Abstract Interpreters: a Monadic Approach to Modular Verification (DRAFT)

Sébastien Michellan
ENS Lyon
France

Laure Gonnord
LCIS (UGA, Grenoble INP)
France

Yannick Zakowski
Inria Lyon
France

Abstract

We argue that monadic interpreters built as layers of interpretations stacked atop the free monad constitute a promising way to implement and verify abstract interpreters in dependently-typed theories such as the one underlying the Coq proof assistant.

The approach enables modular proofs of soundness of the resulting interpreters. We provide generic abstract control flow combinators proven correct once and for all against their concrete counterpart. We demonstrate how to relate concrete handlers implementing effects to abstract variants of these handlers, essentially capturing the traditional soundness of transfer functions in the context of monadic interpreters. Finally, we provide generic results to lift soundness statements via the interpretation of stateful and failure effects.

We formalize all the aforementioned combinators and theories in Coq, and demonstrate their benefits by implementing and proving correct two illustrative abstract interpreters for a structured imperative language and a toy assembly.

1 Introduction

The realm of mechanized verification of programming languages has reached a staggering degree of maturity. Backing up meta-theoretical results with a formalization in a proof assistant has become increasingly routine in the programming language research community [32]. But such formalization efforts have not only become more common, they have grown in scale and ambition: large-scale software is verified against faithful semantics of existing industrial-strength languages [13, 17, 23, 24].

When it comes to formalized proofs, details of representation matter greatly. Propositionally specified transition systems are by and large the most popular: typically, the small-step semantics is specified through proof rules, using a binary relation between dynamic configurations, before considering its transitive closure. While extremely successful, such approaches have drawbacks. On the practical side, these semantics are non-executable at their core, hence requiring some significant extra work to support crucial practice such as differential testing against industrial reference interpreters. In reaction, frameworks such as Skeletal Semantics [2] or the K framework [34] have been designed notably in order to support the automatic derivation of executable interpreters from the formal semantics. On the theoretical side, they tend to lack support for equational reasoning, and often give up on compositionality—recursive definition on the syntax—and modularity—independent definition and combination of the features of the language.

These shortcomings become increasingly painful when formal developments scale. In contrast, when applicable, monads and subsequently algebraic effects have long been recognized as an appealing approach to modeling the semantics of effectful programs. The monad laws, extended with algebraic domain-specific equations capturing the semantics of the effects at hand, yield powerful reasoning principles. Monads have been both a pen-and-paper theoretical tool and a practical programming paradigm for decades, but have also become increasingly popular in the mechanized realm. In particular, *free monads* [37] have been at the root of flexible, general-purpose reasoning frameworks. Variations on this idea have appeared throughout the literature, for instance as the *program monad* in the FreeSpec project [26], as *I/O-trees* [14], and as McBride's *general monad* [27].

In this paper, we focus on *interaction trees* [39] (ITrees), a recent realization of this approach as a Coq library. ITrees are defined as a coinductive variant of the *freer monad* [21] and are also closely related to *resumption monads* [31]. The library provide rich reusable components to model and reason about effectful, recursive, interactive computations, while supporting extraction. In particular, they make the definition of denotational semantics for first order languages with first order effects straightforward.

ITrees have been applied in a wide range of projects, such as modeling network servers [22, 41], transactional objects [25], concurrency [4], or non-interference [36]. Their largest application is arguably the Vellvm project [40, 42], providing a compositional, modular and executable semantics for a large sequential subset of LLVM's intermediate representation. This application leverages the approach's

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

modularity heavily, structuring the semantics into a series of layers, each plugging in an independent implementation of a feature of the language.

In the present work, we seek to offer similar benefits of modularity and reusable components for writing verified static analyses against ITree-based formal semantics. We place ourselves more specifically in the *abstract interpretation* framework [6, 7]. Abstract interpretation is well known for providing rich ways of combining abstractions, through products [5] or communication-based protocols [8, 16]. In this paper, we do not focus our attention on such construction of rich abstract domains. Rather, we follow the *big-step abstract interpreter* line of works [2, 9, 18, 20] in seeking to provide rich reusable combinators to lighten the construction of *verified* abstract interpreters.

Our contributions can be crystallized as follows:

- we identify `aflow`, an extensible monad for monadically programming abstract interpreters in Coq;
- we capture a composable notion of soundness suitable for expressing the correctness of partially interpreted monadic interpreters;
- we define and certify a collection of flow combinators;
- we demonstrate our library by proving correct two abstract interpreters for a structured imperative language and for a control-flow graph language;
- we emphasize that most of the proof effort is internalized in the library: components and their proofs of correctness are reused.

All of our results are formalized in Coq and provided as an open source library.

Section 2 starts by providing necessary background on ITrees and abstract interpretation. Section 3 illustrates the challenges and motivates our design, whose programmatic component is described in more detail in Section 4. It is exemplified in Section 5, describing our case studies. Finally, Section 6 provides details on the meta-theory provided by our library, and the structure of a proof of soundness of an abstract interpreter from the perspective of a user of our library. We conclude with related work.

2 Background

Typographic remarks. For clarity and conciseness, we take some light liberties with Coq code included in this paper. When clear from context, we omit implicit arguments. We use mathematical notations in lieu of traditional identifiers. Furthermore, we present simplified versions of the code such as specialized definitions where the artifact is parametrized, or **`Fixpoint`** instead of **`Equations`**. We hope it will create no confusion, and systematically reference the accompanying with hyperlinks symbolized by 🐓 [1]. We make use of functions between type families, writing `E ⇝ F ::= ∀ {X}, E X → F X` for such a function between `E, F : Type → Type`. We write 𝟙 and `()` for the unit type and its inhabitant.

[1]https://gitlab.inria.fr/sebmiche/itree-ai

```
(* Embedding of pure computations *)
  ITree.ret (v : R) : itree E R.
(* Sequencing computations *)
  ITree.bind (u: itree E T) (k: T → itree E U) : itree E U.
(* Atomic itrees triggering a single event. *)
  ITree.trigger : E ⇝ itree E.
(* Fixed-point combinator *)
  ITree.iter (body : I → itree E (I+R)) : I → itree E R.
```

Figure 1. ITrees: type signature of the main combinators

2.1 Interaction Trees and Monadic Interpreters

Interaction Trees [39] (ITrees) have emerged in the Coq ecosystem as a rich toolbox for building compositional and modular monadic interpreters for first order languages. The library also provides an equational theory for reasoning about equivalence and refinement of computations. Through this section, we introduce the programmatic side of this framework.

ITrees are a data structure for representing computations interacting with an external environment through *visible events*, defined as:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
  | Ret (r: R)         (* terminating computation *)
  | Tau (t: itree E R) (* "silent" tau transition *)
  | Vis {A: Type} (e : E A) (k : A → itree E R).
                (* event e yielding an answer in A *)
```

The datatype takes two parameters: a signature `E` that specifies the set of interactions the computation may have with the environment, and the type `R` of values that it may return. ITree computations can be thought of as trees built out of three constructors. Leaves, via the `Ret` constructor, model pure computations, carrying `R` values. `Vis` nodes model an effect `e` being performed, before yielding to the continuation `k` with the value resulting from `e`. Finally, ITrees are defined coinductively, allowing them to model diverging computations as non-well-founded trees. Accordingly, the `Tau` constructor represents a non-observable internal step that occurs, much as in Capretta's delay monad [3].

One may think of ITrees as a low level imperative programming language embedded inside of Gallina. The library exposes the primitive combinators shown in Figure 1. ITrees have a monadic structure: pure computations can be embedded via `ret`, and computations can be sequenced with the traditional `bind` construct.[2] A minimal effectful computation can be written `ITree.trigger e`, yielding control to the environment to perform an effect `e` and returning the result. By virtue of their coinductive nature, ITrees form what is sometimes referred to as a completely iterative monad [1]. From the eye of the programmer, this captures the ability to write fixpoints using the `iter` combinator. Operationally, `iter f i` is the computation performing `f i`, checking whether the result

[2]We use `x ← c;; k` as a notation for `bind c (fun x ⇒ k x)`.

```
op ∈ Op ::= ⊕ | ⊖ | ⊗
e  ∈ 𝔼 ::= x ∈ 𝕏 | n ∈ 𝕍 | e1 op e2
s  ∈ ℂ ::= skip | assert e | x := e | s1 ; s2
         | if e then s1 else s2 | while e do s
```

Figure 2. Imp: abstract syntax

is a new accumulator `inl j` and continuing with `iter f j`, or if it is a final value `inr r` and returning `r`.

To make things concrete, we turn to our main running example: a traditional Imp language [30] whose abstract syntax is depicted on Figure 2. Arithmetic expressions contain variables in 𝕏, literals in 𝕍 and binary operations. Statements include the usual assignments, sequencing, conditionals and loops, as well as an assert statement acting as a no-op if the condition is valid, as failure otherwise.

We model Imp's dynamic semantics using ITrees. The process, already illustrated in [39], and at scale notably in Vellvm [40], is split into two main phases.

Representation. First, we represent the abstract syntax into an interaction tree: one can think of it as a coinductive representation of the labeled transition system denoting the program. We hence collect the labels, that is the effects that the program may perform:[3]

```
Variant arithE : Type → Type :=
  | Compute (op : Op) (l r : 𝕍) : arithE 𝕍.
Variant memE : Type → Type :=
  | Read (a: 𝕏)                  : memE 𝕍
  | Write (a: 𝕏) (v: 𝕍)          : memE 𝟙.
Variant assertE : Type → Type :=
  | Assert (v : 𝕍)               : assertE 𝟙.
```

The interface specifies for each event its arguments and its return type. We take stock of three families. `arithE` events perform a binary arithmetic computation, expecting a value back. `memE` events interact with the memory through reads—expecting back a value—and writes—expecting an acknowledgement (encoded via 𝟙). `assertE` events expect an acknowledgement in case of success of the test. Let `impE` be the disjoint sum of all events (`arithE +' memE +' assertE`). For each event, we use its lowercase counterpart as a shortcut for triggering it: for instance, `write x 1` is a shortcut for `ITree.trigger (Write x 1)`.

Figure 3 shows the representation functions for Imp. Note that they are defined by recursion on the syntax. Using the ITree combinators, the code is mostly straightforward, closely resembling an interpreter written in a language such as Haskell. The ternary `cond` combinator simply desugars to a Coq level `if` construct. The main subtlety resides in the representation of loops: we define on top of `iter` a `while` combinator using the accumulator as a single bit of information informing the combinator when to escape:

```
Definition while (guard: itree E 𝕍) (body: itree E 𝟙) :=
  ITree.iter (fun (_ : 𝟙) ⇒
    v ← guard;;
    if v =? 0 then ITree.ret (inr ())) ()
    else body;; ITree.ret (inl ())
```

Interpretation. By representing Imp's abstract syntax as ITrees, we have given a semantics to its control flow, but its effects remain purely syntactic. We now provide *handlers* for each category of effects, implementing them through an appropriate monad transformer, as shown on Figure 4.

The arithmetic operations we consider here are pure, hence `h_arith` does not introduce any transformer; it only relies on a pure implementation `compute_binop` omitted here. Memory interactions are stateful, which we implement with the traditional state transformer over a concrete map `mem` providing `mem_store` and `mem_get` operations (the latter returning `0` by default). Finally, asserts may fail, hence `h_arith` introduces failure via the usual `failT` transformer.

We are finally ready to define Imp's denotation, `eval`, by successively interpreting all three layers of effects. Each interpretation removes an event family from the signature and adds a monad transformer. The resulting semantic domain is hence `failT (stateT S (itree ∅))` (where `∅` is the empty signature), i.e. a stateful computation that may diverge or end in a state of error.

Getting there requires two final ingredients. First, the `hoist` monadic combinator lifts a monad morphism `f: m ⇝ n` under a transformer `t:hoist f: t m ⇝ t n`. Second, ITrees's `interp` function lifs an implementation of events as a handler to a whole tree: `interp (h: E ⇝ M): itree E ⇝ M`.[4] Putting all the ingredients together:

```
Definition eval (s: ℂ) : failT (stateT S (itree ∅)) 𝟙:=
  hoist (fun u ⇒ hoist (interp_pure h_arith)
                       (interp_state h_state u))
        (interp_fail h_fail ⟦s⟧).
```

2.2 Abstract Interpretation

Abstract interpretation [7] provides a simple and elegant way to compute sound approximations of a program's semantics, by mimicking the concrete evaluation of the program in an abstract fashion. The analysis defines an over-approximation of the set of states and control flow of the concrete program, trading accuracy in exchange for guaranteed termination.

An abstract domain defines approximations of program objects (values); for simplicity in this paper we consider *non-relational numerical domains.* To further exemplify, we shall consider the Interval domain, which abstracts *sets of numerical values* $V \subseteq \mathbb{Z}^d$ by $V^\# \subseteq \text{Interval}^d$, where Interval $= (\mathbb{Z} \cup -\infty) \times (\mathbb{Z} \cup +\infty)$.

[3]Divergence is the only effect hard-coded in the structure, it is not accounted for in the signature.

[4]The `interp` function is parametric in `M`. However, to avoid ambiguity in the remaining, we postfix its name by the effect introduced by the handler.

```
Fixpoint ⟦·⟧ₑ (e: 𝔼): itree impE 𝕍:=          Fixpoint ⟦·⟧ (s: ℂ): itree impE 𝟙:=
  match e with                                   match s with
  | (x: 𝕏)  ⇒ read x                             | skip               ⇒ ITree.ret ()
  | (n: 𝕍)  ⇒ ITree.ret n                        | assert b           ⇒ v  ← ⟦b⟧ₑ;; assert v
  | a op b  ⇒ l  ← ⟦a⟧ₑ;;                        | x := e             ⇒ v  ← ⟦e⟧ₑ;; write x v
              r  ← ⟦b⟧ₑ;;                        | c1; c2             ⇒ _  ← ⟦c1⟧;; ⟦c2⟧
              compute op l r                     | if b then t else e ⇒ v  ← ⟦b⟧ₑ;; cond (v =? 0) ⟦t⟧ ⟦e⟧
  end.                                           | while b do c       ⇒ while ⟦b⟧ₑ ⟦c⟧
                                                 end.
```

Figure 3. Imp: ITree-representation (with embedded Imp syntax)

```
Definition compute_binop (op : Op) (l r : 𝕍) : 𝕍:= (..).
Definition h_arith `{Monad M} : arithE ⤳ M :=
  fun '(Compute op l r) ⇒
    ret (compute_binop op l r).

Definition h_assert `{Monad M} : assertE ⤳ failT M :=
  fun '(Assert v) ⇒
    ret (if v =? 0 then None else (Some tt)).

Definition h_mem `{Monad M} mem: memE ⤳ stateT mem M :=
  fun e m ⇒ match e with
  | Read a    ⇒ ret (m, mem_get m a)
  | Write a v ⇒ ret (mem_store m a v, tt)
  end.
```

Figure 4. Imp: effect handlers

We use the standard formalization of domains as *lattices* equipped with union (join, $\sqcup$), minimal and maximal elements ($\bot$, $\top$), and a decidable order denoted by $\subseteq$?. A pair of abstraction and concretization functions (α, γ) forming a Galois connection is expected to relate the abstract domain to the concrete one, although we follow Pichardie [29]'s γ-only encoding as summed up Figure 5. To ensure termination during the analysis of loops, abstract domains come with a widening operator equipped with a well-founded measure over vectors of naturals.

Beyond the fact that computations operate over abstract values and stores, we use Imp to informally highlight, in a big-step style, how uncommon the control flow of the resulting interpreter is, a crucial difficulty for the framework we develop in the following sections. Conditionals must run both branches "in parallel", from the same initial memory, and join the results:[5]

$$\llbracket \texttt{if b then ct else ce} \rrbracket^\#(m^\#) = \begin{array}{l} m_1^\# \leftarrow \llbracket \texttt{ct} \rrbracket^\#(m^\#) \\ m_2^\# \leftarrow \llbracket \texttt{ce} \rrbracket^\#(m^\#) \\ \texttt{return}\ (m_1^\# \sqcup m_2^\#). \end{array}$$

Loops `while b do c1` could naively perform an unbounded number of (abstract) iterations.[6] Termination is hence ensured by the usage of the widening operator, which converges due to its well-founded measure:

$$\llbracket \texttt{while b do c1} \rrbracket^\#(m^\#) = \text{repeat}\ \begin{array}{l} m_1^\# \leftarrow \llbracket \texttt{c1} \rrbracket^\#(m^\#) \\ m_2^\# \leftarrow \texttt{widen}\ m^\#\ (m^\# \sqcup m_1^\#) \\ \text{if}\ (m_2^\# \subseteq m^\#) \\ \quad \texttt{return}\ m^\# \\ m^\# \leftarrow m_2^\#. \end{array}$$

That is, $\llbracket \texttt{while b do c1} \rrbracket^\#(m^\#)$ is the least fixpoint of iterating the loop body with widening, applied on $m^\#$.

From these ingredients (replacing computations with abstract domain operations and control flow with abstract transformers), the abstract interpretation framework [7] guarantees that the computation of the abstract semantics *always terminates* and is *safe*, in the sense that the concretization of the obtained semantics is always larger than the (usually untractable) concrete semantics.

3 Design of a layered abstract interpreter

We are now ready to consider the contribution of this paper: designing a monadic abstract interpreter built in a modular fashion and resulting in a static analysis proven correct against the concrete semantics defined in Section 2.1. This section focuses on how surface-level requirements influence the design of the abstract interpreter. We build up to Figure 6, providing a bird's eye view of the interpreter instantiated on a simple Imp program, to prepare for formal details in Sections 4 and 6.

Perhaps the most striking feature of the approach is that we build a hybrid *abstract program* (Figure 6, ❶), generated from the source program while also embedding key components of an abstract interpreter, such as lattices and a fixpoint approximation scheme. It is similar to an abstract interpreter partially evaluated on a chosen input program. As a result, the abstract program exhibits behaviors from both its source and these generic abstract interpretation components.

Interpreting before unfolding control flow. As an immediate consequence of this duality, consider the C-like source expression ⟨*condition*⟩ ? ⟨*true-value*⟩ : ⟨*false-value*⟩,

[5] Guards are typically accounted for; for simplicity, we ignore them here.

[6] Similarly, we ignore the loop guard, for the sake of simplicity.

Operation or relation	Axioms	
$\in: V \to V^\# \to \texttt{Prop}$	Relates to the Galois connection by $v \in x \triangleq v \in \gamma(x)$	
$\subseteq? : V^\# \to V^\# \to \texttt{bool}$	Preorder	$c \in x \to x \subseteq? y \to c \in y$
$\texttt{join}\ (\sqcup) : V^\# \to V^\# \to V^\#$	$x \subseteq x \sqcup y$	$y \subseteq x \sqcup y$
$\texttt{widen} : V^\# \to V^\# \to V^\#$	$x \subseteq \texttt{widen}\ x\ y$	$y \subseteq \texttt{widen}\ x\ y$
$\texttt{measure\_N} : \texttt{nat}$	$\texttt{measure}\ (\texttt{widen}\ x\ y) \leq \texttt{measure}\ x$ (lexicographic order)	
$\texttt{measure} : V^\# \to \texttt{nat}^{\texttt{measure\_N}}$	$\neg(y \subseteq? x) \to \texttt{measure}\ (\texttt{widen}\ x\ y) < \texttt{measure}\ x$ (lexicographic order)	
$\top, \bot : V^\#$	$\forall v,\ v \in \top$	$\texttt{measure}\ \top = (0, \ldots, 0)$
$\texttt{const} : V \to V^\#$	$v \in \texttt{const}\ v$	
$\texttt{istrue}, \texttt{isfalse} : V^\# \to \texttt{bool}$	$\texttt{istrue}\ x \to v \in x \to v \neq 0$	$\texttt{isfalse}\ x \to v \in x \to v = 0$
$\texttt{opp} : V^\# \to V^\#$	$v \in x \to -v \in \texttt{opp}\ x$	
$\texttt{add}, \texttt{sub} : V^\# \to V^\# \to V^\#$	$v_1 \in x_1 \to v_2 \in x_2 \to v_1\ \{+,-\}\ v_2 \in \{\texttt{add}, \texttt{sub}\}\ x_1\ x_2$	

Figure 5. Common lattice operations and numerical domain for Imp. V and $V^\#$ represent concrete and abstract values.

which evaluates to *true-value* when the *condition* is true, and *false-value* otherwise. Assuming that the condition is not statically determined, the abstract program computes an approximation of both options using the lattice's join operator. However, a particular order must be chosen, e.g., the *true* branch first. Thus, it is tempting to denote the abstract program as the following ITree:

$$\begin{aligned} t &\leftarrow [\![\mathit{true\text{-}value}]\!]^\#\ ;; \\ f &\leftarrow [\![\mathit{false\text{-}value}]\!]^\#\ ;; \\ &\texttt{ret}\ (t \sqcup f). \end{aligned}$$

But *unfolding* the conditional choice in this way leads to an issue incompatible with the modular construction we seek: a *sequence point* between the computations of t and f is introduced. Since the semantics of "sequence" change with the current monad, this hypothetical abstract program denoted as an ITree ceases to abstract the behavior of the concrete program once we interpret its effects. For instance, interpreting into the state monad w.r.t. a handler h yields

$$\begin{aligned} s \mapsto (s', t) &\leftarrow \texttt{interp\_state}\ h\ [\![\mathit{true\text{-}value}]\!]^\#\ s\ ;; \\ (s'', f) &\leftarrow \texttt{interp\_state}\ h\ [\![\mathit{false\text{-}value}]\!]^\#\ s'\ ;; \\ &\texttt{ret}\ (s'', t \sqcup f), \end{aligned}$$

which incorrectly uses the final state s' of the *true* branch as the initial state of the *false* branch, instead of the original state s. This results from a confusion between source-level sequence (where we want to introduce new monadic effects by interpretation) and an internal sequenced computation inside an abstract interpretation algorithm (where we don't). More generally, we need to be able to differentiate control flow from the source program and control flow from abstract interpretation components. Since an ITree cannot capture this difference, we use a different structure where source control flow remains symbolic during interpretation, and unfold it to an ITree only once all events are gone.

The abstract control flow monad. We first denote the program into an inductive freer monad with symbolic control flow operations dubbed `aflow` (formally defined in Section 4.1). In this form, abstract programs can be seen as a tree of control flow *combinators* such as *abstract cond* ❷ and *abstract sequence* ❸, with atomic computations as leaves.

We maintain this structure throughout monadic interpretation by applying monad transformers and interpreting events without changing the combinator tree. Once all events are interpreted, we unfold control flow combinators, collapsing the abstract program into a pure ITree computation ❹.

Preservation by interpretation. Our goal of retaining the `aflow` structure when we switch monads raises the question of control flow combinators being "preserved by interpretation". We show in Section 4.2 and 4.3 that we can do so *syntactically*, in that the interpretation of any abstract combinator, e.g. `abstract_cond`, can be expressed as another instance of the same combinator. This relies on combinators being able to internalize the monadic effects added when interpreting, thanks to extra parameterization.

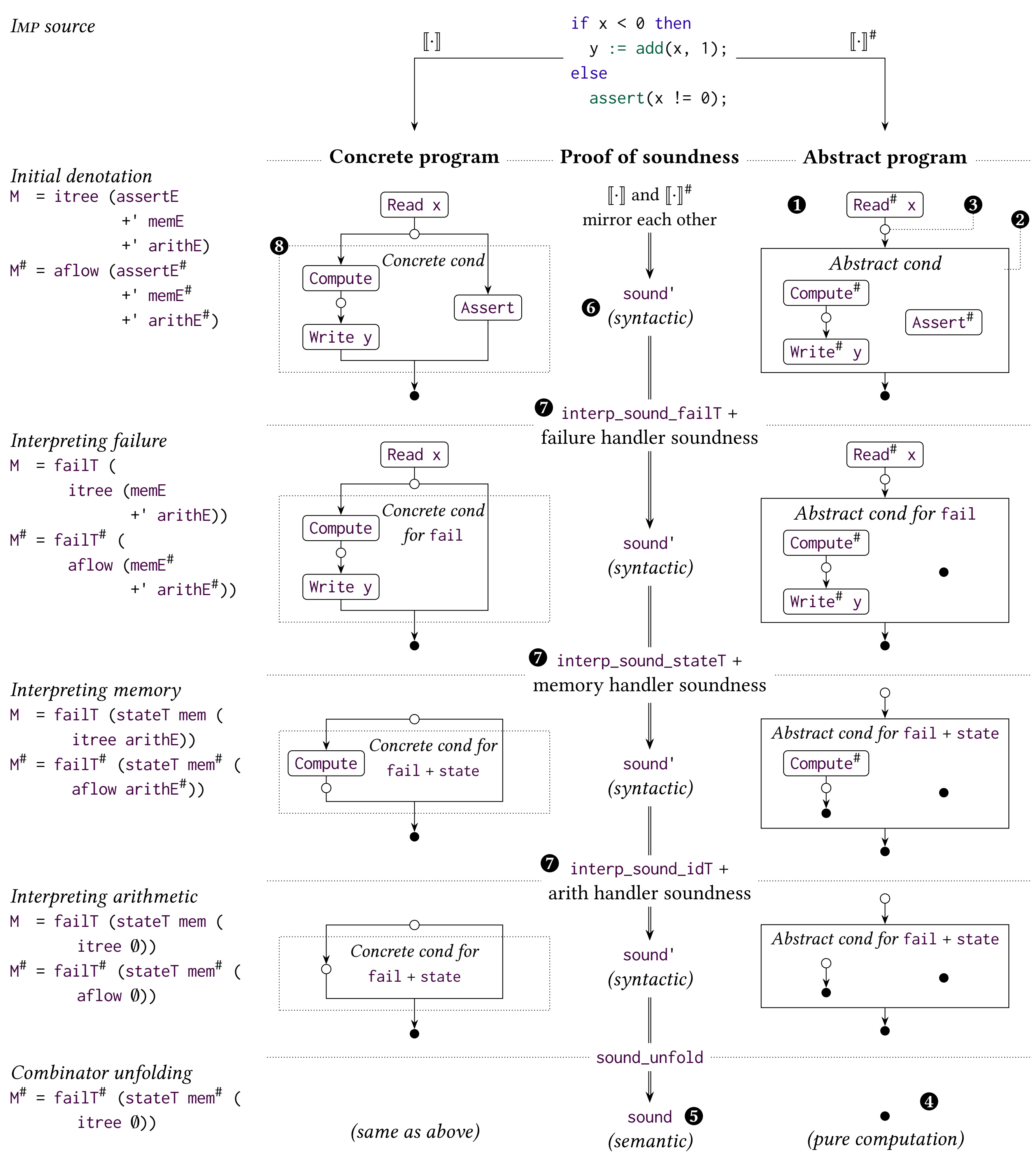
Galois connections for events. This work focuses on proving an abstract interpreter sound by showing that the individual interpretations of each layer (i.e., each source language feature) are sound in isolation, before composing these proofs together. The soundness at each layer captures that "identical" events should get interpreted into sound subprograms. However, most events have parameters, such as writing to a variable in Imp:

$$\texttt{Write}\colon \mathbb{X} \to \mathbb{V} \to \texttt{memE}\ \mathbb{1}.$$

Hence, the signature for the corresponding abstract event must be different:[7]

$$\texttt{Write}^\#\colon \mathbb{X} \to \mathbb{V}^\# \to \texttt{memE}^\#\ \mathbb{1}^\#.$$

[7]The return type could remain as $\mathbb{1}$, but uniformly using a lattice type Galois-connected to the original makes things more consistent.



Imp source

```
if x < 0 then
  y := add(x, 1);
else
  assert(x != 0);
```

Initial denotation

```
M  = itree (assertE
           +' memE
           +' arithE)
M# = aflow (assertE#
           +' memE#
           +' arithE#)
```

Interpreting failure

```
M  = failT (
      itree (memE
          +' arithE))
M# = failT# (
      aflow (memE#
          +' arithE#))
```

Interpreting memory

```
M  = failT (stateT mem (
        itree arithE))
M# = failT# (stateT mem# (
        aflow arithE#))
```

Interpreting arithmetic

```
M  = failT (stateT mem (
         itree ∅))
M# = failT# (stateT mem# (
         aflow ∅))
```

Combinator unfolding

```
M# = failT# (stateT mem# (
         itree ∅))
```

Figure 6. Overview of the denotation process for a simple Imp program.

Denotation from top to bottom. The concrete program is a standard ITree. The abstract program is an aflow *until combinators are unfolded and it gets compiled to an event-less ITree. In both columns, ○ represents the concrete/abstract sequence combinator, with initial computation on top and continuation(s) on the side and bottom. Concrete combinators are not materialized in the ITree but tracked propositionally by* sound'*.*

And so we need to relate events through a Galois connection, typically by matching arguments:

$$\texttt{Write}\ x\ v \in \texttt{Write}^{\#}\ y\ v^{\#} \ \triangleq\ x = y \wedge v \in v^{\#}$$

This is done for each individual event when defining the source language.

Syntactic soundness. Soundness of an abstract interpreter w.r.t. a concrete semantics expresses that the abstract value computed by the analyzer correctly over-approximates all possible concrete executions. This final statement ❺ is formalized as the `sound` predicate in Section 6.1. However, this notion cannot be used for partially-interpreted programs because it ignores events (and traces are not comparable due to differences in control flow unfolding).

We solve this issue by relying on the syntactic preservation of control flow combinators. We introduce an intermediate soundness predicate, dubbed `sound'`, which matches the control flow combinators of the concrete and abstract programs syntactically, while relating raw values and events through Galois connections at the leaves. Programs are initially related by `sound'` because $[\![\cdot]\!]$ and $[\![\cdot]\!]^{\#}$ mirror each other. ❻ Then, since combinators are syntactically preserved by interpretation (and handlers are sound), `sound'` can be maintained through each layer; this is formalized by a collection of `interp_sound_*T` theorems. ❼

Summary. Figure 6 summarizes the steps of the monadic interpretation process guided by these observations, for an example Imp program exhibiting effects from all three categories `assertE`, `memE`, and `arithE` introduced in Section 2.1.

Starting at the top, an Imp program is ascribed concrete and abstract semantics by $[\![\cdot]\!]$ and $[\![\cdot]\!]^{\#}$, with mirrored structures that use the concrete and abstract form of each value, event, and control flow combinator. See for instance, the matching *Concrete cond* ❽ and *Abstract cond* ❷ combinators, and the pair of identically-placed *concrete sequence* and *abstract sequence* (represented as $\circ$ for readability). At this *Initial denotation* stage, the programs are an `itree` and an `aflow`, with all events still in their symbolic form.

Each of the next three layers sees one of the event families get interpreted, switching the monads `M` and `M`$^{\#}$ by cutting the event signature and adding a new monad transformer. Handled events are replaced with pure computations, while keeping the flow structure because control flow combinators are syntactically preserved by interpretation. This fact combined at each layer with a proof of soundness of Imp's event handlers implies the preservation of `sound'`.

Finally, the combinators of the abstract program are unfolded into a proper ITree with no events left. This is when abstract interpretation components such as joins and post-fixpoint approximations are added to the program. Also at this *Combinator unfolding* stage, the proof of soundness carried by `sound'` is finally proven to imply `sound`, which ends up being a standard analysis of abstract interpretation algorithms, independent of the language features at hand.

```
Inductive aflow (E: Type → Type) (R: Type) :=
  | Ret (r: R)
  | Vis {T} (e: E T) (k: T → aflow E R)
  | Seq {U T: Type#}
      (f1: aflow E U) (f2: U → aflow E T) (early: U → bool)
      (post: U → T → T) (k: T → aflow E R)
  | Join {T: Type#}
      (fleft fright: aflow E T) (k: T → aflow E R)
  | Fixpoint {U T: Type#}
      (body: U → aflow E T) (step: T → U) (init: U)
      (k: T → aflow E R)
  | FixpointN (* ... *).
```

Figure 7. Definition of the `aflow` monad.

With this overview in mind, we now dive into the deeper details of implementing this structure in Coq.

4 Implementing the abstract interpreter

We now describe the programmatic side of the library in more details. We first introduce the `aflow` monad, in which the abstract interpreters are represented, and its effectful interpretations. We showcase the control flow combinators used to program both interpreters. Finally, we brush on the unfolding of these combinators into ITree implementations.

4.1 The `aflow` monad

The `aflow` monad is defined on Figure 7. We write **Type**$^{\#}$ for a dependent pair of a **Type** along with a `Lattice` instance. The monadic structure of `aflow E` is based on the `Ret` constructor, with a bind operation recursively propagated through each constructor's continuation `k: T → aflow E R`. We emphasize that this bind represents *a sequence in the abstract interpreter* as discussed in Section 3 (which does not carry monadic effects). The `Vis` constructor provides the freer monad structure and corresponds directly to the `Vis` constructor of `itree`.

The remaining constructors represent the control flow structures which have dedicated algorithms for abstract interpretation, and are used to build higher-level control flow combinators. `Seq` sequences two computations (which is not trivial when effects like failure are involved); it is used by the `sequence` combinator. `Join` joins the results of two computations; it is used by the `cond` combinator (which also adds a condition). `Fixpoint` computes a post-fixpoint of a loop body; it is used by the `do` and `while` combinators. Finally, `FixpointN` computes a post-fixpoint of a family of mutually-tail-recursive functions; it is used by the `cfg` combinator.

These constructors have a few unfamiliar parameters. Focusing on `Seq` as an illustrative example, we have two extra functions. The parameter `early: U → bool` indicates whether the initial computation in the sequence may have failed; and `post: bool → U → T → T` joins the intermediate value `u:U` and the final value `t:T` if failure may have occurred,

Monad	Description
`stateT s`	`stateT s m t = s → m (s * t)` Threads the global state `s` through computations; each step takes the current state as input and returns an updated one.
`failT`	`failT m t = m (option t)` Allows for an early exit (failure path) by returning `none`.
`stateT# s`	`stateT# s m t = s → m (s * t)` Same as `stateT`.
`failT#`	`failT# m t = m (unit# * t)` Adds an extra return value indicating whether the failure path might have been taken, with two possible values: ⊥ (not taken), ()# (maybe taken).

Figure 8. Summary of the monads in our implementation.

as indicated by a boolean parameter. We will shortly show how these functions capture the data-flow paths that arise as a result of interpreting into either the state or failure monads.

Overall, `aflow` will keep track of the source program's control flow structure through the interpretation process, until it can be unfolded into the appropriate abstract interpretation algorithms once all events have been interpreted.

4.2 Monadic interpretation in `aflow`

Monadic effects in abstract programs are quite different from their counterparts in concrete programs. For instance, `failT` allows a concrete Imp program to crash. It goes without saying that the corresponding abstract program will not itself crash. Instead, it will simply add crashing states to the set of potential final states with a lattice join. In general, monadic interpretation in the abstract world boils down to two things: 1. using richer lattice to model new effects (e.g., whether the failure path might have been taken), and 2. adding new data-flow paths (e.g., joining potential failure states with the final state).

We implement support for two effects: stateful and failing computations. The definitions for both concrete and abstract transformers are summarized in Figure 8. The latter are part of the analyzer's interface, so there isn't one single correct definition; for instance, `failT#` could separate the return values from the success and failure cases.

The interpreters for `stateT#` and `failT#` are shown on Figure 9. Unlike in ITree where a single `interp` function handles all monads, in `aflow` each monad transformer is unique. This is because they apply their monadic effect in each flow-related constructor by updating the extra functions (here, the `early` and `post` parameters in `Seq`—`Join` has none). Notice, crucially, how this enables all sub-programs and continuations to be interpreted transparently.

```
Fixpoint interp_state# (h: E ⤳ stateT S (aflow F))
    (f: aflow E R): stateT S (aflow F) R := fun s ⇒
  match f with
  | Ret r ⇒ Ret (s, r)
  | Vis e k ⇒ '(st,t) ← h e s;; interp_state# h (k t) st
  | Seq f1 f2 early post k ⇒
      Seq (interp_state# h f1 s)
          (fun '(su,u) ⇒ interp_state# h (f2 u) su)
          (fun '(_,u) ⇒ early u)
          (fun b '(su,u) '(st,t) ⇒
            (if b then su ⊔ st else st, post b u t))
          (fun '(st,t) ⇒ interp_state# h (k t) st)
  | Join fleft fright k ⇒
      Join (interp_state# h fleft s)
           (interp_state# h fright s)
           (fun '(st,t) ⇒ interp_state# h (k t) st)
  (* ... *)

Fixpoint interp_fail# (h: E ⤳ failT# (aflow F))
    (f: aflow E R) (err: unit#): failT# (aflow F) R :=
  match f with
  | Ret r ⇒ Ret (err, r)
  | Vis e k ⇒ '(et,t) ← h e;;
              interp_fail# h (k t) (err ⊔ et)
  | Seq f1 f2 early post k ⇒
      Seq (interp_fail# h f1 err)
          (fun '(eu,u) ⇒ interp_fail# h (f2 u) eu)
          (fun '(eu,u) ⇒ eu =? ()#|| early u)
          (fun b '(eu,u) '(et,t) ⇒
            (eu ⊔ et, post (eu =? ()#|| b) u t))
          (fun '(et,t) ⇒ interp_fail# h (k t) et)
  | Join fleft fright k ⇒
      Join (interp_fail# h fleft err)
           (interp_fail# h fright err)
           (fun '(et,t) ⇒ interp_fail# h (k t) et)
  (* ... *)
```

Figure 9. Equations for representative cases of the state and failure monadic interpreters for `aflow`.

In the state monad, an extra global state `s:S` is provided as input and returned along the output. `Vis` supplies it to the event handler, which allows state events to be substituted with pure computations. `Seq`'s extra functions are updated to indicate that state does not cause failure (`early` unchanged) but it is affected if a failure occurs elsewhere (`post` joins it in addition to the return values when `b = true`).

The failure monad follows a similar structure. Notice how in `Vis` there is no early exit between `h e` (the interpreted event) and the continuation: this is because bind in `aflow` is a sequence in the abstract interpreter. By contrast, an early exit *is* added in `Seq`, which is what makes it is more complex than a sequence in a classical analyzer. `early` is updated to announce potential failure if the error flag `eu` is ()#, and `post` propagates this information to other transformers by setting `b = true` in its call to the wrapped `post`.

Importantly, both interpretations of `Seq` and `Join` are other instances of `Seq` and `Join`. This stability under interpretation

```
Inductive SeqResult T R :=
  | SR_Continue (value: T)
  | SR_Fail (err: R).
Definition sequence (t: itree E U) (k: T → itree E R)
    (dist: U → SeqResult T R) :=
  ITree.bind t (fun result ⇒
    match dist result with
    | SR_Continue value ⇒ k value
    | SR_Fail err ⇒ ITree.ret err
    end).

Definition sequence# (f: aflow E T) (k: T → aflow E R)
    (early: T → bool) (post: bool → T → R → R) :=
  Seq f k early post Ret.
```

Figure 10. Concrete /abstract sequence combinators.

```
Fixpoint unfold (f: aflow ∅ R): itree ∅ R :=
  match f with
  | Ret r ⇒ ITree.ret r
  | Seq f1 f2 early post k ⇒
     u  ← unfold f1;;
     t  ← unfold (f2 u);;
     unfold (k (post (early u) u t))
  | Join fleft fright k ⇒
     rl  ← unfold fleft;;
     rr  ← unfold fright;;
     unfold (k (rl ⊔ rr));
  (* ... *)
```

Figure 11. Equations for unfolding Ret, Seq and Join.

is later lifted to flow combinators, a key fact for establishing the preservation of sound' by interpretation.

4.3 Implementing control flow combinators

Although the ITree library already provides standard combinators to write concrete interpreters such as the one of Section 2.1, maintaining the tight connection between the concrete and abstract interpreters at each layer, as illustrated on Figure 6, requires a precise control over the way the concrete semantics is defined. Our library hence provides control flow combinators in concrete/abstract pairs,[8] systematically providing (1) a proof of syntactic preservation by pure, stateful, and failure interpretation for each, and (2) a proof that the unfolded abstract combinator soundly approximates the concrete one. In practice, the first requirement means that combinators must express the most general form of a condition/loop/etc, accounting for all supported monadic effects.

Our library currently supports five combinators: sequence ; cond , a binary conditional branching; do , a do/while loop which also supports passing an accumulator value from each iteration to the next; while , a simple wrapper around do which showcases how combinators can build upon each other; and cfg , a control-flow graph structure with a variable number of basic blocks as arguments in the style of assembler, as a more advanced example.

Each combinator is adequately parametrized such that it can internalize pure, stateful and failure effects.[9] This informal statement is captured by establishing for each version of each combinator, and for each of the three interpretations of this combinator, an equation expressing the interpreted result in terms of the initial combinator. We illustrate the sequence combinator, and refer the interested reader to our formal development for the others.

Sequence. The sequence combinators are depicted on Figure 10. On the concrete side, the combinator essentially refines the ITree bind with a device allowing it to absorb state and failure transformers. Rather than immediately distributing the result of the first computation to the continuation, the function dist decides whether to halt midway in case of failure. When interpreting, dist absorbs the monadic effect: for instance, interpreting into failT turns U and T into option types, and dist is updated to map a None return from the first half to SR_Fail, thus taking care of the early exit.

The abstract version is straightforward since we took care of designing the Seq aflow combinator with sufficient parametrization to internalize the effects. It therefore wraps around the constructor with an empty continuation.

Other combinators. The concrete versions of the other combinators present no surprise to readers familiar with other works based on ITrees: cond relies on Coq's meta-level **if**, do and while on the iter combinator as illustrated in Section 2.1, and cfg on the way Xia et al. [39] for Asm or Zakowski et al. [40] for LLVM IR resolve calls in a CFG.

Similarly to what happens for sequence#, elementary versions of cond#, do#, and cfg# are direct wrappers around the corresponding aflow constructors. While it is not the focus on this paper, we illustrate in our library that more precise combinators, for instance variants of cond# taking the guard into account can of course be built. Improving the precision of a combinator in an existing verified analyzer written in our framework only requires to locally reestablish the preservation under the three interpreters, and the soundness after unfolding. Finally, while# illustrates a simple example of building a combinator on top of another (do#).

4.4 Unfolding flow combinators

Once interpretation is finished, flow combinators are unfolded by the unfold function, recursively mapping aflow computations with empty interfaces to itree ones. Figure 11 details Ret, Seq and Join: leaves are trivially translated, while Seq and Join are finally free to be threaded as simple binds. There are no longer any Vis at this stage, since there are

[8]As usual, we distinguish the abstract from the concrete version of each combinator by a superscript #.

[9]Except cfg which does not yet support failure in our implementation.

```
Definition sequence'# t k :=
  sequence# t k (fun _ ⇒ false) (fun b t r ⇒ r).
Definition while'# b c :=
  sequence# (while# (fun _ ⇒ b)
                    (fun _ ⇒ sequence'# c (fun _ ⇒ b))
                    (fun _ ⇒ ()#) ()#)
            (fun _ ⇒ ret ()#).

Fixpoint ⟦·⟧#_e (e: 𝔼): aflow impE# 𝕍#:= (* ... *).

Fixpoint ⟦·⟧# (s: ℂ): aflow impE# 𝟙#:=
  match s with
  | skip       ⇒ ret ()#
  | assert b   ⇒ sequence'# ⟦b⟧#_e (fun v ⇒ assert# v)
  | x := e     ⇒ sequence'# ⟦e⟧#_e (fun v ⇒ write# x v)
  | c1; c2     ⇒ sequence'# ⟦c1⟧# (fun _ ⇒ ⟦c2⟧#)
  | if b then t else e ⇒
      sequence'# ⟦b⟧#_e (fun v ⇒ cond# v ⟦t⟧# ⟦e⟧#)
  | while b do c  ⇒ while'# ⟦b⟧#_e ⟦c⟧#
  end.
```

Figure 12. Imp: representation of commands into aflow.

no events. The last two constructors unfold into fixpoint approximations—we omit them here.

The soundness of this unfolding process, including that of fixpoint approximation schemes, is established when proving the soundness for each combinator—independent of the particular features of the source language.

5 Case studies

Imp. We finally have all the tools in hand to write our abstract interpreter for the Imp language introduced in Section 2. Following the same canvas as in the concrete case, we first craft our event interface. Naturally, the events are identical to the concrete ones, with one difference: they operate over an abstract domain of values $\mathbb{V}^\#$ equipped with a lattice structure parametrizing the analysis.

```
Variant arithE# : Type → Type :=
  | Compute# (op : Op) (l r : 𝕍#) : arithE 𝕍#.
Variant memE# : Type → Type :=
  | Read#  (a: 𝕏)              : memE# 𝕍#
  | Write# (a: 𝕏) (v: 𝕍#)      : memE# 𝟙#.
Variant assertE# : Type → Type :=
  | Assert# (v : 𝕍#)           : assertE# 𝟙#.
```

Writing again impE# for (arithE# +' memE# +' assertE#), we write the analyzer in the aflow monad using the library's abstract combinators, as described in Section 4.3. Depicted on Figure 12, the code follows precisely the structure of its concrete counterpart, but relying on abstract combinators. We wrap the sequence combinator into a top-level one whose extra parameters are preset to embody the initial absence of failure nor global state. The wrapper for the while combinator is more complex, but follows a similar principle. In addition to setting up the arguments for the initial absence of effects, it further specializes it to the present case where neither the condition nor the body being iterated take arguments, hence carrying unit values around instead. Of course, these higher level combinators are meant to be directly exposed to the users, but we keep these parameters explicit here to stress that they are enriched during monadic interpretation.

Remains to code the three abstract handlers for impE#: they capture standard bits of abstract interpreters. The handler for arithmetic contains the corresponding transfer functions over the abstract domain of values considered. The memory handler boils down to the implementation of the abstract map over abstract values considered. Finally, handling asserts specifies how failure is treated in the abstract domain. As in the concrete case, hoist and the three (abstract) effectful interpreters achieve to define the abstract interpreter, eval#:

```
Definition eval# (s: ℂ): failT# (stateT# S# (aflow ∅)) 𝟙#:=
  hoist (fun u ⇒ hoist (interp_pure# h_arith#)
                       (interp_state# h_state# u))
        (interp_fail# h_fail# ⟦s⟧ bot).
```

We can then extract this interpreter into an OCaml program using Coq's extraction feature and run it as a standalone program. As a minimal example, consider the following Imp program:

```
x := 2; y := 0;
while x do { y := 1; x := sub(x, 1); }
z := 5; assert(y); z := 6;
```

The analyzer returns a final state indicating x $\in (-\infty, 2]$, y $\in \{0, 1\}$ and z $\in \{5, 6\}$. The lower bound on x is the direct result of widening after decrementing in the loop. The simple abstract condition we use does not notice the decidable condition in the first iteration, thus allowing y $= 0$. This causes the assert to be analyzed as potentially failing, so the final state (which might be at the assert) has either z $= 5$ or z $= 6$.

Asm. To illustrate the expressivity of our framework, we write an abstract interpreter for Asm, a toy control flow graph language featuring registers and memory. This language presents two layers of interpretation, both stateful. Its abstract aflow representation relies on the cfg# combinator, which computes a fixpoint over a vector of blocks. Both its definition and proof are very similar to that of Imp's.

6 Layered proof of soundness

We finally put our framework to use to certify the soundness of a monadic static analyzer w.r.t. a monadic concrete semantics built using the combinators our library provides.

The predicate sound captures the soundness of an abstract programs w.r.t. a concrete one whose interfaces are empty. It describes the traditional intuition that any value that the concrete program could return must be covered through the Galois connection by the abstract value returned by the unfolded abstract program:

```
sound (p : itree ∅ R) (p# : aflow ∅ R#) ≜
∀ r r#, p returns r →
        unfold p# returns r# →
        r ∈ r#,
```

where "p returns r" expresses that the computation terminates with value r.[10] Note that in case of computations obtained by the construction of monadic interpreters, the return types R and R# include at this stage global states and failure flags, so every feature of the source language is covered by this single statement.

The top-level theorem we establish then simply states that the interpreters are related by sound. For instance for Imp:

$$\forall (c : \mathbb{C})\ s\ s^\#,\ s \in s^\# \longrightarrow \texttt{sound}\ (\texttt{eval}\ c\ s)\ (\texttt{eval}^\#\ c\ s^\#).$$

In order to stress the reusability of our approach, we first describe the key results provided by the library, before highlighting the language- and analyzer-specific proof obligations remaining.

6.1 Generic meta-theory

As discussed in Section 3, most of the proof of soundness is conducted over a stronger notion of soundness, and is only lowered down to sound once all events have been interpreted. This notion is dubbed *syntactic soundness*, and is captured by a predicate

```
sound' (p : itree E R) (p' : aflow E# R#) : Prop
```

which asserts that p and p' have identical structure as flow combinator trees, where nodes carry combinators,[11] and leaves carry Galois-connected return values and events. Contrary to sound, it can relate computations with non-empty signatures: it is used to relate the analyzer starting at the first stage of representation.

Soundness preservation by interpretation. The library provides a battery of theorems expressing the preservation of sound' by interpretation by pure, stateful, and failure interpretations. The proof of each such interp_sound_⋆T lemma relies on the syntactic preservation of the aflow combinators (provided in the library), and assumes a language-specific proof of soundness of the event handler used for the interpretation. For instance, the preservation theorem for the state monad is stated as

```
Lemma interp_sound_stateT
  (h: E ⇝ stateT S (itree F))
  (h#: E# ⇝ stateT S# (aflow F#))
  (t: itree E R) (f#: aflow E# R#) s s#:
  s ∈ s# →
  handler_sound_stateT h h# →
  sound' t f# →
  sound' (interp_state h t s) (interp_state# h# f# s#).
```

[10] The signatures being empty, there is at most one such leaf.

[11] Recall from Section 4.3 that the library provides each combinator in a concrete/abstract pair.

where handler_sound_stateT h h# asserts that the interpretations of Galois-connected events by h and h# are related by sound'.

From syntactic to semantic soundness. Each pair of combinators provided by the library is proven to be sound: intuitively, given semantically sound inputs, they lead to semantically sound computations. These individual lemmas capture the soundness of the abstract interpretation algorithms that flow combinators unfold into. Given the high degree of parametrization of the combinators, these statements are slightly intricate to state, but strictly follow this intuition. For instance, the case of sequence is

```
Lemma sound_seq
  (dist: U → SeqResult T R) (k: T → itree E R)
  (f#: aflow E# T#) (k#: T# → aflow E# R#)
  (early#: T# → bool) (post#: bool → T# → R# → R#)
  (Hsound_tf: sound t f#)
  (Hsound_k: ∀t t#, t ∈ t# → sound (k t) (k# t#))
  (Hdist: ∀u t# r#, u ∈ t# →
    match dist u with
    | SR_Continue t ⇒ t ∈ t#
    | SR_Fail r ⇒ early# t#=true ∧ r ∈ post# true t# r#)
    end)
  (Hpost: ∀b t# r#, r# ⊆ post# b t# r#):
    sound (sequence t dist k)
          (sequence# f# k# early# post#).
```

Its hypotheses (which we do not detail here) are either about the soundness of its sub-programs, or formalize the requirements for the extra functions (dist, early and post), which are carried by sound' along with the syntactic correspondence of combinators.

These individual combinator theorems culminate in the library-provided sound_unfold theorem:

```
Lemma sound_unfold : ∀ (p : itree ∅ R) (p# : aflow ∅ R#),
                     sound' p p# → sound p p#,
```

which allows to conclude a formal proof that the abstract program safely approximates its concrete original.

6.2 User-specific proof obligations: the case of Imp

All control flow combinators needed to evaluate Imp are provided by the library. We plug in a simple interval domain, provided by the library as well. The remaining proof effort is hence minimal. Following Figure 6 from top to bottom, the proof is built in three pieces.

First, we establish the soundness of the representations, i.e. ∀ c, sound' ⟦c⟧ ⟦c⟧#. This proof is entirely mechanical, by induction on c, each case reducing to the definition of sound': it simply captures the structural similarity between both interpreters.

We then transport this syntactic soundness through the three layers of interpretation. In each, the corresponding sound_interp_⋆ lemma is provided by the library, though it still expects us to prove that each pair of handlers is sound: for assertE, by soundness of num_isfalse; for memE, based on properties of the map data structures used to associate variables with concrete and abstract values; for `arithE`, by soundness of the transfer functions over intervals. By chaining these proofs, we obtain the syntactic soundness of the whole abstract interpreter.

Finally, since all events have been interpreted away, we derive the semantic soundness of the interpreter 🐓 by application of `sound_unfold`.

6.3 Extending the library

While the combinators provided are generic and expressive enough to cover a wide range of applications, realistic languages will call for new combinators. We sketch the process of extending the library itself to support more constructions.

While it has not been the focus of this work, new non-relational abstract domains can be added by instantiating the `Lattice` class and a relevant domain class such as `NumericalDomain`.

Adding a new control flow structure requires to craft its monad-generic form as a new pair of combinators, possibly building upon existing ones. This process should usually not require to extend `aflow`—but if needed, the unfolding of the new constructor must be additionally defined. The new combinators must be proved to be preserved by each interpreter, and to be semantically sound one with another. Finally, the syntactic soundness must be extended with a new constructor in `sound'` capturing the new pair, and the new case in `sound_unfold` must be discharged.

Adding support for a new effect is naturally more transversal. A new monad transformer would require extending all control flow structures to ensure they can internalize the new monadic effect; this is the most challenging extension. Once this design question is resolved, each combinator must be proved to preserve the new interpreter—which is typically straightforward.

7 Related Work

The seminal paper by Cousot and Cousot [7] has spawned an exceptionally rich literature around the abstract interpretation framework. We refer the interested reader to recent introductory books [6, 33], and focus on works directly related to the peculiarities of our approach: mechanization and modularity.

Mechanized abstract interpreters. The first attempt at mechanizing abstract interpretation in type theory is probably due to Monniaux [28]. Later on, Pichardie identified during his PhD [29] that the asymmetric γ-only formulation of the framework was the key to alleviating issues with the non-constructivity of the abstraction function encountered in Monniaux's approach. We inherit from this design.

The approach eventually culminated in the Verasco [16] static analyzer: a verified abstract interpreter for the C language combining rich abstract domains to attain an expressiveness sufficient for establishing the absence of undefined behavior in realistic programs. In particular, the analyzer is plugged into CompCert [24] in order to discharge the precondition to its correctness theorem. Verasco supports a notion of modularity essentially orthogonal to the one we propose in the present work: they introduce a system of inter-domain communication based on channels inspired by `Astrée` [8]. Extending our work to support such complex abstract domain combinations and scaling from toy languages to realistic analyzers like Verasco is naturally a major perspective. In contrast, we emphasize that Verasco offers none of the core contributions we propose in our approach: no code reuse, no modularity in terms of effects, and a fuel-based analyzer to avoid having to prove the termination of the analyzer.

Skeletal semantics [2] have been leveraged to derive abstract interpreters in a modular fashion that shares commonalities with our approach. Skeletons and their interpretations provide a reusable meta-language in which to code the concrete and abstract semantics of the languages in a similar way we exploit ITrees and `aflow` with handlers. Despite this superficial similarity, the technical implementations are completely different: in-depth comparison of the two approaches would cause for a fruitful avenue.

Restricting ourselves to γ-only formulations sacrifices part of the abstract interpretation theory: the so-called "computational" style, deriving an abstract interpreter correct by construction from a concrete one. Darais and Van Horn have introduced Constructive Galois Connections [11, 12] to tackle this issue, and formalized their work in Agda.

Big-step abstract interpreters. A wide body of work has sought to modularize and improve code reuse in the design and verification of abstract interpreters. Most of them share conceptually with our work the use of a monadic encoding relying on uninterpreted symbols that gets refined in alternate ways. Bodin et al. [2], previously mentioned, falls into this category, but numerous other non-mechanized contributions have been done in this realm.

Most notably, Darais et al. [9] adapt Van Horn and Might's so-called *Abstracting Abstract Machine* [35, 38] methodology to build abstract interpreters for higher order languages using definitional interpreters written in a monadic style, rather than low level machines. Written in a general purpose functional language, their approach relies on a representation of the program with open recursion and uninterpreted operations, further refined into concrete, collecting and abstract semantics. In order to ease the construction of such monadic interpreters, Darais et al. have also identified so-called *Galois Transformers* [10], well behaved monad transformers that transport Galois connections and mappings to suitable executable transition systems.

Keidel et al. [18, 20] have proposed a framework for modularizing the concrete and abstract semantics based on *arrows* [15], a generalization of monads. Arrows roughly play the role of Skeletons in [2], and of the combination of concrete signatures and `aflow` in ours. The connection between these abstractions would deserve a more thorough analysis.

Recently, Keidel et al. have considered the modular construction of fix-point algorithms for big-step abstract interpreters [19]. This endeavor is orthogonal to our contributions and could hopefully be formalized and incorporated.

References

[1] Peter Aczel, Jiří Adámek, Stefan Milius, and Jiří Velebil. 2003. Infinite trees and completely iterative theories: a coalgebraic view. *Theoretical Computer Science* 300, 1 (2003), 1–45. https://doi.org/10.1016/S0304-3975(02)00728-4

[2] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

[3] Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* Volume 1, Issue 2 (July 2005). https://doi.org/10.2168/LMCS-1(2:1)2005

[4] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL (2023), 1770–1800. https://doi.org/10.1145/3571254

[5] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. 2013. A Survey on Product Operators in Abstract Interpretation. *Electronic Proceedings in Theoretical Computer Science* 129 (09 2013). https://doi.org/10.4204/EPTCS.129.19

[6] Patrick Cousot. 2021. *Principles of Abstract Interpretation.* The MIT Press.

[7] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Principles of programming languages (POPL)*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[8] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4435)*, Mitsu Okada and Ichiro Satoh (Eds.). Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

[9] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (aug 2017), 25 pages. https://doi.org/10.1145/3110256

[10] David Darais, Matthew Might, and David Van Horn. 2015. 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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 552–571. https://doi.org/10.1145/2814270.2814308

[11] David Darais and David Van Horn. 2016. Constructive Galois Connections: Taming the Galois Connection Framework for Mechanized Metatheory. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 311–324. https://doi.org/10.1145/2951913.2951934

[12] David Charles Darais. 2017. *Mechanizing Abstract Interpretation.* Ph. D. Dissertation. University of Maryland, College Park, MD, USA. https://doi.org/10.13016/M2J96097D

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669. https://doi.org/10.5555/3026877.3026928

[14] Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*. Springer-Verlag, Berlin, Heidelberg, 317–331.

[15] John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37 (05 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[16] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 247–259. https://doi.org/10.1145/2676726.2676966

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[18] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages* 3 (10 2019), 1–28. https://doi.org/10.1145/3360602

[19] Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. 2023. Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. *Proc. ACM Program. Lang.* 7, ICFP, Article 221 (aug 2023), 27 pages. https://doi.org/10.1145/3607863

[20] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. *Proc. ACM Program. Lang.* 2, ICFP, Article 72 (jul 2018), 26 pages. https://doi.org/10.1145/3236767

[21] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (aug 2015), 94–105. https://doi.org/10.1145/2887747.2804319

[22] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

[23] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[24] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[25] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–31. https://doi.org/10.1145/3527324

[26] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *FM 2018 - 22nd International Symposium on Formal Methods (LNCS, Vol. 10951)*. Springer, Oxford, United Kingdom, 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

[27] Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 257–275.

[28] David Monniaux. 1998. *Réalisation mécanisée d'interpréteurs abstraits.* Master's thesis. Université Paris 7.

[29] David Pichardie. 2005. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés.* Ph. D. Dissertation. Université Rennes 1. In french.

[30] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations.* Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[31] Maciej Piróg and Jeremy Gibbons. 2014. The coinductive resumption monad. *Electronic notes in theoretical computer science* 308 (2014), 273–288.

[32] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281.

[33] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective.* The MIT Press.

[34] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[35] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. *ACM SIGPLAN Notices* 48, 399–410. https://doi.org/10.1145/2491956.2491979

[36] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees (Artifact). *Dagstuhl Artifacts Ser.* 9, 2 (2023), 06:1–06:2. https://doi.org/10.4230/DARTS.9.2.6

[37] Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 4 (2008), 423–436.

[38] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. *SIGPLAN Not.* 45, 9 (sep 2010), 51–62. https://doi.org/10.1145/1932681.1863553

[39] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119

[40] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[41] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

[42] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (jan 2012), 427–440. https://doi.org/10.1145/2103621.2103709