Abstract Interpreters: a Monadic Approach to Modular Verification

SÉBASTIEN MICHELLAND, ENS de Lyon, France
YANNICK ZAKOWSKI, Inria, France
LAURE GONNORD, LCIS (UGA, Grenoble INP) and LIP (EnsL, UCBL, CNRS, Inria), France

We argue that monadic interpreters built as layers of handlers stacked atop the free monad, as advocated notably by the ITree library, also 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 both code reuse across projects and 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 into a Coq library, and demonstrate their benefits by implementing and proving correct two illustrative abstract interpreters respectively 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 [33]. 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 [14, 18, 24, 25].

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, and its transitive closure describes executions. While extremely successful, such approaches have drawbacks. On the practical side, these semantics are non-executable at their core, hence requiring 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 [35] 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

Authors' addresses: Sébastien Michelland, ENS de Lyon, France, sebastien.michelland@ens-lyon.fr; Yannick Zakowski, Inria, France, yannick.zakowski@inria.fr; Laure Gonnord, LCIS (UGA, Grenoble INP) and LIP (EnsL, UCBL, CNRS, Inria), France, laure.gonnord@lcis.grenoble-inp.fr.

2024. ACM XXXX-XXXX/2024/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

practical programming paradigm for decades, but have also become increasingly popular in the mechanized realm. In particular, *free monads* [38] 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 [27], as *I/O-trees* [15], and as McBride's *general monad* [28].

In this paper, we focus on *interaction trees* [40] (ITrees), a recent realization of this approach as a Coq library. ITrees are defined as a coinductive variant of the *freer monad* [22] and are also closely related to *resumption monads* [32]. 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 [23, 42], transactional objects [26], concurrency [5], or non-interference [37]. Their largest application is arguably the Vellvm project [41, 43], providing a compositional, modular and executable semantics for a large sequential subset of LLVM's intermediate representation. This application leverages the approach's 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 static analyses against ITree-based formal semantics and proving their soundness. We place ourselves more specifically in the *abstract interpretation* framework [7, 8]. Abstract interpretation is well known for providing rich ways of combining abstractions, through products [6] or communication-based protocols [9, 17]. 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, 10, 19, 21] in seeking to provide rich reusable combinators to lighten the construction of *verified* abstract interpreters.

Our contributions can be crystallized as follows:

- we demonstrate that the paradigm of defining languages as layered monadic interpreters can be used to simultaneously define a concrete and an abstract interpreter from a single denotation of the source language;
- we identify that parameterizing abstract interpretation algorithms over the concrete language's monadic effects is key to this construction, and formalize a monad of abstract control flow `aflow` capturing this idea;
- we express a notion of soundness applicable to layered monadic interpreters at intermediate stages of their definition, which enables certification from composing independent soundness proofs for each language feature;
- we provide a Coq implementation of the scheme as a reusable library, featuring a number of pre-certified abstract interpretation algorithms;
- finally, we showcase this work by modularly certifying abstract interpreters for a standard Imp language and a toy assembly language.

Our Coq development with meta-theoretical results and the library is available as open-source software.

Section 2 starts by providing necessary background on ITrees and abstract interpretation. Section 3 presents an overview of the library through the motivating example of a small Imp language. Section 4 illustrates the challenges and motivates our design, whose programmatic component is described in more detail in Section 5. Finally, Section 6 provides details on the meta-theory, and the process of certifying an abstract interpreter from the perspective of a user of our library. We conclude with related work and a quick summary.

```
(* Embedding of pure computations *)
  ITree.ret (r: R): itree E R := Ret r.
(* Sequencing computations *)
  ITree.bind (u: itree E T) (k: T → itree E U): itree E U.
(* Atomic execution of an event. *)
  ITree.trigger: E ⇝ itree E := fun e ⇒ Vis e Ret.
(* Fixed-point combinator *)
  ITree.iter (body: I → itree E (I+R)): I → itree E R.
```

Fig. 1. ITrees: type signature of the main combinators

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`**. To clear any accidental confusion, we systematically reference the accompanying code 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.

2.1 Interaction Trees and Monadic Interpreters

Interaction Trees [40] (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, return values of type `R`. `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 [4].

One may think of ITrees as a low level imperative programming language embedded inside 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

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

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

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

Fig. 2. Imp: abstract syntax

iter combinator. Operationally, iter f i is the computation repeatedly performing f i, each time checking whether the result 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 [31] whose abstract syntax is depicted on Figure 2. Arithmetic expressions contain variables in $\mathbb{X}$, literals in $\mathbb{V}$ 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 nonzero and failure otherwise.

We model Imp's dynamic semantics using ITrees in Figure 3. The process, already illustrated in [40], and at scale notably in Vellvm [41], is split into two main phases.

Denotation. We first want to denote the syntax into an interaction tree. At this stage, events are still symbolic (one can think of them as the labels in a labeled transition system modeling the program). The "Event signature" section of Figure 3 shows the three features for which we choose to use events in Imp:[3]

- Arithmetic (arithE) for binary arithmetic computations. The Compute event takes the parameters to the operation and returns a value.
- Memory access (memE) for reading and writing variables. Read x returns a value while Write x v returns unit. Both will access the global memory state as a side-effect once implemented.
- Assertions (assertE). Assert v evaluates that v is non-zero and returns unit. It causes the program to fail and abort as a side-effect if v is zero.

We call E the disjoint sum of these events, which forms the full event signature of Imp.

We can then move on to the denotation functions, defined by recursion on the syntax tree. The code is mostly straightforward for expressions, with only binary computations requiring multiple steps, and closely resembles a monadic interpreter as one would write e.g. in Haskell. Statements are more of the same, but notice the use of *control-flow combinators* cond and while to implement conditionals and loops. The ternary cond combinator simply desugars to a Coq-level **if** construct. The only 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 ()))
```

We emphasize this notion of control flow combinators because they will later be key to the definition and proof of abstract interpreters.

Handling. 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.

[3]ITrees also hardcode divergence in the structure, making it available at all times.

```
Event signature
Variant arithE: Type → Type :=
  | Compute (op: Op) (l r: 𝕍) : arithE 𝕍.
Variant memE: Type → Type :=
  | Read (x: 𝕏)               : memE 𝕍
  | Write (x: 𝕏) (v: 𝕍)       : memE 𝟙.
Variant assertE: Type → Type :=
  | Assert (v: 𝕍)             : assertE 𝟙.
Definition E := assertE +' memE +' arithE.

Denotation as ITree
Fixpoint ⟦·⟧e (e: 𝔼): itree E 𝕍:=
  match e with
  | (x: 𝕏)   ⇒ ITree.trigger (Read x)
  | (v: 𝕍)   ⇒ ITree.ret v
  | e1 op e2 ⇒ v1 ← ⟦e1⟧e;; v2 ← ⟦e2⟧e;; ITree.trigger (Compute op v1 v2)
  end.
Fixpoint ⟦·⟧ (s: ℂ): itree E 𝟙:=
  match s with
  | skip                  ⇒ ITree.ret ()
  | assert e              ⇒ v ← ⟦e⟧e;; ITree.trigger (Assert v)
  | x := e                ⇒ v ← ⟦e⟧e;; ITree.trigger (Write x v)
  | s1; s2                ⇒ _ ← ⟦s1⟧;; ⟦s2⟧
  | if e then s1 else s2  ⇒ v ← ⟦e⟧e;; cond (v =? 0) ⟦s1⟧ ⟦s2⟧
  | while e do s          ⇒ while ⟦e⟧e ⟦s⟧
  end.

Event handlers
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 {S: Type} `{Monad M}: memE ⇝ stateT S M :=
  fun e s ⇒ match e with
  | Read x    ⇒ ret (s, mem_get s x)
  | Write x v ⇒ ret (mem_store s x v, tt)
  end.

Handling
Definition eval (s: ℂ): failT (stateT S (itree ∅)) 𝟙 :=
  hoist (fun u ⇒ hoist (handle_pure h_arith)
                       (handle_state h_mem u))
        (handle_fail h_assert ⟦s⟧).
```

Fig. 3. Imp: Building an ITree-based concrete interpreter

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 state `s: S` providing `mem_store` and `mem_get` operations (the latter returning `0` for undefined values). Finally, asserts may fail, hence `h_assert` introduces failure via the usual `failT` transformer. These additions of monadic transformers enable the implementation of events' side-effects as pure computations.

We are finally ready to define our concrete interpreter for Imp, `eval`, by successively handling all three layers of effects. Each handling removes an event family from the signature and adds a

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

Fig. 4. Common lattice operations and numerical domain for Imp. V and $V^\#$ represent concrete and abstract values. The `measure` order is lexicographic for both measure axioms.

monad transformer. The resulting semantic domain is hence `failT (stateT S (itree ∅))` (where ∅ is the empty signature), i.e. a stateful computation that may fail (or diverge).

Getting there requires two final ingredients. First, the `hoist` monadic combinator lifts a monad morphism `f: m ⇝ n` under a transformer `t`, giving us `hoist f: t m ⇝ t n`. This allows us to chain our monad transformations. Second, ITrees' `handle` function[4] applies an event handler to a program:

`handle (h: E ⇝ M): itree E ⇝ M.`

In this paper, `M` will always be `monadT (itree F)` for some monad transformer `monadT` and a smaller event signature `F`. For clarity, we write `handle_state`, `handle_fail`, etc. to recall the monad transformer being applied with each use of the function. `handle` substitutes events with the specified handler's implementation and applies the monadic transformer transparently. Putting all the ingredients together, we get the `eval` function from the end of Figure 3, which is a concrete interpreter for Imp.

When evaluated on an Imp program and an initial state, `eval` returns an `option (S * 𝟙)`, i.e. the final state and return value, if the program doesn't fail. We can obtain an executable interpreter by using Coq's extraction feature.

2.2 Abstract Interpretation

Abstract interpretation [8] 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 $\text{Interval} = (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\})$.

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

[4]The function is called `interp` in the ITree library. We reserve the words interpreter and interpretation for the resulting concrete or abstract executable process through this paper to avoid any confusion.

concretization functions (α, γ) forming a Galois connection is expected to relate the abstract domain to the concrete one, although we follow Pichardie [30]'s γ-only encoding as summed up in Figure 4.

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. It implies the common widening insurance of "non-infinite increasing sequence of abstract values" [34].

Abstract interpretation has mainly two implementation variants; we elect to expose here in pseudo-code "big-steps abstract interpretation", which "mimics" the big steps operational semantics of a language and replaces each concrete operation with its abstract counterpart. As for Imp, abstract interpretation with the Interval abstract domain computes and propagate an Interval for each variable (in $\mathbb{X}$) of the program.

Assignment `x := e` simply performs an explicit update for x in the memory $m^\#$:

$$[\![\texttt{x := e}]\!]^\#(m^\#) = v^\# \leftarrow [\![e]\!]^\# \\ \texttt{return}\; m^\#[x := v^\#].$$

Abstract skip is a no-op, abstract sequence is also a sequence in the interpreter. Conditionals must run both branches independently (from the same initial memory) and join the results (here not showing the case of decidable conditions):[5]

$$[\![\texttt{if e then s}_1\;\texttt{else s}_2]\!]^\#(m^\#) = m_1^\# \leftarrow [\![\texttt{s}_1]\!]^\#(m^\#) \\ m_2^\# \leftarrow [\![\texttt{s}_2]\!]^\#(m^\#) \\ \texttt{return}\; (m_1^\# \sqcup m_2^\#).$$

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

$$[\![\texttt{while e do s}]\!]^\#(m^\#) = \text{repeat}\; m_1^\# \leftarrow [\![\texttt{s}]\!]^\#(m^\#) \\ m_2^\# \leftarrow \texttt{widen}\; m^\#\; (m^\# \sqcup m_1^\#) \\ \text{if}\; (m_2^\# \subseteq m^\#) \\ \quad \texttt{return}\; m^\# \\ m^\# \leftarrow m_2^\#.$$

That is, $[\![\texttt{while e do s}]\!]^\#(m^\#)$ 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 specific algorithms), the abstract interpretation framework [8] 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 intractable) concrete semantics.

We aim to fit this framework into a layered, monadic setting, in the style of Section 2.1. Looking back, we can superficially notice that we would like to be able to see assignment as an effect, but also that the control flow of the abstract interpreter differs vastly from the concrete one: in particular, the required independent execution of branches seems at first incompatible with the naive threading of state through binds. The ITrees toolkit seems to fall short: we illustrate how to nonetheless recover similar techniques.

3 A TASTE OF OUR LIBRARY

Before getting into technical challenges and our solutions, let us illustrate the end result of using our library to simultaneously define a concrete and abstract interpreters for Imp (). The

[5]We ignore guards in conditions and loops for now because the expression language will be user-supplied and inverting arbitrary expressions is beyond the scope of this contribution.

User code

```
Abstract events
Definition 𝕍#:= Interval. (* intervals of Z, provided by library *)
Variant arithE#: Type → Type :=
  | Compute# (op: Op) (l r: 𝕍#) : arithE 𝕍#.
Variant memE#: Type → Type :=
  | Read# (x: 𝕏)                : memE# 𝕍#
  | Write# (x: 𝕏) (v: 𝕍#)       : memE# 𝟙#.
Variant assertE#: Type → Type :=
  | Assert# (v: 𝕍#)             : assertE# 𝟙#.
Definition E# := assertE# +' memE# +' arithE#.

Handlers
(* h_arith# and h_mem# are identical to the concrete case, they just use abstract values *)
Definition h_assert# `{Monad M} : assertE# ⇝ failT# M := fun '(Assert v) ⇒
  ret (if isfalse v then (⊤, ⊥) else                         (* statically failing *)
       if istrue v then (⊥, ⊤) else                          (* statically passing *)
       (⊤, ⊤)).                                                        (* unknown *)

Semantics as SurfaceAST
Fixpoint ⟦·⟧e {b: bool} (e: 𝔼): SurfaceAST E E# 𝕍 𝕍# b :=
  match e with
  | (x: 𝕏)   ⇒ do (Read v) and (Read# v)
  | (n: nat) ⇒ ret (num_nat n) and (num_nat n)
  | e1 op e2 ⇒ v1 ← ⟦e1⟧e;; v2 ← ⟦e2⟧e;; do (Compute op) and (Compute# op) on v1 and v2
  end.
Fixpoint ⟦·⟧ {b: bool} (s: ℂ): SurfaceAST E E# 𝟙 𝟙# b :=
  match s with
  | skip                ⇒ ret tt and tt#
  | x := e              ⇒ v ← ⟦e⟧e;; do (Write x) and (Write# x) on v
  | s1; s2              ⇒ _ ← ⟦s1⟧;; ⟦s2⟧
  | if e then s1 else s2 ⇒ v ← ⟦e⟧e;; AST_If v ⟦s1⟧ ⟦s2⟧
  | while e do s        ⇒ AST_While_unit ⟦e⟧e ⟦s⟧
  | assert e            ⇒ v ← ⟦e⟧e;; do (Assert) and (Assert#) on v
  end.

Handling
Definition imp_interp (s: ℂ): failT (stateT S (itree ∅)) 𝟙:=
  hoist (fun u ⇒ hoist (handle_pure h_arith)
                       (handle_state h_state u))
        (handle_fail h_fail (ast2itree ⟦s⟧#)).

Definition eval# (s: ℂ): failT# (stateT# S# (aflow ∅)) 𝟙#:=
  hoist (fun u ⇒ hoist (handle_pure# h_arith#)
                       (handle_state# h_state# ⊤))
        (handle_fail# h_assert# (ast2aflow ⟦s⟧)).
Definition imp_interp# := unfold ∘ eval#.
```

Fig. 5. Imp: Deriving both interpreters from a dual concrete/abstract denotation (🐓).

code for this, given in Figure 5, is similar to Section 2.1 but derives both interpreters from a single representation.

Having previously defined concrete events and their handlers, we now define matching abstract events and their handlers. Some of these definitions are shared; for instance, the memory and arithmetic handlers are parameterized on a map data structure and a numerical type respectively, and their abstract implementation is identical to their concrete one. For `h_assert#` however, the implementation differs because the abstract failure monad transformer `failT#` is different from the standard `failT`: instead of adding failure to the analyzer, it turns return values into pairs of an error value and a normal value. The *error value* indicates whether the failure path might have been taken, while the *normal value* approximates the concrete program's return value in non-failing cases—we come back to `failT`'s definition in Section 5.1. The `h_assert#` handler attempts to decide the assertion's condition using functions from the `NumericalDomain` class (which is implemented by $\mathbb{V}^\#$ but not shown here) before returning such a pair.

We then proceed to denote Imp again, except this time we produce an object of the library-provided type `SurfaceAST E E# 𝟙 𝟙# b`,[6] whose definition we delay until Section 5.2. Elements of this datatype can be thought of as mixed representations that can be projected into either a concrete or an abstract denotation depending on the value of `b`. This duality is most apparent in the `ret-and` and `do-and` (event) statements, where both the concrete and the abstract values or events are provided.

As with the denotation from Section 2.1, control flow in this representation relies on predefined *combinators*, here `AST_if` and `AST_while_unit`. The name `SurfaceAST` comes from the fact that these combinators are still arranged in a syntax tree at this stage. A key insight of this work is that while control flow combinators can be unfolded into their ITree-based implementation immediately for the *concrete* program, we must keep them symbolic throughout event handling for the *abstract* program.

The construction of the final concrete evaluator for Imp differs from Section 2.1 only in that we go through this intermediate representation ⟦s⟧ before recovering the original ITree via the generic `ast2itree` ( ) function.

```
ast2itree: SurfaceAST E E# R R# false ⟶ itree E R.
```

Naturally, there is another side to this coin; we can now as easily extract an abstract program, which is represented into another monad, dubbed `aflow` ( ):

```
ast2aflow: SurfaceAST E E# R R# true ⟶ aflow E R.
```

This new structure, a monad for abstract-interpretation control flow, will be thoroughly motivated through Section 4 and formally defined in Section 5.1. We shall show (also in Section 5.1) that it supports its own notion of event handling, allowing us to mirror the handling process of the concrete interpreter. An executable abstract interpreter in the form of an ITree is obtained by eventually unfolding control flow structures after handling abstract events (`unfold`, defined in Section 5.1).

Proving sound the abstract interpreter. All in all, we provide tools to define concrete and abstract interpreters simultaneously in Coq as layered monadic interpreters. This is not all, however—we also prove the abstract interpreter's soundness! For pairs of denotations derived from a shared `SurfaceAST`, the library proves the soundness of control flow structures and allows users to derive the soundness of the abstract interpreter with minimal obligations. These are, specifically: values returned in `ret-and` and events emitted in `do-and` should be related by the appropriate Galois connection; and (more importantly) abstract event handlers should be sound w.r.t. their concrete counterparts. Section 6 is dedicated to making this claim precise and substantiated.

Running the abstract interpreter. Since ITrees are executable, we can extract the proven sound abstract interpreter into an OCaml program using Coq's extraction feature[7] 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;
```

[6] $\mathbb{1}^\#$ is the two-element unit lattice.

[7] Naturally, we can also extract the concrete interpreter, as is usual.

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 $z = 6$.

Another case study: Asm (🐓). To illustrate the expressivity of our framework, we also write and prove correct an abstract interpreter for Asm, a toy control flow graph language featuring registers and memory. This language presents two layers of handling into the state monad and is centered around a CFG control flow structure (also provided by the library). Both its definition and proof are very similar to that of Imp's, and in fact the theorems for the soundness of memory handling are shared.

4 DESIGN OF A LAYERED ABSTRACT INTERPRETER

We now discuss the theoretical ideas that enable the construction of a layered abstract interpreter in Figure 5 and its modular proof of soundness. This section focuses on how the integration of monadic event handling influences the design of the abstract interpreter. We build up to Figure 6, which provides a bird's-eye view of the dual concrete/abstract denotation process, and Figure 7, which explains control flow structures that are key to our support for event handling. These figures will underpin our presentation of the Coq implementation and proof in Sections 5 and 6.

4.1 Layered handling and intermediate proofs of soundness

Through Section 3, we have focused on embedding Imp into our surface language, and concluded by summarily applying a series of event handlers on the resulting monadic denotation. Figure 6 shows the details of this handling process, including a preview of the modular proof mechanism.

One thing to keep in mind is that our abstract denotation (Figure 6, ❶) is a *hybrid* between a given source program and a traditional abstract interpreter: it is akin to an abstract interpreter partially evaluated on a chosen input program. In particular, it will use algorithms like joins and fixpoint approximations in lieu of conditions and loops. So while it mirrors the structure of the concrete program, *there isn't a one-to-one match* between the computations performed by these two programs. There is only a one-to-one match between their trees of control flow combinators, which is why we shall reflect this tree into a data-structure in order to guide the event handling and proof.

The Imp program in Figure 6 features three control flow constructions: a conditional test through `if`, and two slightly-hidden sequence points: one after reading `x` for the condition, and one between the evaluation of `add(x,1)` and the assignment to `y`. These are materialized in the concrete and abstract program as a box for the conditional[8] (❷ and ❸) and as a circle ○ for sequence points (e.g. ❹).

The source program also exhibits events from all three families `assertE`, `memE`, and `arithE` introduced in Section 2.1. Each family is handled in turn by one of the *"Handling"* layers, where a monad transformer is applied to supply the features required to implement its events. These events disappear each time from the representation as they are substituted with pure computations. In terms of typing, each handler trades part of the event signature for a monad transformer (except for `arithE` which is pure). This continues until there are no more events left, at which point we "unfold" the abstract interpreter into an executable form ❺, by implementing the control flow nodes into the ITree monad.

[8]The concrete box is dotted because it is tracked propositionally, through `sound'`, instead of being part of the data structure.

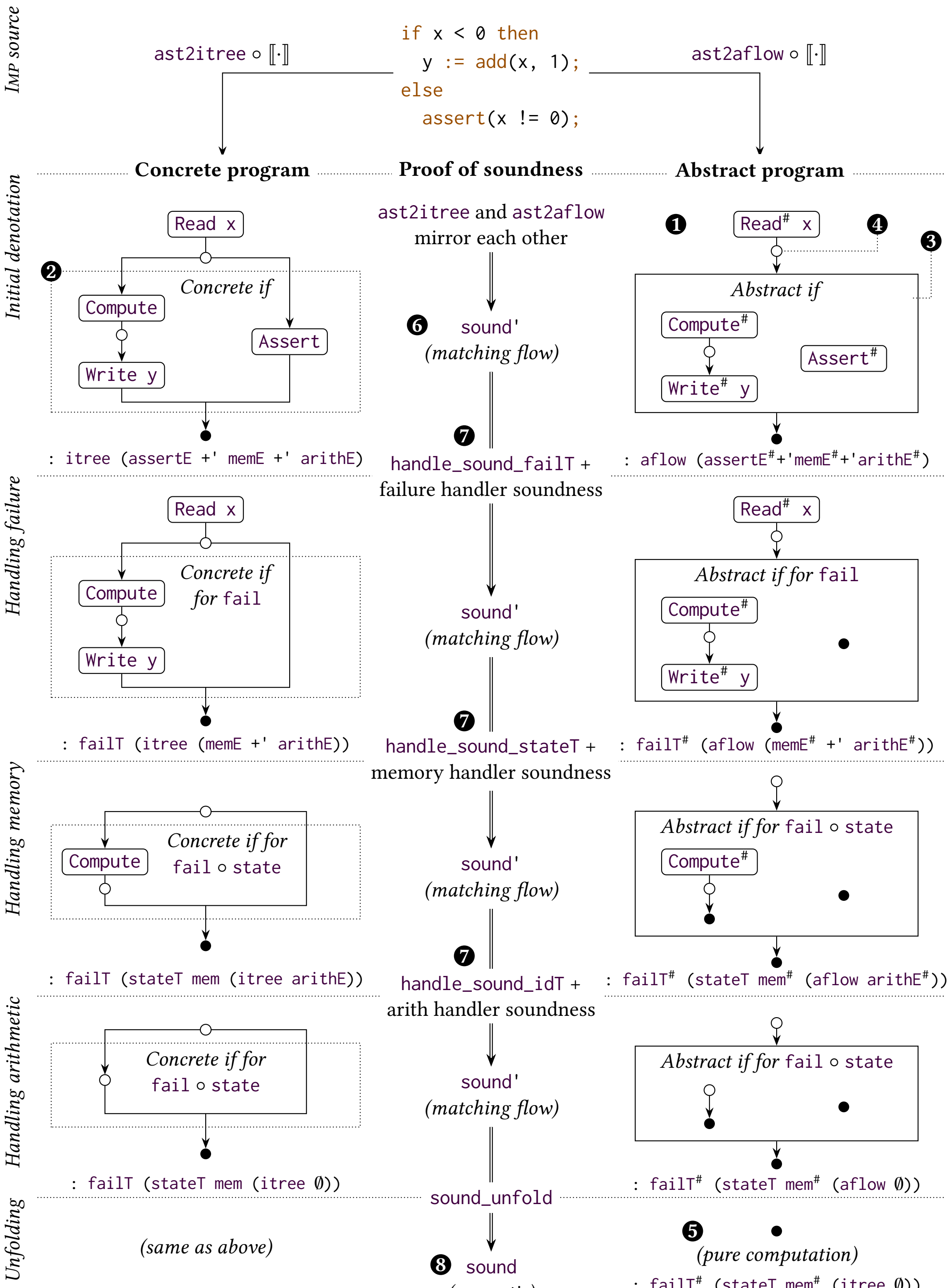

Fig. 6. Overview of the event handling process for a simple Imp program.

Reads from top to bottom; for legend and notations, please see Section 4.1.

Since each concrete event handler introduces a language feature, and each abstract handler a corresponding analysis, each handling layer will contribute a key part of the proof of soundness for the abstract interpreter. In general, 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 ❽ is formalized as the `sound` predicate in Section 6.1. However, this notion cannot be used for partially-handled programs because it ignores events (and traces are not comparable due to differences in control flow).

Extending the concept of soundness to events is not too difficult; all we need is to formalize that "identical" events be used on the concrete and abstract side. Naturally, events with parameters will have different signatures, such as `Write` and `Write#` in Imp:[9]

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

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

This leads to defining Galois connections for events (🍊), which we do for each individual event in the source language, typically by matching arguments:

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

We can now address the soundness of partially-handled programs by introducing an intermediate soundness predicate (dubbed `sound'` ❻) which consists in matching the control flow combinator trees and proving soundness only at the leaves. This means relating return values and events (as in `ret-and` and `do-and` in Section 3) through Galois connections. This new predicate is initially true as a result of the mirroring between `ast2itree` and `ast2aflow`, and the soundness of the user's `ret-and` and `do-and` arguments. It is preserved through each round of handling by a combination of preserving the combinator tree (discussed just below) and the soundness of each abstract event handler w.r.t. its concrete counterpart (❼).

Only at the unfolding stage do we argue that the abstract interpretation algorithms used by the abstract program are correct, which finally implies `sound`. The definitions and proofs for both predicates are properly detailed in Section 6.

4.2 Preserving the combinator tree during event handling

So far in this section, we have assumed that event handling did not affect the nested structure of control flow combinators in either program. Upon closer inspection however, it is not obvious that this tree should remain the same when handling events. There are two reasons for this:

(1) Some monadic handlers add new data (e.g. global state) in the concrete program. Since the abstract program explores multiple (often independent) paths of the concrete program, new data in the abstract program should flow along the control flow paths of the analyzed program, not the control flow paths of the analysis algorithms.
(2) Some monadic handlers simply add new control flow (e.g. failure) in the concrete program. In this case, it is not even immediately clear whether a loop that might fail midway still counts as a loop and whether the associated algorithms implemented in the abstract program correctly account for this option.

To illustrate (1), consider the C-like expression ⟨*condition*⟩ ? ⟨*true-value*⟩ : ⟨*false-value*⟩, 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 will compute an approximation of both

[9] The abstract return type could also be $\mathbb{1}$, but using a lattice Galois-connected to the original type is more consistent.

options using a joining "algorithm", along these lines:

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

Notice how this models two different paths of the concrete program, but in terms of the abstract program it is simply a normal sequence. If we tried to implement this abstract program as an ITree and use the normal state monad handler w.r.t. some handler h to add some global state, we would get the following incorrect data flow:

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

The final state s' of the *true* branch is used as the initial state for evaluating the *false* branch (instead of s) and ignored in the join (instead of being joined with s''). This happens because adding state to a joining algorithm for analyzing pure programs does *not* result in a joining algorithm for analyzing stateful programs. As a result, event handling in the abstract world requires each algorithm (and therefore each abstract control flow combinator) to change in subtle ways to account for new monadic effects.

With this in mind, reason (2) is more of the same, except that the changes needed in algorithms to account for new concrete control flow are usually more substantial than to account for new data.

Our solution to this issue is to use parameterized control flow structures that always account for the possibility of added state and non-local exits; they generalize, in a way, the usual notion of what a "conditional" or a "loop" is. Figure 7 shows our models for two of these structures, the sequence and `do`-loop. The left column shows the concrete version of each combinator while the right column shows the interpretation algorithm that approximates it. Each of the computations relies on *parameters* which are pure computations used to selectively enable or disable features of the algorithm.

Focusing on the concrete sequence first (top left), we see that the sequence point after the execution of $\texttt{p}_1$ runs a "step" function that determines whether a non-local exit was taken (labeled `ERROR`), in which case the program exits immediately, returning a value of type $\texttt{U}_2$. Otherwise, the second half $\texttt{p}_2$ is run (path labeled `OK`) with the normal return value of $\texttt{p}_1$ of type $\texttt{T}_1$.

Writing the `step` function requires some introspection into the return type $\texttt{U}_1$ of $\texttt{p}_1$; in practice this type is isomorphic to $\texttt{T}_1 + \texttt{U}_2$ at any given time, but interestingly it is not practical to *define* it that way. This is because during handling $\texttt{T}_1$, $\texttt{U}_1$ and $\texttt{U}_2$ change (e.g. into pairs when handling into the state monad, some of them into options when handling in the failure monad), so the isomorphism also changes, and it is significantly easier to factor that into the `step` function than to add post-processing into $\texttt{p}_1$. This explains why there are more intermediate types in these combinators than datatypes manipulated by the program.[10]

The concrete `do`-loop combinator works in much the same way. The loop `body` first runs on the initial value `init` of the loop variable, and returns a result of type $\texttt{U}_1$ indicating either a successful step or a non-local exit. This time the successful value of type $\texttt{U}_2$ comes equipped with a true-ness predicate (from a `BooleanDomain` class not discussed in detail here) which drives the decision to keep looping or return. As before, $\texttt{U}_2$ is basically isomorphic to $\texttt{Value} * (\texttt{T} + \texttt{U}_3)$ but defining it this way creates issues during handling, so a new introspection function `dist` is introduced.

[10]As a convention, we name $\texttt{T}_n$ types that contain only material data manipulated by the concrete program (return values, states...) and $\texttt{U}_n$ types that also contain control flow information (here, failures).

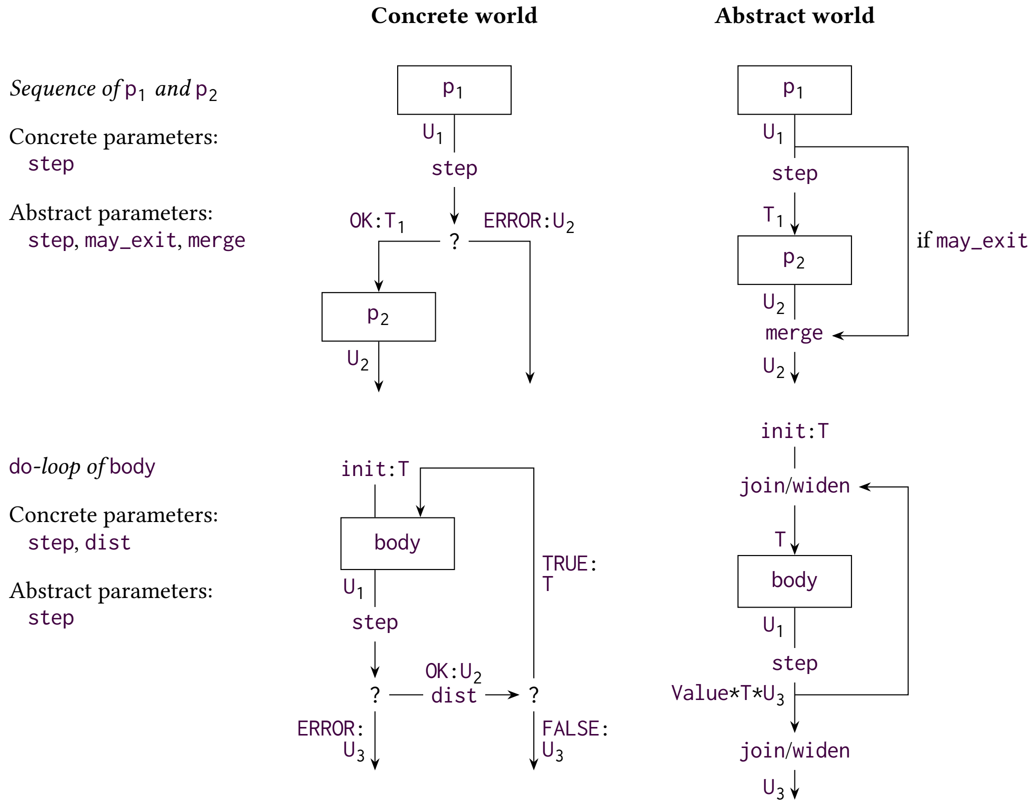

Fig. 7. Diagram representation of parameterized seq and do-loop combinators.

The abstract combinators replicate much of this formalism, but are generally more linear because they analyze all paths. For instance, the abstract sequence combinator evaluates both the `ERROR` and `OK` paths and joins the result with the `merge` function at the end. `merge` is essentially a lattice join except that it avoids joining when the error path is statically not taken, as conservatively evaluated by the `may_exit` function. (It is not enough to rely on join-with-⊥ identities here because the error path can carry extra data such as global states.) The abstract `do`-loop combinator follows the same logic, but this time, a mixture of joining and widening is used to implement the fixpoint approximation scheme. Note how $\mathtt{U}_1$ is introspected as a *product* $\mathtt{Value*T*U}_3$ since the abstract interpreter looks at all paths.

Having these parameterized flow combinators allows us to perform event handling by adding new monadic effects into the abstract program simply by reinstantiating them all with updated parameters. These combinators are the least modular part of the design as they need to deal with every monadic effect supported by the source language. Still, many control flow mechanisms derive from simple primitives (non-local exits for instance model `abort()` but also `break`, `continue`, the first halves of `goto` and `try-catch`, and so on), making this a worthwhile endeveavor in our view.

5 IMPLEMENTING THE ABSTRACT INTERPRETER

We now describe the programmatic side of the library in more detail. We first introduce the `aflow` monad, and its primitive control flow structures. Then, we discuss the `SurfaceAST` previously shown

Library code

```
Inductive aflow (E: Type → Type) (R: Type): Type :=
  | Ret (r: R)
  | Vis {T} (e: E T) (k: T → aflow E R)
  | Seq {U₁ T₁ U₂: Type#}
      (step: U₁ → T₁) (may_exit: U₁ → bool) (merge: bool → U₁ → U₂ → U₂)      (* monadic params *)
      (p₁: aflow E U₁) (p₂: T₁ → aflow E U₂) (k: U₂ → aflow E R)
  | Fixpoint {T U₁ U₃ Value: Type#} `{BooleanDomain Value}
      (step: U₁ → Value * T * U₃)                                            (* monadic params *)
      (body: T → aflow E U₁) (init: T) (k: U₃ → aflow E R)
  | TailMRec (* ... *).
```

Fig. 8. Definition of the aflow monad (🔥).
Type# *is a* **Type** *equipped with a* Lattice.

in Section 3 that enables dual concrete/abstract denotations. Finally, we summarize our supported control flow combinators and their varied implementation methods.

5.1 Primitive control flow structures and the aflow monad

The aflow monad (defined on Figure 8) is a variant of the freer monad with built-in control flow structures. Its monadic structure is based on the Ret constructor and a bind operation that propagates recursively through each constructor's continuation k: _ → aflow E R. We emphasize that this bind represents *a sequence in the abstract interpreter* which, as discussed in Section 4, does not carry monadic effects during handling—unlike the sequence combinator. The Vis constructor provides the freer monad structure and behaves identically to the Vis constructor of itree.

The remaining constructors represent primitive control flow structures, which are the basic building blocks of the abstract versions of the control flow combinators.

- Seq computes the sequence of two computations, accounting for monadic effects added by event handling (and is the abstract sequence combinator from Figure 7);
- Fixpoint computes a post-fixpoint of a loop body (and is the abstract do-loop combinator from Figure 7);
- TailMRec computes a post-fixpoint of a family of mutually-tail-recursive functions.

The key feature of aflow is that it implements event handling by using the parameters discussed in Section 4.2. Now, 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 handling in the abstract world boils down to two things: (1) using richer lattices 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: state (with the state monad transformer) and non-local exits (with the failure monad transformer). The definitions for the traditional concrete transformers and our chosen implementations for the abstract ones are summarized in Figure 9. The handlers for stateT# and failT# are shown on Figure 10—we only show the Ret, Vis and Seq for conciseness and refer the interested reader to the formal development. Most of their work consists in reinstantiating the intermediate types and combinator parameters to enable the effect-internalization features provided by the algorithms. Note that while we focus on the abstract program here, the effect of handling in the concrete program can also be written as a parameter update, mirroring the process.

A detailed look at the code shows that in the state monad, handle_state# on Figure 10, an extra global state s: S is provided as input and returned along the output. Vis supplies it to the event

Transformer	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).

Fig. 9. Summary of the monads in our implementation.

handler, which allows state events to be substituted with pure computations. Seq's introspection functions are updated to indicate that state doesn't cause failure (may_exit unchanged) but it is affected if a failure occurs elsewhere (merge joins it when b = true).

The failure monad, $handle\_fail^{\#}$ on Figure 10, follows a similar structure. In all constructors except Ret, the return value et of the sub-program is passed to the continuation via the failT_kmerge helper, preparing for an eventual merge with the return value er of the continuation. This construction encodes the fact that the program may fail either *before* or *during* k. Notice, however, that it doesn't add a failure in-between (k is always executed). By contrast, a non-local exit *is* added in Seq by programming may_exit to recognize the error flag from p_1 and merge to propagate this information to other transformers.

Of course, these transformations only make sense in light of the associated parameterized algorithms. We first introduced these in the block diagrams of Figure 7, and now conclude their exposition with their implementation at the end of Figure 10. This unfolding step is executed after all events have been handled, such that the event signatures are $E = E^{\#} = \emptyset$ at this point.

5.2 The Surface AST and dual denotation

As illustrated on Figure 6, the key to our dual denotation scheme is to have the concrete and abstract programs use the same flow combinator tree. Each of these structure can be crafted independently by a user of the library, but this targeted structural similarity invites deriving both programs from a single tree representation. We have briefly seen that representation as the SurfaceAST datatype in Section 3. The Surface AST is mostly a convenience feature wrapping control flow combinators, and its most remarkable feature is its dual concrete/abstract parameterization over events and return values. Its simplified definition is shown on Figure 11.

The type is doubly parameterized (E, $E^{\#}$ and R, $R^{\#}$), selecting one denotation with the boolean b through the use of the csum type. An easy way to think about this typing is that for a fixed value of b, occurrences of csum collapse either all to their first argument or all to their second argument.[11]

When passed to ast2itree and ast2aflow, leaves at AST_Ret and AST_Event convert directly to the Ret and Vis constructors of itree E and aflow $E^{\#}$, while combinators are substituted with either the concrete or the abstract implementation. Note how the extra parameterization is gone at this level (save for AST_Do's dist which enables macros, see below) so the complexity of handling monadic events is hidden from the user.

[11]Which means that technically a SurfaceAST is *either* the concrete or abstract program, not both; but since all constructions generate it for both values of b we still treat it as both.

Library code

Handling into the state monad

```
Fixpoint handle_state# (h: E⤳ stateT S (aflow F)): aflow E R → stateT S (aflow F) R :=
  fun p s ⇒ match p with
  | Ret r ⇒ Ret (s, r)
  | Vis e k ⇒ '(s, t) ← h e s;; handle_state# h (k t) s
  | @Seq U1 T1 U2 step may_exit merge p1 p2 k ⇒
     @Seq (S * U1) (S * T1) (S * U2)
          (fun '(s, u1) ⇒ (s, step u1)) (may_exit ∘ snd)
          (fun b '(su1, u1) '(su2, u2) ⇒ (if b then su1 ⊔ su2 else su2, merge b u1 u2))
          (handle_state# h p1 s)
          (fun '(s, t1) ⇒ handle_state# h (p2 t1) s)
          (fun '(s, u2) ⇒ handle_state# h (k u2) s)
  (* ... *).
```

Handling into the failure monad

```
Definition failT_kmerge (et: unit# * T) (p: aflow E (unit# * R)): aflow E (unit# * R) :=
  er ← p;; ret (fst et ⊔ fst er, snd er).
Fixpoint handle_fail# (h: E ⤳ failT# (aflow F)): aflow E R → failT# (aflow F) R := fun p ⇒
  match p with
  | Ret r ⇒ Ret (⊥, r)
  | Vis e k ⇒ et ← h e;; failT_kmerge et (handle_fail# h (k (snd ex)))
  | @Seq U1 T1 U2 step may_exit merge p1 p2 k ⇒
     @Seq (unit# * U1) T1 (unit# * U2)
          (step ∘ snd) (fun '(et, t) ⇒ unit#_to_bool et || may_exit t)
          (fun b '(eu, u) '(et, t) ⇒ (eu ⊔ et, merge (unit#_to_bool eu || b) u t))
          (handle_fail# h p1)
          (handle_fail# h ∘ p2)
          (fun et ⇒ failT_kmerge et (handle_fail# h (k (snd et))))
  (* ... *).
```

Algorithm implementations

```
Fixpoint unfold (p: aflow ∅ R): itree ∅ R :=
  match p with
  | Ret r ⇒ ITree.ret r
  | Seq step may_exit merge p1 p2 k ⇒
     u1 ← unfold p1;;
     u2 ← unfold (p2 (step u1));;
     unfold (k (merge (may_exit u1) u1 u2))
  | Fixpoint step body init k :=
     pfp_u1 ← ITree.iter
       (fun t ⇒ u1 ← unfold (body t);;
                let next_t := t ⊔ proj3_2 (step u1) in
                ret (if next_t ⊆? t then inr u1 else inl (widen t next_t)))
       init;;
     unfold (k (proj3_3 (step pfp_u1)))
  (* ... *).
```

Fig. 10. Key operations in aflow: updating parameterized algorithms when handling into the state ( ) and failure ( ) monads, and their eventual implementations ( ).

5.3 Implementing control flow combinators

The final piece in our puzzle for implementing the interpreters is defining concrete and abstract flow combinators symmetrically at the top-level. We have delayed this presentation until now because we actually provide multiple mechanisms for defining combinators, as a way to balance flexibility (in using accurate abstract interpretation algorithms) with proof effort. Figure 13 shows the six combinators that we implement, which can be categorized into three tiers.

Library code

```
Definition csum (R R#: Type): bool → Type := fun b ⇒ if b then R# else R.
Definition mksum (r: R) (r#: R#) (b: bool): csum R R# b := (* omitted *).

(* All constructors produce a [SurfaceAST E E# R R# b].
   Typeclass instances for BooleanDomain and Galois connections omitted. *)
Inductive SurfaceAST {E E#: Type → Type} (R R#: Type): bool → Type :=
  | AST_Ret {b} (r: csum R R# b)                                          (* ret v and v# *)
  | AST_Event {b} (e: csum (E R) (E# R#) b)                                (* do e and e# *)
  | AST_Seq {b} (p: SurfaceAST T T# b) (k: csum T T# b → SurfaceAST R R# b)     (* v ← p;; k *)
  | AST_If {b} (v: csum Value Value# b) (pthen pelse: SurfaceAST R R# b)   (* if combinator *)
  | AST_Do {b} (dist: csum (U → Value*R*R) (U# → Value#*R#*R#) b)       (* do-loop combinator *)
               (init: csum R R# b) (body: csum R R# b → SurfaceAST U U# b)
  | AST_CFG (* omitted *).
```

Fig. 11. Surface level DSL (🔥) for dual concrete/abstract denotations, with notations in comments.

Library code

Sequence

```
Definition seq (step: U1 → T1 + U2) (p1: itree E U1) (p2: T1 → itree E U2) :=
  u1 ← p1;; match step u1 with
            | inl t1 ⇒ p2 t1
            | inr err ⇒ ret err
            end).
Definition seq# (step: U1 → T1) (may_exit: U1 → bool) (merge: bool → U1 → U2 → U2)
    (p1: aflow E U1) (p2: T1 → aflow E U2) :=
  aflow.Seq step may_exit merge p1 p2 aflow.Ret.
```

do-loop

```
Definition do `{BooleanDomain Value} (step: U1 → U2 + U3) (dist: U2 → Value*T*U3)
  (body: T → itree E U1) (init: T): itree E U3 :=
  ITree.iter (fun t ⇒ u1 ← body t;;
                      ret match step u1 with
                          | inl u2 ⇒ let '(v, loop, leave) := dist u2 in
                                      if BooleanDomain_isfalse v then inr leave else inl loop
                          | inr err ⇒ inr err
                          end) init.
Definition do# `{BooleanDomain Value} (step: U1 → Value*T*U3) (body: T → aflow E U1) init :=
  aflow.Fixpoint step body init aflow.Ret.
```

while-loop macros

```
Definition AST_While {b} init dist (cond body : csum R R# b → SurfaceAST E E# U U# b) :=
  AST_Seq (cond init) (fun c ⇒
    let '(value, r_true, r_false) := (* dist c, but through csum *) in
    AST_If value (AST_Do r_true dist body) (AST_Ret r_false)).

Definition AST_While_unit
    (cond : SurfaceAST E E# Value Value# b)
    (body : SurfaceAST E E# unit unit# b): SurfaceAST E E# unit unit# b :=
  AST_While (mksum tt tt#)
            (mksum (fun v ⇒ (v,tt,tt)) (fun v ⇒ (v,tt#,tt#)))
            (fun _ ⇒ cond)
            (fun _ ⇒ AST_Seq body (fun _ ⇒ cond)).
```

Fig. 12. Simplified definitions of the seq, do-loop and while-loop combinators.

Primitive control flow structures: Some control flow combinators are provided directly with an ITree implementation and a primitive structure in aflow; these are seq, do and tailmrec. The definitions for seq and do, which implement the diagrams from Figure 7, are given at the start of Figure 12. The concrete version in each is similar to the standard implementation discussed in

Combinator	Type	Description
`seq`	Primitive	Pure sequence (with intermediate value)
`do`	Primitive	Do-loop with return value (fixpoint)
`tailmrec`	Primitive	Family of mutually-tail-recursive sub-programs
`if`	Specializes `tailmrec`	Conditional (with pure condition)
`cfg`	Specializes `tailmrec`	Assembly/IR-style Control Flow Graph
`while`	AST macro	C-style while loop (with effectful condition)

Fig. 13. Summary of combinators provided by the library.

Section 2, but with state and failure parameterized. The abstract version is just the corresponding `aflow` constructor, since it's only substituted with an implementation *after* handling events.

Specializations of primitive structures: The `tailmrec` combinator is fairly expressive, as it captures all kinds of intra-functional control flow that traditionally gets compiled down to CFGs in IRs, such as C-style conditions, loops, `switch` statements, etc. We define `cfg` as an instance of it (essentially just encoding the fact that there is only one entry point) and `if` as the restricted case of a `tailmrec` with two non-recursive subprograms.

Naturally, the abstract interpretation of an `if` statement is more accurately computed with a simple join than with `tailmrec`'s general fixpoint approximation scheme. The library allows us to specify and use such a specialized algorithm once it is proven correct. This allows for maintaining accuracy with significantly less proof effort than defining a new primitive structure because `if` gets `tailmrec`'s preservation-through-handling properties for free.

AST macros: Finally, we provide a `while`-loop combinator as a "macro" in the surface AST. `while(c) b` unfolds to the equivalent of `if(c) { do b while(c) }` before the concrete and abstract interpreters are extracted. The code for this as well as the unit-version used by Imp in Section 3 is shown at the end of Figure 12. With this method, no specialized algorithm can be used for interpreting the loop in the abstract program, but no proofs are required at all.

6 LAYERED PROOF OF SOUNDNESS

After having defined a concrete and an abstract interpreter together, we finally turn out attention to the formal certification of the latter w.r.t. the former.

The core property to prove is the following `sound` predicate, which captures the soundness condition for programs with empty event signatures. 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:

$$\begin{aligned}&\texttt{sound}\ (p : \texttt{itree}\ \emptyset\ \texttt{R})\ (p^\# : \texttt{aflow}\ \emptyset\ \texttt{R}^\#) \triangleq\\ &\forall\, r\, r^\#,\ p \text{ returns } r \longrightarrow\\ &\qquad \texttt{unfold}\ p^\# \text{ returns } r^\# \longrightarrow\\ &\qquad r \in r^\#,\end{aligned}$$

where "p returns r" expresses that the computation terminates with value r.[12] 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.

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

The top-level theorem we establish then simply states that the interpreters are related by `sound` after supplying suitable initial states. For instance for Imp (🔥):

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

We go over the entire proof process in the next section and clarify which proof obligations need to be provided by the user in Section 6.2.

6.1 Generic meta-theory

As discussed in Section 4, most of the proof of soundness is conducted over a stronger notion of soundness that tracks the control flow tree, and is only lowered down to `sound` once all events have been interpreted. We call this tree-aware predicate *flow soundness* (🔥), and define it with

$$\texttt{sound'}\ (p : \texttt{itree E R})\ (p' : \texttt{aflow}\ \texttt{E}^\#\ \texttt{R}^\#) : \texttt{Prop}$$

which asserts that p and p' have identical tree structure, and:

- Pairs of leaves (pure value and events) are related by appropriate Galois connections;
- Pairs of nodes (always the concrete and abstract version of the same combinator) use abstract parameters that correctly approximate the concrete parameters. The definition of "sound parameters" depends on each algorithm and we shall treat it as opaque here.

This predicate is initially true for nodes because `ast2itree` and `ast2aflow` produce identical trees with sound initial parameters, and it is also true for leaves if the user's denotation supplies appropriate values and events in the `ret-and` and `do-and` statements.

Preserving `sound'` *during handling.* The key property of `sound'` is that it is preserved during event handling as long as related concrete/abstract get handled into sound sub-programs. This requirement is, of course, where the soundness of the abstract analysis for each language feature comes into play: the user needs to prove that their abstract handlers are sound. The predicate for this varies slightly depending on the monad transformer at hand, but is otherwise straightforward; it is shown below. The predicate `evl_in` is the Galois connection for events.

```
Definition handler_sound_idT (h: E ⤳ itree F) (h#: E# ⤳ aflow F#) :=
  ∀ U U# (e: E U) (e#: E# U#), evl_in e e# → sound' (h e) (h# e#).

Definition handler_sound_stateT (h: E ⤳ stateT S (itree F)) (h#: E# ⤳ stateT S# (aflow F#)) :=
  ∀ U U# (e: E U) (e#: E# U#) (s: S) (s#: S#),
    evl_in e e# → s ∈ s# → sound' (h e s) (h# e# s#).

Definition handler_sound_failT (h: E ⤳ failT (itree F)) (h#: E# ⤳ failT# (aflow F#)) :=
  ∀ U U# (e: E U) (e#: E# U#), evl_in e e# → sound' (h e) (h# e#).
```

The library provides theorems for lifting this handler soundness from individual events to entire programs, thus accomplishing the preservation step shown by double arrows in the middle column of Figure 6. There is one such theorem per monad transformer in which we handle events, named `handling_sound_*`. Their proofs mostly express that the process of updating control flow combinators' parameters during event handling maintains their (opaque) soundness property. As an example, the preservation theorem for the state monad (🔥) is stated as

```
Lemma handling_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' (handle_state h t s) (handle_state# h# f# s#).
```

After handling all events, we finish by unfolding the abstract combinators. It is at this stage that we finally prove that our parameterized abstract interpretation algorithms are sound approximations of their associated concrete control flow structures. These lemmas are slightly intricate to state due to the parameterization, but are otherwise unsurprising. Here is for example the statement for the soundness of the sequence combinator ().

```
Lemma sound_seq
    step (p1: itree E U1) (p2: T1 → itree E U2)                          (* Concrete parameters *)
    step# may_exit# merge# (p1#: aflow E# u1#) (p2#: T1# → aflow E# U2#): (* Abstract parameters *)
  sound p1 p1# →
  (∀ t t#, t ∈ t# → sound (p2 t) (p2# t#)) →
  (* step#, may_exit#, merge# sound w.r.t. step *) →
  sound (seq step p1 p2) (seq# step# may_exit# merge# p1# p2#).
```

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, as used at the bottom of Figure 6.

6.2 User-specific proof obligations: the case of Imp

In general for a pair of interpreters defined in the style of Section 3 which use monad transformers and flow combinators from the library, the user has three kinds of proof obligations.

(1) After providing lattices for abstract types and Galois connections relating them to concrete types, prove related algebraic laws;
(2) Prove that values and events specified in `ret-and` and `do-and` are sound;
(3) Prove that pairs of concrete and abstract handlers are sound.

In Imp's case, the proof effort is particularly minimal because both the interval domain and the handling of the finite memory storage for variables are also part of basic library utilities. In fact, all the lemmas proven in the example file `ImpArithFail.v` () (excluding two boilerplate one-liners) are listed in Figure 14.

The first step is to establish the soundness of the ASTs. The proof proceeds by induction on the input program's syntax, but all nodes in the tree are handled by appropriate constructors of a library-side `SoundAST` predicate. The only non-trivial obligations are for leaves, for which we apply either `SoundAST_Ret` or `SoundAST_Event`, which reduce to goals about Galois connections. In `stmt_sound`, these are all closed by `now` or `easy`.

A slightly more exciting proof can be found in the next section showing that the handlers are sound. This is where most of the analysis for interesting language features is (in this case: arithmetic, memory, and assertions), and also where our modular proof design shines. We show the proof for `h_arith_sound`, which after listing the cases for each event reduces to proving that the interval domain provides sound approximations of integer arithmetic operators. `h_assert_sound` is similar but goes through more API layers not shown in this paper. Finally, the memory handler is already proven sound by a library utility.

We emphasize that the ability to separate these three proofs is a direct benefit of using layered monadic handling when defining the interpreters. Reusing language components with proven analyses such as our basic memory handlers, while seemingly innocuous, is also made possible by the unique modularity of this design.

Which leads into the final theorem, `imp_interp_sound`. This theorem is a direct transcript of Figure 6 from bottom to top. It goes through every layer by chaining handlers until it reaches the

User code

Soundness of initial ASTs

```
Lemma expr_sound (e: expr): SoundAST ⟦e⟧_e ⟦e⟧_e.
Proof. (* 7 lines *) Qed.

Lemma stmt_sound (s: stmt): SoundAST ⟦s⟧ ⟦s⟧.
Proof.
  induction s.
  - apply SoundAST_Seq; [apply expr_sound|]. intros; now SoundAST_Event.
  - now apply SoundAST_Ret.
  - apply SoundAST_Seq; [apply expr_sound|]. intros; now SoundAST_Event.
  - now apply SoundAST_Seq.
  - cbn. apply SoundAST_Seq; [apply expr_sound|]. intros. now apply SoundAST_If.
  - cbn. apply SoundAST_While; try easy.
    * intros; apply expr_sound.
    * intros. apply SoundAST_Seq; auto. intros; apply expr_sound.
Qed.
```

Handlers

```
Lemma h_arith_sound: handler_sound_idT (h_arith (itree ∅)) (h_arith# (aflow ∅)).
Proof.
  intros ? ? [] [] H; try now intuition auto.
  - apply sound'_ret. destruct H as [→ H]. now apply num_unary_sound.
  - apply sound'_ret. destruct H as [→ H]. now apply num_binary_sound.
Qed.

Lemma h_assert_sound: handler_sound_failT (h_assert (itree ∅)) (h_assert# (aflow ∅)).
Proof. (* 9 lines *) Qed.
```

Final soundness theorem

```
Theorem imp_interp_sound (s: stmt) s1 s2:
  s1 ∈ s2 → sound (imp_interp (ast2itree ⟦s⟧) s1) (eval# (ast2aflow ⟦s⟧) s2).
Proof.
  intros Hinit.
  apply sound_unfold.                                               (* Unfolding *)
  apply handling_sound_idT.                                        (* Arith layer *)
  { apply h_arith_sound. }
  apply handling_sound_stateT.                                    (* Memory layer *)
  { apply Hinit. }
  { apply handler_sound_stateT_case, IMPMemory.handle_amem_sound. }
  apply handling_sound_failT.                                     (* Assert layer *)
  { apply handler_sound_failT_case, h_assert_sound. }
  apply sound_ast2itree_ast2aflow, stmt_sound.                    (* Initial ASTs *)
Qed.
```

Fig. 14. Proof of soundness of the Imp interpreters defined in Section 3.

surface AST, at which point the combination of `stmt_sound` and a library theorem concludes. This leaves us with an executable reference interpreter and a certified analyzer both derived from a single denotation of a simple language.

6.3 Extending the library

While we are confident that the theory underlying this paper is expressive enough to cover a wide range of applications, scaling to realistic languages and analyses will naturally require support for more control flow combinators and monad transformers.

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`.

The effort needed to add a new control flow combinator depends on the applicable method, as discussed in Section 5.3. AST macros are free and specializations of existing combinators like `tailmrec` require limited effort. The only proof obligations in this case are the preservation of the specialized shape (e.g. an `if` being two non-recursive blocks) during handling, and the soundness of the specialized algorithm (e.g. a join of both branches). Adding a new primitive control flow structure in `aflow` requires providing an associated parameterized analysis algorithm, which is a non-trivial abstract interpretation question, but otherwise follows a recurring template.

Adding support for a new monad transformer is the most transversal extension, mostly because this requires analysis algorithms to account for any new data- or control-flow from that transformer, which impacts a lot of code. We believe that this approach can still scale to capture common control flow mechanisms, which are for the most part consistent across large numbers of real-world programming languages.

7 RELATED WORK

The seminal paper by Cousot and Cousot [8] has spawned an exceptionally rich literature around the abstract interpretation framework. We refer the interested reader to recent introductory books [7, 34], 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 [29]. Later on, Pichardie identified during his PhD [30] 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 [17] 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 [25] 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` [9]. 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 similarly to how 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 [12, 13] 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 and is mechanized in Coq as well. Although Skeleton share some similarities with our approach, the derivation of abstract interpreters from them is essentially ad-hoc. In particular, no principled treatment of effects of the kind our framework offers is supported. Albeit with quite distinct objectives, Boulmé and Maréchal [3] have also explored the use of monadic semantics to justify the soundness of abstract computations, in the polyhedral domain. Their approach is significantly different to ours: they fix globally the domain, and hence the monad, of interest; they rely on external oracles to embed an impure monad in Coq; they use a form of Dijkstra monad to characterize abstractly the abstract domain itself. Their work may offer hints to expand ours to richer domains.

Out of the realm of type theory, a wide range of non-mechanized, but paper-proved, frameworks for the modular construction of sound abstract interpreters have been built in general purpose programming languages.

Most notably, Darais et al. [10] adapt Van Horn and Might's so-called *Abstracting Abstract Machine* [36, 39] 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* [11], well-behaved monad transformers that transport Galois connections and mappings to suitable executable transition systems.

Keidel et al. [19, 21] have proposed a framework for modularizing the concrete and abstract semantics based on *arrows* [16], 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 [20]. This endeavor is orthogonal to our contributions and could hopefully be formalized and incorporated.

8 CONCLUSION

We have presented a new way of building modular abstract interpreters in dependent type theory: by language features, following the paradigm of layered monadic interpreters. Having identified unique challenges in handling control flow structures, we adapted the paradigm by using parameterized abstract interpretation algorithms as carriers of monadic effects during the handling process. This enabled us to mirror the process for defining concrete and abstract interpreters, and eventually derive them both from a shared denotation. Additionally, the approach provides the expected benefit of decoupling soundness proofs for each language feature, breaking down the complexity of certification significantly. We have packaged all the contributions presented in this paper into a reusable, freely available as an open-source, Coq library.

While we have demonstrated the viability of the approach with mechanized proofs for two simple languages, Imp and Asm, much work remains to be done to scale it up to realistic languages such as C or LLVM IR, and to realistic analyses such as relational domains and complex fixpoint iteration strategies. Some of these extensions appear orthogonal at first (such as iteration strategies), but others present challenges either in terms of abstract interpretation (e.g. designing parameterized algorithms that support more monadic effects) or in the framework itself (e.g. defining relations between values for an arbitrary language with arbitrary operations). Future work will explore these avenues both theoretically and in implementation.

REFERENCES

[1] P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theoretical Computer Science*, 300(1):1–45, 2003.

[2] M. Bodin, P. Gardner, T. P. Jensen, and A. Schmitt. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.*, 3(POPL):44:1–44:31, 2019.

[3] S. Boulmé and A. Maréchal. Refinement to certify abstract interpretations: Illustrated on linearization for polyhedra. *J. Autom. Reason.*, 62(4):505–530, 2019.

[4] V. Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, Volume 1, Issue 2, July 2005.

[5] N. Chappe, P. He, L. Henrio, Y. Zakowski, and S. Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proc. ACM Program. Lang.*, 7(POPL):1770–1800, 2023.

[6] A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. *Electronic Proceedings in Theoretical Computer Science*, 129, 09 2013.

[7] P. Cousot. *Principles of Abstract Interpretation.* The MIT Press, September 2021.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of programming languages (POPL)*, pages 269–282. ACM, 1979.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In M. Okada and I. Satoh, editors, *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*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[10] D. Darais, N. Labich, P. C. Nguyen, and D. Van Horn. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[11] D. Darais, M. Might, and 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*, OOPSLA 2015, page 552–571, New York, NY, USA, 2015. Association for Computing Machinery.

[12] D. Darais and D. Van Horn. Constructive galois connections: Taming the galois connection framework for mechanized metatheory. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 311–324, New York, NY, USA, 2016. Association for Computing Machinery.

[13] D. C. Darais. *Mechanizing Abstract Interpretation.* PhD thesis, University of Maryland, College Park, MD, USA, 2017.

[14] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[15] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, page 317–331, Berlin, Heidelberg, 2000. Springer-Verlag.

[16] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 05 2000.

[17] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.

[18] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017.

[19] S. Keidel and S. Erdweg. Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3:1–28, 10 2019.

[20] S. Keidel, S. Erdweg, and T. Hombücher. Combinator-based fixpoint algorithms for big-step abstract interpreters. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023.

[21] S. Keidel, C. B. Poulsen, and S. Erdweg. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018.

[22] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. *SIGPLAN Not.*, 50(12):94–105, aug 2015.

[23] N. Koh, Y. Li, Y. Li, L.-y. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic. 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*, CPP 2019, page 234–248, New York, NY, USA, 2019. Association for Computing Machinery.

[24] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[25] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[26] M. Lesani, L. Xia, A. Kaseorg, C. J. Bell, A. Chlipala, B. C. Pierce, and S. Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–31, 2022.

[27] T. Letan, Y. Régis-Gianas, P. Chifflier, and G. Hiet. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *FM 2018 - 22nd International Symposium on Formal Methods*, volume 10951 of *LNCS*, pages 338–354, Oxford, United Kingdom, July 2018. Springer.

[28] C. McBride. Turing-completeness totally free. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction*, pages 257–275, Cham, 2015. Springer International Publishing.

[29] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Master's thesis, Université Paris 7, 1998.

[30] D. Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés.* PhD thesis, Université Rennes 1, 2005. In french.

[31] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Logical Foundations.* Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[32] M. Piróg and J. Gibbons. The coinductive resumption monad. *Electronic notes in theoretical computer science*, 308:273–288, 2014.

[33] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, Z. Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[34] X. Rival and K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective.* The MIT Press, February 2020.

[35] G. Rosu and T. Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[36] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. volume 48, pages 399–410, 06 2013.

[37] L. Silver, P. He, E. Cecchetti, A. K. Hirsch, and S. Zdancewic. Semantics for noninterference with interaction trees (artifact). *Dagstuhl Artifacts Ser.*, 9(2):06:1–06:2, 2023.

[38] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[39] D. Van Horn and M. Might. Abstracting abstract machines. *SIGPLAN Not.*, 45(9):51–62, sep 2010.

[40] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2020.

[41] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[42] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L. Xia, L. Beringer, W. Mansky, B. C. Pierce, and S. Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[43] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, jan 2012.