalities requires an additional pass that follows the projection, involving operators `weaken` and `merge` of the factory. We omit this step in the paper in order to keep our explanations simple. Moreover, in practice, encoding (3) could be done more efficiently by considering less variables, exploiting the fact that $\alpha'' = 1 - \alpha'$ and $\boldsymbol{y}'' = \boldsymbol{x} - \boldsymbol{y}'$. But as this complicates the understanding and does not affect much the certification, we will not consider this improvement.

In the following, we compare certificate style to PFS for proving `join` from results of `proj`. In order to have a simpler presentation, we limit here to the case where polyhedra contain only non strict inequalities.

5.3 Proving `join` with Certificates

As previously explained about Definition 5.1, the correctness of `join` is ensured by building the convex hull P from two Farkas combinations, one of P' and one of P''. Fouilhé et al. [2013] described how to extract such combinations from the result of the projection of P_H. As in the rest of the polyhedra library they developed, they proceed in a skeptical way with certificates. Thus, their `join` has the following type:

```
Back.join1 : (BackCstr.t * int) list -> (BackCstr.t * int) list ->
             pexp * pexp
```

It takes the two polyhedra P' and P'' as input, and each of their constraints is attached to a unique identifier, as explained in Section 3.2. It returns two certificates of type `pexp`, one for each inclusion $P' \Rightarrow P$ and $P'' \Rightarrow P$ of Definition 5.1.

Let us now detail how Fouilhé et al. retrieve such certificates from the projection of P_H. Consider operator `Back.proj1_list` that extends `Back.proj1` from Section 3 by eliminating several variables one after the other instead of a single one. Assume that `Back.proj1_list` $P_H\ [x_1, \ldots, x_q]$ returns $(P,\ \mathbf{\Lambda})$ where $\mathbf{\Lambda}$ is a certificate of type `pexp` showing that $P_H \Rightarrow P$. $\mathbf{\Lambda}$ can be represented as a matrix where each row $\mathbf{\Lambda}_{i\cdot}$ contains the coefficients of a Farkas combination showing $P_H \Rightarrow C_i$, where C_i is the i^{th} constraint of P.

Since $\alpha',\ \alpha'',\ \boldsymbol{y}'$ and $\boldsymbol{y}''$ do not appear in P (they have been eliminated by projection), $P_H \Rightarrow P$ holds whatever the value of these variables. In particular, certificate $\mathbf{\Lambda}$ remains valid for any assignment of $(\alpha',\ \alpha'',\ \boldsymbol{y}',\ \boldsymbol{y}'')$. The key idea is to find well-chosen assignments in order to retrieve certificates for $P' \Rightarrow P$ and $P'' \Rightarrow P$ out of $\mathbf{\Lambda}$. Indeed, recall that P_H is the set of convex combinations $\alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''$ of points $\boldsymbol{x}' \in P'$ and $\boldsymbol{x}'' \in P''$. By setting $\alpha' = 1$ and $\alpha'' = 0$, P_H becomes restricted to P'. More precisely, considering assignment $\sigma_1 \triangleq (\alpha' = 1,\ \alpha'' = 0,\ \boldsymbol{y}'' = \mathbf{0})$, P_H becomes $\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{y}' \geq \boldsymbol{b}',\ \mathbf{0} \geq \mathbf{0},\ 1 \geq 0,\ 0 \geq 0,\ 1 + 0 = 1,\ \boldsymbol{x} = \boldsymbol{y}'\}$ that simplifies into

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{x} \geq \boldsymbol{b}',\ 1 \geq 0\} \tag{4}$$

which is equivalent to P'. By applying the Farkas combinations from $\mathbf{\Lambda}$ onto this new set of constraints, we obtain a certificate showing that $P' \Rightarrow P$. The same reasoning applied with assignment $\sigma_2 \triangleq (\alpha' = 0,\ \alpha'' = 1,\ \boldsymbol{y}' = \mathbf{0})$ leads to $P'' \Rightarrow P$.

5.4 Proving `join` with a Direct Product of Polymorphic Farkas Factories

In PFS, we use the following type for `join`'s oracle:

```
Back.join : 'c1 lcf -> (BackCstr.t * 'c1) list ->
            'c2 lcf -> (BackCstr.t * 'c2) list ->
            ('c1 -> 'c2 -> 'c3) -> 'c3 list
```

In this polymorphic type, variable `'c1` (resp. `'c2`) represents the type of logical consequences of P' (resp. P''), whereas variable `'c3` represents the type of logical consequences of $P' \cup P''$, i.e. the type of constraints that are *both* logical consequences of P' and P'' (see Definition 5.1). The `Back.join` oracle is parametrized by a certified operator (given by the frontend) of type `'c1 -> 'c2 -> 'c3` and called `unify`. This `unify` operator simply tests whether the two input constraints are syntactically equal: in this case, this constraint is trivially a logical consequence of $P' \cup P''$. Otherwise, `unify` *fails*: it returns a `top` constraint (or raises an error). Hence, `Back.join` builds a convex polyhedron P which includes—*by construction*—the set $P' \cup P''$.

Internally, this oracle first builds a pair of polyhedra (P_1, P_2) of type `('c1 list)*('c2 list)` and then computes P by pairwise applying `unify` to these two lists. Hence, alternatively to a result of type `('c1 -> 'c2 -> 'c3) -> 'c3 list`, we could also design `Back.join` for a result of type

```
let factory_product (lcf1: 'c1 lcf) (lcf2: 'c2 lcf): ('c1 * 'c2) lcf =
{
  top = (lcf1.top, lcf2.top);
  add  = (fun (c1,c2) (c1',c2') -> lcf1.add c1 c1', lcf2.add c2 c2');
  mul = (fun r (c,c') -> lcf1.mul r c, lcf2.mul r c');
  weaken = (fun (c,c') -> lcf1.weaken c, lcf2.weaken c');
  cte = (fun r cmp (c,c') -> lcf1.cte r cmp c, lcf2.cte r cmp c');
  merge = (fun (c1,c1') (c2,c2') ->  lcf1.merge c1 c1', lcf2.merge c2 c2');
}
```

Fig. 5. Direct product of two Farkas Factories

`('c1 list)*('c2 list)`, and let the frontend build P from this result. These two alternatives are more or less equivalent, because building P from the pair (P_1, P_2) is very easy to implement and prove correct in Coq.

We said that for computing the convex hull, `join` eliminates variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$ from P_H. Recall that the projection operator that we defined for PFS in Section 3.4 has type

```
Back.proj: 'c lcf -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

As we did for the certificate approach, let us define `Back.proj_list` that extends `Back.proj` by eliminating a list of variables.

```
Back.proj_list: 'c lcf -> (BackCstr.t * 'c) list -> Var.t list -> 'c list
```

The `Back.join` oracles is parametrized by two Farkas factories, but it needs to call `Back.proj_list` on a single one. To do so, `Back.join` will provide `Back.proj_list` with a combination of its own two factories. Although the parameter `'c lcf` of `Back.proj_list` was originally designed to be provided by the frontend, nothing forbids the backend to tune it. This is where the flexibility of PFS comes into play! More precisely, `Back.join` combines the two factories of types `'c1 lcf` and `'c2 lcf` into a new one of type `('c1 * 'c2) lcf` given in Figure 5. This factory computes with frontend constraints from P' and P'' in parallel: it corresponds to the *direct product* of the two initial Farkas factories.

Now, let us detail how to build P_H. To be compatible with the combined factory, each backend constraint of P_H must be attached to a frontend constraint of type `'c1 * 'c2`. Constraints of P' – that have type `(BackCstr.t * 'c1)` – are converted into type `(BackCstr.t * ('c1 * 'c2))` by being attached to constraint `lcf2.top` of type `'c2`. Similarly, constraints of type `(BackCstr.t * 'c2)` are attached to constraint `lcf1.top` of type `'c1`. Then, we need to apply the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$ as explained above. But we have no way of doing this on frontend constraints: the backend can only apply factory operators on them, and none allows the addition of new variables. Fortunately, we are only interested in specific values for these new variables, values from assignments σ_1 and σ_2 defined in the previous section. These evaluations make both variables $\boldsymbol{y}'$ and $\boldsymbol{y}''$ vanish, as in Equation (4). Thus, to build the frontend version of constraints of P_H, we evaluate them directly on each assignment as follows:

$$
\begin{array}{lcl*{5}{c}}
 & \texttt{BackCstr.t} & \longrightarrow & \texttt{BackCstr.t} & * & (& \texttt{'c1} & * & \texttt{'c2} &) \\
A'\boldsymbol{x}' \geq \boldsymbol{b}' \left\{ \right. & A'_1\boldsymbol{x}' \geq b'_1 & \longrightarrow & A'_1\boldsymbol{y}' \geq b'_1 & , & (& \underbrace{\left[A'_1\boldsymbol{y}' \geq \alpha' b'_1\right]_{\sigma_1}}_{A'_1\boldsymbol{x}' \geq b'_1} & , & [\top]_{\sigma_2} &) \\
 & \cdots & & & & & & & & \\
 & A'_p\boldsymbol{x}'' \geq b'_p & \longrightarrow & A'_p\boldsymbol{y}' \geq b'_p & , & (& \underbrace{\left[A'_p\boldsymbol{y}' \geq \alpha' b'_p\right]_{\sigma_1}}_{A'_p\boldsymbol{x}' \geq b'_p} & , & [\top]_{\sigma_2} &)
\end{array}
$$

$$
\begin{array}{lcl*{5}{c}}
 & \texttt{BackCstr.t} & \longrightarrow & \texttt{BackCstr.t} & * & (& \texttt{'c1} & * & \texttt{'c2} &) \\
A''\boldsymbol{x}'' \geq \boldsymbol{b}'' \left\{ \right. & A''_1\boldsymbol{x}'' \geq b''_1 & \longrightarrow & A''_1\boldsymbol{y}'' \geq b''_1 & , & (& [\top]_{\sigma_1} & , & \underbrace{\left[A''_1\boldsymbol{y}'' \geq \alpha'' b''_1\right]_{\sigma_2}}_{A''_1\boldsymbol{x}'' \geq b''_1} &) \\
 & \cdots & & & & & & & & \\
 & A''_q\boldsymbol{x}' \geq b''_q & \longrightarrow & A''_q\boldsymbol{y}'' \geq b''_q & , & (& [\top]_{\sigma_1} & , & \underbrace{\left[A''_q\boldsymbol{y}'' \geq \alpha'' b''_q\right]_{\sigma_2}}_{A''_q\boldsymbol{x}'' \geq b''_q} &)
\end{array}
$$

Finally, we add constraints $\alpha' \geq 0$, $\alpha'' \geq 0$ and $\alpha' + \alpha'' = 1$ that must be given in type `BackCstr.t * ('c1 * 'c2)`. However, they contain variables α' and α'' that were not present in the input polyhedra P' and P''. Once again, we build directly their evaluation in σ_1 and σ_2. Constraints $1 \geq 0$ and $0 \geq 0$ are built in types `'c1` or `'c2` thanks to operator `cte` from factories `lcf1` and `lcf2`. Note that $\alpha' + \alpha'' = 1$ is not given here because it evaluates to $(\top, \top)$, and can therefore be discarded.

$$
\begin{array}{lcl*{5}{c}}
\texttt{BackCstr.t} & \longrightarrow & \texttt{BackCstr.t} & * & (& \texttt{'c1} & * & \texttt{'c2} &) \\
\alpha' \geq 0 & \longrightarrow & \alpha' \geq 0 & , & (& \underbrace{[\alpha' \geq 0]_{\sigma_1}}_{1 \geq 0} & , & \underbrace{[\alpha' \geq 0]_{\sigma_2}}_{0 \geq 0} &) \\
\alpha'' \geq 0 & \longrightarrow & \alpha'' \geq 0 & , & (& \underbrace{[\alpha'' \geq 0]_{\sigma_1}}_{0 \geq 0} & , & \underbrace{[\alpha'' \geq 0]_{\sigma_2}}_{1 \geq 0} &)
\end{array}
$$

As an example, let us focus on the proof that P' and P'' both imply $-x_1 - x_2 \geq -1$, which is a constraint of $P' \sqcup P''$. We build P_H as described above, and obtain from its projection a frontend constraint, that is

$$(-x_1 \geq 0,\ 0 \geq 0) + (-x_2 \geq 0,\ 0 \geq 0) + (1 \geq 0,\ 0 \geq 0) + (0 \geq 0,\ -x_1 - x_2 \geq -1)$$
$$= (-x_1 - x_2 \geq -1,\ -x_1 - x_2 \geq -1)$$

The left-hand side of each term is the frontend constraint of type `'c1`, and the right hand side is of type `'c2`. From P' point of view, we obtain $-x_1 - x_2 \geq -1$ as the combination of $-x_1 \geq 0$, $-x_2 \geq 0$ and the constant constraint $1 \geq 0$ that comes from $\alpha' \geq 0$. On the other hand, $-x_1 - x_2 \geq -1$ is a constraint of P'' and is directly returned as a frontend constraint of type `'c2`. The projection returns such results for each constraint of the convex hull $P' \sqcup P''$.

In conclusion, with a well chosen factory, we define our PFS `join` as a simple call to `proj_list`. This makes our implementation much simpler than Fouilhé's one, where the two certificates of `join` are obtained from the one of `proj_list` by tedious rewritings that perform on-the-fly renamings of constraint identifiers.

6 GENERATING COMPACT CERTIFICATES FROM A PFS ORACLE

We designed PFS in order to avoid certificate generation in a skeptical approach based on Coq extraction. Yet, certificates are still useful for other applications. This section demonstrates that PFS is also relevant in this case.

For example, Boulmé and Maréchal [2018] have embedded the guard oracle of the VPL inside a Coq tactic that simplifies Coq proofs thanks to polyhedral computations. This tactic requires an OCaml oracle that produces a Coq AST – i.e. a kind of certificate – typechecked by the Coq kernel.

This AST represents a polyhedral computation, itself encoded as a value of a Coq inductive type – similar to the `pexp` type of Section 3.2. The tactic then applies a Coq version of the `Front.run` interpreter of Section 3.2 to this certificate of type `pexp`.

Certificates could also provide a way to reduce the TCB w.r.t. our current approach. We could imagine certifying each run of our OCaml oracles by generating a Coq term representing this run. For example, this term would be dumped in a Coq source file (in Gallina syntax) and checked by the Coq compiler. Coq extraction and OCaml would no longer be part of the TCB. With respect to the above tactic, this would also avoid trusting the dynamic loading of oracles in the Coq runtime. But, obviously, this approach would make our library much more complicated to integrate into realistic software.

Now, let us explain why PFS is very relevant to implement certificate generating oracles. As detailed in Section 3.4 and in Section 5, polymorphic factories provide an abstract layer that simplifies the implementation of oracles. The code generating certificates can then be easily factorized for a family of oracles, as illustrated in Section 6.1. Moreover, by defining a well chosen factory, we produce a compact AST without slowing too much its generation. This factory actually produces a DAG, from which the final AST is extracted after a dependency analysis. For example, intermediate results that are actually not needed for the AST are discarded. Similarly, when an intermediate computation is used at least twice, we define a binder that stores this result into an intermediate variable. These two optimizations, explained in Section 6.3, avoid useless or redundant computations in the AST interpreter. Another optimization is performed on the DAG: `top` nodes are eliminated, and multiplication by constants are factorized. Section 6.2 gives the factory that produces the DAG, and how this last optimization is applied on the fly.

6.1 Factorizing the AST Generation from PFS Oracles

The DAG datastructure provides the interface below, which helps to wrap PFS oracles of the VPL. Type `dsctr` is the type of nodes in the DAG. Constant `dag_factory` provides a factory instance for our PFS oracles. Function `import` converts an input polyhedron into an input suitable for oracles. Finally, function `export` converts the output of oracles into an AST of type `pexp`.

```
type dcstr
val dag_factory: dcstr Back.lcf
val import: BackCstr.t list -> (BackCstr.t * dcstr) list
val export:  ('a * dcstr) list -> pexp
```

From this interface, wrapping a given PFS oracle into an AST producing oracle is straightforward. For example, we define below `ast_proj` which wraps the `Back.proj` PFS oracle of Section 3.4.

```
let ast_proj (p: BackCstr.t list) (x: Var.t): pexp =
  export (Back.proj dag_factory (import p) x)
```

Below, Section 6.2 defines `dag_factory` and `import` that makes the PFS oracle builds the DAG. Section 6.3 describes the analysis of this DAG in `export` to produce a compact AST.

6.2 A Factory Producing a DAG

For simplicity, we illustrate the generation of the DAG on the following sub-factory of the one of Section 5.1.

```
type 'c lcf = { top: 'c; add: 'c -> 'c -> 'c; mul: Rat.t -> 'c -> 'c }
```

During the DAG generation, we eliminate the neutral element `top` that induces useless nodes. We also factorize multiplications by rational constants. These propagations are directly achieved by the operations of `dag_factory`.

The type `dcstr` of nodes in the DAG is implemented on the right hand-side. This is a record type with a field `def` containing the "operation" at this node. An operation of type `op` corresponds either to an input constraint (constructor `_Ident`) or to an operation on constraints. Operations `_Add` and `_Mul` refer to nodes of type `dcstr`, and such a node can be shared between several operations by pointer sharing. Mutable fields of `dcstr`, like `id` and `nbusers`, are only used during function `export`. They represent auxiliary data on the node, which are computed by the dependency analysis and useful to generate the final AST.

```
type dcstr = {
  def: op;
  mutable id: int;
  mutable nbusers: int;
  (* other omitted fields *)
} and op =
  | Ident_
  | Top
  | Add_ of dcstr * dcstr
  | Mul_ of Rat.t * dcstr
```

We call a node `dc1` a *direct ancestor* of a node `dc2` iff `dc2` appears in `dc1.def` (i.e. as arguments of `Add_` or `Mul_`). It corresponds to the fact that the computation represented by `dc1` *depends on* the result of the computation represented by `dc2`. Here, `dc2` is a reference that may have several direct ancestors but, by construction, it can not be a direct or indirect ancestor of itself.

Most new nodes of the DAG are generated through a call to `(make_dcstr d)` where `d` is a value of type `op`. This call initializes field `def` with value `d` and other fields with default values (these latter being only used in `export`). The only exception is on `Ident_` nodes that are created with a positive field `id` giving their name in the final AST.

```
let make_dcstr ?id:(i=0) d : dcstr = { def=d; id=i; nbusers=0; (* ... *) }
```

Let us now detail the implementation of `import` and `dag_factory`. On a given polyhedron `p`, function `import` associates a new `Ident_` node to each constraint `c` of `p`. The name of each of these nodes – given by its field `id` – corresponds to the position of `c` in the list `p`.

```
let import p = List.mapi (fun i c -> (c, make_dcstr ~id:(i+1) Ident_)) p
```

In `dag_factory`, functions `smart_mul` and `smart_add` are *smart constructors* of nodes which eliminate `Top` nodes and factorize `Mul_` nodes as much as possible.

```
let dag_factory = {top = make_dcstr Top; add = smart_add; mul = smart_mul}
```

This process corresponds to applying the rewriting rules of Figure 6, where $\top$, $+$ and $\cdot$ represent a node where the field `def` is respectively `Top`, `Add_` and `Mul_` and where c, c_1 and c_2 are some other existing nodes. Since these smart constructors assume that their node in inputs are *already rewritten*, they only perform $O(1)$ rewriting steps at each call. Moreover, `(smart_mul n c)` assumes that scalar `n` is not zero and that if `n` is negative then `c` is an equality. These two last assumptions are of course valid on our PFS oracles, and they are preserved by the rewriting rules of Figure 6.

For instance, on a witness "$n_1 \cdot c_1 + n_2 \cdot \left(\top + \frac{n_1}{n_2} \cdot c_2\right)$" generated from a PFS oracle (where $n_1 > 0$), the factory builds a node corresponding to "$n_1 \cdot (c_1 + c_2)$". Let us remark that some useless nodes, such as "$\frac{n_1}{n_2} \cdot c_2$", are generated in the DAG during this process. But they do not pollute the final AST, thanks to the dependency analysis of the next section.

$$n \cdot \top \to \top \qquad 1 \cdot c \to c \qquad n_1 \cdot (n_2 \cdot c) \to (n_1 \times n_2) \cdot c$$

$$\top + c \to c \qquad c + \top \to c \qquad (n_1 \cdot c_1) + (n_2 \cdot c_2) \to \begin{cases} n_1 \cdot \left(c_1 + \frac{n_2}{n_1} \cdot c_2\right) & \text{if } n_1 > 0 \\ n_2 \cdot \left(\frac{n_1}{n_2} \cdot c_1 + c_2\right) & \text{if } n_1 < 0 \end{cases}$$

Fig. 6. Elimination of Top Nodes and Factorization of Mul_ Nodes in the DAG

6.3 Producing the AST

We aim here to produce certificates like examples given in Figure 3 at page 8, where derived constraints used in at least two Farkas combinations (of type `fexp`) are named by a `Bind` instead of having their combination duplicated. This is achieved by function `export`. We now summarize how this function builds a compact AST using a named representation in binders, and where unbound names represent input constraints (while giving their position in the input list).

The oracle, instantiated with `dag_factory`, returns a list of output constraints of type `(BackCstr.t * dsctr)`. Function `export` first extracts `dsctr` values from this list, and obtains the list of *roots* from which we start our dependency analysis on the DAG. By analyzing descendants of each root, we look for nodes that have at least two direct ancestors (among the descendants of the roots). Such nodes are then sorted according to a topological sort and are named with unique positive integers (in field `id`) above the maximum name of reachable `_Ident` nodes. These nodes induce a `Bind` node associating their `id` field to their Farkas combination. On the contrary, descendants of roots which have a null `id` field – they have thus exactly one direct ancestor – are directly replaced by their Farkas combination in the AST without an intermediate `Bind` node.

In conclusion, PFS completely hides the issue of handling binders in the core of our oracles. This handling is factorized over our PFS oracles within a dedicated component, able to produce compact certificates.

7 RELATED WORKS AND CONCLUSION

The skeptical approach has been pioneered in the design of two interactive provers, Automath [de Bruijn 1968] and LCF [Gordon et al. 1979]. Both provers reduce the soundness of a rich mathematical framework to the correctness of a small automatic proof checker called the kernel. But, their style is very different. LCF is written as a library in a functional programming language (ML) which provides the type of theorems as an abstract datatype. Its safety relies on the fact that objects of this type can only be defined from a few primitives (i.e. the kernel). Each of them corresponds to an inference rule of Higher-Order Logic in natural deduction. On the contrary, Automath introduces a notion of "*proof object*" and implements the kernel itself as a typechecker, thanks to Curry-Howard isomorphism. LCF style is more lightweight – both for the development and the execution of proof tactics – whereas the proof object style allows a richer logic (e.g. with *dependent types*). Nowadays, the kernel of skeptical interactive provers is still designed according to one of this style: Coq has proof objects whereas HOL provers are in LCF style.

Since the 90's, the skeptical approach is also applied in two kinds of slightly different contexts: making interactive provers communicate with external solvers like Maple [Harrison and Théry 1998], and verifying the safety of untrusted code, like in "*Proof Carrying Code*" [Necula 1997]. In Coq, it is also applied to the design of proof tactics communicating with external solvers [Armand et al. 2011, 2010; Besson 2006; Grégoire et al. 2008; Magron et al. 2015], and to certify stand-alone programs like compilers or static analyzers which embed some untrusted code [Besson et al. 2010; Blazy et al. 2015; Jourdan et al. 2015; Tristan and Leroy 2008].

Actually, there are now so many works related to the skeptical approach that it seems impossible to be exhaustive. With respect to all these works, the contribution of this paper is to propose a design pattern, called Polymorphic Factory Style (abbreviated as PFS), in order to certify in Coq the results of an untrusted ML oracle. This pattern is illustrated on a new implementation of the VPL, a certified abstract domain of convex polyhedra initially developed in [Fouilhé et al. 2013] and used in the certified Verasco static analyzer [Jourdan et al. 2015]. To summarize, the VPL contains a set of oracles producing witnesses that correspond to nonnegative linear combinations which are logical consequences of their inputs.

In PFS, i.e. *polymorphic LCF style*, oracles produce these witnesses as ordinary ML values (e.g. linear constraints). In other words, instead of building a certificate that the Coq frontend uses to compute the certified value, the oracle directly generates this value by using certified operators of the Coq frontend. This provides several benefits over certificates with no counterpart: the Trusted Computing Base induced by PFS remains the same as in the version with certificate generation. First, it makes the oracle development easier. Ill-handling of certified operators is much straightforward to debug. Without a certificate to build, it naturally removes cumbersome details such as handling of binders. As a consequence, in our implementation of the VPL, the length of the OCaml coded devoted to certification has been divided by two! Second, polymorphism ensures that oracle results are sound by construction. In the polyhedra library, it means that oracles can only produce logical consequences of their input. This property is proved for free from the types of the oracles, in the spirit of the "theorems for free" coined by Wadler [1989]. Third, polymorphism makes witness generation very flexible and modular. Generating a compact certificate is still possible if necessary, e.g. for embedding an oracle within a Coq tactic. Moreover, PFS is even more lightweight than standard LCF style, as it does not need to memorize an explicit type of "theorems". Finally, PFS forbids some incorrect oracles *statically* (by OCaml typechecking), while standard LCF style rejects them only at runtime.

In parallel of our work, [Boulmé and Vandendorpe 2019] have also successfully applied PFS for the certification of Boolean SAT-solvers. In their work, the factory is reduced to a single operation based on propositional resolution, while the OCaml oracle is a parser of unsat certificates emitted by a SAT-solver. This illustrates that PFS may be applied in many verification problems.

ACKNOWLEDGMENTS

We thank Michaël Périn and David Monniaux for their fruitful suggestions all along this work.

REFERENCES

Xavier Allamigeon and Ricardo D. Katz. 2017. A Formalization of Convex Polyhedra Based on the Simplex Method. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 10499. Springer, 28–45.

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs (CPP) (LNCS)*, Vol. 7086. Springer, 135–150.

Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. 2010. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 6172. Springer, 83–98.

Henk Barendregt and Erik Barendsen. 2002. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning* 28, 3 (2002), 321–336.

Florence Benoy, Andy King, and Frédéric Mesnard. 2005. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming* 5, 1-2 (January 2005).

Frédéric Besson. 2006. Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In *Types for Proofs and Programs (TYPES) (LNCS)*, Vol. 4502. Springer, 48–62.

Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. 2010. Certified Result Checking for Polyhedral Analysis of Bytecode Programs. In *Trustworthy Global Computing (TGC) (LNCS)*, Vol. 6084. Springer, 253–267.

Sandrine Blazy, Delphine Demange, and David Pichardie. 2015. Validating Dominator Trees for a Fast, Verified Dominance Test. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 9236. Springer, 84–99.

Sylvain Boulmé and Alexandre Maréchal. 2018. A Coq Tactic for Equality Learning in Linear Arithmetic. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 10895. 108–125. https://hal.archives-ouvertes.fr/hal-01505598

Sylvain Boulmé and Thomas Vandendorpe. 2019. Embedding Untrusted Imperative ML Oracles into Coq Verified Code. (March 2019). https://hal.archives-ouvertes.fr/hal-02062288 preprint.

Vasek Chvatal. 1983. *Linear Programming*. W. H. Freeman.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*. ACM Press.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*. ACM Press.

N.G. de Bruijn. 1968. The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions. In *Symposium on Automatic Demonstration (LNM)*, Vol. 125. Springer, 29–61.

Julius Farkas. 1902. Theorie der einfachen Ungleichungen. *Journal für die Reine und Angewandte Mathematik* 124 (1902).

Alexis Fouilhé and Sylvain Boulmé. 2014. A Certifying Frontend for (Sub)Polyhedral Abstract Domains. In *Verified Software: Theories, Tools, Experiments (VSTTE) (LNCS)*, Vol. 8471. Springer, 200–215.

Alexis Fouilhé, David Monniaux, and Michaël Périn. 2013. Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In *Static Analysis Symposium (SAS) (LNCS)*, Vol. 7935. Springer, 345–365.

Joseph Fourier. 1827. Histoire de l'Académie, partie mathématique (1824). *Mémoires de l'Académie des sciences de l'Institut de France* 7 (1827).

Jacques Garrigue. 2002. Relaxing the Value Restriction. In *Asian Programming Languages and Systems Symposium (APLAS) (LNCS)*, Vol. 2998. Springer, 31–45.

Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. 1978. A Metalanguage for Interactive Proof in LCF. In *Principles of Programming Languages (POPL)*. ACM Press, 119–130.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. LNCS, Vol. 78. Springer.

Benjamin Grégoire, Loïc Pottier, and Laurent Théry. 2008. Proof Certificates for Algebra and Their Application to Automatic Geometry Theorem Proving. In *Automated Deduction in Geometry (ADG) (LNCS)*, Vol. 6301. Springer.

John Harrison and Laurent Théry. 1998. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning* 21, 3 (1998), 279–294.

Jacob M. Howe and Andy King. 2012. Polyhedral Analysis using Parametric Objectives. In *Static Analysis Symposium (SAS) (LNCS)*, Vol. 7460. Springer, 41–57.

Jacques-Herni Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Principles of Programming Languages (POPL)*. ACM Press, 247–259.

Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. 2015. Formal Proofs for Nonlinear Optimization. *Journal of Formalized Reasoning* 8, 1 (2015), 1–24.

Alexandre Maréchal, David Monniaux, and Michaël Périn. 2017. Scalable Minimizing-Operators on Polyhedra via Parametric Linear Programming. In *Static Analysis Symposium (SAS) (LNCS)*, Vol. 10422. Springer. https://hal.archives-ouvertes.fr/hal-01555998

George C. Necula. 1997. Proof-Carrying Code. In *Principles of Programming Languages (POPL)*. ACM Press, 106–119.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: a Case Study on Instruction Scheduling Optimizations. In *Principles of Programming Languages (POPL)*. ACM Press, 17–27.

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture (FPCA)*. ACM Press, 347–359.

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.