Abstract interpreters:
A monadic approach to modular verification

Sébastien MICHELLAND [1], Yannick ZAKOWSKI [2], Laure GONNORD [1]

[1] Université Grenoble-Alpes, Grenoble INP, LCIS (Valence)

[2] Inria Lyon

November 25th, 2024

Université Grenoble Alpes

Université de Lyon

ENS DE LYON

Pitching an internship...

Language description

↓

REUSABLE

Semantic components
state, exceptions...

↓

REUSABLE

Control flow combinators

↓

Concrete interpreter

Pitching an internship...

Language description

REUSABLE

Semantic components

state, exceptions...

↓ REUSABLE

Control flow combinators

↓

Concrete interpreter

REUSABLE

(Semantic components)$^{\#}$

state, exceptions...

↓ REUSABLE

(Control flow combinators)$^{\#}$

↓

Abstract interpreter

Pitching an internship...

Language description

REUSABLE

Semantic components

state, exceptions...

↓ REUSABLE

Control flow combinators

↓

Concrete interpreter

CERTIFIED

CERTIFIED

CERTIFIED

REUSABLE

(Semantic components)$^{\#}$

state, exceptions...

↓ REUSABLE

(Control flow combinators)$^{\#}$

↓

Abstract interpreter

1

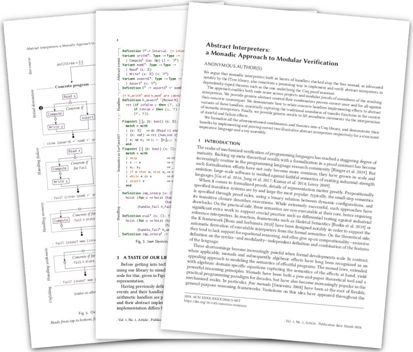
Contributions in this paper

Abstract interpreters: A monadic approach to modular verification

ICFP'24 • HAL • ACM • Source code

1. Abstract interpreters in layered monadic style
   - Imp and Asm
   - Key idea: proper understanding of control flow
   - Analyzer defined by **mirroring interpreter**

2. Proof of soundness is now modular in terms of language features
   - Meta-theorems for **composing components' soundness proofs**
   - Components reusable across languages



Abstract Interpreters: a Monadic Approach to Modular Verification

ANONYMOUS AUTHOR(S)

1 INTRODUCTION

2

Abstract interpreters

a practical recipe

A naive analyzer

How to know possible values of variables at runtime?

- Run the program!

input=4

```
x ← input % 6;        → x=4
y ← 12 - x;           → y=8
z ← 3 * (y / x);      → z=6
```

One output: $(x, y, z) = (4, 8, 6)$

- Ok, but... number of inputs? termination? ⚔️

AI (1/2): from collecting semantics to lattices

- Let's collect all values anyway.

```
x ← input % 6;       → x ∈ {0, 1, 2, 3, 4, 5}
y ← 12 - x;          → (x, y) ∈ {(0, 12), (1, 11), (2, 10), (3, 9), ...}
z ← 3 * (y / x);     → (x, y, z) ∈ {(1, 11, 33), (2, 10, 15), (3, 9, 9), ...}
```

All outputs:
$(x, y, z) \in \{(1, 11, 33), (2, 10, 15), (3, 9, 9), (4, 8, 6), (5, 7, 3)\}$

- Excessive amount of values! ⚔️

AI (1/2): from collecting semantics to lattices

- Let's collect all values and split the variables.

```
x ← input % 6;      → x ∈ {0, 1, 2, 3, 4, 5}
y ← 12 - x;         → y ∈ {7, 8, 9, 10, 11, 12}
z ← 3 * (y / x);    → z ∈ {3, 6, 9, 12, 15, ..., 33, 36}
```

Upper bound on possible outputs:

x ∈ {0, 1, 2, 3, 4, 5}, y ∈ {7, 8, 9, 10, 11, 12}, z ∈ {3, 6, 9, 12, 15, ..., 33, 36}

- Still too many values! ⚔️
- Approximate, but still safe.

AI (1/2): from collecting semantics to lattices

- Let's collect all values and split the variables and approximate sets with intervals.

```
x ← input % 6;      → x ∈ ⟦0, 5⟧
y ← 12 - x;         → y ∈ ⟦12, 12⟧ − ⟦0, 5⟧ = ⟦7, 12⟧
z ← 3 * (y / x);    → z ∈ ⟦3, 3⟧ ∗ (⟦7, 12⟧ / ⟦0, 5⟧) = ⟦3, 36⟧
```

Even upper bound on possible outputs:
$x \in [\![0, 5]\!],\ y \in [\![7, 12]\!],\ z \in [\![3, 36]\!]$

- Tractable
- Approximate, but still safe.

AI (2/2): handling control flow

- Control flow depends on values so we might take multiple paths.

```
x ← input % 6;      → x ∈ {0, 1, 2, 3, 4, 5}
if x < 3            true for x = 0, 1, 2
   y ← x;           → y ∈ {0, 1, 2}
else                true for x = 3, 4, 5
   y ← 12 - x;      → y ∈ {7, 8, 9}
end                 → y ∈ {0, 1, 2} ∪ {7, 8, 9}
```

Bound on possible outputs: $x \in \{0, 1, 2, 3, 4, 5\}$, $y \in \{0, 1, 2, 7, 8, 9\}$

- Join paths with a set union.

AI (2/2): handling control flow

- Control flow depends on values so we use algorithms that account for all paths.

`x ← input % 6;`	$\to$ $x \in [\![0,5]\!]$
`if x < 3`	*true for* $x \in [\![0,2]\!]$
`  y ← x;`	$\to$ $y \in [\![0,2]\!]$
`else`	*true for* $x \in [\![3,5]\!]$
`  y ← 12 - x;`	$\to$ $y \in [\![12,12]\!] - [\![3,5]\!] = [\![7,9]\!]$
`end`	$\to$ $y \in [\![0,2]\!] \sqcup [\![7,9]\!] = [\![0,9]\!]$

Bound on possible outputs: $x \in [\![0,5]\!]$, $y \in [\![0,9]\!]$

- Join paths with the approximation of a set union.

Abstract interpreters: recipe

- Interpret "normally" but replace as follows:

	Concrete	Abstract
Values	`12 : int`	$[\![12, 12]\!]$: `interval`
Operators	$a + b$	$[\![a_1, a_2]\!] +^{\#} [\![b_1, b_2]\!] = [\![a_1 + b_1, a_2 + b_2]\!]$
Conditions	`if e {c1} else {c2} end`	Approximate union of values in c_1 and c_2
Loops	`while e {c}`	Approximate fixpoint of c

1. Replace data types with subset approximations *(lattices)*.
2. Replace control flow structures with specialized algorithms that account for all paths.

3

Layered monadic interpreters

Shallow vs. Deep

(Arguably) more traditional approach:

- Deeply embedded configurations Σ as an inductive
- Specify its semantics $\Sigma \to \Sigma \to \mathbb{P}$

Shallow vs. Deep

(Arguably) more traditional approach:

- Deeply embedded configurations Σ as an inductive
- Specify its semantics $\Sigma \to \Sigma \to \mathbb{P}$

What we consider here:

- Deeply embedded configurations Σ as an inductive
- Shallow representation of those a monadic interpreter: $[\![\cdot]\!] : \Sigma \to M$

Shallow vs. Deep

(Arguably) more traditional approach:

- Deeply embedded configurations Σ as an inductive
- Specify its semantics $\Sigma \to \Sigma \to \mathbb{P}$

What we consider here:

- Deeply embedded configurations Σ as an inductive
- Shallow representation of those a monadic interpreter: $[\![\cdot]\!] : \Sigma \to M$

Potential benefits:

- If M is gentle, *may* be executable;
- $[\![\cdot]\!]$ *may* be built out of reusable components;
- $[\![\cdot]\!]$ *may* be build structurally over Σ.

Monads as models, monads as a programming abstraction

Monad M (for us): a family of types representing a class of effectful programs.

- $M\ R$ is the type of programs returning an R.

Monads as models, monads as a programming abstraction

Monad M (for us): a family of types representing a class of effectful programs.

- $M\ R$ is the type of programs returning an R.

Constructors

- `ret` $(r : R) : M\ R$ — *Pure computation*
- `bind` $(p : M\ T)\ (k : T \to M\ R) : M\ R$ — *Sequence*
- And monad-specific operations.

Monads as models, monads as a programming abstraction

Monad M (for us): a family of types representing a class of effectful programs.

- $M\ R$ is the type of programs returning an R.

Constructors

- ret $(r : R) : M\ R$ — *Pure computation*
- bind $(p : M\ T)\ (k : T \to M\ R) : M\ R$ — *Sequence*
- And monad-specific operations.

Famously central to Haskell, but can also be used in the Coq language (Gallina).

- ret (fibonacci n / 4) : M nat
- bind p (fun $x \Rightarrow$ ret $(x + 1)$) : M nat (*assuming* $p : M$ nat)

Monads as models, monads as a programming abstraction

Monad M (for us): a family of types representing a class of effectful programs.

- $M\ R$ is the type of programs returning an R.

Constructors

- `ret` $(r : R) : M\ R$ — *Pure computation*
- `bind` $(p : M\ T)\ (k : T \to M\ R) : M\ R$ — *Sequence*
- And monad-specific operations.

Famously central to Haskell, but can also be used in the Coq language (Gallina).

- `ret (fibonacci` n `/` $4) : M$ nat
- `bind` p (`fun` $x \Rightarrow$ `ret` $(x + 1)) : M$ nat (*assuming* $p : M$ nat)

And relators, and equations...

A lightweight extension: monad transformer

State monad transformer for state S adds to a given monad M:

- `get` : (`stateT` M) S
- `set` ($s : S$) : (`stateT` M) `unit`

Failure monad transformer adds:

- `abort` : (`failT` M) $\emptyset$

Example (executable inside of Coq).

- `if` $x = 0$ `then abort else set` $(100/x)$: `failT` (`stateT` M) `unit`

`failT` (`stateT` M) is (almost) fine for IMP, but other languages have different features.

- How can theorems talk about "any monad stack"?

The freer monad

Freer monad for events (E : Type $\to$ Type) has `ret`, `bind` and:

- `trigger` (e : E T) : `freerM` E T (Not executable)

The freer monad

Freer monad for events (E : Type $\to$ Type) has `ret`, `bind` and:

- `trigger` $(e : E\ T)$: `freerM` $E\ T$ (Not executable)

E is a description of the language's operations' signatures.

- `Variant stateE := Get : stateE` S | `Set` $(s : S)$: `stateE unit`
- `Variant failE := Abort : failE` $\emptyset$
  - `freerM stateE` $\approx$ `stateT` Id
  - `freerM (failE + stateE)` $\approx$ `failT (stateT` Id`)`

`freerM E` doesn't implement the events, but it's useful as an intermediate representation.

The freer monad

Freer monad for events (E : Type $\to$ Type) has `ret`, `bind` and:

- `trigger` (e : E T) : `freerM` E T (Not executable)

E is a description of the language's operations' signatures.

- `Variant stateE := Get : stateE` S | `Set` (s : S) : `stateE unit`
- `Variant failE := Abort : failE` $\emptyset$
  - `freerM stateE` $\approx$ `stateT` Id
  - `freerM (failE + stateE)` $\approx$ `failT (stateT` Id`)`

`freerM E` doesn't implement the events, but it's useful as an intermediate representation.

Interaction Trees [**XZHH+20**]: `(itree E)` is `(freerM E)` with non-termination

Algebraic effects and their handlers?

Freer monad for events (E : Type → Type) has `ret`, `bind` and:

- `trigger` ($e : E\ T$) : `freerM` $E\ T$ (Not executable)

Operations have signatures, but little semantics. What can we do?

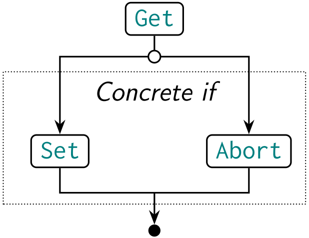
- Extend the signature to a theory: we get algebraic effects;
- Look for their *handlers*.
  - Provide a handler : $E \rightsquigarrow M$;
  - Double check you built a model of your algebra;
  - Get a lifting to computations : $FreerE \rightsquigarrow M$.

If you are happy with one shot continuations, implementing this in Coq is easy.

A layered interpreter for Imp

Imp program: `if x = 0 { abort() } else { x = 100/x }`

Initial denotation

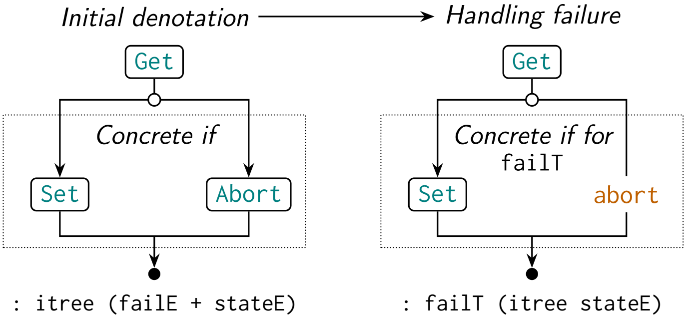


`: itree (failE + stateE)`

- Control flow structures: sequence (drawn ∘) and if change signature when handled.

A layered interpreter for IMP

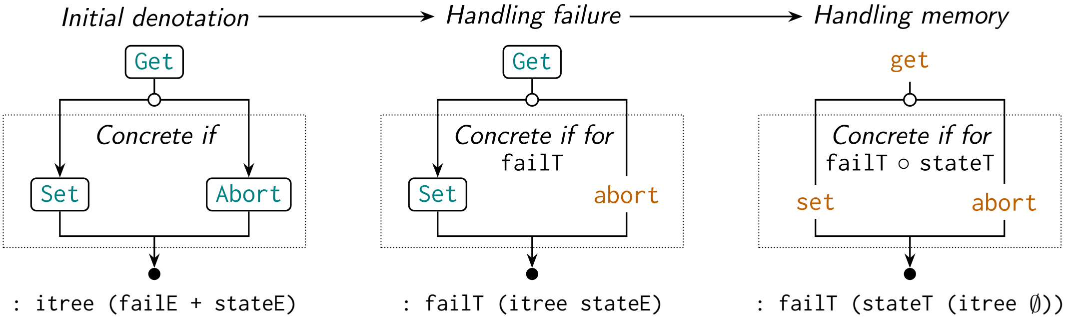
IMP program: `if x = 0 { abort() } else { x = 100/x }`



- Control flow structures: sequence (drawn ∘) and if change signature when handled.

A layered interpreter for Imp

Imp program: `if x = 0 { abort() } else { x = 100/x }`



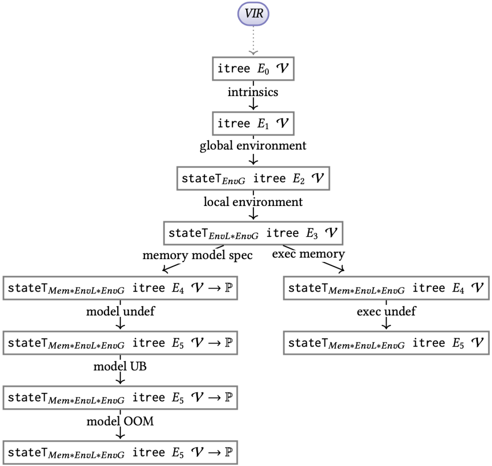
- Control flow structures: sequence (drawn ∘) and if change signature when handled.

Where I would like to get to

A semantics for (sequential) LLVM IR built as a layered interpreter using itrees.

- Jourdan et al. analyse C (Verasco),
- Bodin et al. analyse Javascript,
- We would like to analyze LLVM IR?

For now, we tackle IMP and ASM, but exploring a new methodology.

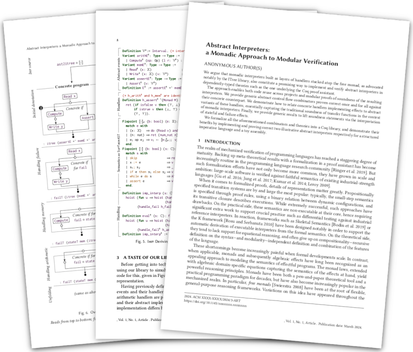


Abstract interpreters: A monadic approach to modular verification

ICFP'24 • HAL • ACM • Source code

1. Abstract interpreters in layered monadic style
   - IMP and ASM
   - Key idea: proper understanding of control flow
   - Analyzer defined by **mirroring interpreter**

2. Proof of soundness is now modular in terms of language features
   - Meta-theorems for **composing components' soundness proofs**
   - Components reusable across languages


Abstract Interpreters: a Monadic Approach to Modular Verification

ANONYMOUS AUTHOR(S)

1 INTRODUCTION

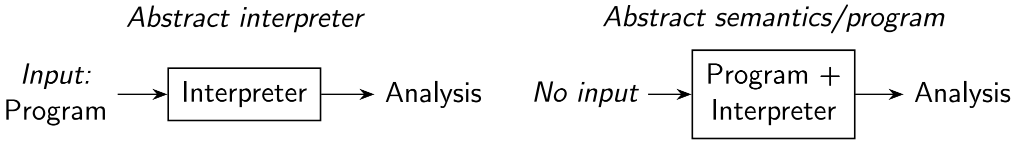
3 A TASTE OF OUR LIB

4

Layered monadic abstract interpreters

there's got to be a better name

On the nature of the "abstract semantics" we build



⚠ Very important

The abstract semantics is a hybrid of **both** the analyzed program and analyzer.

Like an abstract interpreter partially evaluated on a given input program.

- Can we even build abstract programs with the layered event handling process?

Hybrid flow impacts event handling

The abstract semantics is a hybrid of **both** the analyzed program and analyzer.

Handling events with `failT` adds the ability to crash. But:

A potentially-crashing tool that analyses pure programs	**is not**	A tool that analyses potentially-crashing programs

We need "monad transformers" that extend the analysis, not the analyzer.

Hybrid flow impacts event handling

The abstract semantics is a hybrid of **both** the analyzed program and analyzer.

Handling events with `failT` adds the ability to crash. But:

A potentially-crashing tool that analyses pure programs	**is not**	A tool that analyses potentially-crashing programs

We need "monad transformers" that extend the analysis, not the analyzer. So:

1. Implement control flow analyses that know about states/crashes
2. Enable these features during event handling

The key: parameterized control flow algorithms

Example: parametrized sequence.

Can handle pure programs

- may_exit always false, step always OK

Can handle programs in stateT

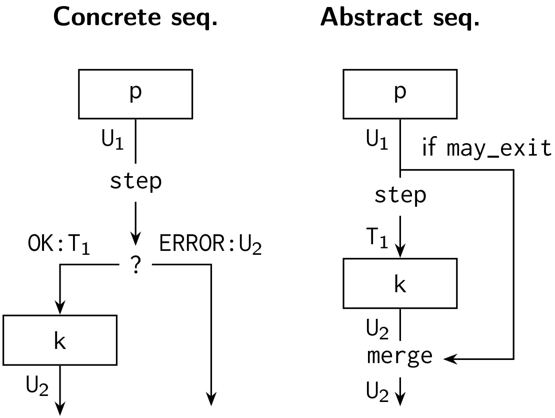
- $T_n = S \times ..., U_n = S \times ...$

Can handle programs in failT

- U_n = option..., use step/may_exit

Event handling in abstract program:

1. Replace events as usual
2. Update control flow algorithms' parameters to add state/failure/etc



The lens that clears it up: monad of control flow

We are in fact describing a freer monad with explicit control flow operations.

Monad of control flow `aflow` for events (E : Type → Type) has `ret`, `bind` and:

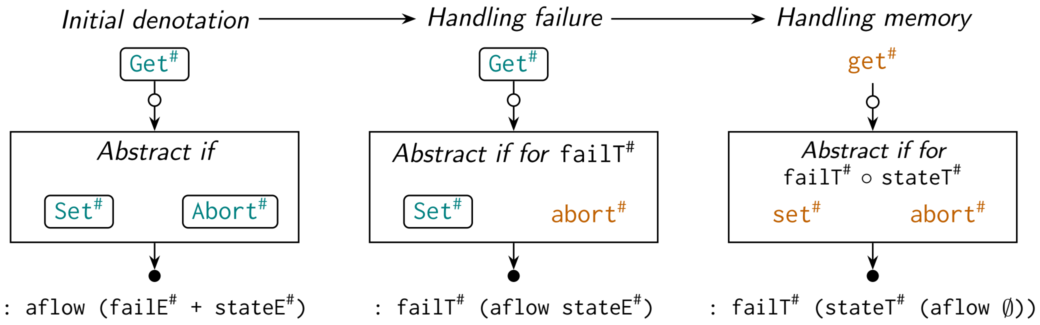
- `trigger` ($e : E\ R$) — *Freer monad*
- `seq` (p : `aflow` $E\ T$) ($k : T \to$ `aflow` $E\ R$) ⟨*params...*⟩ — *Source sequence*
- `if` ($p_1\ p_2$: `aflow` $E\ R$) ⟨*params...*⟩ — *Source conditional*
- ... `do`, `while`, `cfg`...

New notion of event handling:

1. Replace events like before
2. **Also** update parameters of control flow analysis algorithms to enable state/failure

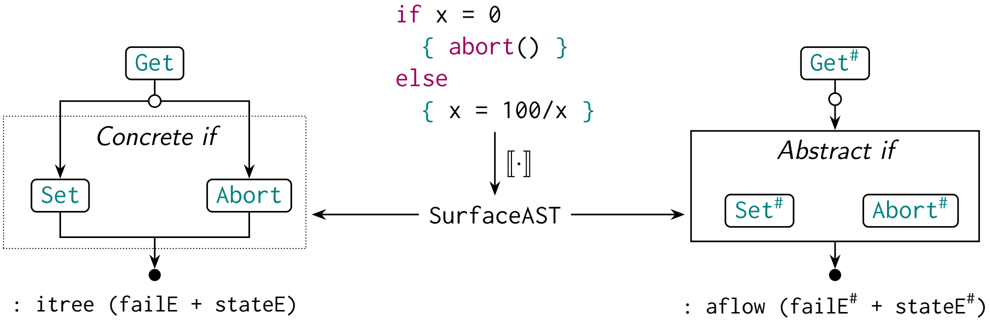
And now: a monadic abstract program

Need abstract events because their parameters/return values become lattices.



This time the if changes a lot with each handling.

Deriving both programs from a single denotation



Shared `SurfaceAST` representation:

- Control flow combinator tree (later projected to `itree` and `aflow`)
- Leaves are `ret` or `trigger` with pairs: $(2, [\![2, 2]\!])$, (`Get`, `Get#`)

Technical aside: combinators, a closer look

We keep `aflow E R` fairly minimal:

- `Ret` $(x : R)$
- `Trigger` $(e : E\ R)$
- `Seq` $(f_1 : \texttt{aflow}\ E\ U_1)\ (f_2 : T_1 \to \texttt{aflow}\ E\ R)$
  $(step : U_1 \to T_1)\ (may\_exit : U_1 \to bool)\ (merge : bool \to U_1 \to R \to R)$
- `Fixpoint` $(\dots)$
- `TailMrec` $(\dots)$

Higher level combinators (`if`, `do`, `cfg`,..) are analyzes implemented directly in `aflow`.
⚠: They still must specify how state and fail update their parameters!

Our implementation

Our Coq development: https://gitlab.inria.fr/sebmiche/itree-ai

Everything formalized and packaged in a library.

- Monad theory, `aflow`, shared denotations with `SurfaceAST`
- Basic lattices and non-relational domains
- Control flow: `seq`, `if`, `do`, `while`, `cfg`

Enough to write two case studies, i.e., abstract interpreters for:

- Imp with arithmetic, state, and failure handled as three successive layers;
- Asm with two layers of state (registers and heap).

⇝ And of course, executable through extraction.

Our implementation

Our Coq development: https://gitlab.inria.fr/sebmiche/itree-ai

Everything formalized and packaged in a library.

- Monad theory, `aflow`, shared denotations with `SurfaceAST`
- Basic lattices and non-relational domains
- Control flow: `seq`, `if`, `do`, `while`, `cfg`

Enough to write two case studies, i.e., abstract interpreters for:

- Imp with arithmetic, state, and failure handled as three successive layers;
- Asm with two layers of state (registers and heap).

⇝ And of course, executable through extraction.

But are those analyzer *sound*?

5

All the proofs are now much easier

Trying it out: proving an IMP analyzer

Syntax and semantics USER

Using concrete/abstract control flow pairs from library

Numerical domain: $\mathbb{Z}$ interval lattice REUSABLE

Soundness of layer #1 (`failT`): assertion `assert()`

- IMP-specific because involves truth values USER

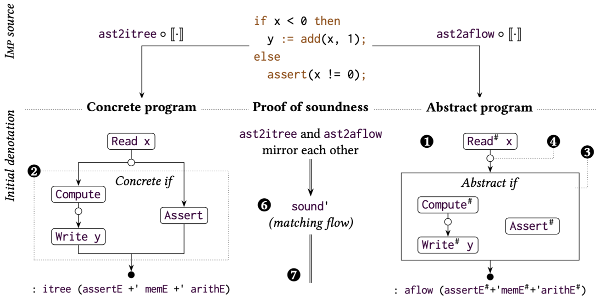
Soundness of layer #2 (`stateT`): variables $x, y, z \leftarrow \ldots$

- Basically just a map lattice `string` $\rightarrow$ value REUSABLE

Soundness of flow analysis algorithms *Meta-theory*

Composing layers' soundness proofs LIBRARY

Bird's eye view

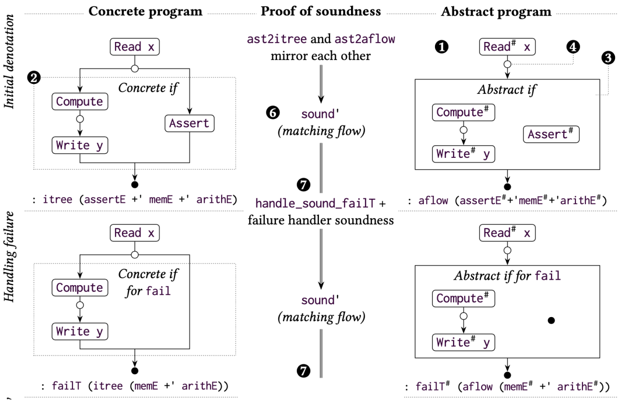


The sound' invariant maintains that:

- the control flow structure of both computations match;
- matching events and values are related through Galois connections.

If programmed through the DSL, for free.

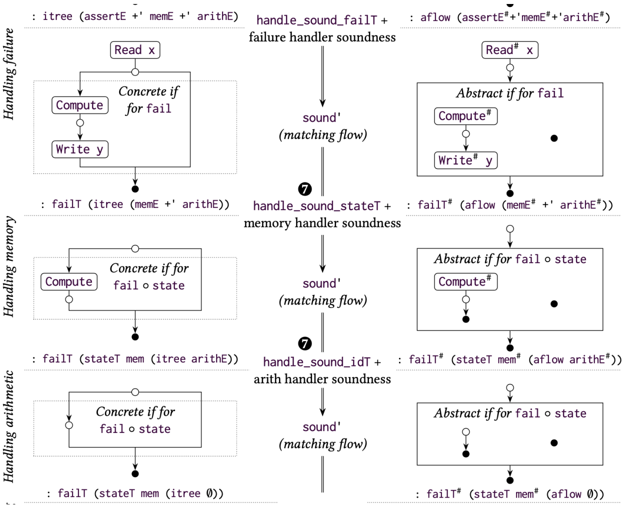
Bird's eye view



We need to preserve sound' by failure interpretation:

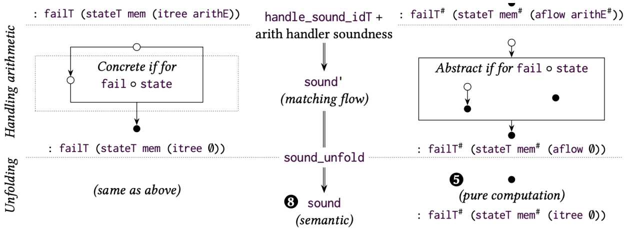
- the user defined handlers must be proven sound;
- the library does the rest.

Bird's eye view



We keep going...

Bird's eye view



Finally, we unfold the implementation of the abstract algorithms: each pair is proven relatively sound in the library, allowing us to conclude.

Certified analysis checklist

Obligation	Who	When
Mirroring of concrete/abstract denotations	User	Every language
Lattice and domains (intervals...)	User	Only once
Language features (state, failure...)	User	Only once
Soundness of parametrized flow algorithms	Library	Every flow structure
Soundness of event handling steps	Library	Every flow structure
Composition of event handling steps	Library	Only one

Analyses for languages features are thus:

- **Modular** (proven independently then composed)
- **Reusable** (`break`, local variables, `abort()`... often the same)

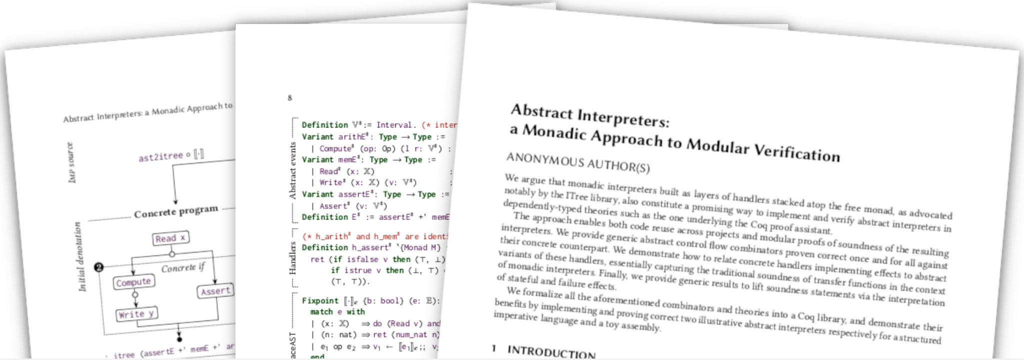
6

Conclusion

More details in the paper, and even more in the code!

The paper: https://hal.science/hal-04628727 (ICFP'24)

The code: https://gitlab.inria.fr/sebmiche/itree-ai



Abstract Interpreters:
a Monadic Approach to Modular Verification

ANONYMOUS AUTHOR(S)

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

Conclusion

Monad-based abstract interpreters are modular and their proofs are too!

Novelties

- Abstract interpreters in layered monadic style + soundness tools
- Identifying the freer monad of control flow

Insights and future work

- Scaling up: new effects; less structured control flow; better analysis algorithms; combining domains...
- Performances: at the moment, `unfold` into itrees, then extract.

Conclusion

Monad-based abstract interpreters are modular and their proofs are too!

Novelties

- Abstract interpreters in layered monadic style + soundness tools
- Identifying the freer monad of control flow

Insights and future work

- Scaling up: new effects; less structured control flow; better analysis algorithms; combining domains...
- Performances: at the moment, `unfold` into itrees, then extract.

Thoughts?