# Toward Certification for Free!

Sylvain Boulmé, Alexandre Maréchal

## ▶ To cite this version:

Sylvain Boulmé, Alexandre Maréchal. Toward Certification for Free!: Correct-By-Construction ML Oracles with Polymorphic LCF Style. 2017. hal-01558252v2

## HAL Id: hal-01558252
## https://hal.science/hal-01558252v2

Preprint submitted on 27 Sep 2017 (v2), last revised 19 Jul 2019 (v3)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# **Toward Certification for Free!** *
Correct-By-Construction ML Oracles with Polymorphic LCF Style

SYLVAIN BOULMÉ and ALEXANDRE MARÉCHAL,
Université Grenoble-Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-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, but producing certificates is a requirement which increases complexity of oracle development.

We propose a design pattern, called *Polymorphic LCF Style*, that removes the need for certificates: ML oracles directly compute the certified result by invoking trusted operators and datastructures extracted from Coq. But, oracles only handle these datastructures as polymorphic values, which forbids oracles to forge incorrect results. This design thus delegates a part of the certification to the ML typechecker. Correctness comes from a weak parametricity property of imperative ML polymorphic types that we call *parametric invariance*. We demonstrate the relevance of Polymorphic LCF 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 INTRODUCTION

This paper provides two contributions. First, we propose a new design pattern for developing certified programs, particularly adapted for problem solvers which solutions are hard to discover but easy to verify. Our design pattern reduces the development effort in Coq (The Coq Development Team 2016) by delegating part of the verification to the OCaml typechecker (Leroy et al. 2016). Second, we apply this design pattern to the development of a realistic library: a certified abstract domain of polyhedra. Let us start by introducing this case study.

*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 overapproximating 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`). 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

*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).
2017. Manuscript submitted to ACM

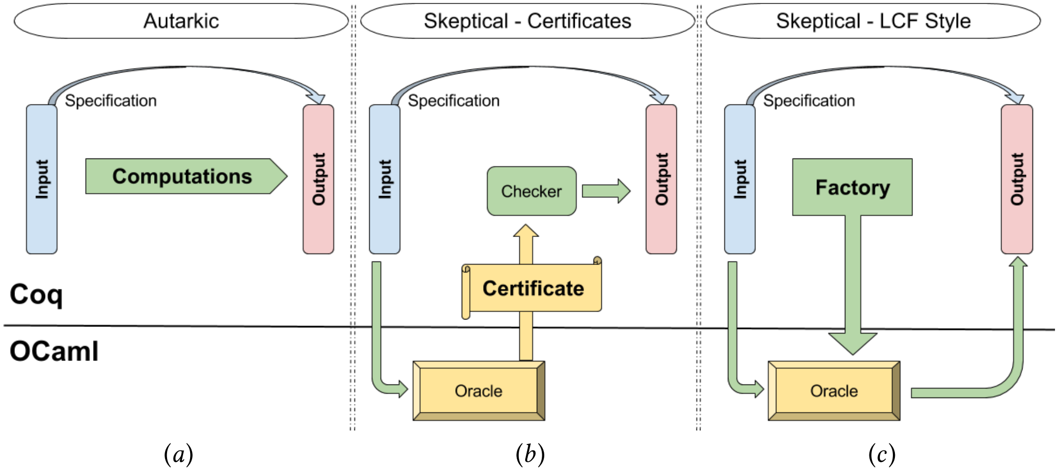



Fig. 1. Three methods of certified computations: autarkic, certificate-driven and with LCF Style.

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.[1]

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. We compare three ways of certifying in Coq that a **`true`** answer from `is_empty` ensures the emptiness of its input polyhedron. The three certification approaches are illustrated on Figure 1.

*Autarkic Approach.* Our polyhedral operators could be directly implemented and proved correct in Coq following the *autarkic* approach (Barendregt and Barendsen 2002). This approach allows a safe offensive programming style (without runtime verifications). It also offers a very reduced Trusted Computing Base (TCB): only the Coq proof-checker needs to be trusted (see Figure 1($a$)). The software can also be used independently from Coq thanks to a built-in Coq process named *extraction*: it is exported to OCaml and then compiled – at the price of adding both the extraction process and the OCaml compiler into the TCB.

However, the autarkic approach is very development-time consuming and restrictive, since it forbids the use of efficient C libraries like GMP (for multi-precision arithmetic) or GLPK (for linear programming). Moreover, it enforces using exclusively Coq datastructures, which forbids many algorithmic optimizations.

*Skeptical Approach with Certificates.* 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, the VPL uses the

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

constraints-only representation of polyhedra in order to ease its certification in Coq.[2] Its abstract domain is designed in a two-tier architecture:

(1) a backend, combining OCaml and C code, provides a set of untrusted oracles that perform efficient but unproved computations, and generate *certificates* driving the reconstruction of a trusted result;
(2) a Coq frontend uses the backend oracles and certificates to provide the certified operators of the abstract domain (see Figure 1($b$)).

The frontend is linked to the backend during its extraction from Coq into OCaml. 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 ML compiler.

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[3] 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 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 the set of points satisfying $x \geq 3 \wedge y \geq 0$ is included in the one that 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 1.1 (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 $P_1 \Rightarrow P_2$ 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). Here, the OCaml oracle for `is_empty` returns a certificate as the list of coefficients which represents the previous Farkas combination. 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 indeed computes the result of the combination with its own certified Coq datastructures and obtains $0 \geq 1$.

[2]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.

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

Certificate generation does not guarantee the absence of bugs in the oracle. For instance, the backend may not terminate normally on some inputs. This approach ensures however a *partial correctness* property: when the oracle terminates and provides a certificate, the frontend uses the certificate to compute a certified result satisfying the *formal specification* of the operator. It detects if the backend went wrong and can then fail or return a trivially correct – but weak – result.

In an informal discussion, A. Fouilhé, the VPL developer, stated that the VPL certificates were more complex than sketched above and that the code generating them was particularly difficult to develop and debug (wait Section 2.2 for more details). He concluded that simplifying this process would be helpful.

*Skeptical Method with LCF Style.* 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.

This style is much lighter than the preceding one, because it avoids the introduction of a certificate format – i.e. an abstract syntax – in order to represent the certified computations. Instead, the OCaml oracle uses a factory of certified operators (i.e. the "Factory" of Figure 1(*c*)) to perform trusted computations. The key idea is that such a factory can only build logical consequences of some given set of axioms. Thus, in our use, "LCF" also means "*Logical Consequences Factories*".

For `is_empty`, the oracle has two versions of each constraint: the untrusted one, named `BackCstr.t`, manipulated by the oracle, and the one extracted from Coq, named `FrontCstr.t`, on which the oracle can only apply factory operators. Given an empty polyhedron, the backend uses an untrusted solver to 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 $0 \geq 1$.

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

This style of certification relies on one assumption: the frontend can trust results produced by an external oracle that uses its certified operators. This is not true in general: making `FrontCstr.t` an abstract datatype is not sufficient to forbid an imperative OCaml program to cheat by returning a contradictory constraint obtained from a previous run. Hence, this naive LCF style is unsound for certification.

*Our contribution.* This paper introduces the *Polymorphic LCF Style*, that we abbreviate as PFS (*Polymorphic Factories Style*) for convenience, for developing correct-by-construction oracles in a skeptical approach. More precisely, a PFS oracle can be trusted to preserve some invariant without the need for an intermediate certificate. We experimented this *design pattern* by reimplementing the certification features of the VPL: it simplifies both Coq and OCaml parts.

On our running example, PFS consists in abstracting the certified datatype `FrontCstr.t` by a polymorphic type `'c` in the oracle, and providing the oracle with operators on this datatype grouped in a factory of type `'c lcf`. Polymorphism ensures that the oracle can only produce correct results by combining its inputs using the operators of the factory. In other words, the type of the `is_empty` oracle becomes

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

Polymorphism of PFS brings the soundness that was missing in the naive LCF style. Results produced by oracles are correct by construction provided that the operators of the factory preserve the desired correctness property. We will explain how such correctness proofs are elegantly

expressed in Coq from the type of PFS oracles (see Section 3). Furthermore, PFS makes our oracles flexible. In particular, we can easily profile or debug oracles, simply by changing the factory. If necessary, we can still produce certificates as in the original VPL using an adequate factory, or even disable the certification for efficiency using a "do-nothing" factory.

The performances of the original version of the VPL were analyzed by Fouilhé et al. (2013). They are comparable with those of PPL (Bagnara et al. 2008) and NewPolka (Jeannet and Miné 2009), two state-of-the-art – but unverified – polyhedra libraries. Our new design seems to have a little benefit on VPL performances. More significantly, it simplifies the development while keeping the same TCB than the original version. Thanks to PFS, the number of lines of code in the VPL modules at the interface of OCaml and Coq has been divided by two, both for OCaml and Coq sides. And it gives simpler and more readable code.[4]

Currently, the soundness proof of our approach is partial. On the one hand, we are able to prove that our reasonings on PFS oracles – which are actually parametricity reasonings (Wadler 1989) – are correct. Indeed, they apply a weak *parametricity* property of polymorphic types, that we call *parametric invariance*. This property has been formalized on system F with higher-order references *a la* ML (Ahmed et al. 2002; Appel et al. 2007; Birkedal et al. 2011). We have adapted this proof on a subset of imperative ML.Our proof is given in Appendix A. On the other hand, we have not yet proved that our particular way to invoke Coq extraction is perfectly sound w.r.t. the actual OCaml compiler. We only conjecture that it is. However, to our best knowledge, none of the real world developments that mix Coq and OCaml code, including the certified compiler CompCert (Leroy 2009), come with such a proof; they rely on similar conjectures. See Section 4 for a detailed discussion.

Actually, we are not the first to relate Coq extraction with parametricity reasoning. In a sense, Bernardy and Moulin (2012, 2013) already looked for a generalization of Coq extraction in order to internalize some parametricity reasonings within dependent type theory. The novelty of our proposal is to use parametricity as a very cheap approach to reason about imperative ML code in Coq. To our knowledge, since the proposal to get "theorems for free" from parametricity by Wadler (1989), this paper describes its first application to the certification of realistic software, indeed implemented within widespread tools like Coq and OCaml.

*Paper Overview.* Section 2 incrementally details PFS on a slightly more complex example: operator `proj`. It also illustrates why the original LCF style is unsound in this context. Section 3 shows how to use PFS in Coq proofs. The soundness of our approach is discussed in Section 4. Section 5 reveals the flexible power of polymorphic factories thanks to operator `join`. Benoy et al. (2005) have shown how to derive a simple implementation of operator `join` from `proj`. We show that the certification of Benoy's `join` boils down to defining a well-chosen instance of the factory expected by operator `proj`. With this approach, the certification of Benoy's `join` becomes elegant and straightforward, whereas the one of Fouilhé et al. (2013) was cumbersome because of many certificate rewritings. Section 6 gives another example of PFS flexibility: using an adequate factory, we generate compact certificates from PFS oracles in order to embed them in Coq tactics.

## 2 PFS ORACLES EXPLAINED WITH A DETAILED EXAMPLE

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 $x$, (`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

[4]Our source code is available at http://github.com/VERIMAG-Polyhedra/VPL.

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 "`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.

In the following, 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 2.1 (Correctness of* `proj`*).* Function `proj` is correct iff any result $P'$ for a computation (`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 2.2. Obviously, such constraints are necessarily implied by $P$.

*Definition 2.2 (Linear Combinations of Constraints).* We define operations $+$ and $\cdot$ 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 3 will then formalize `Front.proj` in Coq.

### 2.1 Simple (but Unsound) LCF Style

In a first step, we follow the LCF style introduced in Section 1. 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 $\cdot$ 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. This representation allows finding 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. For example, the following function builds the certified constraints of $P'_1$ from constraints of $P_1$, according to the previous Farkas combinations. Here, rational constants are written with an informal notation.

```
let build_P'1 (l: FrontCstr.t list): FrontCstr.t list =
  match l with
  | c1::c2::c3::c4::_ ->
    let coeff = 1/3 and tmp = FrontCstr.add c4 (FrontCstr.mul −1 c1) in
    [ c4;
      FrontCstr.mul coeff (FrontCstr.add c3 tmp);
      FrontCstr.mul coeff (FrontCstr.add c2 (FrontCstr.mul −1 tmp)) ]
  | _ -> failwith "unexpected input"
```

But making `Back.proj` return such a function is not so convenient. It is simpler 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 certified functions: `occurs: Var.t -> FrontCstr.t -> bool` such that `occurs x c` tests whether $\mathtt{x} \in V(\mathtt{c})$ and `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'
```

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'
```

### 2.2 Generating an Intermediate Certificate

To be protected against lying backends, we could 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 by Fouilhé et al. (2013) to design the first version of the VPL. 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. In `pexp`, we provide 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 to `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 our `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 understand where the ill-formness comes from in its oracle. Moreover, an oracle like `Back.proj1` needs to handle constraint identifiers for `Bind` according to their semantics in `Front.run`. This is particularly painful in operations like `join` operator of Section 5, which involve several spaces of constraint names (one for each "implication proof"). This consideration motivates the introduction of our new design pattern, where incorrect handling of constraint names is *statically* forbidden, thanks to OCaml typechecker.

### 2.3 Polymorphic LCF Style

We generalize LCF style in order to solve its soundness issue, as justified in Section 4. We also *conjecture* that the approach of Section 3 provides a sound link between the backend and a Coq extracted frontend, without the need for an intermediate AST. Moreover, an AST can still be generated if needed for another purpose (see Section 6).

Our idea 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. Intuitively, the lying function `memofst` from Section 2.1 exploits the fact that we have a *static* type of theorems, defined once for all. But, when we interpret constraints of the result $P'$ as theorems, they are relative to a given set of axioms: the input constraints of $P$. Hence, we need to have 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, such as `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})`.

We precise here that the type of `Back.proj` implementation must generalize the above signature, and not simply unify with it. This directly forbids `memofst` trick. Indeed, if we remove the type coercion from the preceding code of `memofst`, the type system infers `memofst: '_a -> '_a` where `'_a` is an existential type variable introduced for a sound typing of references, see Wright (1995) and Garrigue (2002). Hence, a cheating use of `memofst` would prevent `Back.proj` implementation from having an acceptable type.

## 3 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 3.1 recalls the may-return monad introduced by Fouilhé and Boulmé (2014) to overcome this issue. Section 3.2 explains how PFS oracles are embedded in this approach.

### 3.1 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 0).

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

Lemma congr: test one = test (S 0).
  auto.
Qed.
```

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

Indeed `(oracle one)` returns **`true`** whereas `(oracle (S 0))` 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 $\leadsto_A : ?A \to A \to \text{Prop}$. Intuitively, when "$k : ?A$" is seen as "$k \in \mathcal{P}(A)$", then "$k \leadsto a$" means that "$a \in k$". At extraction, $?A$ is extracted like $A$, and its binding operator is efficiently extracted as an OCaml let-in. See Fouilhé and Boulmé (2014) for more details.

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 0))↝b' → b=b'.
```

### 3.2 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 2.3. 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 2.3.

```
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 2.3, `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.[5]

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 satisfying `s`. In other words, `wcstr(sat p)` is the type of logical consequences of `p`, i.e. the type of its Farkas combination. 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, we can see `rep_sat` above 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. The next section argues for the soundness of this Coq proof.

## 4 SOUNDNESS OF PFS ORACLES IN IMPERATIVE ML

What are the conditions on the OCaml backend to ensure the correctness of our axiomatization/proof in Coq? The backend must be at least *type safe* in the sense defined below. Actually, we will precise this (too weak) first version in Definition 4.2.

*Definition 4.1 (type safety (too weak version)).* An external function is said *type-safe* if it only accesses values of allocated locations according to the type of these locations, and if its result is

[5]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).

compatible with its declared types (see example and counter-example below). An OCaml function is said type-safe if it is well-typed and if it only uses – directly or indirectly – type-safe external functions.

For example, the external constants `Obj.obj: Obj.t -> 'a` and `Obj.magic: 'a -> 'b` – which are implemented by the identity function – are not type-safe, whereas an external constant like `Hashtbl.hash: 'a -> int` – which actually only returns integers – is type-safe. However, at the current state-of-the-art, we do not know whether type safety is sufficient. Indeed, the correctness result on Coq extraction (Letouzey 2004, 2008) expresses that for any closed Coq term, the extracted term performs the same computations as the source term. But, we do not know exactly at which conditions the properties proved on a Coq function can be transferred into its extraction, when the latter is applied to an input that is not itself extracted from Coq. Stating precisely such an extended correctness result for Coq extraction is a challenge which is beyond the scope of this paper. An intermediate step is to consider how we could prove the soundness of the particular meta-reasoning done in Coq on our backend. This section shows that this meta-reasoning exploits a weak parametricity property of ML polymorphic types, which we call *parametric invariance*. This property expresses that (imperative) ML functions preserve any data-invariant attached to their polymorphic type variables. It is proved with denotational models of ML reference types of (Ahmed et al. 2002; Appel et al. 2007; Birkedal et al. 2011).

Section 4.1 recalls on examples the original parametricity of Wadler on pure system F. Section 4.2 illustrates how a Wadler's proof based on parametric invariance can be expressed in Coq, using the style of Section 3.2. Section 4.3 explains how parametric invariance is proved in the framework provided by Ahmed et al. (2002); Appel et al. (2007); Birkedal et al. (2011); Hobor et al. (2010). At last, Section 4.4 discusses the difficulties of establishing a full proof that our Coq design is sound.

### 4.1 Wadler's Theorems for Free

Wadler (1989) demonstrates – on pure polymorphic lambda-calculus (i.e. system F with basic types) – how to automatically deduce theorems about functions from their polymorphic types. These theorems are themselves a consequence of a meta-theorem called parametricity, given here as two statements:

(1) Any type $T$ induces a relation $(\!|T|\!)$ between values of $T$. Relation $(\!|T|\!)$ is here noted as an element of $\mathcal{P}(T \times T)$. If $T$ is a closed monomorphic type, then $(\!|T|\!)$ is the extensional equality. On polymorphic types, universal quantification over types is interpreted as conjunction over relations.
(2) for all closed terms $t : T$, we have $(t, t) \in (\!|T|\!)$.

This last property gives a "theorem for free" about $t$ derived only from its type $T$. We illustrate this on the following two examples $\texttt{pid}{:}\forall\alpha, \alpha \rightarrow \alpha$ and $\texttt{discr}{:}\forall\alpha, \alpha \rightarrow \texttt{int}$.

The relation $(\!|\forall\alpha, \alpha \rightarrow \alpha|\!)$ associated to polymorphic type $\forall\alpha, \alpha \rightarrow \alpha$ is

$$\{(f_1, f_2) \mid \forall\, R_\alpha\ x_1\ x_2,\ (x_1, x_2) \in R_\alpha \ \Rightarrow\ (f_1\ x_1, f_2\ x_2) \in R_\alpha\}$$

Hence, we deduce from $(\text{pid}, \text{pid}) \in (\!|\forall\alpha, \alpha \rightarrow \alpha|\!)$, that for any type $T$ and $x : T$, by taking

$$R_\alpha \triangleq \{(x_1, _) \in T \times T \mid x_1 = x\}$$

we have "$\forall x_1, x_1 = x \Rightarrow (\texttt{pid}\ x_1) = x$". In other words, `pid` is identity.

Similarly, $(\!|\forall\alpha, \alpha \rightarrow \texttt{int}|\!)$ is $\{(f_1, f_2) \mid \forall R_\alpha\ x_1\ x_2,\ (x_1, x_2) \in R_\alpha \ \Rightarrow\ (f_1\ x_1) =_{\texttt{int}} (f_2\ x_2)\}$.
Hence, for any type $T$, taking $R_\alpha \triangleq T \times T$, we deduce "$\forall (x_1\ x_2 : T), (\texttt{discr}\ x_1) = (\texttt{discr}\ x_2)$".

Thus `discr` is constant.

Imperative ML languages do not satisfy Wadler's parametricity, because they allow defining non-constant functions of type `'a -> int`, e.g. by returning a value depending on their number of calls. This is also the case of ML languages providing a function like OCaml `Hashtbl.hash` (e.g. a non-constant function of type `'a -> int`). However, it seems that any OCaml implementation of `pid` – that does not invoke (directly or indirectly) any external constant – is a pseudo-identity. Actually, such a `pid` may not exactly be the identity because it may not terminate normally or produce side-effects. More formally, we say that "`pid` *is a pseudo-identity*" when it satisfies "*if* (`pid x`) *returns normally a result* `y` *then* `y` *equals to* `x`".

In the following, we only consider *parametric invariance*: a weak version of parametricity that associates to each type a *predicate* instead of a *relation*, i.e. a unary relation like $\{(x_1, _) \in T \times T \mid x_1 = x\}$ above. This predicate is here called an *invariant*: all values of a given ML type satisfy its corresponding invariant.

### 4.2 Theorems for Free on ML Code through Coq Extraction

Wadler's proof that `pid` is a pseudo-identity is now mimicked in Coq and its extraction process. We actually follow the style of Section 3.2: a theorem about a ML polymorphic function is proved "*for free*" by instantiating its polymorphic type variable on a dependent type.

Let us build a Coq function `cpid` which *extraction* is "**`let`** `cpid x = pid x`", and which is proved to be a pseudo-identity. In the Coq source, for a type `B` and a value `x:B`, `(cpid x)` invokes `pid` on the type `{ y | y = x }`, which constrains it to produce a value that is equal to `x`. Below, operators `>>=` and `ret` are respectively the bind and unit operators of the may-return monad presented in Section 3.1. Function `proj1_sig` returns the first component of a dependent pair of type `{y:B | y=x}`: its result has type `B`.

```
Axiom pid: ∀ {A}, A → ? A.

Program Definition cpid {B} (x:B): ? B :=
  (pid (A:={ y | y = x }) x) >>= (fun z ⇒ ret (proj1_sig z)).

Lemma cpid_correct A (x y:A): (cpid x) ⇝ y → y=x.
```

Let us point out that we cannot prove in Coq that `pid` – declared as the axiom given above – is a pseudo-identity. Indeed, we provide a model of this axiom where `pid` detects – through some dynamic typing operators – if its parameter `x` has a given type `Integer` and in this case returns a constant value, or otherwise returns `x`. Such a counter-example already appears in (Vytiniotis and Weirich 2007). This function is now provided in Java syntax.

```
final static Integer o = new Integer(0);
static <A> A pid(A x) {
  if (x instanceof Integer) // A is Integer, because Integer is final
    return (A)o;
  return x;
}
```

The soundness of `cpid` extraction is thus related to a nice feature of ML: type-safe polymorphic functions cannot inspect the type to which they are applied. In other words, type erasure in ML semantics ensures that ML functions handle polymorphic values in a uniform way: this is the motivation for parametricity reasoning.

However, a similar counter-example can be built for OCaml by using an external C function that is sound with type `'a -> 'a` (i.e. for all ML type $T$, it behaves like a function of type $T \rightarrow T$). Such a function inspects the bit of its parameter that tags unboxed integers, and returns integer 0 when instantiated on type `int`, or behaves like an identity otherwise. Here, we see that Definition 4.1 is not strong enough, since it allows such an implementation of type `'a -> 'a`. We will thus fix this definition in Section 4.4.

In summary, our Coq proof is not about `pid`, but about `cpid` which instantiates `pid` on a dependent type. Actually, `cpid` and `pid` coincide, but only *in the extracted code*. We recover here an idea of Bernardy and Moulin (2012, 2013): our parametricity proofs correspond to the fact that the invariants instantiating polymorphic type variables in the Coq proofs are syntactically removed by Coq extraction. The parametric invariance theorem presented in the next section ensures that these invariants are still preserved on the extracted ML code.

### 4.3 Formalization of Parametric Invariance in Step-Indexed Kripke Models

The formalization of parametric invariance appears in (Birkedal et al. 2011) for a variant of System F extended with higher-order references *a la* ML. Their article provides a denotational model for this type system. The semantics of a type is called "*unary logical relation*" by Birkedal et al.: it actually formalizes what we have called "*invariant*". From our applications-centric point-of-view, we prefer "*invariant*" to "*unary logical relation*".

Let us now motivate why such a denotational model is necessarily complex. Let us consider the following `fixpoint` function of type `(('a -> 'b) -> ('a -> 'b)) -> 'a -> 'b` which builds a generic fixpoint operator from a higher-order ML reference, but without explicit recursion.

```
let fixpoint f =
  let fix = ref (fun x -> failwith "init") in
  (fix := fun x -> f (!fix) x);
  !fix;;
```

Typically, any recursive function is then definable from `fixpoint` like the following Fibonacci function:

```
let fib: int -> int =
  fixpoint (fun f n -> if n <= 1 then n else f(n-1)+f(n-2))
```

In order to represent values storable at location `fix` in the heap, the set of heaps must itself be represented as a kind of recursive set. Indeed, heaps are map from memory locations to values, and values can contain functions depending on heaps. But such a recursive set can not be defined in naive set-theoretic models of typed lambda-calculi. Thus, Ahmed et al. (2002) have proposed to represent heaps by using a family of sets, itself indexed by a number of "*computation steps*" defined from small-step semantics. Alternatively, in (Birkedal et al. 2011), heap invariants correspond to worlds of a Kripke model that are recursively defined in a category of ultra-metric spaces.

This section summarizes how we have adapted the proof of parametric invariance outlined by Birkedal et al. (2011) on a tiny subset of imperative ML. See Appendix A for the full details. Actually, our proof is not fully faithful to Birkedal et al., because we do not define worlds through ultra-metric spaces. We instead keep the original step-indexed approach of Ahmed et al. (2002) and further developed in (Appel et al. 2007; Hobor et al. 2010) that Birkedal et al. have generalized. This more elementary approach is sufficient to prove parametric invariance. Birkedal et al. are indeed more ambitious: they prove the soundness of a separation logic.

The step-indexed model of Ahmed et al. provides a denotational model for *types* of polymorphic and imperative lambda-calculus. It also provides a proof of *type safety* w.r.t a small-steps semantics.

Birkedal et al. reformulates this model into a model of *invariants*. In other words, whereas Ahmed et al. have a purely semantic definition of types, Birkedal et al. introduces a syntax for types. This allows distinguishing between *types* (syntax) and *invariants* (semantics). This clarifies the powerful interpretation of polymorphic types where *type variables* are substituted by *invariants*, which are not necessarily themselves associated to an existing type. Let us recall the `pid` example where we have replaced type variable `'a` by invariant $\{ y \mid y = x \}$ which is not the one of a ML type. In particular, the invariant of such a polymorphic type – called here a "*parametric invariant*" – is impredicative (its variables range over all invariants, including itself). Hence, "*type preservation*" theorem of Ahmed et al. is reformulated into an "*invariant preservation*" theorem. As underlined above, this theorem demonstrates that ML evaluation preserves richer "types" than ML types. This is a required step on the path to prove that, under some conditions that are still to precisely determine, ML evaluation preserves types given in Coq through extraction.

A minor contribution of our formalization is to target a ML type system with the standard *value-form restriction* (Wright 1995), which provides a sound type inference of polymorphic types in presence of references. On the contrary, the type system studied by Ahmed et al. provides unrestricted polymorphism, which requires explicit annotations from users. For example, in ML, an expression like "`ref(fun x -> x)`" is of type "`('_a -> '_a) ref`" where type variable `'_a` can not be generalized without breaking type preservation. This problem does not appear in imperative SYSTEM F. Indeed, either the user introduces a reference to a polymorphic value like "$\mathbf{ref}(\Lambda\alpha, \lambda x{:}\alpha, x)$" of type "$\mathbf{ref}(\forall\alpha, \alpha \to \alpha)$" which has no equivalent in ML (because this type is not in *prenex* form). Or, the user introduces a polymorphic allocation like "$\Lambda\alpha, \mathbf{ref}(\lambda x{:}\,\alpha, x)$" of type "$\forall\alpha, \mathbf{ref}(\alpha \to \alpha)$" and which semantics is to allocate a reference at each instantiation of $\alpha$. Polymorphic ML values of type "`unit -> (('a -> 'a) ref)`" have actually the same behavior: a reference is allocated at each function call.

Moreover, as we consider the *progress* property as an orthogonal issue, we have slightly simplified the original framework by considering big-steps semantics instead of small-steps. This makes straightforward our adaptation of Ahmed et al. proof.

Let us now detail the results of our formalization. It defines a partially ordered Kripke frame $(W, \sqsubseteq)$ where $W$ is a set equipped with a partial order $\sqsubseteq$. A world $w$ of $W$ represents a *heap context*: a finite map from heap locations to sets of values. And $w_1 \sqsubseteq w_2$ means that $w_2$ extends $w_1$ by allocating new locations.[6] This Kripke frame leads to a Kripke model of intuitionistic logic where propositions are set of worlds that are closed for $\sqsubseteq$

$$\text{Prop} \triangleq \{ p \in \mathcal{P}(W) \mid \forall\, w_1\, w_2, (w_1 \sqsubseteq w_2 \,\wedge\, w_1 \in p) \Rightarrow w_2 \in p \}$$

Given $V$ the set of ML values, an *invariant* $\iota$ is by definition a function of $V \to \text{Prop}$. An *invariant context* $I$ is a finite map from type variables $\alpha$ to invariants. Given a polymorphic type $\sigma$ and an invariant context $I$ such that $\text{FV}(\sigma) \subseteq \text{dom}(I)$, we associate to $\sigma$ an invariant written $[\![\sigma]\!]_I$. We do not detail here the formal definition of $[\![\sigma]\!]_I$ and only use its properties detailed below.

The theorem of *parametric invariance* depends on the polymorphic typing of *closed* expressions $\vdash e : \sigma$ – meaning that expression $e$ has type $\sigma$ – and usual big-steps semantics $\langle e/h \rangle \Downarrow \langle v/h' \rangle$ – meaning that expression $e$ for initial heap $h$ evaluates to value $v$ with final heap $h'$. This theorem is actually a simple consequence of the "*invariant preservation*" theorem mentioned above. It suffices to ensure that any value computed from a well-typed closed expression satisfies the invariant associated to its type.

[6] Worlds also contain a *stratification* (or *step-indexed*) level in order to build the recursive structure of $W$ by approximations. This level bounds the number of dereference *steps* allowed to the current evaluation. Hence, $w_1 \sqsubseteq w_2$ also forbids the stratification level of $w_2$ to increase w.r.t the one of $w_1$. But, these details can remain hidden when applying our main results.

Theorem (Parametric Invariance).
*Under assumptions* $\mathrm{FV}(\sigma) \subseteq \mathrm{dom}(I)$ *and* $\vdash e : \sigma$ *and* $\langle e/h \rangle \Downarrow \langle v/h' \rangle$
*we have* $[\![\sigma]\!]_I\,(v) \neq \emptyset$

The following lemma of *invariants instantiation* is convenient for reasoning with parametric invariants, as shown on the example below. It abstracts big-steps $\langle e/h \rangle \Downarrow \langle v/h' \rangle$ as the may-return judgment $w \Vdash e \leadsto v$ – formally defined in Appendix – where the final heap $h'$ is hidden and the initial heap $h$ is abstracted as a world $w$. This is formally expressed by the following lemma.

Lemma (May-Return Abstraction). $\exists w, w \Vdash e \leadsto v \quad \Leftrightarrow \quad \exists h, \exists h', \langle e/h \rangle \Downarrow \langle v/h' \rangle$

Lemma (Invariants Instantiation). *Assuming* $\mathrm{FV}(\sigma) \subseteq \mathrm{dom}(I)$*, invariant* $[\![\sigma]\!]_I$ *satisfies the following properties according to the syntax of* $\sigma$*:*

- *for type variable* $\alpha$*,* $[\![\alpha]\!]_I = I[\alpha]$
- *for basic type* $\beta$*,* $[\![\beta]\!]_I\,(v) = \text{if } \vdash v : \beta \text{ then } W \text{ else } \emptyset$
- *for two open monomorphic types* $\tau_1$ *and* $\tau_2$*, property* $w \in [\![\tau_1 \to \tau_2]\!]_I\,(v)$ *implies that for all value* $v_1$ *such that* $w \in [\![\tau_1]\!]_I\,(v_1)$ *and all value* $v_2$ *such that* $w \Vdash (v\ v_1) \leadsto v_2$*, there exists* $w'$ *such that* $w \sqsubseteq w'$ *and* $w' \in [\![\tau_2]\!]_I\,(v_2)$
- *for all* $\sigma$*, for all invariant* $\iota$*, if* $\alpha \notin \mathrm{dom}(I)$ *then* $[\![\forall\alpha, \sigma]\!]_I\,(v) \subseteq [\![\sigma]\!]_{I;\{\alpha\mapsto\iota\}}\,(v)$ *where* $I; \{\alpha \mapsto \iota\}$ *is the map extending* $I$ *by associating invariant* $\iota$ *to type variable* $\alpha$*.*

Example (Reasoning with Parametric Invariants).
*Let us assume* $w \in [\![\forall\alpha, \alpha \to \alpha]\!]_I\,(v)$ *and* $w \Vdash (v\ v_1) \leadsto v_2$*. We can prove* $v_2 = v_1$*.*

Proof. We instantiate the parametric invariant in hypothesis $w \in [\![\forall\alpha, \alpha \to \alpha]\!]_I\,(v)$. Let us choose $\alpha \notin \mathrm{dom}(I)$ and let us define $\iota(v_0) \triangleq \{\, w \in W \mid v_0 = v_1 \,\}$. We get $w \in [\![\alpha \to \alpha]\!]_{I;\{\alpha\mapsto\iota\}}\,(v)$. Since $[\![\alpha]\!]_{I;\{\alpha\mapsto\iota\}}\,(v_1) = W$, we have $[\![\alpha]\!]_{I;\{\alpha\mapsto\iota\}}\,(v_2) \neq \emptyset$. Hence, $v_2 = v_1$. □

Our formalization also provides an inversion lemma to reason about may-return relations (see Appendix). This allows conducting correctness proofs on frontend computations involving oracles – like `front_proj_correctness` – in the style of Section 3.2.

### 4.4 Toward a Soundness Result about Coq Extraction in Presence of External Code

For the sake of simplicity, our formalization in Appendix involves only pure external constants in OCaml (i.e. external constants that do not access the heap). But we can accept a large class of external constants as soon as they satisfy our "invariant preservation" property. This leads us to the following revised version of type safety.

*Definition 4.2 (Type Safety (Revised Version)).* An OCaml function is said *type-safe* if it does use – directly or indirectly – only external constants that satisfy their ML type and their associated parametric invariant.

This revised version is thus strictly stronger than the unsound Definition 4.1. For instance, it may accept external polymorphic constants like "`Weak.get: 'a Weak.t -> int -> 'a option`" only if some expert is able to ensure that they satisfy the associated parametric invariants.

However, for constants with a *closed monomorphic type*, the two versions of type safety are *equivalent*. In other words, the invariant associated to a closed monomorphic type trivially holds for any constant satisfying the type preservation property. Hence, we may hope to check such a property on monomorphic external constants with standard static analysis tools.

Conjecture 4.3. *Assuming that our oracle is type-safe (following Definition 4.2), we conjecture that our proof in Coq of Section 3.2 is sound: the property proved on the frontend in Coq cannot be wrong when the front-end is extracted and linked to the actual oracle.*

In order to prove this conjecture, we would need to extend the correctness of Coq extraction proved by Letouzey (2004). Typically, we would expect a result stating that under some conditions which remain to be precisely established, a function extracted from Coq could be applied to impure ML values while still satisfying the properties proved in Coq.

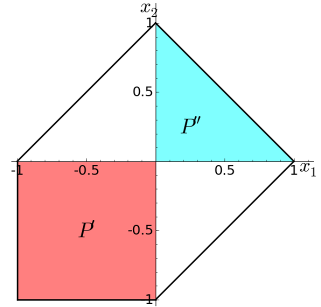
The proof of (Letouzey 2004) suggests that this result probably requires to define a type system that both embeds CIC (the type system behind Coq) and impure ML, in order to define the extraction as a transformation within this type system. Such a type system would both allow expressing the typing judgments and the evaluation rules of CIC and ML, with ML computations boxed in a monad. Moreover, in this framework, CIC types would abstract step-indexed Kripke semantics of their extraction.

Building such a framework seems far beyond the scope of this paper. Moreover, there would probably still have a big gap between the ML fragment embedded in this hypothetical type system and the actual OCaml implementation.

## 5 THE FLEXIBLE POWER OF PFS ILLUSTRATED ON CONVEX-HULL

This section provides an advanced usage of polymorphic factories through the `join` operator. It illustrates the flexible power of PFS, by deriving `join` from the projection operator of Section 2.3. 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




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

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

then, as illustrated on the right hand-side figure,
$P' \sqcup P'' \triangleq \{x_1 \geq -1,\ x_2 \geq -1,\ x_1{+}x_2 \leq 1,\ x_2{-}x_1 \geq -1,\ x_2{-}x_1 \leq 1\}$

The correctness of `join`, given in Definition 5.1, is reduced to two implications themselves proved by Farkas' lemma. More precisely, on a computation (`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 front-end 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 (`join` $P'$ $P''$) satisfies $(P' \Rightarrow P) \ \wedge\ (P'' \Rightarrow P)$ .

### 5.1 Extended Farkas Factories

The factory operations of Definition 2.2 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 1.1 for polyhedra with equalities and strict inequalities.

*Definition 5.2 (Extended Farkas Combination).* Besides operations + and $\cdot$ of Definition 2.2, an extended Farkas combination may invoke one of the three operations:

- `weaken`: $\Downarrow (t \bowtie 0) \triangleq t \geq 0$, for all linear term $t$ and $\bowtie \in \{=, \geq, >\}$.
- $\mathtt{cte}(n, \bowtie) \triangleq n \bowtie 0$ assuming $n \in \mathbb{Q}$ and $n \bowtie 0$.
- `merge`: $(t \geq 0)\ \&\ (-t \geq 0) \triangleq (t = 0)$, for all linear term $t$.

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. The 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 2.2.
(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 $\{\geq, >\} \backslash \{\bowtie\}$, 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 $\mathtt{cte}(\lambda_0, \geq)$ 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 $\mathtt{cte}(\lambda_0, >)$ (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 2.2.
(3) If $P_1$ is satisfiable. Let "$t = 0$" be a constraint of $P_2$. 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 $P_1 \Rightarrow -t \geq 0$.

□

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. Here, constant `top` of the factory corresponds to a constraint noted $\top$ and defined as a shortcut for $\mathtt{cte}(0, =)$ that encodes constraint $0 = 0$. Hence, $\top$ is neutral for operations + and $\cdot$ on constraints. It is thus a very convenient default value in our PFS oracles.

```
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;
}
```

Fields `weaken` and `merge` correspond respectively to operators $\Downarrow$ and &. Type `cmpT` is our enumerated type of comparisons representing $\{\geq, >, =\}$.

### 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.

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

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 projecting variables $\alpha'$, $\alpha''$, $\boldsymbol{x}'$ 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 $\boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''$. To overcome this issue, we apply 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 $\alpha'$ and $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$ by $\alpha''$, 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

$$\begin{cases} -y'_1 \geq 0,\ -y'_2 \geq 0,\ y'_1 \geq -\alpha',\ y'_2 \geq -\alpha' \\ y''_1 \geq 0,\ y''_2 \geq 0,\ -y''_1 - y''_2 \geq -\alpha'' \\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1 \\ x_1 = y'_1 + y''_1,\ x_2 = y'_2 + y''_2 \end{cases}$$

Operator `join` finally consists in projecting variables $\alpha'$, $\alpha''$, $\boldsymbol{y}'$ 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 $P$ 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 proceeds 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 constraint is attached to a unique identifier, as explained in Section 2.2. 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.proj2_list` that extends `Back.proj2` from Section 2 by projecting several variables one after the other instead of a single one. Assume that `Back.proj2_list` $P_H$ $[x_1, \dots, x_q]$ returns $(P,\ \mathbf{\Lambda})$ where $\mathbf{\Lambda}$ is a certificate of type `pexp` showing that $P$ is a logical consequence of $P_H$. Actually, $\mathbf{\Lambda}$ can be viewed as a matrix where each line contains the coefficients of a Farkas combination of $P_H$, and it fulfills

$$\mathbf{\Lambda} \cdot P_H = P \tag{4}$$

Fouilhé et al. showed that $\mathbf{\Lambda}$ can be decomposed into three parts: $\mathbf{\Lambda}_1$ speaking about constraints of $P'$, $\mathbf{\Lambda}_2$ speaking about constraints of $P''$ and $\mathbf{\Lambda}_3$ speaking about remaining constraints.

$$\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}''}_{\mathbf{\Lambda}_2},\ \underbrace{\alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''}_{\mathbf{\Lambda}_3}\}$$

Note that equation (4) holds whatever the value of variables $\alpha',\ \alpha'',\ \boldsymbol{y}'$ and $\boldsymbol{y}''$. The key idea is to assign values to the projected variables $\alpha',\ \alpha'',\ \boldsymbol{y}'$ and $\boldsymbol{y}''$ in encoding (3). Considering assignment $\sigma_1 \triangleq (\alpha' = 1,\ \alpha'' = 0,\ \boldsymbol{y}'' = \boldsymbol{0})$, it becomes $\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{y}' \geq \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{\boldsymbol{0} \geq \boldsymbol{0}}_{\mathbf{\Lambda}_2},\ \underbrace{1 \geq 0,\ 0 \geq 0,\ 1 + 0 = 1,\ \boldsymbol{x} = \boldsymbol{y}'}_{\mathbf{\Lambda}_3}\}$

that simplifies into

$$\{\boldsymbol{x} \mid \underbrace{\boldsymbol{A}'\boldsymbol{x} \geq \boldsymbol{b}'}_{\mathbf{\Lambda}_1},\ \underbrace{1 \geq 0}_{\mathbf{\Lambda}_3}\} \tag{5}$$

which is equivalent to $P'$. Let us call $\lambda$ the coefficient of $1 \geq 0$ in $\mathbf{\Lambda}_3$. Then, we deduce from $(\mathbf{\Lambda}_1 \cdot \boldsymbol{A}')\boldsymbol{x} \geq \boldsymbol{b}' + \lambda \cdot (1 \geq 0) = P$ that $P' \Rightarrow P$. The same reasoning applied with assignment $\sigma_2 \triangleq (\alpha' = 0,\ \alpha'' = 1,\ \boldsymbol{y}' = \boldsymbol{0})$ leads to $P'' \Rightarrow P$.

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

In PFS, the oracle of `join` has the following type :

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

Polyhedra $P'$ and $P''$ come with their own polymorphic type, respectively `'c1` and `'c2`. The polymorphic type of `Back.join` ensures that it returns a pair of polyhedra $(P_1, P_2)$ of type `'c1 list * 'c2 list` such that $P' \Rightarrow P_1$ and $P'' \Rightarrow P_2$. In practice, $P_1$ and $P_2$ should represent the same polyhedron. As a consequence, `Back.join` must take as parameters two factories, one for each polymorphic type.

We said that for computing the convex hull, `join` projects variables $\alpha',\ \alpha'',\ \boldsymbol{y}'$ and $\boldsymbol{y}''$ from $P_H$. Recall that the projection operator that we defined for PFS in Section 2.3 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 projecting a list of variables.

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

At this point, we instantiate the factory of `Back.proj_list` in order to produce the pair of polyhedra with their two distinct polymorphic types. Indeed, although the parameter `'c lcf` of `Back.proj_list` is designed to be provided by the frontend, nothing prevents it from being tuned

by the backend. This is where the flexibility of PFS comes into play! We combine the two factories of types `'c1 lcf` and `'c2 lcf` into a new one of type `('c1*'c2) lcf` as follows.

```
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');
}
```

This new factory computes with frontend constraints from $P'$ and $P''$ in parallel: it corresponds to the *direct product* of the two Farkas factories. Still, to be able to use such a factory, each backend constraint 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 apply the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$ as explained above. But actually, we *do not need* to apply these changes of variable on frontend constraints. As mentioned earlier, the two Farkas combinations of `join` are found by evaluating the result of the projection of $P_H$ on two assignments, respectively $\sigma_1$ and $\sigma_2$. This evaluation makes variables $\boldsymbol{y}'$ and $\boldsymbol{y}''$ both vanish, as in Equation (5). Thus, to build the frontend version of constraints of $P_H$, we evaluate them directly on each assignment as follows:

$$
\begin{array}{cccccccccc}
 & \texttt{BackCstr.t} & \longrightarrow & \texttt{BackCstr.t} & * & ( & \texttt{'c1} & * & \texttt{'c2} & ) \\
\multirow{3}{*}{$\boldsymbol{A'x'} \geq \boldsymbol{b'} \left\{ \right.$} & A'_1\boldsymbol{x}' \geq b'_1 & \longrightarrow & A'_1\boldsymbol{y}' \geq b'_1 & , & ( & \underbrace{\left[A'_1\boldsymbol{y}' \geq \alpha' b'_1\right]_{\sigma_1}}_{A'_1\boldsymbol{x}' \geq b'_1} & , & [\top]_{\sigma_2} & ) \\
 & \dots & & & & & & & & \\
 & A'_p\boldsymbol{x}'' \geq b'_p & \longrightarrow & A'_p\boldsymbol{y}' \geq b'_p & , & ( & \underbrace{\left[A'_p\boldsymbol{y}' \geq \alpha' b'_p\right]_{\sigma_1}}_{A'_p\boldsymbol{x}' \geq b'_p} & , & [\top]_{\sigma_2} & ) \\
\multirow{3}{*}{$\boldsymbol{A''x''} \geq \boldsymbol{b''} \left\{ \right.$} & A''_1\boldsymbol{x}'' \geq b''_1 & \longrightarrow & A''_1\boldsymbol{y}'' \geq b''_1 & , & ( & [\top]_{\sigma_1} & , & \underbrace{\left[A''_1\boldsymbol{y}'' \geq \alpha'' b''_1\right]_{\sigma_2}}_{A''_1\boldsymbol{x}'' \geq b''_1} & ) \\
 & \dots & & & & & & & & \\
 & A''_q\boldsymbol{x}' \geq b''_q & \longrightarrow & A''_q\boldsymbol{y}'' \geq b''_q & , & ( & [\top]_{\sigma_1} & , & \underbrace{\left[A''_q\boldsymbol{y}'' \geq \alpha'' b''_q\right]_{\sigma_2}}_{A''_q\boldsymbol{x}'' \geq b''_q} & )
\end{array}
$$

Finally, we add constraints $\alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1$. As others, these constraints need to have type `BackCstr.t * ('c1 * 'c2)`. However, they contain variables $\alpha'$ and $\alpha''$ that were not present in the input polyhedra $P'$ and $P''$. Here again, we build directly their evaluation in $\sigma_1$ and $\sigma_2$. 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.

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

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 one on 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 (2017) 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 2.2. The tactic then applies a Coq version of the `Front.run` interpreter of Section 2.2 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 2.3 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 2.3.

```
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; (* ... *) }
```

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

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

Fig. 4. Elimination of Top Nodes and Factorization of Mul_ Nodes in the DAG

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 4, 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 4.

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.

### 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).

Beyond interactive provers, producing certificates of unsatisfiability has become mandatory for state-of-the-art Boolean SAT-solvers. Indeed, certificates of unsatisfiability have been required for the UNSAT tracks since SAT Competition 2013. In 2016, they were required – in DRAT format (Wetzler et al. 2014) – for all solvers in the *Main* track of the SAT Competition.[7]

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 LCF 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 non-negative linear constraints which are logical consequences of their inputs.

In Polymorphic LCF style, oracles produce these witnesses as ordinary ML values (e.g. linear constraints). In other words, instead of building an AST 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 advantages over AST style. First, it makes the oracle development easier. Ill-handling of certified operators is much straightforward to debug. Without an AST to build, it naturally removes cumbersome details such as handling of binders. 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). At last, polymorphism makes witness generation very flexible and modular. Generating a compact AST is still possible if necessary, e.g. for embedding an oracle within a Coq tactic.

We strongly believe that PFS could be used with other applications. For instance, the nonlinear support based on Handelman's theorem that was added into the VPL by Maréchal et al. (2016) could be easily certified using a factory that provides nonlinear multiplications. More significantly,

[7] http://baldur.iti.kit.edu/sat-competition-2016

certifying UNSAT answers of a Boolean SAT-Solver can be achieved by a similar approach, by using resolution proofs instead of Farkas certificates (Keller 2013). This seems to indicate than our approach is also relevant to a large class of coNP-hard problems.

Our approach applies parametric invariance of ML polymorphic types in order to simplify the certification in CoQ of oracle-based computations. Another interesting application of parametricity is developed in (Anand and Morrisett 2017; Cohen et al. 2013; Keller and Lasson 2012). These works directly apply relational parametricity on CoQ types. Their parametricity proofs typically help to automate changes of data representation inside CoQ developments.

## ACKNOWLEDGMENTS

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

## REFERENCES

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References Embeddable in Higher-Order Logic. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 75.

Abhishek Anand and Greg Morrisett. 2017. Revisiting Parametricity: Inductives and Uniformity of Propositions. *CoRR* abs/1705.01163 (2017). http://arxiv.org/abs/1705.01163

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Principles of Programming Languages (POPL)*. ACM Press, 109–122.

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.

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming* 72, 1–2 (2008). arXiv:cs/0612085

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).

Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society.

Jean-Philippe Bernardy and Guilhem Moulin. 2013. Type-theory in color. In *International Conference on Functional programming (ICFP)*. ACM Press.

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.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke Models over Recursive Worlds. In *Principles of Programming Languages (POPL)*. ACM Press, 119–132.

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. 2017. A Coq Tactic for Equality Learning in Linear Arithmetic. (April 2017). https://hal.archives-ouvertes.fr/hal-01505598 preprint.

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems (PLS) (LNCS)*, Vol. 7792. Springer, 41–60.

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

Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Certified Programs and Proofs (CPP) (LNCS)*, Vol. 8307. Springer, 147–162.

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.

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. 2010. A Theory of Indirection via Approximation. In *Principles of Programming Languages (POPL)*. ACM Press, 171–184.

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

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*.

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.

Chantal Keller. 2013. Extended Resolution as Certificates for Propositional Logic. In *Proof Exchange for Theorem Proving (EPiC Series in Computing)*, Vol. 14. EasyChair, 96–109.

Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (LIPIcs)*, Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 381–395.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). arXiv:inria-00415861

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2013-2016. *The OCaml System*. INRIA.

Pierre Letouzey. 2004. *Certified functional programming, Program extraction within Coq proof assistant*. Ph.D. Dissertation. Université de Paris XI Orsay.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Computability in Europe (CiE) (LNCS)*, Vol. 5028. Springer, 359–369.

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, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. 2016. Polyhedral Approximation of Multivariate Polynomials Using Handelman's Theorem. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*. Springer, 166–184.

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.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

The Coq Development Team. 2016. *The Coq proof assistant reference manual – version 8.6*. INRIA.

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.

Dimitrios Vytiniotis and Stephanie Weirich. 2007. Free Theorems and Runtime Type Representations. *Electronic Notes in Theoretical Computer Science* 173 (2007), 357–373.

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

Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Theory and Applications of Satisfiability Testing (SAT) (LNCS)*, Vol. 8561. Springer, 422–429.

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

## A PARAMETRIC INVARIANCE FOR A TINY ML

We formalize parametric invariance for the tiny expression language of Definition A.3. This language supports polymorphic higher-order functions with references, which allows deriving a polymorphic fixpoint like in Section A.1. It is an "intermediate" language that only aims to provide a simple framework to formalize parametric invariance. A usual source expressions like "`ref` $e$" needs to be translated into this intermediate language as "`let` $x$ `=` $e$ `in ref` $x$" (using a fresh variable $x$). See Example A.8 for other examples. Such a translation should preserve operational semantics and static typing.

Our framework is quite closed to usual ML frameworks – like for instance (Pottier and Rémy 2005) – except that we specialize the definitions in order to simplify our formalization of parametric invariance. Our framework is thus not well-suited to prove other desirable properties of ML languages like progress or type inference. See Section A.5 for a discussion about extensions of this framework. It is also inspired by (Birkedal et al. 2011).

This formalization is organized as follows. Section A.1 provides a non-trivial example of reasoning by parametric invariance. Section A.2 defines a ML subset: its syntax, its big-step semantics, and its type system. Section A.3 defines semantics of our ML types as invariants. It also proves the core theorem of our formalization – called *invariant preservation* (Theorem A.32). This theorem states that well-type expressions of type $\sigma$ can only compute values satisfying invariant $[\![\sigma]\!]$ associated to type $\sigma$. It is thus very similar to type preservation theorems. Its proof relates "evaluation-steps" to "typing-steps" and shows that evaluation preserves invariants (and types). Section A.4 reformulates Theorem A.32 with a layer of abstraction, in order to hide explicit reasoning on heaps. This allows correctness proofs based on Polymorphic Factory Style. In particular, parametric invariance (Theorem A.37) is simply a reformulation of Theorem A.32 in this more abstract layer.

### A.1 Parametric Invariance and Higher-Order References

Before presenting the proof of parametric invariance on a small subset of imperative ML in next sections, let us illustrate its expressive power on higher-order imperative code. This example suggests that this proof cannot be elementary. First, we define below a generic `fixpoint` function of type `(('a -> 'b) -> ('a -> 'b)) -> 'a -> 'b` from a higher-order ML reference, but without explicit recursion.

```
let fixpoint f =
  let fix = ref (fun x -> failwith "init") in
  (fix := fun x -> f (!fix) x);
  !fix;;
```

Typically, any recursive function is then definable from `fixpoint` like the following Fibonacci function:

```
let fib: int -> int =
  fixpoint (fun f n -> if n <= 1 then n else f(n-1)+f(n-2))
```

Second, we certify – by parametric invariance on the type of `fixpoint` – a `while_loop` function which is extracted on the following ML function of type `('a -> bool)-> ('a -> 'a)-> 'a -> 'a`:

```
let while_loop cond body init =
  fixpoint (fun loop s -> if (cond s) then loop (body s) else s) init
```

Our definition of `while_loop` in Coq proves a "loop-invariant" theorem which mimics the usual Hoare logic rule for *partial correctness*. Below, `S` is the type of explicit states (corresponding to the

above variable `'a` on OCaml extraction) and `I` is the invariant. The Hoare's logic rule is embedded through the dependent types of `body`, of `init` and of the result. Here, `cond` is restricted to be pure.

```
Axiom fixpoint: ∀ {A B}, ((A → ? B) → (A → ? B)) → ? (A → ? B).

Program Definition while_loop {S} (I: S → Prop) (cond: S → bool)
  (body: {s | I s ∧ cond s = true } → ? {s | I s})
  (init: {s | I s })
: ? {s | I s ∧ cond s = false}
:= fixpoint (A:={s | I s}) (B:={s | I s ∧ cond s = false})
      (fun loop s ⇒
         match (cond s) with
         | true ⇒ (body s) >>= loop
         | false ⇒ ret (A:={s | I s ∧ cond s = false}) s
         end)
       >>= (fun f ⇒ f init).
```

Note that this "**`Program Definition`**" generates proof obligations that are automatically discharged by Coq.

Let us insist on the fact that this parametricity reasoning is valid for any `fixpoint` function of the above type, even for the following *wrong* memoized fixpoint which mixes up the outputs of recursive calls.

```
let wrong_fixpoint f =
  let memo = ref None in
  let rec fp x =
    match !memo with
    | None -> let y=(f fp x) in (memo:=Some y); y
    | Some y -> y
  in fp
```

A ML function of type `(('a -> 'b)-> ('a -> 'b))-> 'a -> 'b` – such as `wrong_fixpoint` above – may not correctly compute the fixpoint of its argument. Typically, when replacing `fixpoint` by `wrong_fixpoint` in `fib` example, it does not compute the expected Fibonacci function. However, parametric invariance makes ML typing ensures "for free" that such a function computes a correct fixpoint (modulo normal termination) in the `while_loop` example. This would be also the case for any tail-recursive computation.

This example illustrates the power of ML typing, expressed by parametric invariance. From the code of `fixpoint` or `wrong_fixpoint` above, ML typechecker is able to infer a type which corresponds to a kind of Hoare logic rule for such "loop operators".

## A.2 Semantics and Typing of a Tiny ML

Our intermediate language is parameterized by a simple set of basic monomorphic constants and their type. For the sake of simplicity, we restrict these constants to be $n$-ary functions with $n \leq 1$ like the one of Example A.1. By using other constructs of our intermediate language of Definition A.3, we can embed standard programming constructs as macros – like in Example A.8 – or as definitions like in `fixpoint` example.

*Example A.1 (Basic Values and their Type).* Our set of basic constants contains at least the three disjoint types:

$$\mathtt{unit} \triangleq \{()\} \qquad \mathtt{bool} \triangleq \{\mathtt{true}, \mathtt{false}\} \qquad \mathtt{nat} \triangleq \mathbb{N}$$

The set of basic constants also includes the following unary functions

$$\texttt{not}\left|\begin{array}{ccc}\texttt{bool} & \to & \texttt{bool}\\ \texttt{true} & \mapsto & \texttt{false}\\ \texttt{false} & \mapsto & \texttt{true}\end{array}\right. \qquad \texttt{pos}\left|\begin{array}{ccc}\texttt{nat} & \to & \texttt{bool}\\ 0 & \mapsto & \texttt{false}\\ n \in \mathbb{N}^+ & \mapsto & \texttt{true}\end{array}\right.$$

$$\texttt{assert}\left|\begin{array}{ccc}\texttt{bool} & \to & \texttt{unit}\\ \texttt{true} & \mapsto & ()\end{array}\right. \qquad \texttt{succ}\left|\begin{array}{ccc}\texttt{nat} & \to & \texttt{nat}\\ n & \mapsto & n+1\end{array}\right. \qquad \texttt{pred}\left|\begin{array}{ccc}\texttt{nat} & \to & \texttt{nat}\\ n \in \mathbb{N}^+ & \mapsto & n-1\end{array}\right.$$

*Definition A.2 (Basic Values and their Type).*

- We assume an (enumerable) set $C$ of *basic constants* written $c$ (e.g. implicitly $c \in C$).
- We assume a finite set of *basic types* such that each basic type $\beta$ is a subset of $C$ which is disjoint other basic types.
- A type $b$ of constants follows the syntax $b ::= \beta \mid \beta \to \beta$
- We assume a total *typing* function $\Delta$ from $C$ to types $b$ such that forall $c$ if $\Delta(c) = \beta$ for some $\beta$, then $c \in \beta$.
- Moreover, for each constant $c$ such that $\Delta(c) = \beta_1 \to \beta_2$, we assume an evaluation function $E_c$ given as a partial map from $\beta_1$ to $\beta_2$.

*Definition A.3 (Syntax of Intermediate Expressions).* We define the abstract syntax of expressions $e$ using the following BNF

$$\begin{array}{rl} e ::= & \texttt{ref}\,a \mid a := a \mid\ !a \\ & \mid \texttt{fail} \mid (a\ a) \mid e \sqcup e \\ & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid a \end{array}$$

$$a ::= v \mid x \qquad v ::= c \mid \ell \mid \lambda x, e$$

- non-terminal $a$ represents *atomic expressions*;
- non-terminal $v$ represents *values*;
- terminal $x$ represents *variables*;
- terminal $c$ represents basic constants of Definition A.2;
- terminal $\ell$ represents *locations.*

We note $\mathrm{FV}(e)$ the set of unbound variables in $e$.

In our formalization, we use several kinds of finite maps. The following definition provides generic notations for all these kinds of maps.

*Definition A.4 (Generic Finite Maps).* We provide the following notations for finite maps $m$ from keys $k \in K$ to data $d \in D$:

- $K \rightharpoonup D$ is the set of finite maps from $K$ to $D$
- $\mathrm{dom}(m) \subseteq K$ for the set of keys in $m$;
- $\mathrm{im}(m) \subseteq D$ for the set of data in $m$;
- $m[k]$ for the data associated to $k$ in $m$ (if $k \in \mathrm{dom}(m)$);
- $\emptyset$ for the empty map; $\{k \mapsto d\}$ for a singleton;
- $(m_1; m_2)$ for the update of $m_1$ by bindings of $m_2$;
- $m \backslash E$ for the update of $m$ by removing all keys belonging to set $E$;
- $m_1 \sqsubseteq m_2$ for the property $\mathrm{dom}(m_1) \subseteq \mathrm{dom}(m_2) \ \wedge\ \forall k \in \mathrm{dom}(m_1), m_1[k] = m_2[k]$

*Definition A.5 (Heaps).* A heap $h$ is a finite map from locations $\ell$ to values $v$.

*Definition A.6 (Stacks).* A stack $s$ is a finite map from variables $x$ to values $v$. Partial operator "$s[x]$" is extended into a *total* operator "$s[e]$" returning an expression where each unbound occurrence of a variable $x$ in $\mathrm{dom}(s)$ has been substituted in $e$ by $s[x]$. Indeed, as we will only put *closed* values into stacks, we do not need to use a fully capture-avoiding operator. For example, we define $s[\lambda x, e] \triangleq \lambda x, (s \backslash \{x\})[e]$. The other straightforward cases are left to the reader.

Definition A.7 specializes usual big-step semantics of ML to our expression language. Our semantics "$\langle e/h \rangle \Downarrow_n \langle v/h' \rangle$" is parameterized by a natural number $n$ expressing the number of dereference steps performed by the evaluation. This number $n$ will be used in Section A.3 to approximate heap invariants by an indirection model (Hobor et al. 2010). Usual big-step semantics is property $\exists n, \langle e/h \rangle \Downarrow_n \langle v/h' \rangle$.

*Definition A.7 (Big-Step Semantics).* Big-steps semantics is inductive property "$\langle e/h \rangle \Downarrow_n \langle v/h' \rangle$" expressing that expression $e$ for initial heap $h$ terminates normally on a heap $h'$ and returns a value $v$ in $n$ steps (where $n$ is a natural number).

$$\frac{\ell \notin \mathrm{dom}(h)}{\langle \mathtt{ref}\, v/h \rangle \Downarrow_0 \langle \ell/h; \{\ell \mapsto v\} \rangle} \qquad \frac{\ell \in \mathrm{dom}(h)}{\langle !\ell/h \rangle \Downarrow_1 \langle h[l]/h \rangle} \qquad \frac{\ell \in \mathrm{dom}(h)}{\langle \ell := v/h \rangle \Downarrow_0 \langle ()/h; \{\ell \mapsto v\} \rangle}$$

$$\frac{c_1 \in \mathrm{dom}(E_{c_2})}{\langle (c_2\ c_1)/h \rangle \Downarrow_0 \langle E_{c_2}(c_1)/h \rangle} \qquad \frac{\langle \{x \mapsto v\}[e]/h \rangle \Downarrow_n \langle v'/h' \rangle}{\langle (\lambda x, e\ v)/h \rangle \Downarrow_n \langle v'/h' \rangle}$$

$$\frac{\langle e_1/h \rangle \Downarrow_n \langle v/h' \rangle}{\langle e_1 \sqcup e_2/h \rangle \Downarrow_n \langle v/h' \rangle} \qquad \frac{\langle e_2/h \rangle \Downarrow_n \langle v/h' \rangle}{\langle e_1 \sqcup e_2/h \rangle \Downarrow_n \langle v/h' \rangle}$$

$$\frac{}{\langle v/h \rangle \Downarrow_0 \langle v/h \rangle} \qquad \frac{\langle e_1/h \rangle \Downarrow_{n_1} \langle v_1/h_1 \rangle \qquad \langle \{x \mapsto v_1\}[e_2]/h_1 \rangle \Downarrow_{n_2} \langle v/h' \rangle}{\langle \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2/h \rangle \Downarrow_{n_1+n_2} \langle v/h' \rangle}$$

There is no rule associated to "`fail`", because this operator raises an error.

*Example A.8 (Standard Macros).* In each macros below, variables $x$, $x_1$ and $x_2$ are variables which do not occur free in expressions at the left of the "$\triangleq$".

$$\mathtt{ref}\ e \triangleq \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{ref}\ x \qquad !e \triangleq \mathtt{let}\ x = e\ \mathtt{in}\ !x$$

$$(e_2\ e_1) \triangleq \begin{array}{l} (\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ \mathtt{let}\ x_2 = e_2\ \mathtt{in}\ (x_2\ x_1)) \\ \sqcup (\mathtt{let}\ x_2 = e_2\ \mathtt{in}\ \mathtt{let}\ x_1 = e_1\ \mathtt{in}\ (x_2\ x_1)) \end{array} \qquad e_1; e_2 \triangleq \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$$

$$\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \triangleq \begin{array}{l} \mathtt{let}\ x_1 = e\ \mathtt{in} \\ \mathtt{let}\ x_2 = \mathtt{not}\ x_1\ \mathtt{in} \\ (\mathtt{assert}\ x_1; e_1) \sqcup (\mathtt{assert}\ x_2; e_2) \end{array}$$

*Definition A.9 (Syntax of Types).* We note type variables as $\alpha$. A finite set of type variables is written $A$. The abstract syntax $\tau$ (resp. $\sigma$) of monomorphic (resp. polymorphic) types is given by:

$$\tau ::= \alpha \mid \beta \mid \tau \to \tau \mid \mathtt{ref}\ \tau \qquad \sigma ::= \Pi\, A, \tau$$

A type $b = \beta_1 \to \beta_2$ is syntactically identified as a $\tau$ type. We also abusively identify "$\tau$" and "$\Pi\, \emptyset.\tau$". Moreover, types $\sigma$ are implicitly considered modulo $\alpha$-renaming. At last, we define $\mathrm{FV}(\sigma)$ as the set of unbound type variables in $\sigma$.

*Definition A.10 (Typing Contexts).* A *typing context* is a finite map $\Gamma$ from variables $x$ to types $\sigma$. We define $A \vdash \Gamma$ as the property $\mathrm{FV}(\mathrm{im}(\Gamma)) \subseteq A$

*Definition A.11 (Type Substitution).* A typing substitution $\eta$ is a finite map from type variables $\alpha$ to monotypes $\tau$. We define the following notions:

- property $A \vdash \eta$ is $\mathrm{FV}(\mathrm{im}(\eta)) \subseteq A \wedge \mathrm{dom}(\eta) \cap A = 0$

- type $\eta[\sigma]$ is the *capture-avoiding* substitution in type $\sigma$ replacing each unbound occurrence of $\alpha$ of $\mathrm{dom}(\eta)$ by $\eta[\alpha]$

Definition A.12 specializes the usual ML typing rules to our expression language. Following standard approach (Wright 1995), we apply *value-form restriction* to automatic type generalization on "`let/in`". For our language, this restriction corresponds to generalize only the type of bound expressions that are syntactically *atomic*. Without restrictions on the bound expression, a function like `memofst` described in Section 2.3 would have a polymorphic type `'a -> 'a`. This would break type safety, since after a first call "`(memofst true)`", `memofst` will always return Boolean **`true`** whatever is its argument.

In ML semantics, only values in the stack can be polymorphic. The values in the heap are monomorphic: their type is fixed at allocation. It could be an open type however. The issue in `memofst` example comes from the fact that a bound expression allocates a reference on a open type, and generalizing over this type makes it "escape from its scope". The value-form restriction is a straightforward way to avoid type generalization when the bound expression allocates new references.

Definition A.12 defines judgment "$\langle A/\Gamma\rangle \vdash e : \sigma$" expressing that if $A \vdash \Gamma$, expression $e$ is a "source" expression compatible with type $\sigma$. In particular, $e$ can not contain any location $\ell$. Expression with locations are only generated during evaluation. And, in our simple formalization, we do not need to type them. Moreover, we deduce from the rules that when $e$ is a group expression, then $\sigma$ can actually be only some monomorphic type $\tau$ (since only rule Gen can build a non-monomorphic type).

*Definition A.12 (Typing).* Inductive property $\langle A/\Gamma\rangle \vdash e : \sigma$ is defined by the rules below.

$$\frac{}{\langle A/\Gamma\rangle \vdash c : \Delta(c)} \qquad \frac{\langle A/\Gamma; \{x \mapsto \tau_1\}\rangle \vdash e : \tau_2}{\langle A/\Gamma\rangle \vdash \lambda x, e : \tau_1 \to \tau_2} \qquad \frac{A \vdash \eta \qquad \{x \mapsto \Pi\, \mathrm{dom}(\eta), \tau\} \sqsubseteq \Gamma}{\langle A/\Gamma\rangle \vdash x : \eta[\tau]}$$

$$\frac{\mathrm{FV}(\tau) \subseteq A}{\langle A/\Gamma\rangle \vdash \mathtt{fail} : \tau} \qquad \frac{\langle A/\Gamma\rangle \vdash a : \tau}{\langle A/\Gamma\rangle \vdash \mathtt{ref}\, a : \mathtt{ref}\, \tau} \qquad \frac{\langle A/\Gamma\rangle \vdash a : \mathtt{ref}\, \tau}{\langle A/\Gamma\rangle \vdash\, !a : \tau}$$

$$\frac{\langle A/\Gamma\rangle \vdash a_1 : \mathtt{ref}\, \tau \qquad \langle A/\Gamma\rangle \vdash a_2 : \tau}{\langle A/\Gamma\rangle \vdash a_1 := a_2 : \mathtt{unit}} \qquad \frac{\langle A/\Gamma\rangle \vdash a_1 : \tau_1 \qquad \langle A/\Gamma\rangle \vdash a_2 : \tau_1 \to \tau_2}{\langle A/\Gamma\rangle \vdash (a_2\ a_1) : \tau_2}$$

$$\frac{\langle A/\Gamma\rangle \vdash e_1 : \tau \qquad \langle A/\Gamma\rangle \vdash e_2 : \tau}{\langle A/\Gamma\rangle \vdash e_1 \sqcup e_2 : \tau} \qquad \textsc{Gen}\ \frac{A \cap A' = \emptyset \qquad \langle A \cup A'/\Gamma\rangle \vdash a : \tau}{\langle A/\Gamma\rangle \vdash a : \Pi\, A'.\tau}$$

$$\frac{\langle A/\Gamma\rangle \vdash e_1 : \sigma \qquad \langle A/\Gamma; \{x \mapsto \sigma\}\rangle \vdash e_2 : \tau}{\langle A/\Gamma\rangle \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau}$$

*Definition A.13 (Typing of Closed Expressions).* Property "$A \vdash e : \sigma$" for polymorphic typing of closed source expressions is a short cut of $\langle A/\emptyset\rangle \vdash e : \sigma$.

Lemma A.14 (Auxiliary Results on Typing). *Under the assumptions $\langle A/\Gamma\rangle \vdash e : \sigma$ with $A \vdash \Gamma$, we have the following properties:*

**wellformed type** $\mathrm{FV}(\sigma) \subseteq A$
**wellformed expression** $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$
**weakening** $A \subseteq A'$ *and* $\Gamma \sqsubseteq \Gamma'$ *implies* $\langle A'/\Gamma'\rangle \vdash e : \sigma$

Proof. Each of result is proved – in the above order – by induction on $\langle A/\Gamma\rangle \vdash e : \sigma$. □

### A.3 An Step-Indexed Kripke Model of ML Types

Our main goal is to interpret ML types as invariants about ML values. We first introduce informally the notion that we have in mind.

*Definition A.15 (Informal Notion of Invariant).* An *invariant context* $I$ is a finite map from type variables to subsets of values. For type $\sigma$ such that $\mathrm{FV}(\sigma) \subseteq \mathrm{dom}(I)$, *invariant* $[\![\sigma]\!]_I$ is the set of (closed) values of type $\sigma$ that preserve $I$.

For system F, a value $v : \tau_1 \to \tau_2$ satisfies invariant $[\![\tau_1 \to \tau_2]\!]$ iff for all values $v_1$ of $[\![\tau_1]\!]$, any value computed from $(v\ v_1)$ satisfies $[\![\tau_2]\!]$. However, extending this approach to ML is not straightforward. For example, let us consider the ML value returned by "`(!fix)`" expression inside `fixpoint` example. This value depends on the heap location "`fix`" in which it is stored. Hence, the invariant on "`fix`" depends on the invariant "`(!fix)`' which itself depends on the invariant of "`fix`". More generally, an invariant associated to a value may depend on an invariant associated to a set of locations in the heap, which in turns may depend on invariant associated to their value in the heap, etc. Hence, we need to introduce a notion of "value invariant" and a notion of "heap invariant" which are mutually recursive. This does not seem possible to define this directly in a consistent way.

Inspired by stratified semantics (Ahmed et al. 2002), we refine our informal notion of invariant by using a stratified notion of heap invariant. We consider each invariant as an "observer" using evaluation in order to deconstruct values. We stratify our invariants according to a natural number – called here the "fuel" – which bounds the number of dereferences that they need to observe values. A value invariant of fuel 0 can not check any invariant on the heap. A heap invariant of fuel $n$ can only apply to locations which values satisfy a value invariant of fuel $n$. A value invariant of level $n$ can only check heap invariants of fuel (strictly) below $n$.

We aim to associate to each well-typed value an invariant of the appropriate fuel level. In other words, the invariant satisfied by a value is computed in same time than this value. The level of invariants is computed dynamically, thanks to the "$n$" parameter of $\Downarrow_n$ evaluation. Actually, we produce all invariants inside an evaluation run, starting from an empty heap invariant – a heap invariant that does not constrain any location of the heap. Such an empty heap invariant can be defined at any fuel level and is satisfied by any (initial) heap. Hence, once an initial fuel $n$ has been chosen, we can produce only invariants of levels lower or equals to $n$. We know from $\Downarrow$ definition that such a $n$ exists for any value produced by $\Downarrow$ and any finite observation about this value.

In the next, we formalize these ideas by adapting the step-indexed model of polymorphic higher-order imperative functions developed in (Hobor et al. 2010). Our step-indexed model is a Kripke model, where "world" represents "heap predicate". Such a heap predicate is a finite map associating heap locations to a predicate satisfied by the value at this location. Theorem A.32 expresses that heap updates preserve the world progressively built by successive allocations.

As explained above, in order to define this notion of world in a consistent way, we need to stratify it according to a notion a fuel level. Hence Definition A.16 introduces $W_n$ as the set of "heap predicates" with fuel (strictly) below "$n$". Set $P_n$ is the set of value predicates depending on heap predicates in $W_n$.

*Definition A.16 (Worlds = Heap Predicates).* Given $L$ the set of locations $\ell$ and $V$ the set of values $v$, we define sets $W_n$ and $P_n$ by mutual induction over natural number $n$

$$W_0 \triangleq \emptyset \qquad W_{n+1} \triangleq W_n \ \cup\ \{n\} \times (L \rightharpoonup P_n) \qquad P_n \triangleq V \to \mathcal{P}(W_n)$$

From the definitions above, we now define $W$ the set of heap predicates and $P$ the set of value predicates

$$W \triangleq \bigcup_{n \in \mathbb{N}} W_n \qquad\qquad P \triangleq V \to \mathcal{P}(W)$$

A world $w \in W$ is thus of the form $(n, \Psi)$: we note $w.n$ (resp. $w.\Psi$) this first (resp. second) projection of $w$. Such a $w.n$ is an approximation level, while $w.\Psi$ corresponds to a predicate for the heap at $w.n$ level.

Here, let us remark that we have:

- $w \in W_{w.n+1}$ and $w.\Psi \in L \rightharpoonup P_{w.n}$.
- forall $n$, $W_n \subseteq W_{n+1} \subseteq W$ and $P_n \subseteq P_{n+1} \subseteq P$.

While evaluation consumes fuels, we need to approximate already-computed invariants to a lower level of fuel. Definition A.17 formalizes this notion of approximation.

*Definition A.17 (World Approximation).* For a predicate $p \in P$, we define $\lfloor p \rfloor_n$ of $P_n$ satisfying

$$\lfloor p \rfloor_n (v) \triangleq \{w \in p(v) \mid w.n < n\}$$

We extends this operator for $\Psi \in L \rightharpoonup P$ into $\lfloor \Psi \rfloor_n$ of $L \rightharpoonup P_n$ by

$$\lfloor \Psi \rfloor_n \triangleq \{\ell \mapsto \lfloor \Psi[\ell] \rfloor_n\}_{\ell \in \mathrm{dom}(\Psi)}$$

At last, given $w \in W$, we define $\lfloor w \rfloor_n$ of $W_{n+1}$ by

$$\lfloor w \rfloor_n \triangleq (n, \lfloor w.\Psi \rfloor_n)$$

From these definitions, we deduce immediately the following properties:

- If $n \geq n'$ and $p \in P_{n'}$, $\lfloor p \rfloor_n = p$. Moreover if $n_2 \leq n_1$, then for all $v$, $\left\lfloor \lfloor p \rfloor_{n_1} \right\rfloor_{n_2} (v) = \lfloor p \rfloor_{n_2} (v)$.
- If $n > n'$ and $w \in W_{n'}$ and $p \in P$, $\;w \in p(v) \Leftrightarrow w \in \lfloor p \rfloor_n (v)$
- If $w \in W$, then $\lfloor w \rfloor_{w.n} = w$. Moreover if $n_2 \leq n_1 \leq n$, then $\left\lfloor \lfloor w \rfloor_{n_1} \right\rfloor_{n_2} = \lfloor w \rfloor_{n_2}$.
  For $n \leq w.n$, computing $\lfloor w \rfloor_n$ is called "*aging* $w$", following terminology of (Hobor et al. 2010).
- If $\Psi, \Psi' \in L \rightharpoonup P_n$ such that $\Psi \sqsubseteq \Psi'$, then $\lfloor \Psi \rfloor_{n'} \sqsubseteq \lfloor \Psi' \rfloor_{n'}$.
  This property is called *stability of* $\sqsubseteq$ *by approximations.*

*Definition A.18 (World Satisfiability).* Property $w \Vdash h$ expresses the condition at which a heap $h$ satisfies a world $w$. It is defined as

$$\mathrm{dom}(h) = \mathrm{dom}(w.\Psi) \;\wedge\; \forall \ell \in \mathrm{dom}(h), \forall n \in \mathbb{N}, n < w.n \Rightarrow \lfloor w \rfloor_n \in w.\Psi[\ell](h[\ell])$$

This property is closed by aging: if $n \leq w.n$ and $w \Vdash h$ then $\lfloor w \rfloor_n \Vdash h$. This ensures that satisfiability of heap predicates is preserved while evaluation consumes fuel (by dereferencing heap locations).

Informally, a heap predicate $w_2$ extends a heap predicate $w_1$ if the evaluation of expression may lead from heaps satisfying $w_1$ to heaps satisfying $w_2$. Hence, the domain of the heap predicate may increase (new locations are allocated) while the fuel may decrease (less dereferences are authorized).

*Definition A.19 (World Extension).* The order $w \sqsubseteq w'$ is defined as

$$w'.n \leq w.n \;\wedge\; \lfloor w.\Psi \rfloor_{w'.n} \sqsubseteq w'.\Psi$$

Proof. Verifying that $\sqsubseteq$ is an order on worlds is straightforward. In particular, for transitivity, from $w_1 \sqsubseteq w_2$ and $w_2 \sqsubseteq w_3$, we get by transitivity of $\leq$ on $\mathbb{N}$, by transitivity of $\sqsubseteq$ on finite maps, and by stability of approximations that

$$w_3.n \leq w_1.n \ \wedge \ \left\lfloor \lfloor w_1.\Psi \rfloor_{w_2.n} \right\rfloor_{w_3.n} \sqsubseteq w_3.\Psi$$

We conclude by noticing $\left\lfloor \lfloor w_1.\Psi \rfloor_{w_2.n} \right\rfloor_{w_3.n} = \lfloor w_1.\Psi \rfloor_{w_3.n}$. □

This partially ordered Kripke model leads to an intuitionistic logic for the standard Kripke intuitionistic propositions.

*Definition A.20 (Propositions).* By definition a "proposition over heaps" is a subset of $W$ closed for world extension:

$$\text{Prop} \triangleq \{ p \in \mathcal{P}(W) \mid \forall\, w_1\, w_2, w_1 \sqsubseteq w_2 \wedge w_1 \in p \Rightarrow w_2 \in p \}$$

Let us remark that for $n \leq w.n$, we have $w \sqsubseteq \lfloor w \rfloor_n$. Hence, all propositions are also closed by aging.

*Definition A.21 (Invariant).* An invariant $\iota$ is a function of $V \to \text{Prop}$.

However, our model also involves predicates over worlds which are not closed for world extension. This is in particular the case of the predicate "$w \Vdash e \Downarrow \iota$" of Definition A.22 – meaning that *"any value computed by e satisfies ι in an extension of w"*. Typically, if $w = (0, \emptyset)$ then $w \Vdash !\ell \Downarrow \iota$ holds for any invariant $\iota$. This may not be the case if $w.n \geq 1$ and $\ell \in \text{dom}(w.\Psi)$. This predicate is not even closed for aging.

*Definition A.22 (Evaluation Invariant).* For a given invariant $\iota$, the property $w \Vdash e \Downarrow \iota$ is defined as

$$\begin{array}{l} \forall\, n\, h\, v\, h',\ \langle e/h \rangle \Downarrow_n \langle v/h' \rangle \wedge w \Vdash h \wedge 0 \leq n \leq w.n \\ \Rightarrow \exists w'.w \sqsubseteq w' \wedge w'.n = w.n - n \wedge w' \Vdash h' \wedge w' \in \iota(v) \end{array}$$

Lemma A.23.

$$w \Vdash v \Downarrow \iota \ \Leftrightarrow \ w \in \iota(v)$$

Proof. Assuming $e = v$ in Definition A.22, we deduce by inversion of $\Downarrow_n$ that $h = h'$ and $n = 0$. Hence, $w'.n = w.n$ and $\text{dom}(w'.\Psi) = \text{dom}(h) = \text{dom}(w.\Psi)$. Thus $w' = w$ and $w \in \iota(v)$.
The inverse implication is straightforward. □

Now, we define our semantics for types as invariants. We start from type variables: this leads to the notion of invariant context.

*Definition A.24 (Invariant Context).* An *invariant context I* is a finite map from type variables $\alpha$ to invariants $\iota$.

We now lift this interpretation from type variables to an arbitrary type $\sigma$ as an invariant $[\![\sigma]\!]_I$.

*Definition A.25 (Types as Invariants).* Given $\sigma$ such that $\text{FV}(\sigma) \subseteq \text{dom}(I)$, the proposition $[\![\sigma]\!]_I\,(v)$ is defined recursively over $\sigma$ syntax and by case analysis over $v$ (missing cases correspond implicitly

to proposition $\emptyset$).

$$[\![\alpha]\!]_I \triangleq I[\alpha] \qquad\qquad [\![\beta]\!]_I(c) \triangleq \{ w \in W \mid \Delta(c) = \beta \}$$

$$[\![\texttt{ref}\,\tau]\!]_I(\ell) \triangleq \left\{ w \mid \{\ell \mapsto \lfloor[\![\tau]\!]_I\rfloor_{w.n}\} \sqsubseteq w.\Psi \right\}$$

$$[\![\tau_1 \to \tau_2]\!]_I(v) \triangleq \{ w \mid \forall\, w'\, v_1, (w \sqsubseteq w' \wedge w' \in [\![\tau_1]\!]_I(v_1)) \Rightarrow w' \Vdash (v\; v_1) \Downarrow [\![\tau_2]\!]_I \}$$

$$[\![\Pi\, A, \tau]\!]_I(v) \triangleq \bigcap_{\substack{I' \mid A=\mathrm{dom}(I') \\ \wedge\ \mathrm{dom}(I)\cap A=\emptyset}} [\![\tau]\!]_{I;I'}(v)$$

Proof. We check $[\![\sigma]\!]_I : V \to \mathrm{Prop}$ by induction on $\sigma$:

- for $\sigma \equiv \alpha$, by hypothesis on $I$, we have $I[\alpha] \in V \to \mathrm{Prop}$.
- for $\sigma \equiv \beta$, trivial.
- for $\sigma \equiv \texttt{ref}\,\tau$, we have the following hypotheses

  $$w \sqsubseteq w' \qquad\qquad \{\ell \mapsto \lfloor[\![\tau]\!]_I\rfloor_{w.n}\} \sqsubseteq w.\Psi$$

  By transitivity of $\sqsubseteq$ on finite maps and stability of approximations, we get

  $$\left\lfloor\{\ell \mapsto \lfloor[\![\tau]\!]_I\rfloor_{w.n}\}\right\rfloor_{w'.n} \sqsubseteq w'.\Psi$$

  which reduces to

  $$\{\ell \mapsto \lfloor[\![\tau]\!]_I\rfloor_{w'.n}\} \sqsubseteq w'.\Psi$$

- for $\sigma \equiv \tau_1 \to \tau_2$, we have the following hypotheses

  $$w_1 \sqsubseteq w_2 \qquad w_1 \in [\![\tau_1 \to \tau_2]\!]_I(v) \qquad w_2 \sqsubseteq w' \qquad w' \in [\![\tau]\!]_I(v_1)$$

  By transitivity of $\sqsubseteq$, we have $w_1 \sqsubseteq w'$ and thus

  $$w' \Vdash (v\; v_1) \Downarrow [\![\tau_2]\!]_I$$

  which proves that $w_2 \in [\![\tau_1 \to \tau_2]\!]_I(v)$.
- for $\sigma \equiv \Pi\, A, \tau$, we conclude from the fact that – by induction hypothesis – $[\![\tau]\!]_{I;I'}(v) \in \mathrm{Prop}$.

□

At last, we lift our interpretation of types into an interpretation of typing contexts.

*Definition A.26 (Typing Contexts as Stack Invariants).* For an invariant context $I$, a context $\Gamma$ such that $\mathrm{dom}(I) \vdash \Gamma$, and a stack $s$, we define $[\![\Gamma]\!]_I(s) \in \mathrm{Prop}$ as

$$[\![\Gamma]\!]_I(s) \triangleq \{ w \mid \mathrm{dom}(s) = \mathrm{dom}(\Gamma)\ \wedge\ \forall\, x \in \mathrm{dom}(s), w \in [\![\Gamma[x]]\!]_I(s[x]) \}$$

Proof. Property $[\![\Gamma]\!]_I(s) \in \mathrm{Prop}$ is directly deduced from the fact that for all $x \in \mathrm{dom}(s)$, we have $[\![\Gamma[x]]\!]_I(s[x]) \in \mathrm{Prop}$. □

Now, we introduce auxiliary lemmas in order to prove our theorem of "invariant preservation" below.

Lemma A.27 (Extension of Invariant Context). *Given two invariant contexts $I$ and $I'$ such that* $\mathrm{dom}(I) \cap \mathrm{dom}(I') = \emptyset$. *We have the following properties:*

- *if* $\mathrm{FV}(\sigma) \subseteq \mathrm{dom}(I)$ *then* $[\![\sigma]\!]_I = [\![\sigma]\!]_{I;I'}$
- *if* $\mathrm{dom}(I) \vdash \Gamma$ *then* $[\![\Gamma]\!]_I = [\![\Gamma]\!]_{I;I'}$

Proof. Immediate by induction on $\sigma$ syntax (and definition of $[\![\Gamma]\!]_I$ from $[\![\Gamma[x]]\!]_I$). □

*Definition A.28 (Type Substitution in Invariant Context).* Let us assume $I$ and $\eta$ such that $\mathrm{dom}(I) \vdash \eta$. Invariant $\eta[I] \in \mathrm{dom}(\eta) \rightharpoonup (V \to \mathrm{Prop})$ is defined by

$$\eta[I] \triangleq \{\alpha \mapsto [\![\eta[\alpha]]\!]_I\}_{\alpha \in \mathrm{dom}(\eta)}$$

We have the following property:

$$\mathrm{FV}(\tau) \subseteq \mathrm{dom}(I) \cup \mathrm{dom}(\eta) \text{ implies } [\![\tau]\!]_{I;\eta[I]} = [\![\eta[\tau]]\!]_I$$

Proof. Immediate by induction on $\tau$ syntax. □

Lemma A.29 (Type Substitution in Invariants). *Under assumptions*

$$\sigma = \Pi\,\mathrm{dom}(\eta), \tau \qquad\qquad \mathrm{dom}(I) \vdash \eta$$

*we have*

$$[\![\sigma]\!]_I\,(v) \subseteq [\![\eta[\tau]]\!]_I\,(v)$$

Proof. In $[\![\sigma]\!]_I\,(v)$ definition (see Definition A.25), we instantiate $I'$ by $\eta[I]$, defined at Definition A.28. □

Lemma A.30 (Substitutions of Atomic Expressions). *Let us assume a stack $s$ and an atomic expressions $a$ such that* $\mathrm{FV}(a) \subseteq \mathrm{dom}(s)$. *Then $s[a]$ is a value.*

Proof. From $a$ syntax, either $a$ is already a value $v$: then $s[v]$ is still a value. Otherwise $a$ is variable $x$ such that $x \in \mathrm{dom}(s)$. Thus $s[x]$ is value. □

Lemma A.31 (Simple Small Steps). *Given $f$ a finite map from $\mathbb{N}$ to expressions, the property $e \Downarrow f$ is defined as*

$$\begin{array}{l}\forall\, n\, h\, v\, h', \langle e/h \rangle \Downarrow_n \langle v/h' \rangle \\ \Rightarrow \exists i \in \mathrm{dom}(f), \langle f[i]/h \rangle \Downarrow_n \langle v/h' \rangle\end{array}$$

*If $e \Downarrow f$ and $\forall i \in \mathrm{dom}(f), w \Vdash f[i] \Downarrow \iota$, then $w \Vdash e \Downarrow \iota$.*

Proof. Let us assume $e \Downarrow f$ and $\langle e/h \rangle \Downarrow_n \langle v/h' \rangle \wedge w \Vdash h \wedge 0 \leq n \leq w.n$.
Given $i \in \mathrm{dom}(f)$ such that $\langle f[i]/h \rangle \Downarrow_n \langle v/h' \rangle$.
Then $w \Vdash f[i] \Downarrow \iota$ gives a $w'$ that we can use directly for proving conclusion of $w \Vdash e \Downarrow \iota$. □

Theorem A.32 (Invariant Preservation). *Assuming the following hypotheses*

$$\langle A/\Gamma \rangle \vdash e : \sigma \qquad A = \mathrm{dom}(I) \qquad A \vdash \Gamma \qquad w \in [\![\Gamma]\!]_I\,(s)$$

*Then,*

$$w \Vdash s[e] \Downarrow [\![\sigma]\!]_I$$

Proof. By induction on $\langle A/\Gamma \rangle \vdash e : \sigma$.

- $\langle A/\Gamma \rangle \vdash c : \Delta(c)$ where $\Delta(c) = \beta$.
  For any $w \in W$, we have $w \in [\![\beta]\!]_I\,(c)$. Since $s[c] = c$, by lemma A.23, we have $w \Vdash s[c] \Downarrow [\![\beta]\!]_I$.
- $\langle A/\Gamma \rangle \vdash c : \Delta(c)$ where $\Delta(c) = \beta_1 \to \beta_2$.
  Assuming $w'$ and $v$ such that

  $$w \sqsubseteq w' \qquad\qquad w' \in [\![\beta_1]\!]_I\,(v)$$

  Assuming $\langle (c\ v)/h \rangle \Downarrow_n \langle v'/h' \rangle$ and $w' \Vdash h$, we get

  $$n = 0 \qquad h = h' \qquad v \in \mathrm{dom}(E_c) \qquad v' = E_c(v) \in \beta_2$$

By Definition A.2, we have $\Delta(v') = \beta_2$.
Hence, we have $w' \in [\![\beta_2]\!]_I\,(v')$.
Since $s[c] = c$, we conclude that $w \in [\![\beta_1 \to \beta_2]\!]_I\,(s[c])$.

- $\langle A/\Gamma\rangle \vdash \lambda x, e : \tau_1 \to \tau_2$. Assuming
$$w \in [\![\Gamma]\!]_I\,(s) \qquad w \sqsubseteq w' \qquad w' \in [\![\tau_1]\!]_I\,(v_1)$$
we define $s' \triangleq s; \{x \mapsto v_1\}$ or equivalently, $s' \triangleq (s\backslash \mathrm{dom}(x)); \{x \mapsto v_1\}$.
Because $[\![\Gamma]\!]_I\,(s) \in \mathrm{Prop}$, we have $w' \in [\![\Gamma]\!]_I\,(s)$. And thus, $w' \in [\![\Gamma; \{x \mapsto \tau_1\}]\!]_I\,(s')$.
By induction, we have $w' \Vdash s'[e] \Downarrow [\![\tau_2]\!]_I$.
Because $(s[\lambda x, e]\ v_1) \Downarrow \{1 \mapsto s'[e]\}$, we deduce from Lemma A.31 that $w' \Vdash (s[\lambda x, e]\ v_1) \Downarrow [\![\tau_2]\!]_I$.
Hence $w' \in [\![\tau_1 \to \tau_2]\!]_I\,(s[\lambda x, e])$.
- $\langle A/\Gamma\rangle \vdash x : \eta[\tau]$.
From hypothesis $w \in [\![\Gamma]\!]_I\,(s)$, we deduce
$$x \in \mathrm{dom}(s) \qquad w \in [\![\Pi\,\mathrm{dom}(\eta), \tau]\!]_I\,(s[x])$$
By Lemma A.29, we have $w \in [\![\eta[\tau]]\!]_I\,(s[x])$
- $\langle A/\Gamma\rangle \vdash \texttt{fail} : \tau$.
We have $\texttt{fail} \Downarrow \emptyset$. Thus, from Lemma A.31, $w \Vdash \texttt{fail} \Downarrow [\![\tau]\!]$.
- $\langle A/\Gamma\rangle \vdash \texttt{ref}\,a : \texttt{ref}\,\tau$.
Since $\langle A/\Gamma\rangle \vdash a : \tau$, we have $\mathrm{FV}(a) \subseteq \mathrm{dom}(\Gamma) = \mathrm{dom}(s)$. Thus, by Lemma A.30, $s[a]$ is a value.
By induction on hypothesis $\langle A/\Gamma\rangle \vdash a : \tau$ and by Lemma A.23, value $s[a]$ satisfies $w \in [\![\tau]\!]_I\,(s[a])$.
Assuming $\langle \texttt{ref}\,s[a]/h\rangle \Downarrow_n \langle v/h'\rangle$, we deduce that $v$ is a location $\ell$ such that
$$\ell \notin \mathrm{dom}(h) \qquad n = 0 \qquad h' = h; \{\ell \mapsto s[a]\}$$
We define
$$w' \triangleq (w.n, w.\Psi; \{\ell \mapsto (\tau, \lfloor [\![\tau]\!]_I\,(s[a]))\rfloor_{w.n}\})$$
Assuming $w \Vdash h$, we have
$$w' \Vdash h' \qquad w' \in [\![\texttt{ref}\,\tau]\!]_I\,(\ell)$$
Since $w \sqsubseteq w'$ and $w'.n = w.n$, we conclude that
$$w \Vdash s[\texttt{ref}\,a] \Downarrow [\![\texttt{ref}\,\tau]\!]_I$$
- $\langle A/\Gamma\rangle \vdash !a : \tau$.
By induction on hypothesis $\langle A/\Gamma\rangle \vdash a : \texttt{ref}\,\tau$, value $s[a]$ is a location $\ell$ satisfying $w \in [\![\texttt{ref}\,\tau]\!]_I\,(\ell)$.
Assuming $\langle !\ell/h\rangle \Downarrow_n \langle v/h'\rangle$, we deduce that
$$n = 1 \qquad \ell \in \mathrm{dom}(h) \qquad v = h[\ell] \qquad h' = h$$
Assuming $w.n \geq 1$ and $w \Vdash h$, we have
$$\lfloor w\rfloor_{w.n-1} \in \lfloor [\![\tau]\!]_I\rfloor_n\,(h[\ell])$$
Hence, we define $w' \triangleq \lfloor w\rfloor_{w.n-1}$ and we have
$$w \sqsubseteq w' \qquad w'.n = w.n - 1 \qquad w' \Vdash h \qquad w' \in [\![\tau]\!]_I\,(h[\ell])$$
We conclude that
$$w \Vdash s[!a] \Downarrow [\![\tau]\!]_I$$

- $\langle A/\Gamma\rangle \vdash a_1 := a_2 : \texttt{unit}$. By induction on hypotheses
$$\langle A/\Gamma\rangle \vdash a_1 : \texttt{ref}\,\tau \qquad\qquad \langle A/\Gamma\rangle \vdash a_2 : \tau$$
values $s[a_1]$ and $s[a_2]$ satisfy
$$w \in [\![\texttt{ref}\,\tau]\!]_I\,(s[a_1]) \qquad\qquad w \in [\![\tau]\!]_I\,(s[a_2])$$
In particular, $s[a_1]$ is some location $\ell$.
Assuming $\langle \ell := s[a_2]/h\rangle \Downarrow_n \langle v/h'\rangle$, we deduce that
$$n = 0 \qquad\qquad v = () \qquad\qquad h' = h; \{\ell \mapsto s[a_2]\}$$
Assuming $w \Vdash h$, we have $w \Vdash h'$.
Since, $w \in [\![\texttt{unit}]\!]_I\,(v)$, we conclude that
$$w \Vdash s[a_1 := a_2] \Downarrow [\![\texttt{unit}]\!]_I$$
- $\langle A/\Gamma\rangle \vdash (a_2\ a_1) : \tau_2$. By induction on hypotheses
$$\langle A/\Gamma\rangle \vdash a_2 : \tau_1 \to \tau_2 \qquad\qquad \langle A/\Gamma\rangle \vdash a_1 : \tau_1$$
values $s[a_2]$ and $s[a_1]$ satisfy
$$w \in [\![\tau_1 \to \tau_2]\!]_I\,(s[a_2]) \qquad\qquad w \in [\![\tau_1]\!]_I\,(s[a_1])$$
From the definition of $[\![\tau_1 \to \tau_2]\!]_I$, we have
$$w \Vdash (s[a_2]\ s[a_1]) \Downarrow [\![\tau_2]\!]_I$$
- $\langle A/\Gamma\rangle \vdash e_1 \sqcup e_2 : \tau$. By induction hypotheses, we have
$$w \Vdash s[e_1] \Downarrow [\![\tau]\!]_I \qquad\qquad w \Vdash s[e_2] \Downarrow [\![\tau]\!]_I$$
Since $(e_1 \sqcup e_2) \Downarrow \{1 \mapsto e_1\}; \{2 \mapsto e_2\}$, we deduce from Lemma A.31 that $w \Vdash (e_1 \sqcup e_2) \Downarrow [\![\tau]\!]_I$
- $\langle A/\Gamma\rangle \vdash a : \Pi\,\mathrm{dom}(A'), \tau$.
Let us consider any invariant context $I'$ such that $\mathrm{dom}(I') = A'$. Because $A \vdash \Gamma$ and $A \cap A' = \emptyset$, we have $[\![\Gamma]\!]_{I;I'}\,(s) = [\![\Gamma]\!]_I\,(s)$ (Lemma A.27).
We apply induction hypothesis on $\langle A \cup A'/\Gamma\rangle \vdash a : \tau$ and $w \in [\![\Gamma]\!]_{I;I'}\,(s)$. Because $s[a]$ is a value (Lemma A.30), we have $w \in [\![\tau]\!]_{I;I'}\,(s[a])$ (Lemma A.23). Hence, we conclude
$$w \in [\![\sigma]\!]_I\,(s[a])$$
- $\langle A/\Gamma\rangle \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau$. Let us assume
$$\langle s[\texttt{let}\ x = e_1\ \texttt{in}\ e_2]/h\rangle \Downarrow_n \langle v/h'\rangle \qquad w.n \geq n \qquad w \Vdash h \qquad w \in [\![\Gamma]\!]_I\,(s)$$
We have
$$n = n_1 + n_2 \qquad \langle s[e_1]/h\rangle \Downarrow_{n_1} \langle v_1/h_1\rangle \qquad \langle (s; \{x \mapsto v_1\})[e_2]/h_1\rangle \Downarrow_{n_2} \langle v/h'\rangle$$
By induction hypothesis on $\langle A/\Gamma\rangle \vdash e_1 : \sigma$, we get $w_1$ such that
$$w \sqsubseteq w_1 \qquad w_1.n = w.n - n_1 \geq n_2 \qquad w_1 \Vdash h_1 \qquad w_1 \in [\![\sigma]\!]_I\,(v_1)$$
Since $w_1 \in [\![\Gamma]\!]_I\,(s)$, we have
$$w_1 \in [\![\Gamma; \{x \mapsto \sigma\}]\!]_I\,(s; \{x \mapsto v_1\})$$
By induction hypothesis on $\langle A/\Gamma; \{x \mapsto \sigma\}\rangle \vdash e_2 : \tau$, we get $w'$ such that
$$w_1 \sqsubseteq w' \qquad w'.n = w_1.n - n_2 = w.n - n \qquad w' \Vdash h' \qquad w' \in [\![\tau]\!]_I\,(v)$$
By transitivity, we have $w \sqsubseteq w'$. Thus, we conclude
$$w \Vdash s[\texttt{let}\ x = e_1\ \texttt{in}\ e_2] \Downarrow [\![\tau]\!]_I$$

□

Let us consider the importance of restricting Gen-rule on this proof. Indeed, let us imagine a too weak rule like

$$\frac{A \cap A' = \emptyset \qquad \langle A \cup A'/\Gamma\rangle \vdash e : \tau}{\langle A/\Gamma\rangle \vdash e : \Pi A'.\tau}$$

Replaying the above proof for Gen-rule (by replacing $a$ with $e$), we get by induction that

$$w \Vdash s[e] \Downarrow [\![\tau]\!]_{I;I'}$$

Here, assuming

$$\langle s[e]/h\rangle \Downarrow_n \langle v/h'\rangle \qquad w.n \geq n \qquad w \Vdash h$$

From $w \Vdash s[e] \Downarrow [\![\tau]\!]_{I;I'}$ we get $w'$ such

$$w \sqsubseteq w' \qquad w'.n = w.n - n \qquad w' \Vdash h' \qquad w' \in [\![\tau]\!]_{I;I'}(v)$$

Now, we are stuck and can not conclude $w' \in [\![\sigma]\!]_I(v)$ because $w'$ depends on $I'$. Restricting $e$ to atomic expressions is a simple way to ensure that $w' = w$ (for any $I'$). However, we may imagine to relax Gen-rule by authorizing expressions that do not allocate new locations. In this case, we should be able to prove $w' = \lfloor w \rfloor_{w.n-n}$ (for any $I'$) and conclude $w' \in [\![\sigma]\!]_I(v)$. This may help to still prove the theorem when value restriction is relaxed thanks to subtyping as suggested by Garrigue (Garrigue 2002).

### A.4 Toolkit for PFS Correctness Proofs

In the following, we distinguish between *well-typed source expressions* and *intermediate expressions* generated during evaluation of the former ones. Indeed, the latter may contain locations, whereas the former can not. In order to reason on intermediate expressions, we thus need to extend our notion of typing. Below, we use a semantic definition of this intermediate typing. Roughly, an intermediate expression is a closed expression obtained by substitutions from a well-typed (but possibly open) expression. The intermediate typing depends on a world ensuring that the substitution satisfies the expected invariant.

*Definition A.33 (Intermediate Typing of Expressions).* Given an invariant context $I$ and an expression $e$, we define $[\![e : \sigma]\!]_I \in \text{Prop}$ by

$$[\![e : \sigma]\!]_I \triangleq \{\, w \in W \mid \exists\, e_0\, \Gamma\, s, e = s[e_0] \;\wedge\; \text{dom}(I) \vdash \Gamma \;\wedge\; \langle \text{dom}(I)/\Gamma\rangle \vdash e_0 : \sigma \;\wedge\; w \in [\![\Gamma]\!]_I(s) \,\}$$

Proof. $[\![e : \sigma]\!]_I \in \text{Prop}$ because $[\![\Gamma]\!]_I(s) \in \text{Prop}$. □

Lemma A.34 (Basic Properties of Intermediate Typing). *Typing of intermediate expressions satisfies the following properties:*

(1) $\text{dom}(I) \vdash e : \sigma$ *implies* $[\![e : \sigma]\!]_I = W$
(2) $[\![v : \sigma]\!]_I \subseteq [\![\sigma]\!]_I(v)$

Proof.

(1) We take $e_0 \triangleq e$ and $\Gamma \triangleq \emptyset$ and $s \triangleq \emptyset$. By definition, $[\![\Gamma]\!]_I(s) = W$.
(2) Direct consequence of Theorem A.32 and Lemma A.23.

□

As in our Coq examples, we use a may-return relation. This relation abstracts big-steps semantics, by hiding the final heap $h'$ and abstracting the initial heap $h$ as a world $w$.

*Definition A.35 (May-Return Relation).* Property $w \Vdash e \rightsquigarrow v$ is defined as

$$\exists\, n\, h\, h', \langle e/h \rangle \Downarrow_n \langle v/h' \rangle \;\wedge\; w \Vdash h \;\wedge\; 0 \leq n \leq w.n$$

LEMMA A.36 (MAY-RETURN ABSTRACTION).

$$\exists w, w \Vdash e \rightsquigarrow v \;\Leftrightarrow\; \exists\, n\, h\, h', \langle e/h \rangle \Downarrow_n \langle v/h' \rangle$$

PROOF. $\Rightarrow$-way is straightforward from the definition of may-return relation.
Let us prove the $\Leftarrow$-way. Let us assume $\langle e/h \rangle \Downarrow_n \langle v/h' \rangle$. We define trivial predicate $\top_n \in V \to P_n$ by

$$\top_n(v) \triangleq W_n$$

Then, we define $w_0$ a world of fuel $n$ associating a trivially true predicate to each location of $h$:

$$w_0 \triangleq (n, \{\ell \mapsto \top_n\}_{\ell \in \mathrm{dom}(h)})$$

By definition, for $\ell \in \mathrm{dom}(h)$, for any $w \in W_n$ and for all $v$, we have $w \in w_0.\Psi[\ell](v)$.
Hence, $w_0 \Vdash h$ and $w_0.n = n$. Thus, $w_0 \Vdash e \rightsquigarrow v$. □

Parametric Invariance has two statements given below. The weaker statement only applies to "source" expressions, whereas the stronger one also applies to "intermediate" expressions. Of course, the stronger one implies the weaker one.

THEOREM A.37 (PARAMETRIC INVARIANCE). *The two following statements hold.*

**weaker statement** *Under assumptions*

$$\mathrm{dom}(I) \vdash e : \sigma \qquad\qquad \langle e/h \rangle \Downarrow_n \langle v/h' \rangle$$

*we have*

$$[\![\sigma]\!]_I\,(v) \neq \emptyset$$

**stronger statement** *Under assumptions*

$$w \in [\![e : \sigma]\!]_I \qquad\qquad w \Vdash e \rightsquigarrow v$$

*there exists w′ such that*

$$w \sqsubseteq w' \qquad\qquad w' \in [\![\sigma]\!]_I\,(v)$$

PROOF. The stronger statement is a direct consequence of the definitions and of Theorem A.32.
Using Lemma A.34 and A.36, we deduce the weaker statement from the stronger one. □

LEMMA A.38 (INVARIANT OF FUNCTIONS). *Under assumptions*

$$w \in [\![\tau_1 \to \tau_2]\!]_I\,(v) \qquad\qquad w \in [\![\tau]\!]_I\,(v_1) \qquad\qquad w \Vdash (v\; v_1) \rightsquigarrow v_2$$

*There exists w′ such that*

$$w \sqsubseteq w' \qquad\qquad w' \in [\![\tau_2]\!]_I\,(v_2)$$

PROOF. This a simple consequence of Definition of $[\![\tau_1 \to \tau_2]\!]_I$ (in Definition A.25) and A.35. □

*Example A.39 (Reasoning with Parametric Invariants).* Let us assume

$$w \in [\![\Pi\{\alpha\}, \alpha \to \alpha]\!]_I\,(v) \qquad\qquad w \Vdash (v\; v_1) \rightsquigarrow v_2$$

We can prove $v_2 = v_1$.

PROOF. Let us choose $\alpha \notin \mathrm{dom}(I)$ and let us define

$$I'[\alpha](v_0) \triangleq \{\, w \in W \mid v_0 = v_1 \,\}$$

We get $w \in [\![\alpha \to \alpha]\!]_{I;I'}\,(v)$.
Since $[\![\alpha]\!]_{I;I'}\,(v_1) = W$, we deduce from Lemma A.38 that $[\![\alpha]\!]_{I;I'}\,(v_2) \neq \emptyset$. Hence, $v_2 = v_1$. □

We have given above a simple example of "parametricity reasoning". However, in PFS, we also need to reason on frontend computations involving oracles and not just have a parametricity reasoning on oracles like above. The lemma below (combined with the preceding ones) allows conducting the proof of `front_proj_correctness`, sketched in Section 3.2.

Lemma A.40 (May-Return Inversion).

(1) $w \Vdash v_1 \rightsquigarrow v_2$ *implies* $v_1 = v_2$

(2) *Under assumptions*

$$w \in [\![(\lambda x, e\ v) : \tau]\!]_I \qquad\qquad w \Vdash (\lambda x, e\ v) \rightsquigarrow v'$$

*We have*

$$w \in [\![\{x \mapsto v\}[e] : \tau]\!]_I \qquad\qquad w \Vdash \{x \mapsto v\}[e] \rightsquigarrow v'$$

(3) *Under assumptions*

$$e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \qquad\qquad w \in [\![e : \tau_2]\!]_I \qquad\qquad w \Vdash e \rightsquigarrow v$$

*There exists* $\tau_1$, $v_1$ *and* $w_1$ *satisfying the following properties*

$$w \in [\![e_1 : \tau_1]\!]_I \qquad w \Vdash e_1 \rightsquigarrow v_1 \qquad w \sqsubseteq w_1 \qquad w_1 \in [\![\{x \mapsto v_1\}[e_2] : \tau_2]\!]_I$$

$$w_1 \Vdash \{x \mapsto v_1\}[e_2] \rightsquigarrow v$$

Proof. By inversions on big-step semantics and typing judgment. □

### A.5 Discussion about Extensions

Following the initial approach of (Ahmed et al. 2002) based on small-steps semantics, Theorem A.32 can be generalized in order to also ensure safety. Of course, this generalization should also be expressible for a pretty big-step semantics (Charguéraud 2013).

Extending the present formalization for product and sum types is straightforward: the product of two types is interpreted as the conjunction of their invariants; the sum of two types is interpreted as the disjoint union of their invariant. More formally,

$$[\![\tau_1 \times \tau_2]\!]_I\,((v_1, v_2)) \triangleq [\![\tau_1]\!]_I\,(v_1) \cap [\![\tau_2]\!]_I\,(v_2) \qquad [\![\tau_1 + \tau_2]\!]_I\,(\texttt{inj}_i\ v) \triangleq [\![\tau_i]\!]_I\,(v) \text{ for } i \in \{1, 2\}$$

Generalization to exception-handling should be also straightforward. Intuitively, exceptional values are monomorphic closed values. Hence, they only embed a trivial invariant. Thus, w.r.t invariant preservation, exception-handling has only an effect on the heap, which is quite similar to "`let − in`" construct.

The original model of (Ahmed et al. 2002) also deals with *recursive types*. However, we have not yet considered how this treatment of recursive types could be adapted in our context.