Verified Functional Programming of an Abstract Interpreter

Lucas Franceschino[1], David Pichardie[2], and Jean-Pierre Talpin[1]

[1] INRIA Rennes, France [2] ENS Rennes, France

Abstract. Abstract interpreters are complex pieces of software: even if the abstract interpretation theory and companion algorithms are well understood, their implementations are subject to bugs, that might question the soundness of their computations.
While some formally verified abstract interpreters have been written in the past, writing and understanding them requires expertise in the use of proof assistants, and requires a non-trivial amount of interactive proofs. This paper presents a formally verified abstract interpreter fully programmed and proved correct in the F* verified programming environment. Thanks to F* refinement types and SMT prover capabilities we demonstrate a substantial saving in proof effort compared to previous works based on interactive proof assistants. Almost all the code of our implementation, proofs included, written in a functional style, are presented directly in the paper.

1 Introduction

Abstract interpretation is a theory of sound approximation. However, most of available abstract interpreters do not formally establish a relation between their algorithmic theory and implementations. Several abstract interpreters have been proven correct. The most notable one is Verasco [10], a static analyser of C programs that has been entirely written, specified and proved in the proof assistant Coq. However, understanding the implementation and proof of Verasco requires an expertise with Coq and proof assistants.

Proofs in Coq are achieved thanks to extensive use of proof scripts, that are very difficult for non expert to read. By contrast with a handwritten proof, a Coq proof can be very verbose, and often does not convey a good intuition for the idea behind a proof. Thus, writing and proving sound a static analyzer is a complex and time-consuming task: for example, Verasco requires about 17k lines [10] of manual Coq proofs. Such an effort, however, yields the strongest guarantees and provides complete trust in the static analyser.

This paper showcases the implementation of a sound static analyser using the general-purpose functional programming language F⋆. Equipped with dependent types and built-in SMT solver facilities, F⋆ provides both an OCaml-like experience and proof assistant capacities. It recently shined with the Project Everest [1], which delivered a series of formally verified, high-performance, cryptographic libraries: HACL* [15], ValeCrypt [3] and EverCrypt [14]; that are for

instance used and deployed in Mozilla Firefox. While F* can always resort to proof scripts similar to Coq's ones, most proof obligations in F* are automatically discharged by the SMT solver Z3 [8].

We present an abstract interpreter equipped with the numerical abstract domain of intervals, forward and backward analyses of expressions, widening, and syntax-directed loop iteration. This paper makes the following contributions.

- It demonstrates the ease of use of F* for verified static analysis: we implement a verified abstract interpreter, and show about 95% of its 527 lines of code (proof included) directly in the paper.
- As far as we know, it is the first time SMT techniques are used for verifying an abstract interpreter.
- We gain an order of magnitude in the number of proof lines in comparison with similar works implemented in Coq.

Related work Efforts in verified abstract interpretation are numerous [7,4,2,13], and go up to Verasco [10], a modular, real-world abstract interpreter verified in Coq. Blazy et al. [2] and Verasco follow closely the modular design of Astrée [5]; we exhibit a similar modularity on a smaller scale. However, such analysers require a non-trivial amount of mechanized proofs: in constrast, this paper shows that implementing a formally verified abstract interpreter with very little manual proofs is possible. So far, verified abstract interpreters have been focused on concretization-based formalizations. The work of Darais et al. [6] is the only one to really consider the use of galois connections. They provide a minimalist abstract inteperter for imperative language but this interpreter seems very limited compared to ours. They use the Agda proof assistant which is comparable to Coq in terms of proof verbosity.

Overview Section 2 defines IMP, the language our abstract interpreter deals with, to which is given an operational semantics in Section 3. Then Section 4 formalizes lattices and abstract domains, while Section 5 instantiates them with the abstract domain of intervals. Section 6 derives more specific abstract domains, for numeric expressions and for memories. The latter is instantiated by Section 7, that implements an abstract weakly-relational memory. Finally, Section 8 presents the abstract interpretation of IMP statements.

The F* development is available on GitHub[1]. The resulting analyser is available online as a web application at `https://w95psp.github.io/verified-abstract-interpreter`.

2 IMP: a Small Imperative Language

To present our abstract intrepreter, we first show the language on which it operates: IMP. It is a simple imperative language, equipped with memories represented as functions from variable names `varname` to signed integers, $\texttt{int}_\texttt{m}$. This

[1] `https://github.com/W95Psp/verified-abstract-interpreter`

presentation lets the reader unfamiliar with F⋆ get used to its syntax: IMP's F⋆ definition looks like OCaml; the main difference is the explicit type signatures for constructors in algebraic data types. IMP has numeric expressions, encoded by the type `expr`, and statements `stmt`. Booleans are represented numerically: 0 represents `false`, and any other value stands for true. The enumeration `binop` equips IMP with various binary operations. The constructor `Unknown` encodes an arbitrary number. Statements in IMP are the assignment, the non-deterministic choice, the sequence and the loop.

```
type varname = | VA | VB | VC | VD    type mem τ = varname → τ
type binop = | Plus | Minus | Mult | Eq | Lt | And | Or
type expr = | Const:  int_m → expr    | Var: varname → expr
            | BinOp: binop → expr → expr → expr | Unknown
type stmt = | Assign: varname → expr → stmt | Assume: expr → stmt
            | Seq:       stmt → stmt → stmt | Loop:   stmt → stmt
            | Choice: stmt → stmt → stmt
```

The type $\texttt{int}_\texttt{m}$ is a *refinement* of the built-in F⋆ type $\mathbb{Z}$: while every integer lives in the type $\mathbb{Z}$, only those that respect certain bounds live in $\texttt{int}_\texttt{m}$. Numerical operations (+, − and ×) on machine integers wrap on overflow, i.e. adding one to the maximal machine integer results in the minimum machine integer. We do not give the detail of their implementation.

3 Operational Semantics

This section defines an operational semantics for IMP. It is also a good way of introducing more F⋆ features.

We choose to formulate our semantics in terms of sets. Sets are encoded as maps from values to propositions `prop`. Those are logical statements and shouln't be confused with booleans. Below, ⊆ quantifies over every *inhabitant* of a type: stating whether such a statement is true or false is clearly not computable. Arbitrarily complex properties can be expressed as propositions of type `prop`.

In the listing below, notice the greek letters: we use them throughout the paper. They denote implicit type arguments: for instance, below, ∈ works for any set `set τ`, with any type τ. F⋆ provides the propositional operators ∧, ∨ and ==, in addition to boolean ones (&&, || and =). We use them below to define the union, intersection and differences of sets.

```
type set τ = τ → prop              let (∈) (x: τ) (s: set τ) = s x
let (∩) s_0 s_1 = λx→x∈s_0 ∧ x∈s_1  let (\) s_0 s_1 = λv→s_0 v ∧ ~(s_1 v)
let (∪) (s_0 s_1: set τ): set τ = λx → x ∈ s_0 ∨ x ∈ s_1
let (⊆) (s_0 s_1: set τ): prop = ∀ (x: τ). x ∈ s_0 ⟹ x ∈ s_1
let set_inverse (s: set int_m): set int_m = λ(i: int_m) → s (-i)
```

To be able to work conveniently with binary operations on integers in our semantics, we define `lift_binop`, that lifts them as set operations. For example, the set `lift_binop (+) a b` (`a` and `b` being two sets of integers) corresponds to $\{va + vb \mid va \in \texttt{a} \wedge vb \in \texttt{b}\}$.

```
let lift_binop (op: τ → τ → τ) (a b: set τ): set τ
  = λr → ∃ (va:τ). ∃ (vb:τ). va ∈ a ∧ vb ∈ b ∧ r == op va vb
unfold let lift op = lift_binop (concrete_binop op)
```

The binary operations we consider are enumerated by `binop`. The function `concrete_binop` associates these syntactic operations to integer operations. For convenience, `lift` maps a `binop` to a set operation, using `lift_binop`. This function is inlined by F⋆ directly when used because of the keyword `unfold`; intuitively `lift` behaves as a macro.

```
unfold let concrete_binop (op: binop): int_m → int_m → int_m
  = match op with | Plus → n_add | Lt → lt_m | ... | Or → ori_m
```

The operational semantics for expressions is given as a map from memories and expressions to sets of integers. Notice the use of both the syntax `val` and `let` for the function $\texttt{osem}_{\texttt{expr}}$. The `val` syntax gives $\texttt{osem}_{\texttt{expr}}$ the type $\texttt{mem}\rightarrow\texttt{expr}\rightarrow\texttt{set int}_{\texttt{m}}$, while the `let` declaration gives its definition. The semantics itself is uncomplicated: `Unknown` returns the set of every $\texttt{int}_{\texttt{m}}$, a constant or a `Var` returns a singleton set. For binary operations, we lift them as set operations, and make use of recursion.

```
val osem_expr: mem → expr → set int_m
let rec osem_expr m e = λ(i: int_m)
  → match e with | Const x → i==x | Var v → i==m v | Unknown → ⊤
  | BinOp op x y → lift op (osem_expr m x) (osem_expr m y) i
```

The operational semantics for statements maps a statement and an initial memory to a set of admissible final memories. Given a statement `s`, an initial memory $\texttt{m}_\texttt{i}$ and a final one $\texttt{m}_\texttt{f}$, $\texttt{osem}_{\texttt{stmt}}$ `s` $\texttt{m}_\texttt{i}$ $\texttt{m}_\texttt{f}$ (defined below) is a proposition stating whether the transition is possible.

```
val osem_stmt (s: stmt): mem → set mem
let rec osem_stmt (s: stmt) (m_i m_f: mem)
  = match s with
  | Assign v e → ∀w. if v = w then m_f v ∈ osem_expr m_i e
                              else m_f w == m_i w
  | Seq a b  → ∃ (m_1: mem). m_1 ∈ osem_stmt a m_i ∧ m_f ∈ osem_stmt b m_1
  | Choice a b → m_f ∈ (osem_stmt a m_i ∪ osem_stmt b m_i)
  | Assume e → m_i == m_f ∧ (∃ (x: int_m). x ≠ 0 ∧ x ∈ osem_expr m_i e)
  | Loop a → closure (osem_stmt a) m_i m_f
```

The simplest operation is the assignment of a variable `v` to an expression `e`: the transition is allowed if every variable but `v` in $\texttt{m}_\texttt{i}$ and $\texttt{m}_\texttt{f}$ is equal and the final value of `v` matches with the semantics of `e`. Assuming that an expression is true amounts to require the initial memory to be such that at least a non-zero integer (that is, the encoding of `true`) belongs to $\texttt{osem}_{\texttt{expr}}$ $\texttt{m}_\texttt{i}$ `e`. The statement `Seq a b` starting from the initial memory $\texttt{m}_\texttt{i}$ admits $\texttt{m}_\texttt{f}$ as a final memory when there exists (i) a transition from $\texttt{m}_\texttt{i}$ to an intermediate memory $\texttt{m}_1$ with statement

a and (ii) a transition from $\mathtt{m}_1$ to $\mathtt{m}_\mathtt{f}$ with statement b. The operational semantics for a loop is defined as the reflexive transitive closure of the semantics of its body. The `closure` function computes such a closure, and is provided by F⋆'s standard library.

4 Abstract Domains

Our abstract interpreter is parametrized over relational domains. We instantiate it later with a weakly-relational [5] memory. This section defines lattices and abstract domains. Such structures are a natural fit for typeclasses [12], which allow for ad hoc polymorphism. In our case, it means that we can have one abstraction for lattices for instance, and then instantiate this abstraction with implementations for, say, sets of integers, then intervals, etc. Typeclasses can be seen as record types with dedicated dependency inference. Below, we define the typeclass `lattice`: defining an instance for a given type equips this type with a lattice srtucture.

Refinement types Below, the syntax $\mathtt{x}{:}\tau\{\mathtt{p\ x}\}$ denotes the type whose inhabitants both belong to τ and satisfy the predicate p. For example, the inhabitant of the type $\mathtt{bot}{:}\mathbb{N}\{\forall(\mathtt{n}{:}\mathbb{N})\,.\ \mathtt{bot}{\leq}\mathtt{n}\}$ is 0: it is the (only) smallest natural number. To typecheck $\mathtt{x}{:}\tau$, F⋆ collects the *proof obligations* implied by "$\mathtt{x}$ has the type τ", and tries to discharge them with the help of the SMT solver. If the SMT solver is able to deal with the proof obligations, then $\mathtt{x}{:}\tau$ typechecks. In the case of "0 is of type $\mathtt{bot}{:}\mathbb{N}\{\forall(\mathtt{n}{:}\mathbb{N})\,.\ \mathtt{bot}{\leq}\mathtt{n}\}$", the proof obligation is $\forall(\mathtt{n}{:}\mathbb{N})\,.\ 0{\leq}\mathtt{n}$.

Below, most of the types of the fields from the record type `lattice` are refined. Typechecking `i` against the type `lattice` τ yields a proof obligation asking (among other things) for `i.join` to go up in the lattice and for `bottom` to be a lower bound. Thus, if "`i` has type `lattice` τ" typechecks, it means there exists a proof that the properties written as refinements in `lattice`'s definition hold on `i`. We found convenient to let `bottom` represent unreachable states. Note `lattice` is under-specified, i.e. it doesn't require `join` to be provably a least upper bound, since such a property plays no role in our proof of soundness. This choice follows Blazy and al. [2].

```
class lattice τ = { corder: order τ
  ; join: x:τ → y:τ → r:τ {corder x r ∧ corder y r}
  ; meet: x:τ → y:τ → r:τ {corder r x ∧ corder r y}
  ; bottom: bot:τ{∀x. corder bot x}; top: top:τ{∀x. corder x top}}
```

For our purpose, we need to define what an abstract domain is. In our setting, we consider concrete domains with powerset structure. The typeclass `adom` encodes them: it is parametrized by a type τ of abstract values. For instance, consider `itv` the type for intervals: `adom itv` would be the type inhabited by correct abstract domains for intervals.

Implementing an abstract domain amounts to implementing the following fields: (i) c, that represents the type to which abstract values τ concretizes;

(ii) $\texttt{adom}_{\texttt{lat}}$, a lattice for τ; (iii) `widen`, a widening operator; (iv) γ, a monotonic concretization function from τ to `set c`; (v) `order_measure`, a measure ensuring the abstract domain doesn't admit infinite increasing chains, so that termination is provable for fixpoint iterations; (vi) $\texttt{meet}_{\texttt{law}}$, that requires `meet` to be a correct approximation of set intersection; (vii) $\texttt{top}_{\texttt{law}}$ and $\texttt{bot}_{\texttt{law}}$, that ensure the lattice's bottom concretization matches with the empty set, and similarly for `top`.

```
class adom τ = { c: Type; adom_lat: lattice τ
  ; γ: (γ: (τ → set c) {∀ (x y: τ). corder x y ⟹ (γ x ⊆ γ y)})
  ; widen: x:τ → y:τ → r:τ {corder x r ∧ corder y r}
  ; order_measure: measure adom_lat.corder
  ; meet_law: x:τ → y:τ → Lemma ((γ x ∩ γ y) ⊆ γ (meet x y))
  ; bot_law: unit → Lemma (∀ (x:c). ~(x ∈ γ bottom))
  ; top_law: unit → Lemma (∀ (x:c). x ∈ γ top)}
```

Notice the refinement types: we require for instance the monotony of γ. Every single instance for `adom` will be checked against these specifications. No instance of `adom` where γ is not monotonic can exist. With a proposition `p`, the `Lemma p` syntax signals a function whose outcome is computationally irrelevant, since it simply produces `()`, the inhabitant of the type `unit`. However, it does not produces an arbitrary `unit`: it produces an inhabitant of `_:unit {p}`, that is, the type `unit` refined with the goal `p` of the lemma itself.

For praticity, we define some infix operators for $\texttt{adom}_{\texttt{lat}}$ functions. The syntax `{|...|}` lets one formulate typeclass constraints: for example, `(⊑)` below ask F* to resolve an instance of the typeclass `adom` for the type τ, and name it `l`. Below, `(⊓)` instantiates the lemma $\texttt{meet}_{\texttt{law}}$ explicitly: $\texttt{meet}_{\texttt{law}}$ `x y` is a unit value that carries a proof in the type system.

```
let (⊑) {|l:adom τ|} = l.adom_lat.corder
let (⊔) {|l:adom τ|} (x y:τ): r:τ { corder x r ∧ corder y r
                                  ∧ (γ x ∪ γ y) ⊆ γ r } = join x y
let (⊓) {|l:adom τ|} (x y:τ): r:τ { corder r x ∧ corder r y
                                  ∧ (γ x ∩ γ y) ⊆ γ r }
  = let _ = meet_lawx y in meet x y
```

Lemmas are functions that produce refined `unit` values carrying proofs. Below, given an abstract domain `i`, and two abstract values `x` and `y`, `join_lemma i x y` is a proof concerning `i`, `x` and `y`. Such an instantiation can be manual (i.e. below, $\texttt{i.top}_{\texttt{law}}$ `()` in `top_lemma`), or automatic. The automatic instantiation of a lemma is decided by the SMT solver. Below, we make use of the `SMTPat` syntax, that allows us to give the SMT solver a list of patterns. Whenever the SMT solver matches a pattern from the list, it instantiates the lemma in stake. The lemma `join_lemma` below states that the union of the concretization of two abstract values `x` and `y` is below the concretization of the abstract join of `x` and `y`. This is true because of γ's monotony: we help a bit the SMT solver by giving a hint with `assert`. This lemma is instantiated each time a proof goal contains `x ⊑ y`.

Because of a technical limitation, we cannot write SMT patterns directly in the $\texttt{meet}_{\texttt{law}}$, $\texttt{bot}_{\texttt{law}}$ and $\texttt{top}_{\texttt{law}}$ fields of the class `adom`: thus, below we reformulate them.

```
let top_lemma (i: adom τ)      (let bot_lemma, meet_lemma = ...)
  : Lemma (∀ (x: i.c). x ∈ i.γ i.adom_lat.top)
          [SMTPat (i.γ i.adom_lat.top)] = i.top_law ()
let join_lemma (i: adom τ) (x y: τ)
  : Lemma ((i.γ x ∪ i.γ y) ⊆ i.γ (i.adom_lat.join x y))
          [SMTPat (i.adom_lat.join x y)]
  = let r = i.adom_lat.join x y in assert (γ x ⊆ γ r ∧ γ y ⊆ γ r)
```

5 An Example of Abstract Domain: Intervals

Until now, the F⋆ code we presented was mostly specificational. This section presents the abstract domain of intervals, and thus shows how proof obligations are dealt with in F⋆. Below, the type `itv'` is a dependent tuple: the refinement type on its right-hand side component `up` depends on `low`. If a pair `(|x,y|)` is of type `itv'`, we have a proof that $\texttt{x} \leq \texttt{y}$.

```
type itv' = low:int_m & up:int_m {low≤up}    type itv = withbot itv'
```

The machine integers being finite, `itv'` naturally has a top element. However, `itv'` cannot represent the empty set of integers, whence `itv`, that adds an explicit bottom element using `withbot`. The syntax `Val?` returns true when a value is not `Bot`. For convenience, `mk` makes an interval out of two numbers, and $\texttt{itv}_{\texttt{card}}$ computes the cardinality of an interval. We use it later to define a measure for intervals. `inbounds x` holds when $\texttt{x}:\mathbb{Z}$ fits machine integer bounds.

```
type withbot (a: Type) = | Val: v:a → withbot a | Bot
let mk (x y: ℤ): itv = if inbounds x && inbounds y && x ≤ y
                       then Val (|x,y|) else Bot
let itv_card (i:itv):ℕ = match i with | Bot→0 | Val i→dsnd i - dfst i + 1
```

Below, $\texttt{lat}_{\texttt{itv}}$ is an instance of the typeclass `lattice` for intervals: intervals are ordered by inclusion, the `meet` and `join` operations consist in unwrapping `withbot`, then playing with bounds. $\texttt{lat}_{\texttt{itv}}$ is of type `lattice itv`: it means for instance that we have the proof that the join and meet operators respect the order $\texttt{lat}_{\texttt{itv}}\texttt{.corder}$, as stated in the definition of `lattice`. Note that here, not a single line of proof is required: F⋆ transparently builds up proof obligations, and asks the SMT to discharge them, that does so automatically.

```
instance lat_itv: lattice itv =
  { corder = withbot_ord #itv' (λ(|a,b|) (|c,d|) → a≥c && b≤d)
  ; join = (λ(i j: itv) → match i, j with
       | Bot, k | k, Bot → k
       | Val (|a,b|), Val (|c,d|) → Val (|min a c, max b d|))
```

```
  ; meet = (λ(x y: itv) → match x, y with
       | Val (|a,b|), Val (|c,d|) → mk (max a c) (min b d)
       | _ → Bot); bottom = Bot; top = mk min_int_m max_int_m }
```

Such automation is possible even with more complicated definitions: for instance, below we define the classical widening with thresholds. Without a single line of proof, `widen` is shown as respecting the order `corder`.

```
let thresholds: list int_m = [min_int_m;-64;-32;-16;-8;-4;4;8;16;32...]
let widen_bound_r (b: int_m): (r:int_m {r>b ∨ b=max_int_m}) =
  if b=max_int_m then b else find' (λ(u:int_m) → u>b) thresholds
let widen_bound_l (b: int_m): (r:int_m {r<b ∨ b=min_int_m}) =
  if b=min_int_m then b else find' (λ(u:int_m) → u<b) (rev thresholds)
let widen (i j: itv): r:itv {corder i r ∧ corder j r}
  = match i, j with | Bot, x | x, Bot → x
  | Val(|a,b|),Val(|c,d|) →Val (| (if a≤c then a else widen_bound_l c)
                                , (if b≥d then b else widen_bound_r d)|)
```

Similarly, turning `itv` into an abstract domain requires no proof effort. Below $\texttt{itv}_{\texttt{adom}}$ explains that intervals concretize to machine integers ($\texttt{c = int}_\texttt{m}$), how it does so (with $\gamma = \texttt{itv}_\gamma$), and which lattice is associated with the abstract domain ($\texttt{adom}_{\texttt{lat}} = \texttt{lat}_{\texttt{itv}}$). As explained previously, the proof of a proposition p in F* can be encoded as an inhabitant of a refinement of `unit`, whence the "empty" lambdas: we let the SMT solver figure out the proof on its own.

```
let itv_γ : itv →set int_m = withbot_γ (λ(i:itv') x →dfst i≤x ∧ x≤dsnd i)
instance itv_adom: adom itv = { c = int_m    ; adom_lat = lat_itv; γ = itv_γ
    ; meet_law = (λ_ _→()); bot_law = (λ_→());  top_law = (λ_→())
    ; widen = widen      ; order_measure={f=itv_card;max=size_int_m}}
```

5.1 Forward Binary Operations on Intervals

Most of the binary operations on intervals can be written and shown correct without any proof. Our operators handle machine integer overflowing: for instance, `add_overflows` returns a boolean indicating whether the addition of two integers overflows, solely by performing machine integer operations. The refinement of `add_overflows` states that the returned boolean `r` should be true if and only if the addition in $\mathbb{Z}$ differs from the one in $\texttt{int}_\texttt{m}$. The correctness of $\texttt{itv}_{\texttt{add}}$ is specified as a refinement: the set of the additions between the concretized values from the input intervals is to be included in the concretization of the abstract addition. Its implementation is very simple, and its correctness proved automatically.

```
let add_overflows (a b: int_m)
  : (r: bool {r ⟺ int_arith.n_add a b ≠ int_m_arith.n_add a b})
  = ((b<0) = (a<0)) && abs a > max_int_m - abs b
let itv_add (x y: itv): (r: itv {(γ x + γ y) ⊆ γ r})
  = match x, y with | Val (|a, b|), Val (|c, d|)
```

```
                → if add_overflows a c || add_overflows b d
                  then top else Val (⟪a + c, b + d⟫) | _→Bot
```

However the SMT solver sometimes misses some necessary lemmas. In such cases, we can either guide the SMT solver by discriminating cases and inserting hints, or go fully manual with a tactic system à la Coq. Below, the `assert` uses tactics: everything within the parenthesis following the `by` keyword is a computation that manipulates proof goals. Our aim is to prove that subtracting two numerical sets a and b is equivalent to adding a with the inverse of b.

Unfortunately, due to the nature of `lift_binop`, this yields existential quantifications which are difficult for the SMT solver to deal with. After normalizing our goal (with `compute ()`), and dealing with quantifiers and implications (`forall_intro`, `implies_intro` and `elim_exists`), we are left with `∃y. b (-y) ∧ r=x+y` knowing `b z ∧ r=x-z` given some `z` as an hypothesis. Eliminating `∃y` with `-z` is enough to complete the proof.

We sadly had to prove that (not too complicated) fact by hand. This however shows the power of F*. Its type system is very expressive: one can state arbitrarily mathematically hard propositions (for which automation is hopeless). In such cases, one can always resort to Coq-like manual proving to handle hard proofs.

```
let set_inverse (s: set int_m): set int_m = λ(i: int_m) → s (-i)
let lemma_inv (a b: set int_m)
  : Lemma ((a-b) ⊆ (a+set_inverse b)) [SMTPat (a+set_inverse b)]
  = assert ((a-b) ⊆ (a+set_inverse b)) by (  compute ();
      let _= forall_intro () in let p_0 = implies_intro () in
      let witX,p_1 = elim_exists (binder_to_term p_0) in
      let witY,p_1 = elim_exists (binder_to_term p_1) in
      let z: ℤ = unquote (binder_to_term witY) in
      witness witX; witness (quote (-z)))
```

Notice the SMT pattern: the lemma $\texttt{lemma}_{\texttt{inv}}$ will be instantiated each time the SMT deals with an addition involving an inverse. Defining the subtraction $\texttt{itv}_{\texttt{sub}}$ is a breeze: it simply performs an interval addition and an interval inversion. Here, no need for a single line of proof for its correctness (expressed as a refinement).

```
let itv_inv (i: itv): (r: itv {set_inverse (γ i) ⊆ γ r})
  = match i with | Val⟪lower, upper⟫ → Val⟪-upper, -lower⟫ | _ → i
let itv_sub (x y:itv):(r:itv{(γ x - γ y) ⊆ γ r}) = itv_add x (itv_inv y)
```

Proving multiplication sound on intervals requires a lemma which is not inferred automatically:

$$\forall x \in [a,b], y \in [b,c]. \left[\min\left(ac,ad,bc,bd\right), \max\left(ac,ad,bc,bd\right)\right]$$

In that case, decomposing that latter lemma into sublemmas $\texttt{lemma}_{\texttt{min}}$ and $\texttt{lemma}_{\texttt{mul}}$ is enough. Apart from this lemma, $\texttt{itv}_{\texttt{mul}}$ is free of any proof term.

```
let lemma_min (a b c d: ℤ) (x: ℤ{a≤x ∧ x≤b}) (y: ℤ{c≤y ∧ y≤d})
  : Lemma (x×y≥a×c ∨ x×y≥a×d ∨ x×y≥b×c ∨ x×y≥b×d) = ()
unfold let in_btw (x: ℤ) (l u: ℤ) = l≤u ∧ x≥l ∧ x≤u
let lemma_mul (a b c d x y: ℤ)
  : Lemma (requires in_btw x a b ∧ in_btw y c d)
    (ensures x×y ≥ (a×c) `min` (a×d) `min` (b×c) `min` (b×d)
           ∧ x×y ≤ (a×c) `max` (a×d) `max` (b×c) `max` (b×d))
    [SMTPat (x×y); SMTPat (a×c); SMTPat (b × d)]
  = lemma_min a b c d x y; lemma_min (-b) (-a) c d (-x) y

let mul_overflows (a b:int_m):(r:bool{r≠inbounds (int_arith.n_mul a b)})
  = a ≠ 0 && abs b > max_int_m `div_m` (abs a)
let itv_mul (x y: itv): r:itv {(γ x × γ y) ⊆ γ r}
  = match x, y with
    | Val (|a, b|), Val (|c, d|) →
        let l = (a×c) `min` (a×d) `min` (b×c) `min` (b×d) in
        let r = (a×c) `max` (a×d) `max` (b×c) `max` (b×d) in
        if mul_overflows a c || mul_overflows a d
        || mul_overflows b c || mul_overflows b d
        then top else Val (|l, r|)
    | _ → Bot
```

The forward boolean operators for intervals require no proof at all; here we only give their type signatures. A function of interest is `itv_as_bool`: it returns `TT` when an interval does not contain 0, `FF` when it is the singleton 0, `Unk` otherwise.

```
let β (x: int_m): itv = mk x x
let itv_eq (x y:itv): r:itv {(γ x `n_eq` γ y) ⊆ γ r} =... let itv_lt =...
let itv_cγ (i: itv) (x:int_m): r:bool {r ⟺ itv_γ i x} =...
let itv_as_bool (x:itv): ubool  // with type ubool = |Unk|TT|FF
  = if β 0=x || Bot?x then FF else if itv_cγ x 0 then Unk else TT
let itv_andi (x y: itv): (r: itv {(γ x `n_and` γ y) ⊆ γ r})
  = match itv_as_bool x, itv_as_bool y with
  | TT, TT → β 1 | FF, _ | _, FF → β 0 | _, _ → mk 0 1
let itv_ori (x y: itv): (r: itv {(γ x `n_or` γ y) ⊆ γ r}) =...
```

5.2 Backward Operators

While a forward analysis for expressions is essential, another powerful analysis can be made thanks to backward operators. Typically, it aims at extracting information from a test, and at refining the abstract values involved in this test, so that we gain in precision on those abstract values. Given a concrete binary operator $\oplus$, we define $\overleftarrow{\oplus}$ its abstract backward counterpart. Let three intervals $x^\#$, $y^\#$, and $r^\#$. $\overleftarrow{\oplus}\ x^\#\ y^\#\ r^\#$ tries to find the most precise intervals $x^{\#\#}$ and $y^{\#\#}$ supposing $\gamma\ x^\# \oplus \gamma\ y^\# \subseteq \gamma\ r^\#$. The soundness of $\overleftarrow{\oplus}\ x^\#\ y^\#\ r^\#$ can be formulated as below. We later generalize this notion of soundness with the type $\mathtt{sound}_{\overleftarrow{\oplus}}$, which is indexed by an abstract domain and a binary operation.

```
let x##, y## = (⟵⊕) x# y# r# in
 ∀ x y. (x ∈ γ x# ∧ y ∈ γ y# ∧ op x y ∈ γ r#)
    ⟹ (x ∈ γ x## ∧ y ∈ γ y##)
```

As the reader will discover in the rest of this section, this statement of soundness is proved entirely automatically against each and every backward operator for the interval domain. For `op` a concrete operator, $\texttt{sound}_{\overleftarrow{\delta p}}$ `itv op` is inhabited by sound backward operators for `op` in the domain of intervals. If one shows that $\overleftarrow{\oplus}$ is of type $\texttt{sound}_{\overleftarrow{\delta p}}$ `itv (+)`, it means exactly that $\overleftarrow{\oplus}$ is a sound backward binary interval operator for `(+)`. The rest of the listing shows how light in proof and OCaml-looking the backward operations are. Below, we explain how $\overleftarrow{\texttt{lt}}$ works: it is a bit complicated because it hides a "$\overleftarrow{\texttt{ge}}$" operator.

```
let ⟵add: sound⟵δp itv n_add = λx y r → x ⊓ (r-y), y ⊓ (r-x)
let ⟵sub: sound⟵δp itv n_sub = λx y r → x ⊓ (r+y), y ⊓ (x-r)
let ⟵mul: sound⟵δp itv n_mul = λx y r →
  let h (i j:itv) = (if j=β 1 then i⊓r else i) in h x y, h y x
let ⟵eq:  sound⟵δp itv n_eq
  = λx y r → match itv_as_bool r with | TT → x⊓y,x⊓y | _ → x,y
let (\) (x y: itv): (r: itv {(γ x \ γ y) ⊆ γ r}) =...
let ⟵and: sound⟵δp itv n_and
  = λx y r → match itv_as_bool r,itv_as_bool x,itv_as_bool y with
    | FF, TT, _ → x, y ⊓ β 0          | FF, _, TT → x ⊓ β 0, y
    | TT,  _, _ → x \ β 0, y \ β 0    | _ → x, y
let ⟵or: sound⟵δp itv n_or
  = λx y r → match itv_as_bool r,itv_as_bool x,itv_as_bool y with
    | TT,FF,Unk | TT,FF,FF → x, y \ β 0 | TT,Unk,FF → x \ β 0, y
    | FF, _, TT | FF, TT, _ → x ⊓ β 0, y ⊓ β 0 | _ → x, y
```

Let us look at $\overleftarrow{\texttt{lt}}$. Knowing whether `x < y` holds, $\overleftarrow{\texttt{lt}}$ helps us refining `x` and `y` to more precise intervals. Let `x` be the interval $[0; \texttt{max}_{\texttt{int}_\texttt{m}}]$, `y` be $[5; 15]$ and `r` be $[0; 0]$. Since the singleton $[0; 0]$ represents `false`, $\overleftarrow{\texttt{lt}}$ `x y r` aims at refining `x` and `y` knowing that `x < y` doesn't hold, that is, knowing $\texttt{x} \geq \texttt{y}$. In this case, $\overleftarrow{\texttt{lt}}$ finds $\texttt{x'} = [5; \texttt{max}_{\texttt{int}_\texttt{m}}]$ and $\texttt{y'} = [5; 15]$. Indeed, when `r` is $[0; 0]$, `itv_as_bool r` equals to `FF`. Then we rewrite $\neg(\texttt{x} < \texttt{y})$ either as $\texttt{y} < \texttt{x} + 1$ (when `x` is `incrementable`) or as $\texttt{y} - 1 < \texttt{x}$. In our case, `x`'s upper bound is $\texttt{max}_{\texttt{int}_\texttt{m}}$ (the biggest $\texttt{int}_\texttt{m}$): x is not incrementable. Thus we rewrite $\neg([0; \texttt{max}_{\texttt{int}_\texttt{m}}] < [5; 15])$ as $[6; 16] < [0; \texttt{max}_{\texttt{int}_\texttt{m}}]$.

Despite of these different case handling, the implementation of $\overleftarrow{\texttt{lt}}$ required no proof: the SMT solver takes care of everything automatically.

```
let ⟵lt_true (x y: itv)
  = match x, y with | Bot, _ | _, Bot → x,y
  | Val⟮a,b⟯, Val⟮c,d⟯ → mk a (min b (d-1)), mk (max (a+1) c) d
let decrementable i=Val?i&&dfst(Val?.v i)>min_int_m  let incr.=...
let ⟵lt: sound⟵δp itv n_lt
  = λx y r → match itv_as_bool r with | TT → ⟵lt_true x y
```

```
| FF → if incrementable x // x < y ⟺ y > x+1
       then let ry, rx = lt←_true y (itv_add x (β 1)) in
            itv_sub rx (β 1), ry
       else if decrementable y // x < y ⟺ y-1 > x
            then let ry, rx = lt←_true (itv_sub y (β 1)) x in
                 rx, itv_add ry (β 1)
            else x,y | _ → x, y
```

6 Specialized Abstract Domains

Abstract domains are defined in Section 4 as lattices equipped with a sound concretization operation. Our abstract interpreter analyses IMP programs: its expressions are numerical, and IMP is equipped with a memory. Thus, this section defines two specialized abstract domains: one for numerical abstractions, and another one for memory abstractions.

6.1 Numerical Abstract Domains

In the section 5.2, we explain what a sound backward operator is in the case of the abstract domain of intervals. There, we mention a more generic type $\texttt{sound}_{\overleftarrow{op}}$ that states soundness for such operators in the context of any abstract domain. We present its definition below:

```
type sound_op← (a:Type) {|l:adom a|} (op:l.c→l.c→l.c)
  = op←: (a → a → a → (a & a)) {
      ∀ (x# y# r#: a). let x##, y## = op← x# y# r# in
         (∀ (x y: l.c). (x ∈ γ x# ∧ y ∈ γ y# ∧ op x y ∈ γ r#)
                  ⟹ (x ∈ γ x## ∧ y ∈ γ y##))}
```

We define the specialized typeclass $\texttt{num}_{\texttt{adom}}$ for abstract domains that concretize to machine integers. A type that implements an instance of $\texttt{num}_{\texttt{adom}}$ should also have an instance of `adom`, with $\texttt{int}_{\texttt{m}}$ as concrete type. Whence the fields $\texttt{na}_{\texttt{adom}}$, and $\texttt{adom}_{\texttt{num}}$. Moreover, we require a computable concretization function `cgamma`, that is, a function that maps abstract values to computable sets of machine integers: $\texttt{int}_{\texttt{m}} \to \texttt{bool}$. The β operator lifts a concrete value in the abstract world. We also require the abstract domain to provide both sound forward and backward operator for every syntactic operator of type `binop` presented in Section 2. The function `abstract_binop` maps an operator `op` of type `binop` to a sound forward abstract operator. Its soundness is encoded as a refinement. Similarly, $\overleftarrow{\texttt{abstract\_binop}}$ maps a `binop` to a corresponding sound backward operator. To ease backward analysis, $\texttt{gt}_0$ and $\texttt{lt}_0$ are abstractions for non-null positive and negative integers.

```
class num_adom (a: Type) =
{ na_adom: adom a; adom_num: squash (na_adom.c == int_m)
; cgamma: x#:a → x:int_m → b:bool {b ⟺ x ∈ γ x#}
```

```
; abstract_binop: op:_ → i:a → j:a → r:a {lift op (γ i) (γ j) ⊆ γ r}
; ←abstract_binop: (op: binop) → sound_{←op} a (concrete_binop op)
; gt_0: x#:a {∀(x:int_m). x>0 ⟹ x ∈ γ x#}
; lt_0: x#:a {∀(x:int_m). x<0 ⟹ x ∈ γ x#}; β: x:int_m →r:a{x ∈ γ r} }
```

For a proposition `p`, the F⋆ standard library defines `squash p` as the type `_:unit{p}`, that is, a refinement of the unit type. This can be seen as a lemma with no parameter.

Instance for intervals The section 5 defines everything required by $\texttt{num}_{\texttt{adom}}$, thus below we give an instance of the typeclass $\texttt{num}_{\texttt{adom}}$ for intervals.

```
instance itv_num_adom: num_adom itv = {
  na_adom = solve; adom_num = (); cgamma = itv_cγ ; β = (λ x → β x);
  abstract_binop = (function | Plus → itv_add ... | Or → itv_ori);
  ←abstract_binop = (function | Plus → ←add ... | Or → ←or );
  lt_0 = (mk min_int_m (-1)); gt_0 = (mk ( 1) max_int_m) }
```

6.2 Memory Abstract Domains

From the perspective of IMP statements, an abstract domain for abstract memories is fairly simple. An abstract memory should be equipped with two operations: assignment and assumption. Those are directly related to their syntactic counterpart `Assume` and `Assign`. Thus, $\texttt{mem}_{\texttt{adom}}$ has a field `assume_` and a field `assign`. The correctness of these operations are elegantly encoded as refinement types.

Let us explain the refinement of `assume_`: let $\texttt{m}_0^\#$ an abstract memory, and `e` an expression. For every concrete memory $\texttt{m}_0$ abstracted by $\texttt{m}_0^\#$, the set of acceptable final memories $\texttt{osem}_{\texttt{stmt}}$ `(Assume e)` $\texttt{m}_0$ should be abstracted by `assume_` $\texttt{m}_0^\#$ `e`.

```
class mem_adom μ = { ma_adom: adom μ; ma_mem: squash (ma_adom.c == mem);
  assume_: m0#:μ → e:expr → m1#:μ
    {∀ (m0: mem{m0 ∈ γ m0#}). osem_stmt (Assume   e) m0 ⊆ γ m1#};
  assign: m0#:μ → v:varname → e:expr → m1#:μ
    {∀ (m0: mem{m0 ∈ γ m0#}). osem_stmt (Assign v e) m0 ⊆ γ m1#}}
```

7 A Weakly-Relational Abstract Memory

In this section, we define a weakly-relational abstract memory. This abstraction is said weakly-relational because the entrance of an empty abstract value in the map systematically launches a reduction of the whole map to `Bot`. Below we define an abstract memory (`amem`) as either an unreachable state (`Bot`), or a mapping (`map` τ) from `varname` to abstract values τ. The mappings `map` τ are equipped with the utility functions `mapi`, $\texttt{map}_1$, $\texttt{map}_2$ and `fold`.

```
type map τ =...                    type amem τ = withbot (map τ)
let get' : map τ → varname → τ =...  let fold: (τ→τ→τ) → map τ → τ =...
let mapi: (varname → τ → β) → map τ → map β =...
let map₁: (τ→β) → map τ → map β = λf → mapi (λ_ → f)
let map₂: (τ→β→γ) → map τ → map β → map γ =...
```

A lattice structure The listing below presents `amem` instances for the typeclasses `order`, `lattice` and $\texttt{mem}_{\texttt{adom}}$. Once again, the various constraints imposed by these different typeclasses are discharged automatically by the SMT solver.

```
let amem_update (k: varname) (v: τ) (m: amem τ): amem τ
  = match m with | Bot → Bot
    | Val m → Val (mapi (λk' v' → if k'=k then v else v') m)
instance amem_lat {| l: adom τ |}: lattice (amem τ) =
  { corder = withbot_ord (λm₀ m₁ → fold (&&) (map₂ corder m₀ m₁))
  ; join = (λx y → match x, y with
      | Val x, Val y → Val (map₂ join x y) | m,Bot | _,m → m)
  ; meet = (λx y → match x, y with
      | Val x, Val y →
        let m = map₂ (⊓) x y in
        if fold (||) (mapi (λ_ v → l.adom_lat.corder v bottom) m)
        then Bot else Val m
      | _ → Bot); bottom = Bot; top = ...}
instance amem_adom {|l:adom τ|}: adom (amem τ) = { c = mem' l.c
  ; adom_lat=solve; meet_law=(λ_ _→()); top_law=(λ_→()); bot_law=(λ_→())
  ; γ = withbot_γ (λm# m → fold (∧) (mapi (λv x → m v ∈ γ x) m#))
  ; widen = (λx y → match x, y with
    | Val x, Val y → Val (map₂ widen x y) | m,Bot | _,m → m)
  ; order_measure = let {max; f} = l.order_measure in
    { f = (function | Bot → 0 | Val m# → 1 + fold (+) (map₁ f m#))
    ; max = 1 + max × 4 }}
```

The rest of this section defines a $\texttt{mem}_{\texttt{adom}}$ instance for our memories `amem`. The typeclass $\texttt{mem}_{\texttt{adom}}$ is an essential piece in our abstract interpreter: it provides the abstract operations for handling assumes and assignments.

Forward expression analysis We define $\texttt{asem}_{\texttt{expr}}$, mapping expressions to abstract values of type τ. It is defined for any abstract domain, whence the typeclass argument $\{|\texttt{num}_{\texttt{adom}}\ \tau|\}$. The abstract interpretation of an expression `e` given $\texttt{m}_0^\#$ an initial memory is defined below as $\texttt{asem}_{\texttt{expr}}\ \texttt{m}_0^\#\ \texttt{e}$. It is specified via a refinement type to be a sound abstraction of `e`'s operational semantics $\texttt{osem}_{\texttt{expr}}\ \texttt{m}_0\ \texttt{e}$. This function leverages the operators from the different typeclasses for which we defined instances just above. $\beta{:}\texttt{int}_{\texttt{m}}{\to}\tau$ and $\texttt{abstract\_binop}{:}\texttt{binop}{\to}\ldots$ come from $\texttt{num}_{\texttt{adom}}$, while $\texttt{top}{:}\tau$ comes from `lattice`.

```
val get: m:amem τ {Val? m}→varname→τ     let get (Val m) = get' m
let rec asem_expr {|num_adom τ|} (m₀#: amem τ) (e: expr)
```

```
: (r: τ { ∀ (m₀: mem). m₀ ∈ γ m₀# ⟹ osem_expr m₀ e ⊆ γ r })
= if m₀# ⊑ bottom then bottom else
  match e with | Const x → β x | Unknown → top | Var v → get m₀# v
  | BinOp op x y → abstract_binop op (asem_expr m₀# x) (asem_expr m₀# y)
```

Backward analysis Our aim is to have an instance for our memory of $\texttt{mem}_{\texttt{adom}}$: it expects an `assume_` operator. Thus, below a backward analysis is defined for expressions. Given an expression `e`, an abstract value $\texttt{r}^\#$ and a memory $\texttt{m}_0^\#$, $\overleftarrow{\texttt{asem}}$ `e` $\texttt{r}^\#$ $\texttt{m}^\#$ computes a new abstract memory. That abstract memory refines the abstract values held in $\texttt{m}_0^\#$ as much as possible under the hypothesis that `e` lives in $\texttt{r}^\#$. The soundness of this analysis is encoded as a refinement on the output memory. Given any concrete memory $\texttt{m}_0$ and integer `v` approximated by $\texttt{r}^\#$, if the operational semantics of `e` at memory $\texttt{m}_0$ contains `v`, then $\texttt{m}_0$ should also be approximated by the output memory.

When `e` is a constant which is not contained in the concretization of the target abstract value $\texttt{r}^\#$, the hypothesis "`e` lives in $\texttt{r}^\#$" is false, thus we translate that fact by outputting the unreachable memory `bottom`. In opposition, when `e` is `Unknown`, the hypothesis brings no new knowledge, thus we return the initial memory $\texttt{m}_0^\#$. In the case of a variable lookup (i.e. `e` = `Var v` for some `v`), we consider $\texttt{x}^\#$, the abstract value living at `v`. Since our goal is to craft the most precise memory such that `Var v` is approximated by $\texttt{r}^\#$, we alter $\texttt{m}_0^\#$ by assigning $\texttt{x}^\# \sqcap \texttt{r}^\#$ at the variable `v`. Finally, in the case of binary operations, we make use of the backward operators and of recursion. Note that it is the only place where we need to insert a hint for the SMT solver: we assert an equality by asking F⋆ to normalize the terms. We state explicitly that the operational semantics of a binary operation reduces to two existentials: we manually unfold the definition of $\texttt{osem}_{\texttt{expr}}$ and `lift_binop`. The `decreases` clause explains to F⋆ why and how the recursion terminates.

```
let rec ←asem {|l:num_adom τ|} (e: expr) (r#: τ) (m₀#: amem τ)
: Tot (m₁#: amem τ { (* decreases: *) m₁# ⊑ m₀# ∧ (* soundness: *)
      (∀(m₀:mem) (v:int_m). (v∈γ r# ∧ m₀∈γ m₀# ∧ v ∈ osem_expr m₀ e)
                                  ⟹ m₀ ∈ γ m₁#)}) (decreases e)
  = if m₀# ⊑ bottom then bottom else match e with
  | Const x → if cgamma r# x then m₀# else bottom | Unknown → m₀#
  | Var v→let x#: τ = r# ⊓ get m₀# v in
          if x#⊑bottom then Bot else amem_update v x# m₀#
  | BinOp op ex ey → let ←op = ←abstract_binop op in
      let x#, y# = ←op (asem_expr m₀# ex) (asem_expr m₀# ey) r# in
      let r#: amem τ = ←asem ex x# m₀# ⊓ ←asem ey y# m₀# in
      assert_norm (∀ (m: mem) (v: int_m). v ∈ osem_expr m e
        ⟺ (∃ (x y:int_m). x ∈ osem_expr m ex ∧ y ∈ osem_expr m ey
                        ∧ v == concrete_binop op x y));
      r#
```

Iterating the backward analysis While a concrete test is idempotent, it is not the case for abstract ones. Our goal is to refine an abstract memory under a hypothesis as much as possible. Since $\overleftarrow{\texttt{asem}}$ is proven sound and decreasing, we can repeat the analysis as much as we want. We introduce `prefixpoint` that computes a pre-fixpoint. However, even if the function from which we want to get a prefixpoint is decreasing, this is not a guarantee for termination. The type `measure` below associates an order to a measure that ensures termination. Such a measure cannot be implemented for a lattice that has infinite decreasing or increasing chains. We also require a maximum for this measure, so that we can reverse the measure easily in the context of postfixpoints iteration.

```
type measure #a (ord: a → a → bool)
  = { f: f: (a → ℕ) {∀ x y. x `ord` y ⟹ x =/= y ⟹ f x < f y}
    ; max: (max: ℕ {∀ x. f x < max}) }
```

Let us focus on `prefixpoint`: given an order $\sqsubseteq$ with its measure `m`, it iterates a decreasing function `f`, starting from a value `x`. The argument `r` is a binary relation which is required to hold for every couple `(x, f x)`. `r` is also required to be transitive, so that morally `r x (f`^{n} `x)` holds. `prefixpoint` is specified to return a prefixpoint `y`, that is, with `r x y` holding.

```
let rec prefixpoint ((⊑): order τ) (m: measure (⊑))
  (r: τ→τ→prop {trans r}) (f: τ→τ {∀e. f e ⊑ e ∧ r e (f e)}) (x:τ)
  : Tot (y: τ{r x y ∧ f y == y ∧ y ⊑ x}) (decreases (m.f x))
  = let x' = f x in if x ⊑ x' then x else prefixpoint (⊑) m r f x'
```

Below is defined $\overleftarrow{\texttt{asem}}$`_fp` the iterated version of $\overleftarrow{\texttt{asem}}$. Besides using `prefixpoint`, the only thing required here is to spell out `t`, the relation we want to ensure.

```
let asem_fp {|num_adom τ|} (e:expr) (r:τ) (m₀#:amem τ)
  : Tot (m₁#: amem τ {(∀ (m₀:mem) (v:int_m). m₁#⊑m₀# ∧
                    (v∈γ r ∧ m₀∈γ m₀# ∧ v∈osem_expr m₀ e) ⟹ m₀∈γ m₁#)})
  = let t (m₀# m₁#: amem τ) = ∀ (m: mem) (v: int_m).
      (v ∈ γ r ∧ m ∈ γ m₀# ∧ v ∈ osem_expr m e) ⟹ m ∈ γ m₁# in
    prefixpoint corder order_measure t (asem e r) m₀#
```

A `mem_adom` instance We defined both a forward and backward analysis for expressions. Implementing an `mem_adom` instance for `amem` is thus easy, as shown below. For any numerical abstract domain τ, `amemory_mem_adom` provides an `mem_adom`, that is, an abstract domain for memories, providing nontrivial proofs of correctness. Still, this is proven automatically.

```
instance amemory_mem_adom {| nd: num_adom τ |}: mem_adom (amem τ) =
  let adom: adom (amem τ) = amem_adom in { ma_adom = adom; ma_mem = ()
  ; assume_ = (λm# e → asem_fp e gt₀ m# ⊔ asem_fp e lt₀ m#)
  ; assign = (λm# v e → let v#: τ = asem_expr m# e in
                if v# ⊑ bottom then Bot else amem_update v v# m#)}
```

8 Statement Abstract Interpretation

Wrapping up the implementation of our abstract interpreter, this section presents the abstract interpretation of IMP statements. For every memory type μ that instantiates the typeclass of abstract memories $\texttt{mem}_{\texttt{adom}}$, the abstract semantics $\texttt{asem}_{\texttt{stmt}}$ maps statements and initial abstract memories to final memories. $\texttt{mem}_{\texttt{adom}}$ is defined and proven correct below.

Given a statement `s`, and an initial abstract memory $\texttt{m}_0^\#$, $\texttt{mem}_{\texttt{adom}}$ `s` $\texttt{m}_0^\#$ is a final abstract memory so that for any initial concrete memory `m` approximated by $\texttt{m}_0^\#$ and for any acceptable final concrete memory `m'` considering the operational semantics, `m'` is approximated by $\texttt{mem}_{\texttt{adom}}$ `s` $\texttt{m}_0^\#$. Here, we give two hints to the SMT solver: by normalization (`assert_norm`), we unfold the operational semantics in the case of choices or sequences. The analysis of an assignment or an assume is very easy since we already have operators defined for these cases. In the case of the sequence of two statements, we simply recurse. Similarly, when the statement is a choice, we recurse on its two possibilities. Then the two resulting abstract memories are merged back together. The last case to be handled is the loop, that is some statement of the shape `Loop body`. We compute a fixpoint $\texttt{m}_1^\#$ for `body`, by widening: it therefore approximates correctly the operational semantics of `Loop body`, since it is defined as a transitive closure. F*'s standard library provides the lemma `stable_on_closure`; of which we give a simplified signature below. The concretization γ $\texttt{m}_1^\#$ is a set, that is a predicate: we use this lemma with γ $\texttt{m}_1^\#$ as predicate `p` and with the operational semantics as relation `r`.

```
val simplified_stable_on_closure: r:(τ → τ → prop) → p:(τ → prop)
  → Lemma (requires ∀ x y. p x ∧ r x y ⟹ p y)
          (ensures ∀ x y. p x ∧ closure r x y ⟹ p y)
```

```
let rec asem_stmt {| md: mem_adom μ |} (s: stmt) (m0#: μ)
  : (m1#:μ{∀(m m':mem). (m∈γ m0# ∧ m'∈osem_stmt s m) ⟹ m'∈γ m1#})
  = assert_norm(∀s0 s1 (m0 mf:mem). osem_stmt (Seq s0 s1) m0 mf
     == (∃(m1:mem). m1 ∈ osem_stmt s0 m0 ∧ mf ∈ osem_stmt s1 m1));
    assert_norm(∀a b (m0 mf:mem). osem_stmt (Choice a b) m0 mf
     == (mf ∈ (osem_stmt a m0 ∪ osem_stmt b m0)));
    if m0# ⊑ bottom then bottom
    else match s with          | Assign v e → assign m0# v e
    | Assume e → assume_ m0# e | Seq s t → asem_stmt t (asem_stmt s m0#)
    | Choice a b → asem_stmt a m0# ⊔ asem_stmt b m0#
    | Loop body→let m1#: μ = postfixpoint corder order_measure
                    (λ(m#:μ) → widen m# (asem_stmt body m# <: μ))
                  in stable_on_closure (osem_stmt body) (γ m1#) (); m1#
```

Below we show the definition of `postfixpoint`, which is similar to `prefixpoint`. However, it is simpler because it only ensures its outcome is a postfixpoint.

```
let rec postfixpoint ((⊑): order τ) (m: measure (⊑))
  (f: τ → τ {∀ x. x ⊑ f x}) (x: τ)
```

```
: Tot (y: τ{f y == y ∧ (⊑) x y}) (decreases (m.max - m.f x))
= let x' = f x in if x' ⊑ x then x else postfixpoint (⊑) m f x'
```

9 Conclusion and further works

We presented almost the entire code of our abstract interpreter for IMP. Our approach to abstract interpretation is concretization-based, and follows the methodology of [2,10]. While using F⋆, we did not encountered any issue regarding expressiveness, and additionally gained a lot in proof automatization, to finally implement a fairly modular abstract interpreter. The table below compares the line-of-proof vs. line-of-code ratio of our implementation compared to some of the available verified abstract interpreters. Ours is up to 17 times more proof efficient. It is very compact, and requires a negligible amount of manual proofs. This comparison has its limits, since the different formalizations do not target the same programming languages: [10] and [2] handles the full C language, while [4] and the curent paper deal with more simple imperative languages. Also, proof effort usually does not scale linearly.

	Code	Proof	Ratio	Feature set
This paper	487	39	0.08	Simple imperative language
Pichardie et al. [4]	3 725	5 020	1.35	Simple imperative language
Verasco [10]	16 847	17 040	1.01	CompCert C langage
Blazy et al. [2]	4 000	3 500	0.87	CompCert C langage

The sources of our abstract interpreter sources are available along with a set of example programs; building it natively or as a web application is easy, reproducible[2] and automated.

This work is very far from the scope of Verasco which required about four years of human time [11,9], but our results, which required 3 months of work with F⋆ expertise, are very encouraging.

Further work We aim at following the path of Verasco by adding real-world features to our abstract interpreter and consider a more realistic target language such as one of the CompCert C-like input languages. One of the weakenesses of Verasco is its efficiency. Using Low⋆, a C DSL for F⋆, it is possible to write (with a nontrivial additionnal effort related to Low⋆) a very efficient C and formally verified abstract interpreter. This development also opens the path for enriching F⋆ automation via verified abstract interpretation.

[2] Our build process relies on the purely functional Nix package manager.

References

1. *Provably secure communication software,* `https://project-everest.github.io/`
2. S. Blazy, V. Laporte, A. Maroneze, D. Pichardie: *Formal verification of a C value analysis based on abstract interpretation.* In: SAS. pp. 324–344. LNCS (2013)
3. B. Bond, C. Hawblitzel, M. Kapritsos, R. Leino, J. Lorch, B. Parno, A. Rane, S. Setty, L. Thompson: *Vale: Verifying high-performance cryptographic assembly code.* In: Proceedings of the USENIX Security Symposium. USENIX (August 2017), distinguished Paper Award
4. D. Cachera, D. Pichardie: *A certified denotational abstract interpreter.* In: Proc. of International Conference on Interactive Theorem Proving (ITP-10). LNCS, vol. 6172, pp. 9–24 (2010)
5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival: *The astrée analyzer.* In: European Symposium on Programming. pp. 21–30 (2005)
6. D. Darais, M. Might, D. Van Horn: *Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis.* In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 552–571. OOPSLA 2015 (2015)
7. P. David: Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés. Ph.D. thesis, Université Rennes 1 (2005), in french
8. L. De Moura, N. Bjørner: *Z3: An efficient smt solver.* In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
9. J.H. Jourdan: Verasco: a Formally Verified C Static Analyzer. Theses, Universite Paris Diderot-Paris VII (May 2016)
10. J.H. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie: *A formally-verified C static analyzer.* In: 42nd symposium Principles of Programming Languages. pp. 247–259. ACM Press (2015)
11. V. Laporte: Verified static analyzes for low-level languages. Theses, Université Rennes 1 (Nov 2015)
12. G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy: *Meta-F*: Proof automation with SMT, tactics, and metaprograms.* In: 28th European Symposium on Programming (ESOP). pp. 30–59 (2019)
13. T. Nipkow: *Abstract interpretation of annotated commands.* In: Beringer, Felty (eds.) Interactive Theorem Proving (ITP 2012). vol. 7406, pp. 116–132 (2012)
14. J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C.M. Wintersteiger, S. Zanella-Béguelin: *Evercrypt: A fast, verified, cross-platform cryptographic provider.* In: IEEE Symposium on Security and Privacy. IEEE (May 2020)
15. J.K. Zinzindohoué, K. Bhargavan, J. Protzenko, B. Beurdouche: *Hacl: A verified modern cryptographic library.* In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1789–1806. CCS '17, Association for Computing Machinery (2017). https://doi.org/10.1145/3133956.3134043